HASKELL METAMODEL FOR ASSESSING SOFTWARE QUALITY

MASTER THESIS

by

ing. H. H. R. F. Vos

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University of the Netherlands, faculty of Management, Science & Technology
Master Software Engineering

to be defended publicly on Thursday 26 October 2017 at 11:00 a.m..

Student number: 838451707
Course code: IM9906
Thesis committee: prof. dr. J. T. Jeuring (chairman), Open University of the Netherlands
dr. B. J. Heeren (supervisor), Open University of the Netherlands
dr. ir. S. Stuurman (supervisor), Open University of the Netherlands
Summary iv

1 Introduction 1

2 Related work 3
  2.1 Metamodels 4
  2.2 Existing source code analysis tools based on a meta-metamodel 5
    2.2.1 Moose 5
    2.2.2 Rascal 6
  2.3 Prior work on software metrics for Haskell 9
  2.4 meta\textsuperscript{HS} metamodel prototype 11
  2.5 Lack of Cohesion of Module Elements (LCOM) source code metric 13

3 Research questions 14

4 meta\textsuperscript{HS} data model layer 18
  4.1 Source code example 19
  4.2 Parser 20
  4.3 Metamodel 21
    4.3.1 Elements 21
    4.3.2 Relations 25
    4.3.3 MetaModel 26
  4.4 Extractors 27
    4.4.1 Program Contains 28
    4.4.2 Module Contains 28
    4.4.3 Module Source 29
    4.4.4 Module Exports 29
    4.4.5 Module Imports 30
    4.4.6 Module Uses 30
    4.4.7 Type classes and instances 32
  4.5 Aggregators 34

5 meta\textsuperscript{HS} EDSL layer 35
  5.1 Metamodel façade 36
    5.1.1 Generation 36
    5.1.2 File operations 36
    5.1.3 Queries 36
    5.1.4 Utility functions 37
  5.2 Graphs 38
    5.2.1 UsesGraph 38
    5.2.2 Utility functions 39
  5.3 EDSL 40
Source code metrics and software visualization are in common use for imperative programming languages and are well supported by the available software tools. Especially tools that are based on a metamodel seem to provide platforms that ease the development of source code metrics and software visualizations.

The availability of a metamodel could provide a boost to the research on software quality for the Haskell programming language by providing a common environment for the development of source code metrics and software visualizations. The development of a metamodel prototype helps to determine whether it is useful to develop such a metamodel and to discover the main design considerations.

This thesis describes the design and implementation of this prototype – named meta^HS – and discusses the main design considerations discovered.

It will conclude that a scalable and extensible metamodel – combined with a well-designed domain specific language layer and a powerful graphical user interface environment – can be a useful tool for Haskell software engineers.
In 1994, Parnas [Par94] describes why software evolution is an important aspect of software engineering: An application that is not modified after the initial release will eventually fail to meet the changing needs of the users and become obsolete. Modifying the software, however, is likely to negatively affect the internal structure of the software, which results in a lower quality of the software and a higher chance of introducing bugs during future modifications.

Therefore, it is necessary to have techniques and tools to perform source code analysis. Examples of these techniques are source code metrics and software visualization. Source code metrics measure certain aspects of the source code. Software visualization allows software engineers to comprehend the internal structure of an application more easily.

Source code metrics and software visualization are in common use for object-oriented programming languages and are well supported by the available software tools. Moose [NDG05] and Rascal [KLV11] are examples of especially powerful environment for source code metrics and software visualization. They are both based on a meta-metamodel and demonstrate that this foundation can result in a platform that eases the development of source code metrics and software visualizations.

Functional programming is a programming paradigm that focusses on the evaluation of expressions instead of the manipulation of state as is common in the object-oriented and imperative programming paradigms. The Haskell programming language is one of the most popular functional programming languages and is often used in research projects on functional programming.

Research on software quality, source code metrics, and software visualization for functional programming languages, and Haskell in particular, is progressing slowly in comparison to the research conducted on software quality for object-oriented programming languages (see Chapter 2).

The availability of a metamodel could provide a boost to the research on software quality for the Haskell programming language by providing a common environment for the devel-
opment of source code metrics and software visualizations. The development of a meta-
model prototype would help to determine whether it is useful to develop such a metamodel
and to discover the main design considerations.

The structure of this document is as follows: Chapter 2 discusses the related work of this
research project. The research questions are presented in Chapter 3. Chapter 4 covers the
realization of the data model layer\(^1\) of the meta\(^{HS}\) prototype \(^2\). The meta\(^{HS}\) EDSL\(^3\) and ex-
tensions\(^4\) layers are discussed in Chapter 5 and Chapter 6 respectively. Chapter 7 describes
the validation results and the conclusions are discussed in Chapter 8.

\(^1\) The data model layer is concerned with extracting information from Haskell source code.
\(^2\) meta\(^{HS}\) is the name given to the metamodel prototype developed during this research project. This name is
inspired by metamodels (the ‘meta’ part) and the common Haskell source file extension (‘hs’).
\(^3\) An embedded domain specific language, as defined by Andy Gill, is “a library in a host language that has the
look, feel, and semantics of its own language, customized to a specific problem domain” [Gil14].
\(^4\) The extensions layer contains the source code metric and visualization algorithms.
This chapter discusses the related work for this research project.

Section 2.1 introduces metamodels and Section 2.2 provides descriptions of existing source code analysis tools based on meta-metamodels.

An overview of the prior work on software metrics for Haskell is given in Section 2.3. This section shows that the current research on software quality for functional programming languages – especially Haskell – is very limited compared to the research on software quality for object-oriented programming languages.

Section 2.4 explains why the creation of a metamodel for Haskell could be beneficiary to the research on software metrics for Haskell and details the main requirements for such a metamodel.

A description of the ‘Lack of Cohesion of Module Elements’ (LCOM) source code metric is provided in Section 2.5. This metric is used to demonstrate the capabilities of the metamodel and is therefore key to the metamodel prototype.
2.1. Metamodels

Using models to represent certain aspects of a software system is a common practice within the field of software engineering. In general, these models make it easier to comprehend the aspects that these models cover than by diving into the source code. For instance, understanding the complex relations between a large number of classes in an object-oriented program is considerably easier when using an UML class diagram derived from the source code than by uncovering these relations from the source code.

For a model to be usable, it is necessary to have a precise definition for all the elements used in the model. This set of definitions is called the metamodel and defines the class of models that can be created in accordance with these definitions. This is analogous to the way that a legend for a roadmap defines the class of roadmaps that can be created and that conform to this legend.

Well-defined metamodels can also improve reusability by allowing the decoupling of producers and consumers of the information contained within a conforming model [Bas+15].

It is also possible to create a meta-metamodel that defines the various properties for a set of metamodels. In the context of this research, this meta-metamodel would define the properties of a set of programming language specific metamodels and therefore this meta-metamodel is programming language independent. In other words; an environment built around a metamodel will be specific for a single programming language, while one that is built around a meta-metamodel could support multiple programming languages.

Four-layer Metamodel Hierarchy

The four-layer metamodel hierarchy [OMG11] defines a useful structure to label the various model types described above.

In the context of this research:

- The M0 layer contains the actual source code.
- The M1 layer contains the models.
  A model represents the source code (e.g. through an abstract syntax tree) and conforms to an M2 metamodel.
- The M2 layer contains the metamodels.
  Such a metamodel defines the properties for a specific programming language. It will conform to the M3 layer meta-metamodel if such a model is available.
- The M3 layer contains the meta-metamodel.
  This meta-metamodel can allow programming language independence if available in the environment.

A source code metrics tool can therefore be based on an M2 layer metamodel if it only targets a single programming language, or it can be based on an M3 meta-metamodel if it targets multiple programming languages.
2.2. Existing Source Code Analysis Tools Based on a Meta-Metamodel

This section provides a systematic description of the Moose and Rascal source code analysis tools that are based on a meta-metamodel. The environment, main requirements, and architecture are described for both tools.

2.2.1. Moose

The Moose reengineering environment\(^1\) is a language-independent environment for reverse- and reengineering legacy software.

Environment

Moose is implemented in the Smalltalk object-oriented programming language and currently runs on the Pharo virtual machine. Pharo provides Moose with an execution environment, a development environment, and a graphical user interface.

Requirements

The main requirements for a software reengineering environment as defined by the designers of the Moose reengineering environment are [DLT01]:

- **Support for reengineering tasks:** the reengineering environment should be well suited for various reengineering tasks. Examples of such tasks are calculating source code metrics, creating software visualizations, and refactoring.

- **Extensible:** an environment for reverse engineering and reengineering should be extensible in many aspects. For instance, it should be possible to gather information from multiple sources, to perform a variety of analyses on this information, and to export this information to multiple external tools (e.g. visualization).

- **Exploratory:** the exploratory nature of reverse engineering and reengineering demands that a reengineering environment does not impose rigid sequences of activities.

- **Scalable:** as legacy systems tend to be huge, an environment should be scalable and provide sufficient performance to prevent long latency times.

- **Information exchange and tool integration:** as a reengineering effort typically requires a combination of specialized tools, it is necessary that these tools can exchange information and can be integrated into a workflow.

Architecture

The high-level architecture of Moose is depicted in Figure 2.1.

Moose maintains a repository of software models and provides services for querying and manipulating these models. External tools can use these models and services to extract the information they require.

\(^1\)http://www.moosetechnology.org
Source code parsing for the Smalltalk programming language is provided directly by the virtual machine. Other programming languages require external parsers that interface with the Moose environment via the MSE interface.

![Diagram of Moose architecture](image)

**Figure 2.1**: The architecture of Moose (based on [NDG05]).

### 2.2.2. Rascal

The Rascal\(^2\) metaprogramming language aims at providing a concise and effective language for performing meta-programming tasks such as the analysis and transformation of existing source code and models, and the implementation of domain-specific languages.

The slogan for this project is to be “the one-stop shop for metaprogramming” [KLV11].

**Environment**

The Rascal environment is implemented as a set of plugins for the Eclipse\(^3\) integrated development environment.

**Requirements**

Klint et al. [KVV09] list the following main requirements for Rascal:

\(^2\)[http://www.rascal-mpl.org]
\(^3\)[https://eclipse.org]
2.2. Existing Source Code Analysis Tools Based on a Meta-Metamodel

- **Expressiveness**: Source code analysis requires a metaprogramming language that is well suited for this domain and that provides suitable primitives (e.g. pattern matching and traversals).

- **Safety**: Source code analysis and transformation is a complex domain where solutions are error-prone. Many applications are both deep (conceptually hard) and wide (many details to consider). A modular language that facilitates encapsulation and reuse helps to deal with such complexity. A static type system that offers safety features such as immutability and well-formedness will also help managing this complexity.

- **Usability**: Usability includes learnability, readability, debugability, traceability, deployability and extensibility. The Rascal language should be extensible by advanced users, have an open design to enable easy interoperability and integration with third-party components, and it should have good encapsulation mechanisms.

- **Performance**: The results of source code analysis are often huge and the Rascal implementation should be fast and lean enough to support such applications.

**Architecture**

Rascal is based on a meta-metamodel called M^3. A high-level architectural overview of Rascal can be seen in Figure 2.2.

Source code is parsed by a suitable tool (e.g. an external and language-specific compiler) to an abstract syntax tree (AST) that is in a format defined by this external tool. A translator can fetch information from this AST and generate one M^3 AST and one M^3 relations data structure per source code file. The extractor might be able to extract more information from an M^3 AST than the translator is capable of. The linker combines the information contained in the various, file-related, M^3 relations data structures to generate a merged M^3 relations data structure [Bas+15].

