Towards Generating Software Architectures

Towards Generating Software Architectures

By

Alexander Schaap, B.Sc.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirements for the Degree

Master of Applied Science

McMaster University

© Copyright by Alexander Schaap, September 2016

Master of Applied Science (2016) McMaster University
(Software Engineering) Hamilton, Ontario

Title:	Towards Generating Software Architectures
Author:	Alexander Schaap, B.Sc. (Twente University)
Supervisors:	Dr. Jacques Carette & Dr. Mark Lawford
Number of Pages:	viii, 91

Abstract

Program generators are programs that produce other programs, thereby saving time and effort. Program families can be encoded in a generator to produce all members of such a family while leveraging automation and reducing duplication. Through multi-stage programming (MSP) in the form of MetaOCaml, a generator has been created for multiple partial implementations of KeyWord In Context (KWIC) example from Parnas' seminal paper on modular decomposition. What makes this work novel is the generation of different software architectures, as well as different programming paradigms (imperative or functional). This is achieved by using a common abstract internal representation of the signatures of the system components, and applying them to a single abstract representation of the commonalities of the components. Thus the work challenges commonly held perceptions of concepts such as software architecture and programming paradigm by being able to generate members of a program family that varies on both these aspects.

Acknowledgments

An expression of gratitude.

Declaration of Academic Achievement

The student will declare his/her research contribution and, as appropriate, those of colleagues or other contributors to the contents of the thesis.

Table of Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

API	application programming interface	3
AST	abstract syntax tree	26
CGAL	Computational Geometry Algorithms Library	33
CPS	continuation-passing style	38
DSL	domain-specific language	6
GE	Gaussian Elimination	38
KWIC	KeyWord In Context	iii
LEDA	Library of Efficient Data types and Algorithms	33
MSP	multi-stage programming	iii

Chapter 1

Introduction

Products belonging to the same family share common features. When the products are software, it is often possible to decompose them into modules in such a way that the common features are separated from the product-specific ones. Automated assembly of different combinations results in all products being created while costs are reduced through reuse of modules as well as said automation. While a generator that does this must be specific to the program family, the idea can be applied wherever program families exist.

In order to identify the commonalities and differences between members of a program family, one needs to decompose them and analyze the resulting components. Decomposition of tasks is documented and rationalized by Parnas (1972), who uses the KeyWord In Context (KWIC) program to illustrate. Since then, (Garlan and Shaw (1994) and van Vliet (2000)) have illustrated more architectures using KWIC.

Program generators typically produce implementations that vary depending on low-level design choices, such as variations in certain algorithms. This thesis explores generating implementations that vary on the higher-level design choices of software architecture and programming paradigm. To achieve this, MSP using MetaOCaml is combined with novel code-generation techniques.

1.1 Motivation

Many industries have to support product lines over many years, and these undergo significant architectural evolutions such as the current switch from single-core to multi-core processors in the automotive industry. To support software product lines, one would likely use generative programming, which has proven effective when there are low-level design variations. However, the question remains whether generative programming is well-suited for high-level design variations such as variation in software architecture and variation in programming paradigm. This thesis is a first step in that direction, demonstrating multi-paradigm multi-architecture generation from a single 'model' – in this case an umbrella architecture – for a simple example well known in literature.

1.2 Goals

One goal is to create a generator that produces multiple implementations of KWIC which reflect different software architectures (or none at all, as Parnas (1972) suggested). Ideally, it would be able to generate code reflecting decompositions including those described by Parnas, as well as 'spaghetti code' resulting from using any of these modularizations and then inlining everything.

In order to generate multiple architectures, all information they contain must be known. This seems like an obvious statement, but the implication is that the generator must contain all architectures combined. This means that there is an umbrella architecture, and all resulting architectures are simplifications of it. Finding this umbrella architecture is a vital subgoal.

Another subgoal was to improve and expand upon the modularizations presented by Parnas through more thorough application of his decomposition criteria. He described the principle and applied it to an example, but pointed out some flaws in this application. Furthermore, due to the age of the publication, certain assumptions are no longer necessarily true.

The last goal is the generation of imperative and functional code from the same application programming interface (API), which implies they can be generalized to the same design concepts. An example would be looping, for which the imperative approach tends to be a for-loop or a while-loop, where typically an index or a condition changes respectively. In functional programming, this would be done through recursion, avoiding the notion of mutable state that the previous examples implied. However, there is still a condition on which to loop, whether the function needs to call itself or the condition of the while-loop changes. Abstracting the implementation-specific details away leaves us with a higher-level design language, as described in Curutan (2013). In theory, these loops have long been known to be equivalent, but to the best of the author's knowledge, this has never been put into practice using a generator providing a unified API.

1.3 Approach

The first intermediate goal is to find the umbrella architecture. The first step towards this is to implement the architectures described. The benefits of this are deepened understanding, better command of the programming language used, and a target for the generator to generate. Deepened understanding leads to better subsequent implementations that encapsulate more design decisions in separate modules, creating increasingly complex architectures. Some notable assumptions to be challenged are: the need for characters, the definition of a word, the use of indices to manipulate arrays, and the need for imperative programming. Once a sufficient umbrella architecture has been implemented, the generator can be created. MetaOCaml will be combined with techniques described by Carette (2006), Carette, Kiselyov, and Shan (2009) and Carette and Kiselyov (2011).

1.4 Contribution

This work explores the possibility of creating a program family encompassing different software architectures using generative programming (specifically, multi-stage programming). A detailed look at the KWIC program and an analysis of some of the published comparisons between different software architectures it illustrates give a foundation for the analysis and a more thorough application of Parnas' criteria for decomposition (Parnas 1972) that follow. It also reconfirms the existence of a paradigm-independent design language that is more abstract than actual code but less abstract than the high-level language used to describe algorithms (Curutan 2013). However, it also shows that this design language can be employed to adequately describe both imperative and functional code in a unified API.

1.5 Outline

This thesis starts with background material in chapter 2, explaining the key concepts and techniques used throughout the document. This is followed by preliminaries and related work in chapter 3, in which software architectures are discussed and illustrated using KWIC, metaprogramming is introduced, and an overview of related work is provided. Subsequently, chapter 4 discusses the goals for the generator and the approach taken before an in-depth analysis resulting in design constraints. The implementation is described in chapter 5. Finally, chapter 6 concludes the thesis and discusses future work. Preliminary research is included in Appendix A.

Note that later chapters reference paths such as /ge-based-kwic/ a number of times; these are both hyperlinks to the relevant file or directory and paths in the accompanying archive file found at `https://www.cas.mcmaster.ca/~schaapal/mthesis/code.tar.gz`. Not including code listings in the appendices reduces the number of pages by approximately 150 pages, saving a significant amount of paper for every hard copy printed.

Chapter 2

Background

This chapter aims to briefly explain the concepts that are key to the work presented in this thesis. First, a brief overview of Parnas' seminal paper is given in section 2.1, along with a description of the KeyWord In Context (KWIC) example program. Decomposition of tasks into modules is the basis for software architectures, which are defined in section 2.2. (Four architectures are illustrated using KWIC in section 3.1.) Given that there are components, the connection to product families becomes apparent. Product families and code generation is elaborated upon in section 2.3. After that, a novel approach towards embedding a domain-specific language (DSL) and generating a result is succinctly described in section 2.4; this is the foundation of the generator presented in this work.