![Figure 2.2: An overview: from code to metrics and visualizations via M^3 models [Bas+15].](image)

The M^3 relations data structure is implemented as an associative array. A number of the keys are predefined by the M3 layer meta-metamodel, while specific M2 layer metamodels can extend the set of keys (e.g. for language specific constructs). The values in this associative array are all binary relations. Location types are used to link to physical and
logical code locations in the source code. For example, in a Java specific model, the binary containment relation `<|class:///foo/bar|,|pkg:///foo|>` would indicate that the class “bar” is contained in the “foo” package [Bas+15].
2.3. Prior work on software metrics for Haskell

Software metrics are used to extract information from code for further analysis (e.g. program comprehension or assessing software quality). A well-known example is the Cyclomatic Complexity metric [McC76] that measures the number of paths through a program's source code.

Most of the research on software metrics has been done for object-oriented and imperative programming languages [Ber95; RT05]. The research activities on software metrics for functional programming languages, and Haskell in particular, have been very limited and are summarized in the following paragraphs.

Klaas van den Berg (1995)
In his PhD thesis [Ber95], Van den Berg describes efforts to perform static analysis on functional programs. A software metric analyzer was created for the Miranda programming language. This analyzer is based on a control-flowgraph model and a callgraph model. Graphical representations for these models are also defined in his thesis.

Chris Ryder and Simon Thompson (2005)
Ryder and Thompson [RT05] conducted research on the usefulness of software metrics for the Haskell programming language. The authors have identified a collection of software metrics for use with Haskell programs and categorized these into the following categories:

- **Patterns**: metrics related to pattern matching.
- **Distance**: metrics related to the distance between the definition of an element and its use.
- **Callgraph Attributes**: metrics related to function calls.
- **Function Attributes**: more general metrics related to the implementation of functions.

These metrics were calculated on two Haskell programs and this has shown that it may be possible to use some of these metrics to indicate functions that may have increased risk of containing errors, and which may therefore benefit from more rigorous testing. In addition, it was determined that there is no single attribute that makes a Haskell program complex, but rather that this is caused by a combination of features.

Harm van den Hoven (2015)
Van den Hoven [Hov15] focussed his research on three maintainability properties of Haskell programs:

- **Coupling**: the level of dependency between modules.
  This property is covered by two software metrics:
  - **Coupling Between Modules (CBM)**: the number of modules that a module depends on.
  - **Coupling Factor (CF)**: the ratio between the actual and maximum number of imported modules for a program.
• **Cohesion**: the degree in which code in a module performs one specific task. This property is covered by two software metrics:
  
  – **Lack of Cohesion of Module Elements (LCOM)**: the number of independent groups of functionality within a module.
  
  – **Cohesion Factor (CHF)**: a ratio determined by the aggregation of the various LCOM values for a program.

• **Information hiding**: concerns hiding the internal working of a module. This property is covered by the following software metric:
  
  – **Hiding Factor (HF)**: the ratio between the actual and maximum entities exposed within a module.

A source code analysis tool was created that calculates these five software metrics.

These metrics have been calculated for 902 Haskell programs to learn the distribution of the resulting scores. Also, the revision histories of seven Haskell applications have been analyzed in depth to conclude that Haskell modules with high coupling, low cohesion and little information hiding are sensitive to change.

His research project can be seen as a continuation of the research done by Ryder and Thompson [RT05]. Both research projects come to the conclusion that software metrics are useful and that multiple software metrics need to be combined to assess the complexity of a Haskell program.
Section 2.2 highlights how the Moose and Rascal metamodels have successfully created common platforms for the calculation and visualization of software metrics for object-oriented and imperative programming languages.

Section 2.3 shows that the current research on software quality for functional programming languages – especially Haskell – is very limited compared to the research on software quality for object-oriented programming languages.

Research projects on software quality for the Haskell programming language could clearly benefit from a metamodel as it provides a higher level of abstraction for the implementation of new software metrics. The metamodel also provides a common platform that should ease sharing and collaboration among different research projects. The metamodel would be a powerful tool for software engineers to assess and improve the quality of their Haskell projects. Therefore, the design of a prototype of a metamodel for the Haskell programming language is chosen as the topic of this research project. This prototype is named meta$^\text{HS}$ and is presented in this section.

**Environment**

From the perspective of the user, as described in Section 2.2, it is essential that a metamodel can perform the desired source code analysis tasks and that it can easily be integrated into the user’s workflow. The Moose and Rascal metamodels are both at a disadvantage in this respect: neither of them target the functional programming paradigm, they both require alternative runtime environments (Smalltalk and Java), and use alternative metaprogramming languages (Smalltalk and Rascal). For these reasons, a metamodel for the Haskell programming language – that targets Haskell developers – is best implemented in the Haskell programming language itself.

Using Haskell as the metaprogramming language should assure that all important language aspects can be covered while implementing software metrics. In addition, Haskell also provides facilities for metaprogramming (e.g., ‘Template Haskell’ [SJ02] and ‘Scrap Your Boilerplate’ [LJ03]). Using Haskell should also ease the integration into the user’s workflow as there is no need for another runtime environment or programming language.

**Requirements**

The main requirements for the meta$^\text{HS}$ metamodel are similar to the main requirements for both Moose and Rascal:

- **Scalability**: The metamodel should be equally usable for small as well as large Haskell source code bases.
- **Extensibility**: The metamodel should be extendible to allow the addition of new functionality.
- **Safety**: The metamodel should strive to provide a safe environment where errors are minimized and where correct results are produced.
- **Expressiveness**: The metamodel should be well suited for source code analysis of Haskell source code.
• **Usability**: The metamodel should not impose restrictions on user workflows.

This research project only covers the following less severe requirements:

• **Scalability**: The meta\textsuperscript{HS} metamodel should be able to extract the relevant information for the seven Haskell applications that were analyzed in depth by Van den Hoven [Hov15]. These applications are: ampersand-3.0.2\textsuperscript{4}, helium-1.8\textsuperscript{5}, ideas-1.2\textsuperscript{6}, pandoc-1.13.1\textsuperscript{7}, parsec-3.1.7\textsuperscript{8}, QuickCheck-2.7.6\textsuperscript{9}, and uulib-0.9.16\textsuperscript{10}.

• **Extensibility**: The meta\textsuperscript{HS} metamodel should be extensible with the data structures and algorithms required for the implementation of the LCOM source code metric and visualization.

• **Safety**: The meta\textsuperscript{HS} metamodel should only provide data that is properly typed to consumers of this data in order to prevent type errors.

• **Expressiveness**: The implementation of the LCOM source code metric and visualization algorithms should be straightforward as the meta\textsuperscript{HS} metamodel should provide a higher level of abstraction.

• **Usability**: It is intended that the Glasgow Haskell Compiler (GHC) is the only requirement on the environment in order to use the meta\textsuperscript{HS} metamodel.

The LCOM source code metric (see Section 2.5) is chosen to demonstrate the capabilities of the metamodel as it does not rely on information from multiple modules and has a graphical representation that can be visualized.

\textsuperscript{4} https://hackage.haskell.org/package/ampersand-3.0.2
\textsuperscript{5} https://hackage.haskell.org/package/helium-1.8
\textsuperscript{6} https://hackage.haskell.org/package/ideas-1.2
\textsuperscript{7} https://hackage.haskell.org/package/pandoc-1.13.1
\textsuperscript{8} https://hackage.haskell.org/package/parsec-3.1.7
\textsuperscript{9} https://hackage.haskell.org/package/QuickCheck-2.7.6
\textsuperscript{10} https://hackage.haskell.org/package/uulib-0.9.16
The Lack of Cohesion of Module Elements (LCOM) source code metric is described in more detail in this section. This source code metric is used to demonstrate the capabilities of the metamodel and is therefore key to the metamodel prototype.

Hitz and Montazeri [HM95] defined the LCOM4 source code metric for object-oriented programming languages. The LCOM4 metric depends on the creation of a graph for each class where the set of vertices correspond to the set of methods of this class. An edge between two vertices indicates that these two methods either access one or more shared variables or that one of these method calls the other method. The LCOM4 value for a class is then defined as the number of connected components within this graph.

The range for this LCOM4 value is a non-negative integer number between zero and the number of methods for this class. An LCOM4 value of zero indicates that this module does not contain any functions and therefore is a potential candidate for removal. An LCOM4 value of one indicates that all methods of this class are connected. An LCOM4 value of two or higher might indicate that this class is concerned with multiple tasks and should probably be refactored.

The Lack of Cohesion of Module Elements (LCOM) source code metric, as defined by Van den Hoven [Hov15], is a translation of the LCOM4 metric to the Haskell programming language.

A simple example is shown in Figure 2.3. The source code is shown on the left and the corresponding call graph is shown on the right of this figure. The graph indicates that there are two connected components in this module and therefore the corresponding LCOM value is 2.

```
magicNumber :: Integer
    magicNumber = 42

isMagic :: Integer -> Bool
    isMagic = (== magicNumber)

loud :: String -> String
    loud = (+ "!")

louder :: String -> String
    louder = loud . loud

evenLouder :: String -> String
    evenLouder = louder . louder

eextremelyLoud :: String -> String
    extremelyLoud = evenLouder . evenLouder
```

Figure 2.3: Lack of Cohesion of Module Elements (LCOM) example.
This project performs exploratory research to determine whether the development of a metamodel for the Haskell programming language is useful and to discover the desirable features for such a metamodel and the metaprogramming environment that would be based on this metamodel. For this purpose, the metahs prototype is developed.

The main research question is:
‘What are the design considerations for a scalable and extensible metamodel for assessing the software quality of Haskell programs?’

This main research question can be decomposed into the following sub research questions:

- **RQ1**: ‘What software architecture is suitable for the metahs metamodel?’
- **RQ2**: ‘How can the metahs metamodel be extended?’
- **RQ3**: ‘What features should the metahs EDSL have for concise implementations of source code metrics and visualizations?’

The remainder of this section discusses these sub research questions in more detail.

**RQ1**: ‘What software architecture is suitable for the metahs metamodel?’

The metahs metamodel requires a scalable and extensible software architecture to meet its requirements. Given that a metamodel is heavily concerned with the flow of information through the metamodel an information flow diagram is an appropriate depiction of the high-level software architecture for the metahs metamodel.

The architecture of the metahs prototype is inspired by the architectural design of Rascal (see subsection 2.2.2). Experiences gained from the implementation of the metahs prototype should give sufficient insights into whether this software architecture is suitable for the metahs metamodel.

A prototype is developed as a part of this research project that will be a proof-of-concept and is used to discover the main design considerations. This prototype is limited to what is required to implement the LCOM source code metric and visualization (see Section 2.4).
Nevertheless, this should provide enough insights into the topic for discovering the main design considerations, the applicability of a metamodel for the Haskell programming language, and the usability for the user.

**RQ2: ‘How can the meta\(^{HS}\) metamodel be extended?’**

The meta\(^{HS}\) metamodel contains an external parser and an extractor function to extract and store relevant information from the source code modules of the application. The requirements for the meta\(^{HS}\) metamodel imply that the metamodel generated by this combination should be scalable and extensible (see Section 2.4).

The scalability of the parser and extractor functions affect the maximum size and complexity of the Haskell applications that can be analyzed by the meta\(^{HS}\) metamodel. If these functions require more time or computer resources (e.g. processing power, memory, or storage space) to analyze a specific Haskell application than acceptable or available, they will fail to meet the scalability requirement for this specific situation.

The meta\(^{HS}\) metamodel should also support aggregator functions that can gather information from the metamodel and combine this into information on a higher and more abstract level. For example, an aggregator function can be defined that calculates the LCOM values (see Section 2.5) for all the modules in a program. Such an aggregator function would simplify the implementation of certain source code metric and visualization algorithms as these algorithms can use the higher level information provided by this aggregator function.

The main tasks associated with this research question are:

- **Parser:** It is important to understand how to interface with the external haskell-src-exts\(^1\) parser, to understand how the resulting Abstract Syntax Trees (AST) are structured, and to assure that this parser is sufficient for the needs of this research project.

- **Extractor:** The data structure generated by the Extractor function, and the Extractor algorithm itself, need to be designed. Given the nature of the extractor function, it seems logical to focus on optimizing the AST traversal and to use techniques for generic programming (e.g. ‘Scrap Your Boilerplate’[LJ03]).

- **Aggregator:** The LCOM aggregator function is designed and integrated into the existing components of the meta\(^{HS}\) metamodel.

- **Scalability:** The scalability of the meta\(^{HS}\) metamodel is checked to assure that the seven Haskell applications that were analyzed in depth by Van den Hoven [Hov15] can be analyzed by the meta\(^{HS}\) metamodel.

The proof-of-concept, and thereby the meta\(^{HS}\) data model proposal, is validated by:

- A manual inspection on a limited number of simple test scripts, to assure that the information contained in the meta\(^{HS}\) metamodel matches the information contained in the test scripts.

\(^1\) [https://hackage.haskell.org/package/haskell-src-exts](https://hackage.haskell.org/package/haskell-src-exts)
• The seven Haskell applications that were analyzed in depth by Van den Hoven [Hov15] are assumed to be an adequate representation of average Haskell applications. Therefore, the prototype must be able to generate a metamodel for these seven applications to meet the scalability requirement of the meta$^\text{HS}$ metamodel.

• The LCOM source code metric requires that the metamodel, at a minimum, contains the containment and usage relations. It is assumed that the addition of these relations to the (empty) metamodel give a sufficient example of the extensibility of the metamodel. A further extension of the metamodel will be provided by the addition of an LCOM aggregator function.

• The LCOM source code metric, which is created as a part of research question RQ3, will validate the meta$^\text{HS}$ metamodel on a higher level.

RQ3: ‘What features should the meta$^\text{HS}$ EDSL have for concise implementations of source code metrics and visualizations?’

The goal for the meta$^\text{HS}$ embedded domain-specific language (EDSL) is to make the implementation of source code metrics and visualizations as concise and straightforward as possible.

The primary reason for the inclusion of this EDSL is that it reduces the implementation effort for the user by providing a higher level interface [Hud98; Gil14]. A secondary consideration is that it creates an additional abstraction layer, which should be beneficial for the evolution of the metamodel itself in the future. The current assumption is that a shallow EDSL will suffice [Gil14].

A prototype of the meta$^\text{HS}$ EDSL is created as a proof-of-concept that augments the prototype made as a part of RQ2. This prototype focuses primarily on the features required for the ‘Lack of Cohesion of Module Elements’ (LCOM) source code metric (see Section 2.5) and the corresponding visualization for this metric. The LCOM source code metric and visualization algorithms are implemented to demonstrate that the meta$^\text{HS}$ EDSL satisfies the reduced usability, expressiveness, and safety requirements (see Section 2.4).

The algorithm for the LCOM source code metric could be summarized as “Determine the number of connected components of the top-level elements of a module”. Therefore, the basic operations required for this metric are: select a module, get the top-level elements of the chosen module, compute the connected components, and compute the number of connected components. The meta$^\text{HS}$ EDSL should make it possible to directly translate these four operations of the LCOM algorithm to actual functions of the EDSL.

The main tasks associated with this research question are:

• **meta$^\text{HS}$ EDSL prototype**: The EDSL prototype, focussing on the LCOM source code metric, is designed. The main design considerations need to be discovered and possible deviations from the meta$^\text{HS}$ EDSL proposal need to be documented.

• **LCOM source code metric and visualization**: The prototype for the LCOM source code metric and visualization is designed to validate the ability of the meta$^\text{HS}$ EDSL to allow for a concise implementation of this metric.
The validity of the proof-of-concept, and thereby the meta^{HS} EDSL proposal, is checked by:

- The conciseness of the code required for the LCOM source code metric and visualization prototypes give a clear indication of the success or failure of the meta^{HS} EDSL to allow for concise implementations of the LCOM metric.

- The LCOM source code metric is calculated for the seven Haskell applications that were analyzed in depth by Van den Hoven [Hov15] and the resulting values are compared to the values reported by Van den Hoven.

- The correctness of the LCOM visualization prototype is manually checked for a selected number of modules.
4

META$^H_5$ DATA MODEL LAYER

This chapter describes the design and implementation of the prototype for the meta$^H_5$ data model layer as depicted in Figure 4.1. The legend for this diagram is shown in Figure C.1.

The meta$^H_5$ data model layer is responsible for extracting and maintaining the information relevant to the metamodel. It consists of a parser and an extractor function. The parser function parses Haskell source code and generates the corresponding abstract syntax tree (AST) on a per module basis. It is implemented using the haskell-src-exts$^1$ package. The extractor function generates a data structure (represented in the information flow diagram as a pentagon) for each abstract syntax tree that contains the information that is of interest to the meta$^H_5$ metamodel (e.g. function calls).

Section 4.1 introduces a simple example that will be used throughout this document to help explain new concepts when these are introduced. The parser is briefly touched upon in Section 4.2. Section 4.3 details the core concepts of the meta$^H_5$ metamodel. The various extractor functions implemented during this research project are described in Section 4.4. Section 4.5 concludes this chapter by discussing aggregators.

$^1$https://hackage.haskell.org/package/haskell-src-exts-1.19.1
4.1. **SOURCE CODE EXAMPLE**

This and subsequent chapters use the Haskell source code in Listing 4.1 as an example to help explain new concepts when these are introduced. The example code contains a variety of different elements that are relevant for the LCOM software metric while still remaining relatively small and easy to comprehend. It is by no means intended to be complete or useful in any other way.

The source code example consists of a single module named `Example` that contains two different sets of functionality. The first set is related to a very simplified abstraction of a car and consists of one type synonym (CarType), two datatypes (Color and Car), three field labels (carType, color, and age), and two functions (newCar and repaint). The second set consists of a pattern that defines a magic number (magicNumber) and a function that can check whether a provided integer number matches this magic number (isMagic).

---

**Listing 4.1: Example source code**

```haskell
module Example
    where

type CarType = (String, String)  -- brandName, typeName

data Color = Red | Green | Blue deriving (Show)

data Car = Car
    { carType :: CarType
    , color :: Color
    , age :: Int
    } deriving (Show)

newCar :: String -> String -> Color -> Car
newCar brandName typeName newColor = Car
    { carType = (brandName, typeName)
    , color = newColor
    , age = 0
    }

repaint :: Car -> Color -> Car
repaint car newColor = car { color = newColor }

magicNumber :: Int
magicNumber = 42

isMagic :: Int -> Bool
isMagic = (== magicNumber)
```

4.2. **Parser**

The meta\textsuperscript{HS} prototype uses version 1.19.1 of the haskell-src-exts\textsuperscript{2} parser to parse Haskell source code modules to the corresponding abstract syntax tree (AST) for further analysis.

Roughly the first ten percent of the abstract syntax tree generated by the parser for the \texttt{Example} module is shown in Listing 4.2 in order to give an impression of the very precise and overwhelming amount of information contained in such an AST.

```
Listing 4.2: Partial AST for the Example module

ParseOk (Module (SrcSpanInfo {srcInfoSpan = SrcSpan "projects/input/example/Example.hs" 1 1 29 1, srcInfoPoints = [SrcSpan "projects/input/example/Example.hs" 1 1 1,SrcSpan "projects/input/example/Example.hs" 1 1 1,SrcSpan "projects/input/example/Example.hs" 4 1 4 1,SrcSpan "projects/input/example/Example.hs" 6 1 6 1,SrcSpan "projects/input/example/Example.hs" 8 1 8 1,SrcSpan "projects/input/example/Example.hs" 14 1 14 1,SrcSpan "projects/input/example/Example.hs" 15 1 15 1,SrcSpan "projects/input/example/Example.hs" 21 1 21 1,SrcSpan "projects/input/example/Example.hs" 22 1 22 1,SrcSpan "projects/input/example/Example.hs" 24 1 24 1,SrcSpan "projects/input/example/Example.hs" 25 1 25 1,SrcSpan "projects/input/example/Example.hs" 27 1 27 1,SrcSpan "projects/input/example/Example.hs" 28 1 28 1,SrcSpan "projects/input/example/Example.hs" 29 1 29 1,SrcSpan "projects/input/example/Example.hs" 29 1 29 1]) (Just (ModuleHead (SrcSpanInfo {srcInfoSpan = SrcSpan "projects/input/example/Example.hs" 1 1 2 10, srcInfoPoints = [SrcSpan "projects/input/example/Example.hs" 1 1 1 7,SrcSpan "projects/input/example/Example.hs" 2 5 2 10]) (ModuleName (SrcSpanInfo (srcInfoSpan = SrcSpan "projects/input/example/Example.hs" 1 8 1 15, srcInfoPoints = [])) "Example") Nothing Nothing)) []) [] [TypeDecl (SrcSpanInfo {srcInfoSpan = SrcSpan "projects/input/example/Example.hs" 4 1 4 32, srcInfoPoints = [SrcSpan "projects/input/example/Example.hs" 4 1 4 5,SrcSpan "projects/input/example/Example.hs" 4 14 4 15]) (DHead (SrcSpanInfo (srcInfoSpan = SrcSpan "projects/input/example/Example.hs" 4 6 4 13, srcInfoPoints = [])) (Ident (SrcSpanInfo {srcInfoSpan = SrcSpan "projects/input/example/Example.hs" 4 6 4 13, srcInfoPoints = []}) "CarType")) (TyTuple (SrcSpanInfo {srcInfoSpan = SrcSpan "projects/input/example/Example.hs" 4 16 4 17, srcInfoPoints = [SrcSpan "projects/input/example/Example.hs" 4 16 4 17,SrcSpan "projects/input/example/Example.hs" 4 23 4 24,SrcSpan "projects/input/example/Example.hs" 4 31 4 32]) Boxed [TyCon (SrcSpanInfo {srcInfoSpan = SrcSpan "projects/input/example/Example.hs" 4 17 4 23, srcInfoPoints = []}) (UnQual (SrcSpanInfo {srcInfoSpan = SrcSpan "projects/input/example/Example.hs" 4 17 4 23, srcInfoPoints = []}) (Ident (SrcSpanInfo {srcInfoSpan = SrcSpan "projects/input/example/Example.hs" 4 17 4 23, srcInfoPoints = []}) "Example"))))]
```

\textsuperscript{2}https://hackage.haskell.org/package/haskell-src-exts
4.3. **METAMODEL**

This section describes the design and implementation of the meta\(^{HS}\) metamodel. It subsequently describes the fundamental units of information, how these can be combined into relations, and finally the `MetaModel` type that forms the basis of all other functionality present in the meta\(^{HS}\) environment.

4.3.1. **ELEMENTS**

`Element` is the fundamental unit of information in the meta\(^{HS}\) metamodel. These elements can currently be used to represent programs, modules, functions, and source locations in the analyzed source code.

Two methods for implementing an element have been investigated: URI and Datatype.

**URI based implementation**

The first implementation is based on using a Uniform Resource Identifier (URI) [BRM05] as the basic element akin to the ‘Location’ types used in Rascal’s M\(^{3}\) meta-metamodel (see subsection 2.2.2).

This implementation requires the definition of several URI schemes to define the concrete syntax of these URI. A number of examples are shown in Listing 4.3. The excerpt in Listing 4.4 shows the implementation of the `srcSpanToLoc` function that encodes the source location information from the parser’s AST to the corresponding URI object.