2.1 On the Criteria to Be Used in Decomposing Systems into Modules

The title of this section is also the title of an often-cited paper by David L. Parnas (1972). In it, he rationalizes the modularization of programs. Modularization in this case means separating the whole into logical parts which connect and communicate through their interfaces (which define the inputs and outputs). The advantages of modularization listed by Parnas are managerial, product flexibility and comprehensibility. The managerial benefit is that it allows multiple people to work on the different parts, potentially allowing for faster completion. Product flexibility hints at creating a family of products, because it would allow for substantial changes to a program by modifying only one module. Comprehensibility would allow for larger systems because one would not have to look at the whole but only study modules one by one. Another benefit one can infer would be faster debugging and maintenance because a problem is first traced to a module, at which point only that module needs to be considered. Furthermore, modules have the potential to be reused, either as a whole or in part. Modularization is typically done intuitively, based on the tasks a program needs to perform; these can also be represented by a flowchart. Parnas argues that the resulting modules should have minimal interfaces, allowing for greater variation of module implementations. This is also called information hiding. These variations should be meaningful, allowing for certain anticipated changes (which dictate the modularization). One anticipated change could be the sorting algorithm, creating the need for a decomposition which enables changing the sorting algorithm by changing only one module, leaving interfaces and other modules untouched. Conversely, the aspects of the program that are expected to stay unchanged can be used as the boundaries for modularization. This approach requires more thought but produces better results than the intuitive approach.

2.1.1 KeyWord In Context (KWIC)

Parnas illustrates his approach on examples, and also shows that the application of his approach is not a simple process and that improvement to the resulting modularization is often possible. The simplest example (and the only one in some versions of this paper) is a small program named “Keyword in Context”, or KWIC. As Parnas succinctly puts it:

> The KWIC index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be “circularly shifted” by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

KWIC is small enough that modularization is optional, but illustrates the point in an overseeable and comprehensible manner.

2.2 Software Architectures

When designing a system, its top-level decomposition and the expected interaction between the resulting components is known as its *software architecture* (van Vliet 2000). This represents early design decisions. Architecture also provides a basis from which one can work when considering product families, providing boundaries for potential software reuse. While some architectures work better than others for certain systems, there is not always one unquestionably best solution. The best solution can conceivably depend on the features requested.

The KWIC program is a commonly used program to explain and differentiate various software architectures as will be shown in section 3.1.

2.3 Code Generation as Applied to Product Families

According to Parnas 1976, program families are "defined (analogously to hardware families) as sets of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members.".

One could write each member of such a family by hand, repeating a lot of work every time a common property is reimplemented. Intuitively, one feels the need for reusable software components. Decomposition of all members using Parnas' criteria (Parnas 1972) while keeping the commonalities between them in mind should yield a set of highly reusable components. Various selections of these along with member-specific components can then be combined into a product family member. Applying automated code generation techniques to the problem of creating members of product families is the next logical step. Through generative programming, one can specify the member to be generated via an abstract configuration and have the generator produce a complete product in the form of code (Czarnecki and Eisenecker 2000).

2.4 Finally Tagless, Partially Evaluated

Another paper that greatly contributed to the results presented in this thesis is "Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages" by Jacques Carette, Oleg Kiselyov and Chung-chieh Shan. When one embeds a domain-specific language (DSL) in another language, there is typically some overhead as a result when executing the embedded language; dispatching on syntax of that language and tagging types of values in that language. Dispatch overhead can be avoided using code generation (through multi-staged programming, for example). The way tagging overhead can be avoided is described in this paper, though it humbly claims credit only for combining ideas present in other papers. When specifying types for the embedded language, one typically creates a number of type constructors for each of them. Types are certainly desirable, because they further restrict the existence of nonsensical code, ensuring proper execution. When evaluating, these type constructors (tags) are first pattern-matched on and then removed, which is not without cost. Instead, the solution presented here uses functions for everything (hence tagless). The word 'Finally' in the title denotes a final approach, which means representing each term by its "denotation in a semantic algebra" rather than its abstract syntax (initial approach).

Chapter 3

Preliminaries & Related Work

First, four software architectures are explored, discussed, analyzed and compared using KWIC to illustrate. Next, various metaprogramming techniques are reviewed, culminating in an overview of MetaOCaml. Finally, an overview of related work is presented.

3.1 KWIC Software Design Architectures

Even though one can easily come up with an implementation of KWIC, one tends to make several implicit assumptions while doing so. However, a more structured approach will quickly lead to considerations such as performance on large data sets, different use cases and whether or not the process is iterative, to name a few. The size of the data set varies, but ultimately one chooses to take a certain upper bound into account. However, this bound may change in the future, and this can be taken into account by allowing certain parts to be swapped out. For example, the sorting module could keep the entire data set in working memory while sorting – this does not scale well, and warrants replacing it with a module that only keeps a small portion in working memory, or perhaps one that distributes sorting over other computers possibly connected over a network. Similarly for other considerations, one determines a number of anticipated changes within the system. Every architecture enables a different (possibly overlapping) set of anticipated changes.

A number of modularizations or decompositions are described and compared below. The two words will be used interchangeably in this document. Comparison between the architectures and further discussion of anticipated changes occurs in subsection 3.1.2 and subsection 3.1.4. The KWIC program is discussed in subsection 2.1.1. To summarize, the tasks that need to be carried out are:

1. Read lines from the input medium
2. Create all possible circular shifts from these lines
3. Sort all shifts alphabetically
4. Output the result

3.1.1 Parnas' Modularizations

Parnas (1972) compares and contrasts two decompositions, namely the more intuitive modularization that typically follows from a flowchart, and the one he proposes which employs information hiding. Others have built on this and proposed and compared more architectures. Garlan and Shaw (1994) elaborate on two more architectures, namely an event-based one attributed to Garlan, Kaiser and Notkin and a pipe-and-filter one "inspired by the Unix index utility". They also provide a more detailed comparison structured around specific criteria.

Flowchart-Based Modularization (Shared Data Architecture)

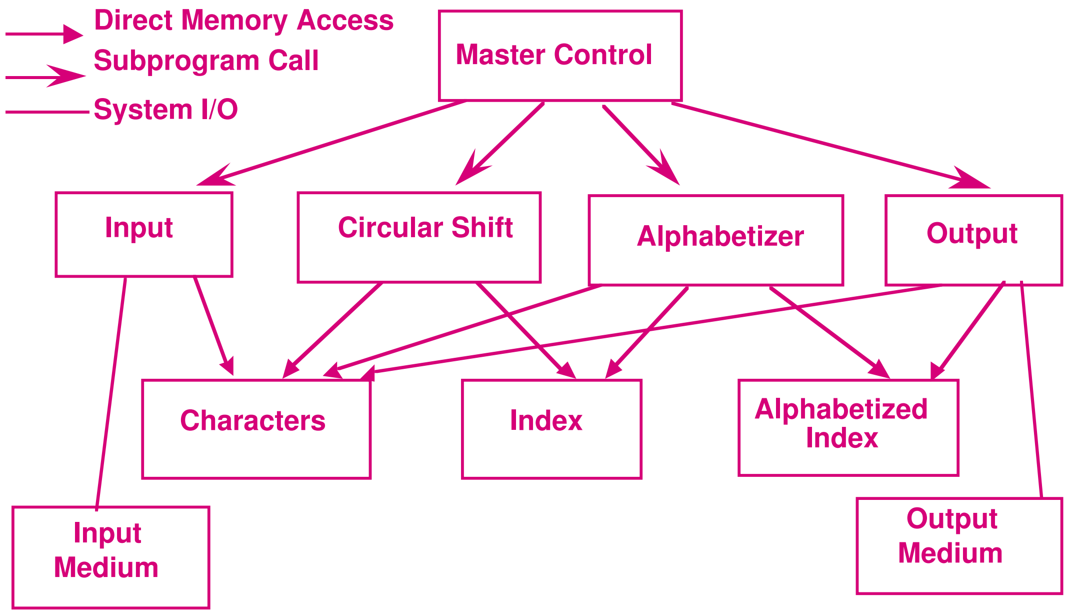

Figure 3.1: Modularization following from a flowchart (Garlan and Shaw 1994)

Following the steps the program needs to carry out (outlined above), one can intuitively see how it will be decomposed into five modules:

1. Input - reads lines from some input medium and stores them in the format accepted by the other modules

2. Circular Shift - stores index of all possible shifts of all lines

3. Alphabetizing - stores index of sorted shifts using the results of the previous two modules

4. Output - prints "nicely formatted" result using the results of modules 1 and 3

5. Master Control - runs aforementioned modules sequentially, possibly taking care of errors, memory allocation and other miscellaneous tasks

Information-Hiding Modularization (Abstract Data Architecture)

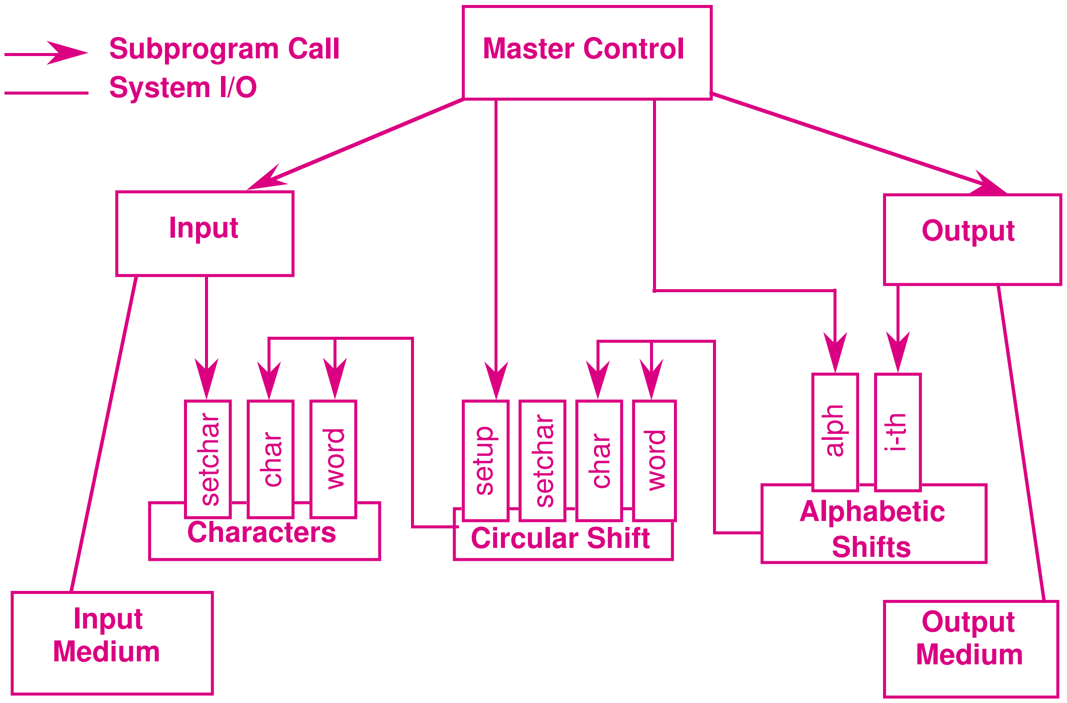

Figure 3.2: Modularization resulting from following the information-hiding principle (Garlan and Shaw 1994)

There are a number of ways the previous decomposition can be improved. Parnas believed that every design decision that might be revisited, or anticipated change, should be contained within its own module. He proposed the following modules, but pointed out some flaws in his choices which illustrate that this is not a trivial task.

1. Line Storage - provides functions that access and modify the data structure in which lines can be stored

2. Input - reads lines from the input medium and calls the appropriate functions from the Line Storage module to store them
3. Circular Shifter - provides functions analog to those in module 1 that give access to shifts, as well as a setup function that must be run first in order to construct the index used by the other functions
4. Alphabetizer - presents function to access sorted shifts, as well as a sort function which must be run first, to create the index used by the access function (analog to the setup function of module 3)
5. Output - prints shifts
6. Master Control - calls the other modules in the appropriate order

3.1.2 Parnas' Comparison

Parnas describes these two decompositions as being "different ways of cutting up what may be the same object". The conclusion he draws from that is that the final result after assembling both decompositions might be identical. The differences are therefore limited to the interfaces between modules and the chunks of work being done at a time by a module. This also means that one program has multiple representations, namely a runnable one and one for making modifications and human reading.

However, differences come to light when considering the property of *changeability*. Parnas provides a partial list of design decisions he calls "questionable and likely to change under many circumstances". Among these are the input format, location where lines are stored, a fixed number of characters per words (four), representing shifts using indices versus writing them out, and whether to sort alphabetically all at once or distributing that computation. More design decisions can be called into question, such as what exactly a word is, which order to sort in, and whether one or multiple locations to store things in are advantageous to name a few. Some of these changes would affect every module of the first decomposition, such as where the lines are stored or how many characters a word contains. In contrast, the information-hiding modularization confines changes to the line storage location to a single module.

Other criteria for comparison are independent development and comprehensibility. Parnas considers interfaces between modules representative of design decisions. The flowchart-based decomposition results in more complex interfaces because it requires more design decisions to be made before the interfaces are finalized. This delays the point at which independent development can commence. Because many details are hidden within modules, less design decisions will need to be made for interfaces in the second decomposition, making them more abstract. Less decisions and less interdependence means less interaction is required between developers, and therefore more independent development which can begin at an earlier time. Parnas argues that the flowchart-based modularization "will only be comprehensible as a whole" due to the interdependence between modules. He believes this is not the case for the information-hiding decomposition.

The criteria used to create each modularization is also taken into account. The first decomposition essentially follows from a flowchart, and will not be sufficient for larger systems. Modules are seen as subroutines. In contrast, information-hiding is used to construct the second decomposition. Every module essentially has a "secret", which consists of the design decision contained. The interfaces between modules therefore try to hide as much of the inner workings as possible. The result is a design that allows for change. Every design decision contained in a module will only require changes to that specific module, should that decision be revisited. Consequently, decisions that are likely to change should be contained in their own module.

Parnas also considers efficiency, noting that the information-hiding decomposition will easily be less efficient for KWIC due to the function call overhead. He therefore hints at an unusual compilation process that assembles code, treating functions as subroutines that are effectively inlined. This results in the decomposition not being apparent from the final result of this process, which adds meaning to his earlier statement describing the possibility of the end result being equal when assembling both decompositions.

While Parnas envisioned this assembly process to be at a low level, a lot of information is already discarded at that stage; multi-stage programming assembles code fragments at a much higher level, and is therefore able to make more guarantees in terms of syntactic correctness and type-safety. It is for this reason that a language like MetaOCaml[1] was chosen.