This method for implementing an element has two significant drawbacks. The syntax components available in an URI do not directly match the information necessary for the implementation of the metamodel and all the information in an URI must be encoded as strings. This therefore necessitates the packing of multiple information units into a single URI syntax component (the pattern `qs` in Listing 4.4 is an example of such an operation) and it also implies that some of the URI syntax components remain unused as these have no useful role in the metamodel (for example, neither the ‘userinfo’ nor the ‘port’ elements of an URI have obvious counterparts in the metamodel). The continuous conversion between Haskell values and their corresponding string representations is a burden on the execution speed of the metamodel that is best avoided.

```
Listing 4.3: URI based Elements examples

datatype://Example.Car
file://projects/input/example/src/Example.hs?startLine=8\&startColumn=1\&endLine=12\&endColumn=22
```

```
Listing 4.4: URI based elements excerpt for source location information

type Loc = URI
locStartLine = "startline"
locStartColumn = "startcolumn"
locEndLine = "endline"
```
locEndColumn = "endcolumn"

srcSpanToLoc :: SrcSpan -> Loc
srcSpanToLoc srcSpan =
  URI { uriScheme = Just "file",
      uriUserInfo = Nothing,
      uriRegName = Just "",
      uriPort = Nothing,
      uriPath = srcSpanFilename srcSpan,
      uriQuery = Just $ pairsToQuery qs,
      uriFragment = Nothing }
where
  qs = [(locStartLine, show (srcSpanStartLine srcSpan)),
      (locStartColumn, show (srcSpanStartColumn srcSpan)),
      (locEndLine, show (srcSpanEndLine srcSpan)),
      (locEndColumn, show (srcSpanEndColumn srcSpan))]

**DATATYPE BASED IMPLEMENTATION**

The second implementation is based on a single datatype named `Element` as shown in Listing 4.5.

The `Element` datatype is native to the Haskell environment, requires no packing into ill-matched containers, and does not require constant conversions to and from string. In addition, this datatype is easy to extend, easy to use in pattern matching, and utilizes Haskell’s strong type system.

The meaning of the value constructors contained in this datatype are:

- **Program**: Represents the program that has been analyzed. The `name` field contains an arbitrary string that should resemble the program under analysis (e.g., "Example"). A future extension could be to allow multiple `Program` objects to exist in a single metamodel to allow for situations in which multiple programs are created from a single code base.

- **Module**: Represents a single module. The `name` field contains the module name (`moddid`) of the module (e.g., "Example").

- **Function**: Represents a pattern, function or value constructor. The `name` field contains the qualified name (e.g., "Example.repaint"). A question mark prefix will be used in those cases where the name cannot be fully qualified (e.g., "?.(<+>)"). Value constructors will have a `name` field that starts with an uppercase letter (e.g., "Example.Red"), while patterns and functions will have a `name` field that is either a symbol or starts with a lowercase letter.

- **DataType**: Represents a datatype declaration. The `name` field contains the qualified name (e.g., "Example.Color").
4.3. **Metamodel**

- **TypeSynonym**: Represents a type synonym. The name field contains the qualified name (e.g., "Example.CarType").

- **UnknownType**: Represents a type for which the exact type cannot be determined. The name field will contain the name of the type with a question mark prefix (e.g., "?.Int").

- **Location**: Represents a source code location. The locationPath field is a string that contains a path to a file. The locationStartLine, locationStartColumn, locationEndLine, and locationEndColumn define a region within this file that is significant for this location (e.g., all the code related to a certain function).

- **StringValue** and **IntValue**: Represents generic String or Int values respectively. These value constructors can be used by aggregator functions (see Section 4.5).

The **MetaModel** module, in which the **Element** datatype is defined, also contains an instance of the **Pretty** type class that creates pretty-printed string representations for these **Elements**. These pretty-printed representations are easier to read by humans and more concise than the string representation generated by the **show** function. Listing 4.6 shows the difference for three elements.

This document uses the pretty-printed version wherever possible.

---

**Listing 4.5: Elements**

```
-- | The various elements that can be used in the metamodel.
data Element
  = Program
    { name :: !String
    -- ^ The name of the program.
    }
    -- ^ Top level program
  | Module
    { name :: !String
    -- ^ The qualified name of the module.
    }
    -- ^ Represents a module
  | Function
    { name :: !String
    -- ^ The qualified name of the function.
    }
    -- ^ Represents a function
  | DataType
    { name :: !String
    -- ^ The qualified name of the datatype declaration.
    }
    -- ^ Represents a data declaration.
  | TypeSynonym
    { name :: !String
    -- ^ The qualified name of the type synonym.
    }
```
Listing 4.6: Pretty-printing of Elements

-- non pretty-printed:
Module {name = "Example"}
DataType {name = "Example.Car"}
Location {locationPath = "projects/input/example/src/Example.hs",
  locationStartLine = 8, locationStartColumn = 1,
  locationEndLine = 12, locationEndColumn = 22}

-- pretty-printed:
Module "Example"
DataType "Example.Car"
Location "projects/input/example/src/Example.hs" 8 1 12 22
4.3.2. Relations

Relations are used to express that two elements are related in some specific way. For example, when a function declaration $f$ is defined inside a module $m$ it can be stated that module $m$ contains function $f$. Similarly, if function $g$ is used by function $f$ it can be stated that function $f$ uses function $g$.

The $\text{meta}^{HS}$ metamodel uses the Relation type synonym to encode relations and this is defined as a Set of Element pairs (see Listing 4.7). The first Element in this pair is designated as the parent, while the second Element is designated as the child. The nature of the relation between the parent and the child is defined on the metamodel level (see subsection 4.3.3). For example, the pair (Function "Example.isMagic", Function $\rightarrow$ "Example.magicNumber") states that the "Example.isMagic" function uses the "Example.magicNumber" function if this pair is contained in the usage relation set.

Listing 4.7: Relations

```
-- | The relation types supported by the MetaModel.
-- | A relation is defined as a set of (Element,Element) pairs.
type Relation = Set (Element,Element)
```

Nonsensical relations

The current implementation of the Relation type allows for the definition of nonsensical relations. For example, (Function "Example.age", Program "example") is nonsensical if this pair is present in the containment relation set.

The approach taken by the $\text{meta}^{HS}$ prototype is that the components used in the generation of the metamodel are implemented in such a way that these nonsensical relations cannot be included into the generated metamodel. However, when the ability to read a metamodel from a file is going to be actively used it will become necessary to implement a validation function that can validate this metamodel. The combination of these two measures should give a high certainty the the metamodel is free of nonsensical relations.

Another approach could be to redesign the Relation datatype in such a way that it becomes impossible to define a nonsensical relation. This option has been considered, but it was decided not to investigate this approach. There are also some concerns on how this additional complexity will affect the higher level components of the $\text{meta}^{HS}$ environment and whether the potential benefits are worth this additional complexity.

It should be noted that even a sensical relation can still be incorrect and therefore ‘free from nonsensical relations’ does not mean ‘free from mistakes’.
4.3.3. **MetaModel**

The `MetaModel` type contains the various `Relation` items that make up the metamodel and is implemented as a newtype wrapper around a `Map` between a string – that indicates the specific relation – and a `Set` of `Element` pairs that define this relation (see Listing 4.8).

```
-- | A metamodel is implemented as a mapping between a String and
-- Relation. The key string will denote the type of relation between
-- the pairs in the value relation.
type MetaModelImpl = Map.Map String Relation

-- | The MetaModel type.
newtype MetaModel = MetaModel { getMetaModelImpl :: MetaModelImpl }
deriving (Read,Show)
```

The complete `MetaModel` for the `Example` module can be found in Appendix A.
4.4. Extractors

This section describes the various extractor function that are defined for the meta$^\text{HS}$ prototype.

Several code listings in this section show parts of the generated metamodel for the Example program. The complete metamodel can be found in Appendix A.

**Top-level declarations**

A utility program – named DeclTest – is created to determine which of the 33 top-level declarations$^3$ defined by version 1.19.1 of the haskell-src-exts are actually in use by the programs used for the validation of the meta$^\text{HS}$ prototype. The results of this analysis are shown in Table 4.1.

Based on this information, the TypeDecl, TypeSig, PatBind, FunBind, DataDecl, and GDataDecl declarations are 'must have' requirements for the meta$^\text{HS}$ prototype. Information on module exports and imports are required to properly process the ClassDecl and InstDecl declarations and therefore these declarations are classified as 'should have' requirements. The remaining five declarations do not relate to the LCOM software metric and are therefore classified as 'could have' requirements.

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Description</th>
<th>ampersand-3.0.2</th>
<th>helium-1.8</th>
<th>ideas-1.2</th>
<th>ideas-1.6</th>
<th>parsec-3.1.7</th>
<th>QuickCheck-2.7.6</th>
<th>uulib-0.9.16</th>
</tr>
</thead>
<tbody>
<tr>
<td>TypeDecl</td>
<td>A type declaration</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>TypeSig</td>
<td>A type signature declaration</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>PatBind</td>
<td>A pattern binding</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>FunBind</td>
<td>A set of function binding clauses</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>DataDecl</td>
<td>A data OR newtype declaration</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>GDataDecl</td>
<td>A data OR newtype declaration, GADT style</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ClassDecl</td>
<td>A declaration of a type class</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>InstDecl</td>
<td>An declaration of a type class instance</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>InfixDecl</td>
<td>A declaration of operator fixity</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>InlineSig</td>
<td>An INLINE pragma</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>SpecSig</td>
<td>A SPECIALISE pragma</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DeprPragmaDecl</td>
<td>A DEPRECATED pragma</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpliceDecl</td>
<td>A Template Haskell splicing declaration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.4.1. **Program Contains**

The Program Contains extractor creates \((\text{Program } "p", \text{ Module } "m")\) Element pairs for each module that is successfully parsed. The output of this extractor for the `Example` program is shown in Listing 4.9.

Listing 4.9: Example MetaModel – part 1

\[(\text{Program } "example", \text{ Module } "Example")\]

4.4.2. **Module Contains**

The Module Contains extractor analyzes the top-level declarations of a module and generates relation items for each supported declaration according to the following logic:

- **TypeDecl**: \((\text{Module } "m", \text{ TypeSynonym } "ts")\)

- **DataDecl and GDataDecl**:
  - \((\text{Module } "m", \text{ DataType } "dt")\) for the datatype declaration.
  - \((\text{Module } "m", \text{ Function } "vc")\) for each value constructor.
  - \((\text{Module } "m", \text{ Function } "fl")\) for each field label.
  - \((\text{DataType } "dt", \text{ Function } "vc")\) for each value constructor.
  - \((\text{DataType } "dt", \text{ Function } "fl")\) for each field label.

- **PatBind and FunBind**: \((\text{Module } "m", \text{ Function } "f")\)

The \((\text{DataType } "dt", \text{ Function } "vc")\) and \((\text{DataType } "dt", \text{ Function } "fl")\) relation items are not necessary for the LCOM software metric and therefore have no purpose at this time. However, they are still retained as they might be useful for other software metrics in the future.

The output of this extractor for the `Example` program is shown in Listing 4.10.

Listing 4.10: Example MetaModel – part 2

\[
\begin{align*}
(\text{Module } "Example", \text{ Function } "Example.Blue") \\
(\text{Module } "Example", \text{ Function } "Example.Car") \\
(\text{Module } "Example", \text{ Function } "Example.Green") \\
(\text{Module } "Example", \text{ Function } "Example.Red") \\
(\text{Module } "Example", \text{ Function } "Example.age") \\
(\text{Module } "Example", \text{ Function } "Example.carType") \\
(\text{Module } "Example", \text{ Function } "Example.color") \\
(\text{Module } "Example", \text{ Function } "Example.isMagic") \\
(\text{Module } "Example", \text{ Function } "Example.magicNumber") \\
(\text{Module } "Example", \text{ Function } "Example.newCar") \\
(\text{Module } "Example", \text{ Function } "Example.repaint") \\
(\text{Module } "Example", \text{ DataType } "Example.Car") \\
(\text{Module } "Example", \text{ DataType } "Example.Color") \\
(\text{Module } "Example", \text{ TypeSynonym } "Example.CarType") \\
(\text{DataType } "Example.Car", \text{ Function } "Example.Car")
\end{align*}
\]
4.4. EXTRACTORS

(dataType "Example.Car", function "Example.age")
(DataType "Example.Car", Function "Example.carType")
(DataType "Example.Car", Function "Example.color")
(DataType "Example.Color", Function "Example.Blue")
(DataType "Example.Color", Function "Example.Green")
(DataType "Example.Color", Function "Example.Red")

4.4.3. Module Source

The Module Source extractor analyzes the top-level declarations of a module and generates an Element pair for each supported top-level declaration that identifies the file path to the corresponding source code file and the region within this file that is significant for this top-level declaration.