3.1.3 Additional KWIC Software Design Architectures by Garlan and Shaw (1994)

Implicit Invocation Modularization (Event-Based Architecture)

Garlan, Kaiser and Notkin argue that the Information-Hiding Modularization (Abstract Data Architecture) is not as conducive to addition of functionality as some might desire. They therefore propose the modularization seen in Figure 3.3, which does not require modifying existing modules to add new

[1]`http://okmij.org/ftp/ML/MetaOCaml.html`

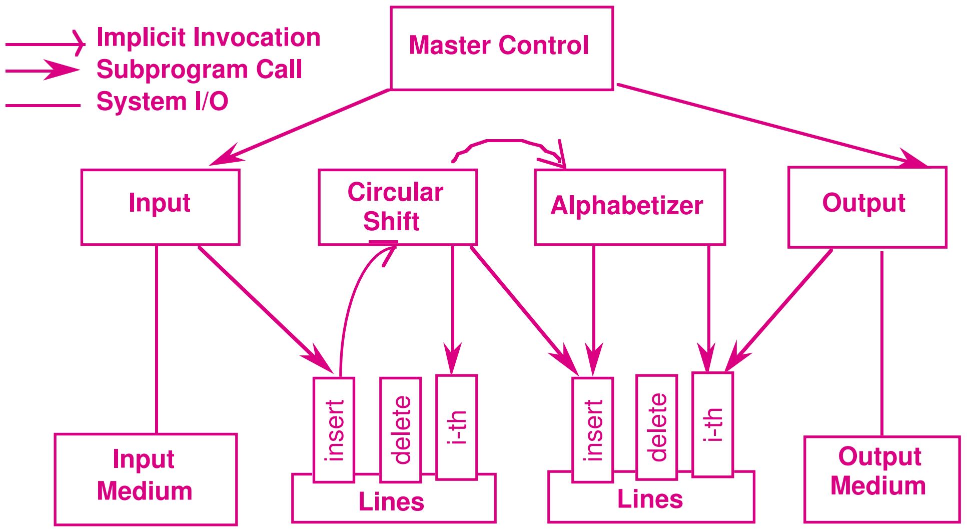

Figure 3.3: Modularization using implicit invocation (Garlan and Shaw 1994)

ones. While there are no new kinds of modules compared to the previous modularization, the interaction is slightly different. There are two places where lines are stored, and every next step is implicitly called. The storage containing lines is now an event, and another module can register with this one to be notified of such an event. The advantage over a conventional function call is that no modification to either module is necessary, as long as they fit this observer pattern; the observer ("interested" module) simply registers itself with an observable (module that can produce events), which maintains a list of observers to notify. The storage modules still hide the actual data structures used to store their contents. One potential issue is the order in which these implicitly invoked modules are executed, since all observers are called at once. The presence of two storage modules also implies more space is required.

Pipe-and-filter Modularization (Dataflow Architecture)

Lastly, this modularization is essentially a sequence of filters. There is no centralized control through a Master Control module, instead it is distributed

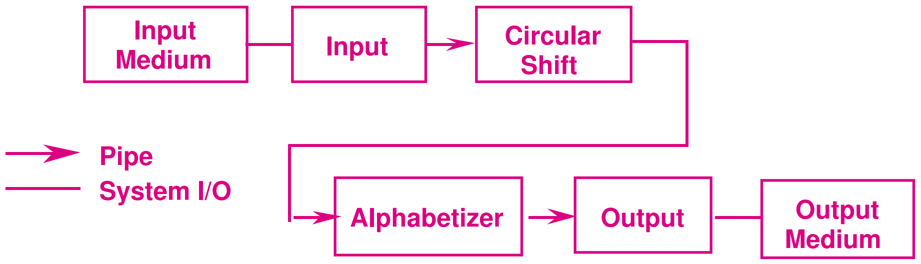

Figure 3.4: Modularization in the style of pipe-and-filter (Garlan and Shaw 1994)

– each module calling the next. New functionality can easily be added by inserting a new module at the appropriate point in the pipeline. However, data representation has to be considered between every pair of modules that are connected, and any kind of interactivity (such as deleting lines) is not what this approach is well-suited for. Garlan and Shaw (1994) also argue that space would be used inefficiently since modules have to pass all data on to the next module instead of manipulating it while it resides in some storage. van Vliet (2000) explains that error handling is also difficult when using this architecture. This architecture could potentially work incrementally, but assuming the output will be sequential, the sorting module will requires all lines to ensure the proper order.

3.1.4 Four-way KWIC Software Design Architecture Comparison

It should be pointed out that the contents of the comparison table presented by Garlan and Shaw (1994) appears to be debatable[2]. For this reason, van Vliet

[2]van Vliet 2000, and possibly Murray Wood in 2004 (according to PDF metadata): http://www.csee.wvu.edu/~ammar/CU/swarch/lecture%20slides/slides%206%20sw%20arch%20design/lecture%20slides%206%20Architecture%20Design/supporting%20slides/KWIC%20example%20Architecture3.pdf

(2000) was also consulted.

The criteria used to compare the four architectures described above are as follows:

Changes in (overall) algorithm such as when the lines are shifted – either when they are read, when the sorting module requires them or all at once

Changes in data representation such as the way lines are stored, and how circular shifts are represented (index and offset or explicitly)

Enhancements such as ensuring the circular shifting module omits shifts that start with noise words such as "a", "an", "and", etc – typically module-specific

Performance in terms of both space and time

Reuse in the sense that modules can serve as reusable entities; van Vliet (2000) explains that while the simplicity of the interaction between modules is an important factor, "whether the abstraction it embodies is useful in another context" is of much more consequence.

Independent Development which can start when all decisions affecting multiple modules have been made; how soon programmers can begin working on their own

Comprehensibility in terms of time and effort required to understand a part of the program

The last two criteria are mentioned in Parnas' comparison, and the table from Garlan and Shaw (1994) is expanded upon by van Vliet (2000) using these two. van Vliet (2000) also adds a neutral or 'somewhere in the middle' evaluation in addition to Garlan and Shaw (1994)'s '+' and '–'.

Shared Data Architecture

Changes to data representation are a lot of work because all modules contain explicit references to this representation. The modules are tightly coupled, containing explicit references to the data structure used to store the words. This means the components are not generally reusable, but efficient. Garlan and Shaw (1994) argue that adding enhancements also accessing the shared data (such as the example above) is easy, but van Vliet (2000) notes that others might not be. Changes to the overall algorithm might be easy or difficult, depending on the change. As noted in Parnas' Comparison, many design decisions pertaining to the format of the shared data need to be made before independent development can begin, resulting in complex interfaces. As Parnas (1972) argues, understanding of a module usually requires understanding of other modules if not the whole program, slowing comprehensibility.

Abstract Data Architecture

As Parnas pointed out, changing the data representation will be easy, since this design decision is encapsulated in its own module. No modification to other modules is required. Since there is looser coupling, modules are better suited for reuse. However, it is probably more difficult to enhance functionality. Removing shifts starting with noise words would be doable, but it might become complicated. Garlan and Shaw (1994) and van Vliet (2000) both argue that the interaction between modules is still explicit within each module, potentially making enhancements more work. For the same reason, changes to the overall algorithm may also prove complex. van Vliet (2000) argues that this depends however, since a change in algorithm that pertains a module's secret will be easy, but cross-module changes in algorithm might prove more challenging. Performance would be good if an assembler as described by Parnas existed. Otherwise, function call overhead and possibly larger space requirements would slow this architecture down. Apparently, both Garlan and Shaw (1994) and van Vliet (2000) do not consider this to have much effect. The interfaces of this decomposition will be much more abstract in comparison to those of the shared data architecture, allowing independent development to commence earlier. As Parnas (1972) argues, the comprehensibility should be much improved for this architecture compared to the shared data one due to better encapsulation of design decisions resulting in requiring less knowledge of other modules.

Event-driven Architecture

Changes to the overall algorithm as well as enhancements are easier now due to the interaction between modules being much less explicit. Identical to section 3.1.4, van Vliet (2000) argues for cross-module algorithm changes still being difficult. Reuse should also benefit from this, but Garlan and Shaw (1994) feel that this architecture is closer to the Information-Hiding Modularization (Abstract Data Architecture) and therefore has poor reusability. For the same reason, they conclude that data representation is not easy to change. Fortunately, van Vliet (2000) disagrees. Hence the † in Table 3.1. A change in data representation should be easy, since this is still encapsulated in a separate module (which Garlan and Shaw (1994) explicitly mentions). Performance will be bad, lines and shifts are both explicitly stored, doubling the space requirement. Function calls and the Observer pattern further increase overhead.

van Vliet (2000) explicitly mentions the event-scheduling overhead as a possible cause for reduced performance. The order of execution as well as potential cycles due to the event-driven nature of this architecture could be a problem. Similar to the abstract data architecture, independent development can begin early because the interfaces are going to be abstract. Comprehensibility suffers from the lack of knowledge regarding which module(s) will respond when an event is raised, and especially the order in which they do so. It can therefore be unclear which module is in control at a given time.

Dataflow Architecture

Garlan and Shaw (1994) are quick to assert that the pipe-and-filter architecture supports changes in the processing algorithm, enhancements and reuse well due to the fact that it allows one to easily add new filters to the pipeline. However, in their words "it is virtually impossible to modify the design to support an interactive system", which is one of the examples they use to illustrate changes in the overall algorithm. The algorithm is also limited to the sequential style of the architecture. Performance is considered poor due to every module having to parse, unparse, and pass on the entire dataset to the next module. For this reason, changes in data representation between two modules are difficult to implement as well, since this would affect both modules, and the changes might cascade (van Vliet 2000). The data representation within a single module can be changed with relative ease though. Only the format of the datastreams between filters will need to be decided, and these formats tend to be simple, so independent development of filters will be easy. The architecture is comparatively straight-forward and sequential, giving little cause for comprehensibility to suffer.

	Shared Data	Abstract Data Types	Events	Dataflow	*Ideal Goal*
Changes in algorithm	–	$0^{-\dagger}$	$0^{+\dagger}$	$+^{\$}$	+
Changes in data representation	–	+	$+^{-\dagger}$	–	+
Enhancements	$0^{+\dagger}$	–	+	+	+
Performance	+	$+^{\$}$	–	–	+
Reuse	–	+	$+^{-\dagger}$	+	+
Independent development	–	+	+	+	+
Comprehensibility	–	+	0	+	+

Note:
$^{+/-\dagger}$ Garlan and Shaw (1994) and van Vliet (2000) disagree, superscript symbol from the former
$^{\$}$ Debatable

Table 3.1: Summarizing the comparison between the various KWIC architectures, based on the table in van Vliet (2000)

3.2 Metaprogramming

Metaprogramming is the act of writing programs that handle and/or produce other programs. The example familiar to most people would be compilers, which take a program in the form of source code and produce a program in the form of bytecode or machine-specific instructions. Another example would be a program that attempts halting analysis, producing either a proof that the program always halts, or a counterexample showing that under some circumstances, it will not. Note that the latter program is an undecidable problem, so there are only partial solutions. Intuitively, one could see the counterexamples posing a potential problem, because one does not have infinite time to check whether it truly does not terminate at some point.

There are various approaches to metaprogramming:

- String munging

- Macro systems
- Template metaprogramming
- Staged metaprogramming

3.2.1 String Munging

This is the simplest technique, consisting of simply cutting and pasting strings, with no guarantees regarding syntactic or type correctness. This is because the resulting program consists of concatenated strings that could contain anything. A basic example of this can be found in Listing 1. It shows a shell script that produces another shell script upon execution. This newly created script will contain 992 echo statements and will print the numbers 1 to 992 upon subsequent execution. However, any character can be inserted into the new file, so this approach is error-prone and difficult to maintain.

```
#!/bin/sh
# metaprogram
OUTPUT='program.sh'
echo '#!/bin/sh' > $OUTPUT
for I in $(seq 992)
do
        echo "echo $I" >> $OUTPUT
done
chmod u+x $OUTPUT
```

Listing 1: Simplest form of metaprogramming by assembling a string. Based on an example from Wikipedia. (https://en.wikipedia.org/wiki/Metaprogramming#Examples)

3.2.2 Macro Systems

Two examples of macro systems are the C preprocessor and the macros found in Lisp. The former are textual macros, which manipulate code at the token level, whereas the latter are syntactic macros, working at the abstract syntax tree (AST) level. Despite its name, Template Haskell also belongs in this category. To demonstrate a To illustrate this concept briefly and in a way least foreign to the reader, a simple example of a C preprocessor macro is shown in Listing 2. This program defines a swap macro that is subsequently called as if it were a function, but the call will be replaced by the three statements from the definition by the preprocessor. The curly braces scope the variable `c` so that it will not be visible outside the routine defined in the macro, but note that this is left up to the programmer. While similar to the previous example in Listing 1, the C preprocessor ensures that the macros consist only of valid tokens, which is a step up from having no guarantees at all. However, this leaves a lot to be desired because not all combinations of tokens result in valid programs.

```c
#include <stdio.h>
#define SWAP(a, b, type) { type c; c = b; b = a; a = c; }

int main(){
int a = 3;
int b = 5;
printf("a is %d and b is %d\n", a, b);
SWAP(a, b, int);
printf("a is now %d and b is now %d\n", a, b);
return 0;
}
```

Listing 2: Simple metaprogramming using the `#define` macro.

3.2.3 Template Metaprogramming

In this approach, the compiler uses templates to generate temporary code, which is then merged into the rest of the source before an executable is generated. Depending on the template system, certain guarantees can be made. In general, syntactic correctness is assumed. However, type safety is optional and often difficult. A well-known language that implements this is C++, but there are many, including D[3]. D is a language that tries to improve over C++, so its syntax will appear to be similar. An easy example introducing template programming in D can be seen in Listing 3 and Listing 4.

```d
module lt;

bool lessthan(T)(T a, T b) {
  return a < b;
}
```

Listing 3: Elementary template definition

Listing 3 defines a module so that it can subsequently be imported by another file. This module contains one function, namely `lessthan`; this function takes a type argument `T`, and two arguments of type `T`. The result is a Boolean value denoting whether `a` is less than `b`. In Java, the first `T` would have been enclosed like so: `<T>`. The implicit assumption here is that arguments `a` and `b` can be ordered in some way.

Listing 4 instantiates imports Listing 3, and instantiates `lessthan` for integers, comparing 3 to 4 in order to get a `true` back.

Generic Quicksort

[3]http://dlang.org/

```
import std.stdio;
import lt;
void main(string[] args) {
  writeln(lessthan!int(3,4));
}
```

Listing 4: Instantiating template defined in Listing 3. Note that `writeln` is a template function as well; it converts all arguments `to!string(arg)`

Listing 5 contains a generic quicksort algorithm implemented in D. It imports a swap function, which simply swaps to elements in an array. This quicksort implementation also takes a comparison function by which the list will be ordered. Arrays of any type will be sorted, as long as the comparison function is applicable to this type.

Listing 6 runs the quicksort presented in Listing 5. The function used to order elements by is from Listing 3. Casting to type `int` on line 7 is the result of the `length` call. It is necessary because it returns a long integer instead of a regular one.

3.2.4 Multi-Stage Programming

The aim of multi-stage programming (MSP) is to develop generic software that does not pay a runtime penalty for this generality (Taha 2004). It does this by generating and executing code at runtime. Unfortunately, there is little support for writing MSP generators in C or Java. However, an extension to OCaml called MetaOCaml (Kiselyov 2015) was created with this specific purpose in mind. MetaOCaml ensures both syntactic correctness and type safety. The basics of MSP with MetaOCaml will be explained briefly below; Taha 2004 provides a more comprehensive tutorial.

```
module quicksort;
import std.algorithm : swap;
void qs2(T)(T[] values, int start, int end, bool function(T, T)
 ↪  compare) {
  if (end - start > 0) {
    int p = partition2(values, start, end, compare);
    qs2(values, start, p - 1, compare);
    qs2(values, p + 1, end, compare);
  }
}
int partition2(T)(T[] values, int start, int end, bool
 ↪  function(T, T) compare) {
  int storeIndex = start;
  for (int i = start; i < end; i++) {
    if (compare(values[i], values[end])) {
      swap(values[i], values[storeIndex]);
      storeIndex++;
    }
  }
  swap(values[end], values[storeIndex]);
  return storeIndex;
}
```

Listing 5: A slightly more complex template example.