The output of this extractor for the Example program is shown in Listing 4.11.

Listing 4.11: Example MetaModel – part 3

(Module "Example", Location "projects/input/example/src/Example.hs"
                              1 1 29 1)
(Function "Example.isMagic", Location "projects/input/example/src/
                                     Example.hs" 28 1 28 27)
(Function "Example.magicNumber", Location "projects/input/example/
                                     src/Example.hs" 25 1 25 17)
(Function "Example.newCar", Location "projects/input/example/src/
                                     Example.hs" 15 1 19 6)
(Function "Example.repaint", Location "projects/input/example/src/
                                     Example.hs" 22 1 22 48)
(DataType "Example.Car", Location "projects/input/example/src/
                      Example.hs" 8 1 12 22)
(DataType "Example.Color", Location "projects/input/example/src/
                      Example.hs" 6 1 6 48)
(TypeSynonym "Example.CarType", Location "projects/input/example/src/
                      /Example.hs" 4 1 4 32)

4.4.4. Module Exports

The Module Exports extractor analyzes the export list of the module under analysis and generates an Element pair for each exported type or value (e.g. (Module "mn", Function "mn.split").)

The Haskell programming language allows for two abbreviations in the export list: the abbreviated form T(...) and the module export. The former exports all the value constructor and value accessor functions defined for type T. The latter exports all the top-level elements exported by a different module. Both will require the extractor to expand these abbreviated forms while taken into account the naming rules listed in section 5.2 of the Haskell 2010 Language Report [Mar+10]. The module export abbreviated form might also necessitate the implementation of a multi-pass extractor.
This extractor has not been implemented in the current meta\textsuperscript{HS} prototype as the information produced by this extractor is not required for calculation of the LCOM source code metric.

4.4.5. Module Imports

The Module Imports extractor analyzes the import declarations of the module under analysis and generates an Element pair for each imported type or value (e.g. (Module "MetaHS\textsuperscript{HS} \rightarrow .DataModel.Utils.Name", Function "Data.List.intercalate").)

The Haskell programming language allows for a programmer to create a flexible import declaration (see section 5.3 of the Haskell 2010 Language Report \cite{Mar+10}). For instance, it is possible to import a module while hiding certain entities and it is also possible to limit the expansion of the top-level environment by using a qualified import. This flexibility will likely complicate the correct implementation of this extractor.

This extractor has not been implemented in the current meta\textsuperscript{HS} prototype as the information produced by this extractor is not required for calculation of the LCOM source code metric.

4.4.6. Module Uses

The Module Uses extractor analyzes the supported top-level declarations of a module to determine what external types and values it uses.

Name Resolution

A Haskell programmer can ease the burden of having to write the fully qualified name of each used external identifier by adding the appropriate import declarations at the top of a module. Consequently, this implies that the Module Uses extractor needs to perform name resolution on the identifiers in the AST in order to retrieve their fully qualified names.

The NameResolutionMap type is defined as a mapping between the name of an incompletely qualified identifier and the corresponding Element. Two of these mappings are created for each module to analyze: one for types and the other for values. The name resolution maps should not be used directly for performing name resolution on an identifier, but only through the resolveType and resolveValue functions.

The resolveType function returns the corresponding fully qualified Element for an identifier if this is present in the NameResolutionMap for types. Otherwise, it returns an UnknownType in which the identifier is preceded with a question mark symbol (e.g. UnknownType "?.Int").

The resolveValue function is the counterpart of the resolveType function that works on values. It returns a Function in which the identifier is also preceded with a question mark symbol (e.g., Function "?.(==)").

The name resolution maps are currently created solely on the information provided by the Module Contains extractor (see subsection 4.4.2) and therefore they are only capable of
name resolution for identifiers contained in the module under analysis. The information of the Module Imports extractor (see subsection 4.4.5) should also be used – when this extractor becomes available – in order to also allow for name resolution of identifiers that are contained by other modules.

**Algorithm**

The Module Uses extractor uses generic helper functions to find declared types for each supported top-level declaration in the module to analyze. The types for a top-level declaration can be declared in another top-level type signature declaration that is applicable to this declaration or in any internal type declaration (such as type synonyms, type signatures, or expression type-signatures). A similar approach is used for value constructors and field labels in a datatype declaration. Every unique type found this way results in the generation of an **Element pair** where the second **Element** is the result of applying the `resolveType` function to the type found.

Patterns and functions can also contain references to external values and therefore require additional logic to find all these external value references. The core of this logic consists of finding all identifiers in the body of the pattern or the function and subsequently removing those that corresponded to either type constructors, local variables, local patterns, or local functions. Every unique external value found this way will results in the generation of an **Element pair** where the second **Element** is the result of applying the `resolveValue` function to the value found.

The output of this extractor for the **Example** program is shown in **Listing 4.12**.

```
Listing 4.12: Example MetaModel – part 4

(Function "Example.Blue", DataType "Example.Color")
(Function "Example.Car", DataType "Example.Car")
(Function "Example.Car", DataType "Example.Color")
(Function "Example.Car", TypeSynonym "Example.CarType")
(Function "Example.Car", UnknownType "?.Int")
(Function "Example.Green", DataType "Example.Color")
(Function "Example.Red", DataType "Example.Color")
(Function "Example.age", DataType "Example.Car")
(Function "Example.age", UnknownType "?.Int")
(Function "Example.carType", DataType "Example.Car")
(Function "Example.carType", TypeSynonym "Example.CarType")
(Function "Example.color", DataType "Example.Car")
(Function "Example.color", UnknownType "?.Int")
(Function "Example.isMagic", Function "?.==(==)")
(Function "Example.isMagic", Function "Example.magicNumber")
(Function "Example.isMagic", UnknownType "?.Bool")
(Function "Example.isMagic", UnknownType "?.Int")
(Function "Example.magicNumber", UnknownType "?.Int")
(Function "Example.newCar", Function "Example.Car")
(Function "Example.newCar", Function "Example.age")
(Function "Example.newCar", Function "Example.carType")
(Function "Example.newCar", Function "Example.color")
```
4.4.7. **Type classes and instances**

The current metaHS prototype does not support type classes and instances. In order to add support for type classes and instances, it is necessary to implement the Module Exports and Module Imports extractors and to decide on how the metamodel should support these concepts This section describes how this support can be added in future research projects.

**Type classes**

Type classes have at least six aspects that might be of interest: the type class has a name, it is defined in a specific module, it is located in a source code file, it has a number of pattern of function type declarations, it might have default pattern or function implementation, and it might be a subclass of another type class.

The implementation of the first three items is relatively simple as it only requires the introduction of a new `Class` or `TypeClass` value constructor to the `Element` `dataType` and some minor additions to the `Module Contains` and `Module Source` extractors.

The fourth and fifth aspects are more complicated. It could be decided that information on default pattern and function implementations are not of interest for the metamodel and in this case it would suffice to treat the pattern and function type declarations in a similar way as patterns and functions are already treated for modules and datatypes Alternatively, a new `Element` could be added to differentiate between the pattern or function type declarations and their provided default implementations.

The last item will necessitate the introduction of a new relation to encode the subclassing relationship. This also requires a decision on whether this relation should have the superclass as the left-hand `Element` and the subclass as the right-hand `Element` or vice versa. This would also probably require the introduction of a new extractor.

It will also be necessary to make the ModuleUses extractor aware of the context of type signatures once type classes are properly supported.

**Type instances**

Adding support for type instances will require the implementation of the Module Exports and Module Imports extractors as it is necessary to connect the type instance to the corresponding type class irrespective of whether these are defined in the same or in different modules. The main reason for this coupling is that type classes can define default implementations for patterns and functions and therefore a type instance alone will not provide enough information on the patterns and functions it truly adds to the top-level of the module.
A new *Instance* or *TypeInstance* value constructor will have to be added to the *Element* data*Type*. Given that the ‘name’ of a type instance consists of the type class it implements and the type it implements this type instance for, it becomes necessary to determine how this ‘name’ is handled by this new *Element*. Probably the most convenient solution is to pack these two items in parentheses (e.g. Instance "A.B.(Eq Int)") as the meta*HS* environment already supports this notation for symbols.

A new *Relation* will also have to be added to link the type instance with the type class that it implements in the metamodel. It might be possible to reuse the relation defined between subclass and superclass if this seems usable to adequately describe this relation as well.

Another decision point is on whether the metamodel should be aware of type instances that reimplement patterns or function that were given default implementations in their respective type class.
4.5. **AGGREGATORS**

An aggregator function can aggregate lower-level information from the metamodel into higher-level information and is defined as a function that takes a `MetaModel` and returns a `MetaModel`. As an example, the `lcomAggregator` analyzes the information generated by the Module Uses extractor (see subsection 4.4.6), calculates the LCOM value for each module in the project (see Section 6.1) and returns a `MetaModel` that contains a new relation that specifies the LCOM value for each module in the project as shown in Listing 4.13.

To differentiate between relations generated by extractor and aggregator functions, it is advised to use strings that start with an underscore character (e.g. "contains") to identify relations generated by extractor functions, while relations generated by aggregator functions should not start with an underscore character (e.g. "LCOM").

---

**Listing 4.13: Example MetaModel – LCOM relation**

```
"LCOM":
  (Module "Example", IntValue "2")
```
This chapter describes the design and implementation of the prototype for the meta$^{HS}$ embedded domain specific language (EDSL) layer as depicted in Figure 5.1. The legend for this diagram is shown in Figure C.1.

![Information flow diagram for the meta$^{HS}$ metamodel.](image)

An embedded domain-specific language (EDSL) is included as the primary interface to the meta$^{HS}$ metamodel for consumers (e.g., a metrics calculation algorithm) and to provide additional common services (e.g., graph generation).

The primary reason for the inclusion of this EDSL is that it reduces the implementation effort for the user by providing a higher level interface [Hud98; Gil14]. A secondary consideration is that it creates an additional abstraction layer, which is beneficial for the evolution of the metamodel itself.

Section 5.1 introduces the metamodel façade. The graph generation support offered is described in Section 5.2. A short reflection on the EDSL concludes this chapter in Section 5.3.
5.1. **Metamodel Façade**
The metamodel façade provides access to the metamodel while shielding the implementation details. This façade is the public interface to the metamodel and should not be bypassed by other functions in the meta\(^H\)\(^S\) EDSL layer or above.

5.1.1. **Generation**
The `generateMetaModel` function is responsible for generating the metamodel. This function attempts to parse all the files in a provided directory hierarchy. Errors reported by the parser will be stored in a text file for further manual analysis if so desired. The results of applying the extractors to the successfully parsed modules will be combined into the `MetaModel` type and returned to the caller.

Aggregator functions (see Section 4.5) can use the `getRelation` function to fetch relations from a `MetaModel` and the `setRelation` function to add new relations to a `MetaModel`.

5.1.2. **File Operations**
Persistent storage of the metamodel to a file is provided by the `writeMetaModel` function. Reading the metamodel from this file can be accomplished by using the `readMetaModel` function.

Additionally, the `writeMetaModelPretty` function can also be used to write a pretty-printed version of the metamodel to a file. This function uses the `pretty` function to create a pretty-printed version of the metamodel. This pretty-printed file is currently geared towards human readers – e.g., for debugging purposes – and therefore there is no equivalent read function available.

As noted in subsection 4.3.2, it is advisable to augment the `readMetaModel` function with a `validate` function to prevent the presence of nonsensical relations in the metamodel.

5.1.3. **Queries**
A number of queries are provided to allow higher level functions to retrieve subsets of the information contained in the metamodel. These functions shield their users from the specific implementation of the metamodel and therefore allow the structure of the metamodel to change independently of the higher layers.

- **modules**: Returns a list of `Module Elements` found in the metamodel. This information can be used to retrieve further module oriented information from the metamodel.

- **numberOfItems**: Returns the number of `Relation` items in the metamodel. Currently, this is only used for debugging.