There are three basic MSP constructs (as first introduced in Lisp; Larjani 2013).

- **Brackets**: `.<` and `>.` – these delay execution of any expression, creating typed code fragments as demonstrated by the MetaOCaml interpreter in Listing 7. While `1+2` is simply `3`, enclosing the expression in brackets creates code, which can subsequently be combined with other code before eventually being executed.

- **Escape** `.~` – for combining code fragments, as demonstrated in Listing 8. It takes the code produced in the demonstration shown in Listing 7

```
import std.stdio;
import lt;
import quicksort;

void main(string[] args) {
int[] values = [5,2,3];
writeln("Initially: ", values);
qs2(values, 0, cast(int) values.length - 1, &lessthan!int);
writeln("Result using lessthan: ", values);
}
```

Listing 6: Running the quicksort from Listing 5 using code from Listing 3 as the comparator

```
# let a = 1+2;;
val a : int = 3
# let a = .<1+2>.;;
val a : int code = .<1+2>.
```

Listing 7: Demonstration of MetaOCaml interpreting brackets (Taha 2004)

and “splices” it into a larger code fragment which multiplies the given expression with itself. The types check out because `a` is of type `int code`, which will be just `int` when escaped, which is in turn exactly what the multiplication operator expects. Note that the parentheses to preserve the order of operations are inserted automatically.

```
# let b = .<.~a * .~a >. ;;
val b : int code = .<(1 + 2) * (1 + 2)>.
```

Listing 8: Demonstration of MetaOCaml’s escape, using the results shown in Listing 7 (Taha 2004)

- **Run** `.!` or `Runcode.run` – for executing a code fragment, as seen in the demonstration contained in Listing 9. Here the code assembled in

Listing 8 is executed.

```
# let c = .! b;;
val c : int = 9
```

Listing 9: Demonstration of MetaOCaml's run using the result of Listing 8 (Taha 2004)

The method for building MSP programs is as follows:

1. single-stage program is developed, implemented and tested
2. Organization and data-structures are studied to ensure they can be used in a staged manner – may require "factoring"
3. Staging annotations are introduced, program is tested

However, one can consider this idealized because in practice it is useful and often necessary to iterate. This process is illustrated below. A simple power function is shown in Listing 10. Assuming that computing x^2 is a common

```
let rec power (n, x) =
match n with
0 -> 1 | n -> x * (power (n-1, x));;
```

Listing 10: Simple power function in pure OCaml

operation, one creates an alias for the power function where `n` is always 2, as shown in Listing 11. Unfortunately, every time `power2` is called, `power` is

```
let power2 = fun x -> power (2,x);;
```

Listing 11: Alias for squaring values

called twice. Ideally, the function shown in Listing 12 is executed, because it avoids function call overhead. Because this is a simple example designed

```
power2 = fun x -> 1*x*x;;
```

Listing 12: Preferred function for squaring values

to illustrate MSP, the function in Listing 10 will not have to be refactored for staging. Adding stage annotations results in Listing 13. Note that when the input is 0, the result still needs to be a code fragment containing 1 because when the result is not 0, the result is a code fragment. Using the staged function, x^2

```
let rec power (n, x) =
match n with
0 -> .<1>. | n -> .<.~x * .~(power (n-1, x))>.;;
```

Listing 13: Staged power function

becomes the function shown in Listing 14. Using this version results in less

```
let power2 = .! .<fun x -> .~(power (2,.<x>.))>.;;
```

Listing 14: Squaring function using MSP

runtime overhead, because the final code fragment produced at compile time is `.<x*x*1>.`. One can see how adding the case `1 -> .<x>.` will further reduce the number of calculations at runtime; calculating x^1 would simply result in `.<x>.`. This means that calculating x^2 would result in `.<x*x>.`, removing the redundant multiplication by one in both and subsequent cases. Arguably, `0 -> .<1>.` could be removed, but this depends on whether x^0 ever needs to be computed. If negative exponents need to be evaluated, relevant cases should be added as well.

3.3 Related Work

This chapter is by no means comprehensive, but serves to provide some context for this work.

As shown in section 3.2, there are various approaches to metaprogramming.

The Library of Efficient Data types and Algorithms (LEDA) described by Mehlhorn and Näher (1999) and the Computational Geometry Algorithms Library (CGAL) described by Fabri et al. (2000) are libraries using C++ templates to improve abstraction. Elsheikh (2010) and Carette, Elsheikh, and Smith (2011) note for both cases that while C++ templates provide genericity, maintainability is an issue due to limited abstraction mechanisms of this technique.

Blitz++ (Veldhuizen 1998) eliminates abstraction costs at runtime through metaprogramming with C++ templates. It preceded LEDA and CGAL, exploring many of the metaprogramming techniques those two use. Unfortunately, metaprogramming with C++ templates means that there are fewer guarantees. The elimination of overhead is dependent on the compiler inlining methods, something that is reportedly difficult to ensure. Many errors such as type errors and composition errors are only detected when compiling the generated code, and finding the source of these errors in the generator is quite a daunting task.

A project that uses string munging to generate code is ATLAS, presented by Whaley, Petitet, and Dongarra (2001). As noted in subsection 3.2.1, this approach guarantees nothing about the resulting code. This makes improvements and maintenance quite complicated. (In contrast, MetaOCaml guarantees that generated code is type-safe and well-formed.)

Object-oriented programming techniques typically achieve abstraction in a more conventional way, namely through dynamic binding at runtime. As Schorn (1991) and Simpson (1999) demonstrate, this can create runtime overhead due to dynamic dispatch.

Multi-stage programming using MetaOCaml addresses many of the issues encountered when using the above techniques. Multi-stage programming ensures that all generated code is well-typed at generation time, and monads combined with the finally tagless technique described by Carette, Kiselyov, and Shan (2009) allow for better maintainability and extensibility.

Program generators have been used successfully to generate program families (Larjani 2013, Carette 2006, Szymczak 2014, Elsheikh 2010 and all of the above). MetaOCaml has already shown its usefulness in this area, as illustrated by Elsheikh 2010 and Carette 2006. Haskell is also demonstrated as an able language for this purpose by Curutan (2013) and Szymczak (2014). Curutan (2013) may look similar at first glance, but uses abstract data types rather than the finally tagless approach presented by Carette, Kiselyov, and Shan (2009) which this work uses. The above examples generate implementations where the differences typically concern lower level design decisions. However, to the best of the author's knowledge, no work has been published on generating a program family where the architecture or other high-level design decisions vary. Many of the techniques for creating a generator found in Carette 2006 and Carette and Kiselyov 2011 still apply. One way to interact with generators is a DSL, as seen in Larjani 2013 and Szymczak 2014, for example.

The monad used in this thesis is identical to the one used in Carette (2006), and as mentioned in that publication, it bears resemblance to the monad from Kiselyov, Swadi, and Taha (2004) and Swadi et al. (2006).

As shown by Curutan (2013), there is a design language that is less abstract than algorithm descriptions but more abstract than implementation languages. Note that this is not just pseudo-code; this language is paradigm-agnostic. This concept is used in this thesis, both to express the functionality of each KWIC module as well as to unify the functional and imperative programming paradigms. The mapping between these two has been theorized and reasoned about, but to the best of the author's knowledge, no generator that can produce both has been implemented until now.

Chapter 4

Analysis and Design

This chapter begins by describing the goals and the approach taken to accomplish them. It then analyzes a number of the design decisions required for the KWIC program, and what the minimal interface of modules encapsulating them should expose.

4.1 Goals & Approach

This section is a more detailed description of what the aim of this thesis is than that provided in the Introduction. Subsequently, an initial approach to achieving this aim is provided.