- **elementContains**: Returns a list of `Elements` that are contained by a specific `Element`. Two variants of this query are provided: `programContains` and `moduleContains`. These perform a check to ensure that the specified `Element` is actually a `Program` or `Module value constructor` respectively.
• **elementUses**: Returns a list of Elements used by a specific Element.

• **elementSource**: Returns the source location for a top-level declaration.

• **domain**: Returns the domain for a specified relation in the metamodel.

• **range**: Returns the range for a specified relation in the metamodel.

Additional operations on relations could be added in a future release of the meta\textsuperscript{HS} prototype.

### 5.1.4. Utility Functions

The **split** utility function is able to split the name field used in most Elements into the qualifier part – if present – and the identifier name. It also can support symbol identifiers.

The **isLocal** function uses the **split** function to determine whether a provided qualified identifier is local to a provided qualifier. This functionality can be used to discriminate between local and non-local references.

The **locationToQuery** function converts the output of the **elementSource** function to a string that can be used as the query component of a Uniform Resource Identifier (URI) \cite{BRM05}.
5.2. Graphs

One of the main requirements for the meta$^{HS}$ prototype is to calculate and visualize the LCOM software metric. As explained in Section 2.5, the LCOM software metric can be calculated by creating a graph of all the elements and usage relationships for a module and counting the number of connected components in this graph. This graph can also be used for the visualization of the LCOM software metric.

The Graphviz$^1$ visualization software [Gan09] is used to generate the LCOM software metric visualizations. As a consequence of this decision, the Data.Graph.Inductive$^2$ package is selected as the basis for the graph and the Data.GraphViz$^3$ package for interfacing with the Graphviz visualization software.

5.2.1. UsesGraph

The Graph module currently focuses on graphs that contain the internal usage relations between elements in a module as required for the calculation of the LCOM source code metric. Chapter 6 describes how these functions are used for the calculation and visualization of the LCOM software.

The internalUsesGraph function analyzes a metamodel to extract the internal usage relations for a specified module to create the corresponding graph.

The visualization of graphs by the Graphviz visualization software can be customized by providing a set of Graphviz parameters. The internalUsesParams function generates a default set of Graphviz parameters for the graph returned by the internalUsesGraph function. This default set of Graphviz parameters will visualize datatypes as green rectangles and type synonyms as blue rectangles. Patterns, functions, and value constructors will be rendered as white rectangles. The internal usage relations will be represented by unnamed arrow or lines depending on whether the internal uses graph is created as a directed or as an undirected graph. Finally, the name of the module will be displayed at the bottom of the image. The result of applying the default set of Graphviz parameters to the internal uses graph of the Example module can be seen in Figure 5.2.

The internalUses function is a convenience function that return the results of the internalUsesGraph and internalUsesParams functions as a pair.

---

1 http://www.graphviz.org
2 https://hackage.haskell.org/package/fgl-5.5.3.1/docs/Data-Graph-Inductive-Graph.html
3 https://hackage.haskell.org/package/graphviz
5.2.2. Utility Functions

Visualizing the graph is implemented by the `graphToImage` utility function that is defined in the `Utils` module. This function converts the graph and Graphviz parameters to an equivalent representation in the DOT\(^4\) graph description language and subsequently uses the Graphviz visualization software to create the image and to store it to a file. The Graphviz layout engine (e.g., `Dot` or `Circo`) and the image format to use for the generation (e.g., `Png` or `Svg`) can be also be specified.

\(^4\)http://www.graphviz.org/content/dot-language
5.3. EDSL

The LCOM source code metric defines the functionality currently exposed by the meta\textsuperscript{HS} EDSL layer. It is assumed that the meta\textsuperscript{HS} EDSL will grow as new source code metrics and visualizations are added in the future.

Nevertheless, the clear separation between the meta\textsuperscript{HS} data model layer and the higher layers provided by the current implementation is already useful. For instance, it allows for flexibility in the implementation of the metamodel and it simplifies the implementation of algorithms that use information from the metamodel. Having this separation in sooner rather than later seems to be a good choice.
This chapter describes the meta$^{HS}$ extensions layer as depicted in Figure 6.1. The legend for this diagram is shown in Figure C.1.

The meta$^{HS}$ extensions layer is dedicated to the various algorithms that calculate or visualize source code metrics.

Section 6.1 describes the LCOM source metric and visualization algorithms that are implemented in the current meta$^{HS}$ prototype. Section 6.2 describes the LCOM report generation features.
6.1. **LCOM ALGORITHM**

The LCOM module exports two functions that calculate the LCOM source code metric for a specified module in the metamodel. The `lcomGraph` function uses the `internalUsesGraph` function (see subsection 5.2.1) to generate the LCOM graph and default Graphviz parameters for a specified module. This information is augmented with the calculated LCOM value and returned to the caller. The `lcom` function is a shorthand version of the `lcomGraph` function that only returns the calculated LCOM value for a specified module.

Rendering the LCOM visualization can be accomplished by applying the `graphToImage` function to the LCOM graph and the default Graphviz parameters returned by the `lcomGraph` function (see subsection 5.2.2).
6.2. LCOM REPORT

A simple program that demonstrates the capabilities of the current meta\textsuperscript{HS} prototype is contained within the source code repository. Although it is not a part of the meta\textsuperscript{HS} extensions layer, it seems appropriate to describe this program here.

This demonstration program will generate the metamodel for a given project and store both the normal and pretty-printed versions of this metamodel to the file system. Next, it will generate the LCOM graph for each module in the metamodel using multiple Graphviz layout engines if desired. Finally, it will create an LCOM report in the form of a HTML table that can be viewed in a web browser.

The index page is shown in Figure 6.2. This page is created manually and its only purpose is to have a central starting point to navigate to the project specific LCOM page. Clicking on the ‘example’ hyperlink will forward the browser to the LCOM page for the Example project.

The source code modules for the Example project are shown in Figure 6.3. This table indicates that the LCOM value for this module is indeed two and a link is provided to the graph image generated by the Graphviz ‘Dot’ layout engine. Clicking on the ‘Dot’ hyperlink for the Example module will forward the browser to the LCOM graph as shown in Figure 6.4.

The LCOM graph for the Example module is shown in Figure 6.4. The default Graphviz parameters generated by the \texttt{lcomGraph} function will display datatypes as green boxes, type synonyms as blue boxes, and patterns and functions as white boxes. Elements for which the source code location is known in the metamodel (currently only top-level declarations) will also be turned into a clickable hyperlink to an HTML editor. Clicking on the ‘Car’ datatype hyperlink will forward the browser to the HTML editor as shown in Figure 6.5.

Figure 6.5 shows the HTML editor in response to a click on the Car datatype as described above. The relevant section of the source code will be highlighted and scrolled into view if necessary. The HTML editor requires that these pages are run on a web server or in a browser where the local file restrictions have been temporarily disabled.
Figure 6.2: The index page.

Figure 6.3: The LCOM report table for the Example project.
6.2. LCOM REPORT

Figure 6.4: The LCOM graph for the Example module.

Figure 6.5: The HTML editor opened on the Car datatype.
This chapter describes the method used to validate the correctness of the generated meta-models and LCOM information.
7.1. Manual validation

The procedure for manual validation consists of three steps. First, the source for a selected module is manually analyzed to determine which items should be present in the metamodel. Next, the generated metamodel is checked to assure that all these expected items are in the metamodel and that no unexpected items are present in the metamodel. Finally, the generated LCOM graphs are checked against this information.

Example

The Example module is used to explain the procedure used for the manual validation.

Two copies of the source code module are converted to the Rich Text Format (RTF). These copies are named after the module with '_contains' and '_uses' appended to indicate the specific analysis performed on these files. In this example, these file are named ‘Example_contains.rtf’ and ‘Example_uses.rtf’.

The various identifiers that are of interest for the Contains relation in the ‘Example_contains.rtf’ file are marked with a background color as shown in Figure 7.1. A blue color is used for type synonyms, a green color for datatypes, and a yellow color for patterns, functions, and value constructors. A similar operation is performed on the ‘Example_uses.rtf’ file as shown in Figure 7.2. Here a green background color is used to mark types that are of interest for the Uses relation and a yellow background is used for referenced patterns, functions or value constructors.

The metamodel is filtered such that only the relations related to the module under validation remain. Subsequently it is checked that the Contains and Uses relations in the metamodel conform to the marked identifiers in the ‘Example_contains.rtf’ and ‘Example_uses.rtf’ files.

Lastly, a copy of the generated LCOM image is made and each relevant Uses relation in the metamodel is annotated in this copy with a green arrow. The result should be an image where each edge is matched by an annotation arrow and vice versa as shown in Figure 7.3.
module Example

where

type CarType = (String, String)

data Color = Red | Green | Blue deriving (Show)

data Car = Car
  { carType :: CarType,
    color :: Color,
    age :: Int
  } deriving (Show)

newCar :: String -> String -> Color -> Car
newCar brandName typeName newColor = Car
  { carType = (brandName, typeName),
    color = newColor,
    age = 0
  }

repaint :: Car -> Color -> Car
repaint car newColor = car { color = newColor }

magicNumber :: Int
magicNumber = 42

isMagic :: Int -> Bool
isMagic = (== magicNumber)

Figure 7.1: The Contains annotations for the Example module.
7.1. Manual validation

Figure 7.2: The Uses annotations for the Example module.

Figure 7.3: The annotated LCOM image for the Example module.
MODIFICATIONS
The parser failed to parse a significant number of the source files in the analyzed projects:

• Most of these failures are caused by missing language pragmas, preprocessor statements or incorrect file encodings. These issues are resolved by adding the suggested language pragmas, manually resolving the preprocessor statements, or by resolving the file encodings.

• One file in the ampersand-3.0.2 project failed with a ‘Parse error: logicalDataModel-Section’ error. This problem is not resolved.

• Two files in the uulib-0.9.16 project failed with an ‘Illegal data/newtype declaration’ error. The problematic data structures are removed as a quick solution.

• The helium-1.8 project contains three Main.hs files that caused confusion for the metamodel. Two of these file are deleted.

• Four files (one for ideas-1.2, two for ideas-1.6, and one for pandoc-1.13.1) failed with an “ambiguous infix expression” error. This problem is not resolved.

These issues are not analyzed in further detail as the parser and the source code of the analyzed projects are not considered to be a part of this research project.

VALIDATED MODULES
The following fifteen modules have been manually validated:

• ampersand-3.0.2:

• helium-1.8:
  – Helium.StaticAnalysis.Directives.TS_Attributes
  – Helium.StaticAnalysis.Miscellaneous.TypeConstraints

• ideas-1.2:
  – Ideas.Encoding.Evaluator

• ideas-1.6:
  – Ideas.Encoding.Evaluator
  – Ideas.Utils.StringRef

• meta\text{HS}:
  – MetaHS.DataModel.MetaModel
  – MetaHS.DataModel.Extractor.Module.Contains
  – MetaHS.DataModel.Extractor.Module.Source
  – MetaHS.DataModel.Extractor.Module.Uses
• pandoc-1.13.1:
  – Text.Pandoc.MediaBag
  – Text.Pandoc.MIME

• parsec-3.1.7:
  – Text.Parsec.Perm

• QuickCheck-2.7.6:
  – Test.QuickCheck.Monadic

• uulib-0.9.16:
  – UU.Parsing.Interface

The following selection criteria were used to select the modules mentioned above:

• The LCOM graph should fit on a single screen.

• The LCOM graph should preferably contain several different special items (e.g., type synonyms, datatypes, symbols, and recursive calls).

• The LCOM graph should preferably have an interesting shape.

• The length of the source code should be within reasonable limits and preferably vary from module to module.

• At least one module must be chosen for each project.

• The most important source code modules from the meta\textsuperscript{HS} project must be present.
7.2. RESULTS COMPARISON

The LCOM values generated by the meta$^\text{HS}$ prototype have also been compared to the LCOM values calculated by Van den Hoven [Hov15].

It is necessary to remove modules that were analyzed by only one prototype. Also, it is necessary to filter out those module that contain type classes or type instances as these are not supported by the current meta$^\text{HS}$ prototype (see subsection 4.4.7) and therefore will result in mismatched LCOM values. Finally, modules that contain preprocessor statements are also removed as it is unclear what strategy is used by Van den Hoven to remove preprocessor statements as this can result in different LCOM values.

The number of modules, per project, that passed these filtering operations are shown in the 'Modules:' column of Table 7.1. The 'Matches:' column shows the number and percentage of modules where the LCOM value calculated by the meta$^\text{HS}$ prototype matches the LCOM values calculated by Van den Hoven [Hov15].

An experiment is performed to see if the removal of type synonyms and data constructors$^1$ in the LCOM calculation could explain the differences. The results of this experiment are shown in Table 7.2.

A number proceeded by a plus sign (e.g., '+2') in the 'Delta:' column indicates the number of modules for which the modified meta$^\text{HS}$ prototype produced a matching LCOM value while the unmodified meta$^\text{HS}$ prototype produced a non-matching value. Similarly, a number proceeded by a minus sign (e.g., ‘-1’) in this column indicates number of modules where the reverse is true. The ‘Union:’ column shows the number and percentage of modules where a matching value is found by the unmodified or the modified meta$^\text{HS}$ prototype.

This comparison provides a reasonable indication that the results generated by the current meta$^\text{HS}$ prototype are acceptable. Further analysis would be required to determine the exact cause of the differences in the LCOM values for some of the modules, but that is outside of the scope of this research assignment.

---

$^1$Value constructors and fields are not removed.
7.3. 

The main threats to validity for this research project are incompleteness, assumption and implementation mistakes.

The current meta\textsuperscript{HS} environment is only an early prototype and therefore incomplete. For example, as described in Section 4.4, a number of identified extractors are not yet implemented. As a result of this incompleteness, there is a possibility that weaknesses in the current metamodel, or the underlying assumptions, are discovered when new functionality is added to the meta\textsuperscript{HS} environment in the future.

The validation activities undertaken as part of this research project have been aimed at gaining a reasonable level of confidence of the validity of the current meta\textsuperscript{HS} prototype and are therefore also incomplete. This implies that it is not unlikely that implementation mistakes can be found during future validation activities on other modules.

Any implementation mistakes discovered during the development and validation of the current meta\textsuperscript{HS} prototype have been resolved. This does not imply that the code for the current meta\textsuperscript{HS} prototype is free from implementation mistakes. However, our current assumption is that the discovery of new implementation mistakes will probably not drastically change the findings of this research project.

<table>
<thead>
<tr>
<th>Modules:</th>
<th>Matches:</th>
<th>Delta:</th>
<th>Union:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ampersand-3.0.2</td>
<td>38</td>
<td>+2</td>
<td>38</td>
</tr>
<tr>
<td>helium-1.8</td>
<td>52</td>
<td>+5</td>
<td>50</td>
</tr>
<tr>
<td>ideas-1.2</td>
<td>55</td>
<td>+4</td>
<td>55</td>
</tr>
<tr>
<td>pandoc-1.13.1</td>
<td>37</td>
<td>-1</td>
<td>37</td>
</tr>
<tr>
<td>parsec-3.1.7</td>
<td>22</td>
<td>+6</td>
<td>22</td>
</tr>
<tr>
<td>QuickCheck-2.7.6</td>
<td>4</td>
<td>-2</td>
<td>4</td>
</tr>
<tr>
<td>uulib-0.9.16</td>
<td>13</td>
<td>+2</td>
<td>13</td>
</tr>
</tbody>
</table>
CONCLUSIONS

This thesis concludes by providing answers to the research questions, a short discussion on the usefulness of the meta\(^{HS}\) metamodel and suggestions for future research projects.

8.1. ANSWERS TO RESEARCH QUESTIONS

This section provides answers to the research questions raised in Chapter 3.

RQ1: ‘What software architecture is suitable for the meta\(^{HS}\) metamodel?’

The development of the meta\(^{HS}\) prototype showed that the information flow diagram shown in Figure 8.1 is suitable as the architecture for a Haskell metamodel for assessing software quality.

Validation of the meta\(^{HS}\) prototype demonstrated that the scalability of this architecture is sufficient to analyze all the Haskell projects specified in Section 2.4. The addition of various extractors (see Section 4.4) and the LCOM aggregator (see Section 4.5) prove that the meta\(^{HS}\) metamodel is easily extended.

The inclusion of the meta\(^{HS}\) embedded domain specific language (EDSL) seems to be a good method for providing a safe environment. All information provided by the EDSL is properly typed to prevent type errors. The prototype also shows that it is relatively easy to add checks to EDSL functions if desired (e.g., the \texttt{programContains} function described in subsection 5.1.3). The meta\(^{HS}\) EDSL also allows for concise and straightforward implementations of source code metrics and visualizations as demonstrated by the prototype for the LCOM source code metric (see Chapter 6).

The prototype should be usable in a wide variety of user workflows as the requirements on the environment are minimal. The current reliance on the Graphviz visualization software is a disadvantage, but this can be resolved in the future as discussed in Section 8.3.
RQ2: ‘How can the meta\textsuperscript{HS} metamodel be extended?’

Validation has shown that the meta\textsuperscript{HS} prototype is capable of analyzing all the Haskell programs that were defined for this research project. The time required to generate the metamodel depends on the complexity of the project to analyze and ranged from just a few seconds to almost nine minutes during testing (see Table 8.1). These numbers suggest that the time required to generate a metamodel does not prevent the regular use of the meta\textsuperscript{HS} prototype.

During this research project, the metamodel has been extended several times as new extractors and aggregators were implemented. These additions only required the definition of a new string identifying the relation added to the metamodel. Therefore, no practical issues have been found that imposed restrictions on the scalability and extensibility of the metamodel.

RQ3: ‘What features should the meta\textsuperscript{HS} EDSL have for concise implementations of source code metrics and visualizations?’

The meta\textsuperscript{HS} EDSL layer should act as a façade to the metamodel. This façade hides the implementation details of the metamodel from the users of this information in the higher layers. This also allows the implementation of the metamodel to change independently and simplifies the implementation of the users.

This layer should also provide basic functionality that is expected to be used by algorithms in the higher layer (e.g., the support for graphs as described in Section 5.2). This will simplify the implementation of the algorithms in the higher layers and could prevent duplication.

RQ: ‘What are the design considerations for a scalable and extensible metamodel for assessing the software quality of Haskell programs?’

It is essential that the data structure for the metamodel is designed with extensibility in mind as the number of extractors and aggregators will grow over time. Given the amount of information that will be added to the metamodel, it is also highly advisable to keep the basic structure of the metamodel as simple as possible.

The inclusion of a metamodel façade is highly advisable as it decouples the meta\textsuperscript{HS} data model layer from the higher level layers and allows for independent changes to the meta\textsuperscript{HS} data model and higher layers.

A future graphical user interface must be very easy to use, and highly interactive, as the information generated by the metamodel becomes complex very quickly. For example, although the Example module (see Section 4.1) is a very simple module, its corresponding LCOM graph (see Figure 6.4) is already starting to become complex.
8. Conclusions

Table 8.1: Throughput times of metamodel and LCOM report generation.

<table>
<thead>
<tr>
<th>Project</th>
<th>MetaModel:</th>
<th>LCOM:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ampersand-3.0.2</td>
<td>108</td>
<td>112</td>
</tr>
<tr>
<td>helium-1.8</td>
<td>520</td>
<td>790</td>
</tr>
<tr>
<td>ideas-1.2</td>
<td>41</td>
<td>22</td>
</tr>
<tr>
<td>ideas-1.6</td>
<td>42</td>
<td>24</td>
</tr>
<tr>
<td>parsec-1.13.1</td>
<td>126</td>
<td>228</td>
</tr>
<tr>
<td>QuickCheck-2.7.6</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>uulib-0.9.16</td>
<td>13</td>
<td>39</td>
</tr>
</tbody>
</table>

The numbers in this table correspond to the throughput times (rounded off to the nearest second) as reported by a run of the compiled ‘timedTest’. The time required to generate the metamodel is shown in the ‘MetaModel’ column and the time required to generate the LCOM report is shown in the ‘LCOM’ column. These tests were performed on a 2016 15-inch Apple MacBook Pro (2.6 GHz Intel Core i7 CPU, 16 GiB RAM, macOS Sierra 10.12.5).
8.2. DISCUSSION

The main goal of this research project is to perform exploratory research to determine whether the development of a metamodel for the Haskell programming language is useful (Chapter 3).

The abstract syntax tree (AST) generated by the haskell-src-exts parser contains an overwhelming amount of very precise information about a Haskell source code module. Understanding the structure of such an AST has therefore been a time consuming and error prone process. Implementing the algorithms to extract pieces of information from an AST has also proven to be quite a challenge as mistakes are easy to make and difficult to debug. In comparison, the metamodel generated by the meta$^{HS}$ prototype is much easier to comprehend and use due to its simpler structure and accompanying embedded domain specific language.

Using the information contained in the metamodel became even easier after the metamodel façade was added. Essentially, all that is required to retrieve information from the metamodel façade is a basic understanding of list comprehensions. This certainly does not hold for retrieving information from the AST.

Generating the first LCOM graph was really exciting as it immediately became clear how useful these graphs are as they make it significantly easier to comprehend the structure of a module (see Figure 6.4). On the other hand, they are also a reminder of how quickly simple things tend to become complex when it concerns software. This also clearly demonstrates why a highly interactive graphical user interface is required.

We are confident that the research performed by Van den Hoven [Hov15] could have benefitted from a working metamodel environment as envisioned for the meta$^{HS}$ environment. It would have allowed him to focus solely on the source code metrics instead of having to deal with all the low-level details of the AST. For these reasons, it is our opinion that the development of a metamodel for the Haskell programming language is indeed useful.
8.3. **Future Work**

The development of the prototype for the meta\textsuperscript{HS} environment generated quite a few suggestions for future research projects.

It would be worthwhile to investigate the errors generated by the haskell-src-exts parser as these errors prevent a number of modules from being analyzed by the meta\textsuperscript{HS} environment. Additionally, it would be advisable to switch to a different parse function that can extract additional information (e.g., comments).

The prevention of nonsensical relations could also be an interesting direction for future research. As discussed in subsection 4.3.2, this could be accomplished by the inclusion of a validation function.

The addition of new extractors and aggregators will greatly increase the usability of the meta\textsuperscript{HS} environment and the number of source code metrics that can be calculated. A number of suggestions can be found in Section 4.4.

Performance improvements could also greatly benefit the usability of the meta\textsuperscript{HS} environment as the growing number of extractors and aggregators will probably slow down the environment over time.

Designing and implementing a graph layout engine – in Haskell – would be very beneficial as it removes the current reliance on an external graph layout engine. The development of a highly interactive graphical user interface would also be advisable.

The meta\textsuperscript{HS} EDSL layer is still in a very early stage of development and it should evolve as the meta\textsuperscript{HS} environment grows over time. Adding new source code metrics will require new support from the meta\textsuperscript{HS} EDSL layer and it would be interesting to see how this will influence the EDSL.
META MODEL FOR THE EXAMPLE MODULE

The pretty-printed versions of the MetaModel for the Example module is shown in Listing A.1.