4.1.1 Goals

The objective is to create a program generator in MetaOCaml. This generator should produce different implementations of KWIC, which embody various software architectures. For example, the generator should produce both implementations that reflect the pipe-and-filter architecture described in section 3.1.3

as well as the abstract data architecture from section 3.1.1. Adhering to a single architecture could also be optional; combining architectures should be a possibility. Furthermore, the generator would ideally also implement Parnas' idea that modularization is not visible in the final code. This means that the architecture would only be visible inside the generator, and the resulting implementation would be "spaghetti code". This result is acceptable because the code should not be maintained after being generated; an adjustment to the generator (where the architecture is visible) would be preferable.

4.1.2 Approach

The goal of generating multiple implementations reflecting various software architectures makes encoding knowledge of all desired architectures into the generator a necessity. This is best done in the form of an umbrella architecture, which contains everything the other architectures consist of. A good way to gain insight into this umbrella architecture is to create a number of implementations manually. To circumvent certain technical challenges presented by the nature of specific programming languages, multiple languages can be used for this. This is especially helpful when the the desired architectures are more easily implemented using different paradigms. (Meta)OCaml is a multi-paradigm language containing both imperative and functional paradigms, so many of the implementations would map to this language without too much trouble. The implementations and corresponding architectures should be analyzed thoroughly to identify the minimal requirements for the umbrella architecture.

Once the umbrella architecture appears to be well-understood, effort can be put into designing and implementing the generator. MetaOCaml's staging facilities have previously been employed to create generators, as noted in section 3.3. One of the most similar examples is the generator presented in the Gaussian Elimination (GE) paper by Carette (2006). Some of the base code should be reused, because it already contains many of the necessary mechanisms such as continuation-passing style (CPS) code combinators (which will be explained further below). In a nutshell, CPS makes the control flow in a program explicit, especially in functional programming languages. Code combinators (presented in Carette 2006 and Carette and Kiselyov 2011)are pieces of code that combine one or more smaller code fragments into a larger one. By combining smaller fragments into larger ones, one eventually 'assembles' a complete implementation. Once familiar with the reused base code and MetaOCaml itself, a very iterative approach should be taken to creating a program generator. Functions should be tested frequently to keep track of intermediate progress. OUnit2 appears to be a commonly used unit-testing framework for OCaml, and its resemblance to Java's JUnit implies a minor learning curve.

4.2 Initial Analysis of KWIC

Architectures presented by Garlan and Shaw (1994) do not introduce any new modules beyond those introduced by Parnas (1972). Pipe-and-filter takes away the explicit concept of storage, forcing all the data to be passed along between modules instead of being held in one or more dedicated modules. This could potentially be worked around by creating a passthrough storage module or something to that effect. Parnas' second decomposition contains a shifter module that presents the sorting module with a interface allowing for access to any of the shifts instead of putting them into either the original or a separate storage module, essentially acting as a more complicated passthrough. While this would potentially reduce reusability, one can explore 'connector' modules (Larjani 2013) and different sets of functor signatures that ensure various valid combinations in order to mitigate this problem. For these reasons, every one of Parnas' modules is considered in this analysis.

4.3 Storage Module(s)

After creating a number of implementations of KWIC in various languages as well as analyzing their architectures, KWIC (specifically the storage module) was considered from the perspective of each of its modules. Parnas admitted that his preferred decomposition was not perfect yet; the goal here was to apply information hiding more thoroughly.

When considering storage, the elements stored are one of the primary concerns. Parnas' smallest unit in KWIC was a character. Characters made up words, words made up lines and lines filled the data structure the preferred modularization hides from other modules. Having characters explicitly be the smallest unit would primarily add complexity and perhaps expose more internals that are preferably hidden, because the smallest unit that is manipulated by any module is a word. While a word consisting of characters makes sense when considering the function of KWIC (creating an index that is easy to search), it is still a design decision. Also, the way these characters are stored has to be agreed upon by all modules in the way Parnas envisioned KWIC. There is no need to expose more than the minimum required regarding lines; they can be ordered, but the individual characters (if they are indeed that) and the form they are stored in are irrelevant to all but the functions that add or output lines. It appears sufficient to make words the smallest unit exposed to other modules in the program.

The input module only needs a storage module to provide an add/insert function, and a storage only needs to hold lines at this point. It is assumed the order of the words in the lines has meaning, but this is not necessary when viewed from the perspective of the input module. It is also unnecessary for the shifter module, which needs to rotate lines. A conventional way to achieve this is to retrieve one line at a time, and while it typically writes into a new storage (or creates a table of indices according to Parnas), a single heterogeneous storage might be considered – the distinction between shifts and original lines that is visible to the shifter module is all that is ultimately required for the latter option. Parnas mentions that shifter produces shifts in some order, which is not strictly necessary, given that the shifts will be reordered by the sorting module. However, this order could be alphabetical, meaning that not allowing for a decomposition in which sorter is empty is a design error. Another piece of information that the shifter requires is the format of a line. One can easily infer that this could also be abstracted away; possibly even by simply reusing a storage module. Hypothetically, this could be accomplished by inserting a storage container into another one, where the former represents a line and the latter is the container that holds lines.

The order of lines in a storage is only significant to the sorting module, and if storage containers are reused to hold words of a line, then there as well due to the need for shifting. Therefore, some (but not necessarily all) storage modules will need to maintain an ordering. For this, as well as shifting lines, the concept of indices seems necessary. The distinction between shifts and originals may not matter, unless required by output (e.g. when printing the keyword and then the original line, if so desired). However, this could matter internally, when using the same storage module for original lines and shifts, in which case this would ideally be contained in the shifting module.

With only a retrieval and insertion function for a storage module, the sorting module (and possibly the shifter) would end up performing common tasks such as swapping. It would be better if this were a function provided by the storage module, both for abstraction and performance reasons. Bringing the sorting module 'closer' to the storage module is only possible if there is a storage (e.g. not pipe and filter). The sorter would be the only module to swap lines, unless the shifter uses this to rotate lines. However, this would be inefficient similar to how swapping using a retrieval and insertion function would be. A rotation function could also be provided by the storage module instead. This would effectively move the design decision specifying the method through which rotation is performed to the storage module, but it would potentially improve performance as well as reduce the need for retrieval and insertion functions. Output intuitively also needs a retrieval function to pretty-print a storage (though it could print a desired shift according to Parnas, presumably working with indices from the sorter).

4.3.1 Conclusion: Multiple Options

Ultimately, there could be three storages. The first primarily for input, just containing lines with only an insertion function for input and a retrieval function for the shifting module. The second storage would be for the shifting module, storing shifts. The shifting module still only require this storage to have insertion and retrieval functionality. The third storage is for the sorting module, containing shifts in a certain order. A function that added new lines would work if all data was available at once and add added lines sequentially, but a modification function would be necessary if that was not the case. Output needs the third storage to have a retrieval function as well, but if the pretty-printing requirement is relaxed it might just request the contents of the entire storage instead. The first storage could be combined with the second if there was a way to differentiate shifts from original lines. The second and third could be combined, but the order is only necessary after sorting; this would also require a swap function (combining retrieval and modification).

4.4 Sorting Module

Instead of calling it an alphabetizing module, which implies the assumption that it only sorts words in alphabetical order, it is more appropriate to call it a sorting module. When considering KWIC and other concordances, these assumptions made by Parnas (1972) make sense. However, from a software perspective, what a word exactly happens to be is just another design decision, and one that is preferably contained in a module rather than spread throughout the program (as seen in /java-kwic/[1] for example). This module should only contain the algorithm by which the contents of a storage module is sorted.

[1]Hyperlink to the directory in question. For convenience, all code is available in an archive at `https://www.cas.mcmaster.ca/~schaapal/mthesis/code.tar.gz`

4.5 Shifting Module

As stated above this module essentially only needs to create rotated lines. But lines in this context only need to be containers that hold elements in a certain order. All other details should be hidden from this module to make it as generic (and therefore reusable) as possible.