Listing A.1: MetaModel for the Example module

```
MetaModel
  "LCOM":
    (Module "Example", IntValue "2")
  "_contains":
    (Program "example", Module "Example")
    (Module "Example", Function "Example.Blue")
    (Module "Example", Function "Example.Car")
    (Module "Example", Function "Example.Green")
    (Module "Example", Function "Example.Red")
    (Module "Example", Function "Example.age")
    (Module "Example", Function "Example.carType")
    (Module "Example", Function "Example.color")
    (Module "Example", Function "Example.isMagic")
    (Module "Example", Function "Example.magicNumber")
    (Module "Example", Function "Example.newCar")
    (Module "Example", Function "Example.repaint")
    (Module "Example", DataType "Example.Car")
    (Module "Example", DataType "Example.Car")
    (Module "Example", DataType "Example.Color")
    (Module "Example", TypeSynonym "Example.CarType")
    (DataType "Example.Car", Function "Example.Car")
    (DataType "Example.Car", Function "Example.age")
    (DataType "Example.Car", Function "Example.carType")
    (DataType "Example.Car", Function "Example.color")
    (DataType "Example.Car", Function "Example.Blue")
    (DataType "Example.Car", Function "Example.Green")
    (DataType "Example.Car", Function "Example.Red")
  "_source":
    (Module "Example", Location "projects/input/example/src/Example.hs" 1 1 29 1)
```
(Function "Example.isMagic", Location "projects/input/example/src/Example.hs" 28 1 28 27)
(Function "Example.magicNumber", Location "projects/input/example/src/Example.hs" 25 1 25 17)
(Function "Example.newCar", Location "projects/input/example/src/Example.hs" 15 1 19 6)
(Function "Example.repaint", Location "projects/input/example/src/Example.hs" 22 1 22 48)
(DataType "Example.Car", Location "projects/input/example/src/Example.hs" 8 1 12 22)
(DataType "Example.Color", Location "projects/input/example/src/Example.hs" 6 1 6 48)
(TypeSynonym "Example.CarType", Location "projects/input/example/src/Example.hs" 4 1 4 32)
"_uses":
(Function "Example.Blue", DataType "Example.Color")
(Function "Example.Car", DataType "Example.Car")
(Function "Example.Car", DataType "Example.Color")
(Function "Example.Car", TypeSynonym "Example.CarType")
(Function "Example.Car", UnknownType ".Int")
(Function "Example.Green", DataType "Example.Color")
(Function "Example.Red", DataType "Example.Color")
(Function "Example.age", DataType "Example.Car")
(Function "Example.age", UnknownType ".Int")
(Function "Example.carType", DataType "Example.Car")
(Function "Example.carType", TypeSynonym "Example.CarType")
(Function "Example.color", DataType "Example.Car")
(Function "Example.color", DataType "Example.Color")
(Function "Example.isMagic", Function ".(==)")
(Function "Example.isMagic", Function "Example.magicNumber")
(Function "Example.isMagic", UnknownType ".Bool")
(Function "Example.isMagic", UnknownType ".Int")
(Function "Example.magicNumber", UnknownType ".Int")
(Function "Example.newCar", Function "Example.Car")
(Function "Example.newCar", Function "Example.age")
(Function "Example.newCar", Function "Example.carType")
(Function "Example.newCar", Function "Example.color")
(Function "Example.newCar", DataType "Example.Car")
(Function "Example.newCar", DataType "Example.Color")
(Function "Example.newCar", UnknownType ".String")
(Function "Example.repaint", Function "Example.color")
(Function "Example.repaint", DataType "Example.Car")
(Function "Example.repaint", DataType "Example.Color")
(TypeSynonym "Example.CarType", UnknownType ".String")
The directory structure of the meta$^\text{HS}$ source code is shown in Table B.1. A short description explaining the main purpose of the file or directory is included.

The content of the ‘project’ subdirectory is detailed in Table B.2. The source code of a project to analyze should be stored in a subdirectory of the ‘input’ directory that is named after the name of the project (‘<ProjectName>’ is used in this table as a placeholder for the project name). The ‘output’ and ‘report’ directories will be generated by the meta$^\text{HS}$ program. The other directories and files are used for the web browser interface.
### Table B.1: Directory structure for the meta\textsuperscript{HS} prototype

<table>
<thead>
<tr>
<th>Path:</th>
<th>Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td>doc/</td>
<td>Haddock documentation folder</td>
</tr>
<tr>
<td>MetaHS/</td>
<td></td>
</tr>
<tr>
<td>DataModel/</td>
<td></td>
</tr>
<tr>
<td>Extractor/</td>
<td></td>
</tr>
<tr>
<td>Module/</td>
<td></td>
</tr>
<tr>
<td>Contains.hs</td>
<td>Extracts Module Contains relations</td>
</tr>
<tr>
<td>Source.hs</td>
<td>Extracts Module Source relations</td>
</tr>
<tr>
<td>Uses.hs</td>
<td>Extracts Module Uses relations</td>
</tr>
<tr>
<td>Program/</td>
<td></td>
</tr>
<tr>
<td>Contains.hs</td>
<td>Extracts Program Contains relations</td>
</tr>
<tr>
<td>MetaModel.hs</td>
<td>Defines the MetaModel</td>
</tr>
<tr>
<td>Utils/</td>
<td></td>
</tr>
<tr>
<td>File/</td>
<td>Utility functions for files and folders</td>
</tr>
<tr>
<td>Language/</td>
<td>Utility functions for the haskell-src-exts parser</td>
</tr>
<tr>
<td>Find.hs</td>
<td>Utility functions for searching AST nodes</td>
</tr>
<tr>
<td>Name.hs</td>
<td>Utility functions for qualified names</td>
</tr>
<tr>
<td>NameResolution.hs</td>
<td>Utility functions for name resolution</td>
</tr>
<tr>
<td>EDSL/</td>
<td></td>
</tr>
<tr>
<td>Graph/</td>
<td></td>
</tr>
<tr>
<td>Types.hs</td>
<td>Defines types for the graph part of the EDSL layer</td>
</tr>
<tr>
<td>UsesGraph.hs</td>
<td>Generates the internal uses graph</td>
</tr>
<tr>
<td>Utils.hs</td>
<td>Utility functions for the graph part of the EDSL layer</td>
</tr>
<tr>
<td>Graph.hs</td>
<td>Exports the graph modules of the EDSL layer</td>
</tr>
<tr>
<td>MetaModel.hs</td>
<td>Functions operating on MetaModel objects</td>
</tr>
<tr>
<td>Utils.hs</td>
<td>Utility functions for the EDSL layer</td>
</tr>
<tr>
<td>Extensions/</td>
<td></td>
</tr>
<tr>
<td>LCOM.hs</td>
<td>Implements the LCOM software metric</td>
</tr>
<tr>
<td>EDSL.hs</td>
<td>Exports the modules of the EDSL layer</td>
</tr>
<tr>
<td>projects/</td>
<td>Top-level directory for the projects to analyze</td>
</tr>
<tr>
<td>Main.hs</td>
<td>Defines the main function to allow for compilation</td>
</tr>
<tr>
<td>Report.hs</td>
<td>Generates the HTML report</td>
</tr>
<tr>
<td>Test.hs</td>
<td>Defines the tests used for validation</td>
</tr>
<tr>
<td>Types.hs</td>
<td>Defines types and associated functions used by Test.hs</td>
</tr>
</tbody>
</table>
### Table B.2: Project directory structure

<table>
<thead>
<tr>
<th>Path:</th>
<th>Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td>css/</td>
<td>Cascading Style Sheets</td>
</tr>
<tr>
<td>images/</td>
<td>Images</td>
</tr>
<tr>
<td>input/</td>
<td>Projects to analyze</td>
</tr>
<tr>
<td>js/</td>
<td>JavaScript files</td>
</tr>
<tr>
<td>output/</td>
<td>LCOM graphs generated by the GraphViz Circo layout engine</td>
</tr>
<tr>
<td></td>
<td>LCOM graphs generated by the GraphViz Dot layout engine</td>
</tr>
<tr>
<td></td>
<td>The generated meta-model</td>
</tr>
<tr>
<td></td>
<td>Prettified version of the generated meta-model</td>
</tr>
<tr>
<td></td>
<td>List of errors reported by the parser</td>
</tr>
<tr>
<td>&lt;ProjectName&gt;/graphs/</td>
<td>The generated LCOM report</td>
</tr>
<tr>
<td>circo/</td>
<td>The source code editor HTML page</td>
</tr>
<tr>
<td>dot/</td>
<td>The main HTML page</td>
</tr>
<tr>
<td>metaModel.txt</td>
<td></td>
</tr>
<tr>
<td>metaModel_pretty.txt</td>
<td></td>
</tr>
<tr>
<td>parseError.txt</td>
<td></td>
</tr>
</tbody>
</table>
The legend for information flow diagrams is shown in Figure C.1.

- **Metal** data model
- Layer name
- Layer divider
- Flow of information
- Haskell source code file
- Function
- Abstract syntax tree (AST)
- Metamodel data

Figure C.1: Legend for information flow diagrams.
ACRONYMS

**AST**  Abstract Syntax Tree.

**CSS**  Cascading Style Sheets.

**EDSL**  Embedded Domain-Specific Language.

**HTML**  Hyper Text Markup Language.

**IDE**  Integrated Development Environment.

**JS**  JavaScript.

**LCOM**  Lack of Cohesion of Module Elements.

**RTF**  Rich Text Format.

**URI**  Uniform Resource Identifier.
**ACADEMIC ARTICLES**


<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The architecture of Moose (based on [NDG05]).</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>An overview: from code to metrics and visualizations via M^3 models [Bas+15].</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Lack of Cohesion of Module Elements (LCOM) example.</td>
<td>13</td>
</tr>
<tr>
<td>4.1</td>
<td>Information flow diagram for the meta^HS metamodel.</td>
<td>18</td>
</tr>
<tr>
<td>5.1</td>
<td>Information flow diagram for the meta^HS metamodel.</td>
<td>35</td>
</tr>
<tr>
<td>5.2</td>
<td>The LCOM graph for the Example module.</td>
<td>39</td>
</tr>
<tr>
<td>6.1</td>
<td>Information flow diagram for the meta^HS metamodel.</td>
<td>41</td>
</tr>
<tr>
<td>6.2</td>
<td>The index page.</td>
<td>44</td>
</tr>
<tr>
<td>6.3</td>
<td>The LCOM report table for the Example project.</td>
<td>44</td>
</tr>
<tr>
<td>6.4</td>
<td>The LCOM graph for the Example module.</td>
<td>45</td>
</tr>
<tr>
<td>6.5</td>
<td>The HTML editor opened on the Car datatype.</td>
<td>45</td>
</tr>
<tr>
<td>7.1</td>
<td>The Contains annotations for the Example module.</td>
<td>48</td>
</tr>
<tr>
<td>7.2</td>
<td>The Uses annotations for the Example module.</td>
<td>49</td>
</tr>
<tr>
<td>7.3</td>
<td>The annotated LCOM image for the Example module.</td>
<td>49</td>
</tr>
<tr>
<td>8.1</td>
<td>Information flow diagram for the meta^HS metamodel.</td>
<td>56</td>
</tr>
<tr>
<td>C.1</td>
<td>Legend for information flow diagrams.</td>
<td>64</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Declaration usage</td>
<td>27</td>
</tr>
<tr>
<td>7.1</td>
<td>Results comparison</td>
<td>52</td>
</tr>
<tr>
<td>7.2</td>
<td>Results comparison including the modified meta^{HS} prototype</td>
<td>53</td>
</tr>
<tr>
<td>8.1</td>
<td>Throughput times of metamodel and LCOM report generation</td>
<td>56</td>
</tr>
<tr>
<td>B.1</td>
<td>Directory structure for the meta^{HS} prototype</td>
<td>62</td>
</tr>
<tr>
<td>B.2</td>
<td>Project directory structure</td>
<td>63</td>
</tr>
</tbody>
</table>
**List of Listings**

4.1 Example source code .................................................. 19  
4.2 Partial AST for the Example module ................................. 20  
4.3 URI based Elements examples ........................................ 21  
4.4 URI based elements excerpt for source location information ... 21  
4.5 Elements ..................................................................... 23  
4.6 Pretty-printing of Elements .............................................. 24  
4.7 Relations ...................................................................... 25  
4.8 MetaModel .................................................................... 26  
4.9 Example MetaModel – part 1 ........................................... 28  
4.10 Example MetaModel – part 2 ......................................... 28  
4.11 Example MetaModel – part 3 ......................................... 29  
4.12 Example MetaModel – part 4 ......................................... 31  
4.13 Example MetaModel – LCOM relation ............................. 34  
A.1 MetaModel for the Example module ................................. 59