4.6 Input and Output Modules

The Input module contains knowledge of the input format and how this should be encoded in the storage module. Similarly, the output module only needs to be able to present the contents of the (final) storage module in some form.

4.7 Combining Modules Into KWIC

Based on the work by Carette (2006) and Larjani (2013), KWIC modules should correspond to parametrized OCaml modules, or *functors*. These take modules as parameters as well as be passed as arguments to other modules. For example, the sorting module takes a storage module as an argument, which in turn takes as a parameter a module that specifies the type of the values being stored. This way, modules can be instantiated and combined into a complete KWIC implementation. Ultimately, this could be done through a simple user-friendly DSL, but the OCaml module language is already a DSL on its own.

4.8 Architectures

The idea is that while a number of modules may be architecture-specific, a lot should be reusable. This would allow mixing and matching to create whatever software architecture is specified as input to the generator. The signatures prevent combinations that do not produce a working implementation. The combinations of modules range from the one generating the implementation reflecting the most complicated architecture to the one that inlines everything so that the simplest implementation (that might not reflect any architecture) is produced. By presenting the user a number of choices in the beginning, they should be able to easily generate an implementation. By doing so, the generator should demonstrate that software architecture is not an unchangeable design decision early in the software development process.

4.9 Incrementality

The stream-like nature of the pipe-and-filter architecture implies the possibility for incremental input and results. To this end, earlier implementations created separate groups of shifts and sorted those. However, all lines and shifts in KWIC must be sorted together. This means that in the pipe-and-filter architecture, all input must be provided at once.

4.10 Subsequent Analysis

While the initial analysis was a good start, some of the details that became apparent through iterative implementation changed. The storage module was developed with sorting in mind, and the use of indices was deemed undesirable.

Parnas used indices because these were the appropriate solution at the time. However, iterators and traversal functions have made their debut since then. These hide more of the data structure than indices and therefore became the preferred approach.

Another way for the shifting module to achieve its goal is to have the storage module provide a function that returns a shifted version of a line. The shifter module then calls this function and stores the results in the appropriate storage module (this could be the same one or a different one). This approach hides the structure of a line, only taking one line as input and producing one as output, whereas the original approach required the shifter and storage modules to agree on some format for lines that included distinguishable tokens in some order. While this seems logical when expecting to shift lines, the decision of what this specifically entails is kept within the storage module when using the latter approach.

4.11 Distinction Between Containers, Elements, and Their Respective Descriptions

There is a storage module that contains functions to manipulate a container for elements, whether it be an array or a list, for example. However, there is also a notion of element descriptors; in the case of an array, this maps to indices. For a list, this is not as clear; typically, one iterates over a list recursively, so a likely choice is to make an element descriptor the same as the element itself. One can access an element from its descriptor; in the case of lists, only the identity function is needed, but in the case of arrays, one would get the element at the provided index. Note that one cannot necessarily use the element descriptor to access another element; while indices can be incremented, a list element would have to provide access to the next element. This is justified because this functionality would imply the elements have an order. The concept of a container descriptor is slightly different; it describes (a portion of) a container. Some sorting algorithms recurse on parts of the original data structure. In the case of lists, it's simply a list, but for arrays it is a tuple of the array and the first and last indices of the portion of the array it describes.

Chapter 5

Implementation

This chapter will discuss the efforts made towards creating a program generator in MetaOCaml which would produce KWIC implementations. Some concepts and code were reused from the GE paper (Carette 2006), which will be explained first. Concluding this chapter is a brief overview of the final implementation, highlighting some noteworthy details, along with difficulties encountered.

This chapter references paths such as /ge-based-kwic/ a number of times; these are both hyperlinks to the relevant file or directory and paths in the accompanying archive file found at `https://www.cas.mcmaster.ca/~schaapal/mthesis/code.tar.gz`.

5.1 Concepts

First, a number of concepts carried over from the GE code (Carette 2006) will be explained. Some familiarity with MetaOCaml and functional programming in general is necessary at this point; see section 3.2 for a brief overview and Taha (2004) for more detail. For OCaml itself, many textbooks exist; Minsky, Madhavapeddy, and Hickey (2013) proved helpful to the author on numerous occasions.

5.1.1 Making the Type of the Generated Code (Fragment) Abstract

As explained in the preliminary section 3.2, MetaOCaml manipulates code fragments of type `'a code`, where the parameter `'a` can be anything. The GE code 'hides' the `'a code` type behind `'b abstract` in the fashion of finally tagless (Carette, Kiselyov, and Shan 2009). This allows for multiple 'backends' to the generator which can produce different output based on the same input. The interface for such backends is the /ge-based-kwic/coderep.mli interface. Two backends are implemented. For one, `'b abstract = 'b code`, and it produces code fragments (/ge-based-kwic/code.ml). This is the backend that makes the program a generator. For the other, `'b abstract = unit -> 'b`. This executes the code that would be generated, simulating lazy evaluation through 'thunks' (`fun () -> ...`). This does not delay execution in the same way MetaOCaml's code fragments do (i.e. until `Runcode.run` is called); instead it delays execution until the current 'fragment' is combined into another one. However, the computed values should be the same, and the latter can be implemented in plain OCaml. Apart from greater separation between the code being generated and the generation process, this also aids in debugging when one of these backends produces errors. Most importantly, allowing the generation of code as well as an immediately executed implementation takes away the burden of showing that both are equivalent, as would be the case when manually writing what one wants to generate and then creating a generator for it. Updating both concurrently is a most time-consuming experience due to the difficulty of achieving exact equivalence. Whereas the former will often be referred to as code, the latter will be referred to as the 'direct' approach. Code will be the main focus.

5.1.2 Continuation-Passing Style & State

The direct style of programming that everyone knows is ubiquitous enough for people to not realize it is named as such. However, there is an alternative called CPS. A function in continuation-passing style always takes one extra argument, which is its continuation. A continuation is just another function which takes the result of the function it is passed to as well as a subsequent continuation function. A CPS function executes its calculation and passes its result as an argument to the continuation. The simplest continuation is the identity function, which simply returns whatever is passed in as its argument.

The benefit of CPS is that it makes the control flow explicit. It thereby allows backtracking to occur in case a certain "path" of continuations ends up yielding an error. A function higher up in the chain can anticipate this and subsequently try an alternative continuation. One clear downside is that readability of the code suffers, especially for novices.

The type of a continuation would be `('v -> 'w) -> 'w`, where `'v` is the value passed in and `'w` is the answer (Larjani 2013). The second `'w` seems redundant, but its presence can be explained through a simple example: if we have a function called `one` that produces the code fragment[1] `.< 1 >.`, and this function is in CPS, then it needs a continuation as (one of) its arguments. Presuming there are no other arguments to this function, its definition would be:

[1]See subsection 3.2.4

`let one k = k .< 1 >.`. The simplest continuation would be the initial continuation, which would be the identity function as stated above; `fun k0 v = v`. Because of the type of this function, namely `'v -> 'v` (where `'w` is `'v`) `one`'s type would be `('v -> 'w) -> 'w`, because it would have to call the continuation on its result, returning a value of the return type of that continuation, in this example `k0`.

This is sufficient for generating functional programs, because all values are passed on as function arguments. However, we need a notion of state in order to generate imperative code.

In order to add state, an additional argument `s` is added. In this case, state is a list manipulated by a number of functions that will add or look up an element. Elements in this case are references to variables in the generated code; this state represents the state of the generated code. One can extend the state with a new variable, look it up in order to manipulate it, as well as modify an existing entry. (This last operation was not available in the GE code, but was added by Carette.) The state will remain empty when generating functional programs. Note that this is a global state. To declare variables in a scoped setting, a different mechanism (the `let!` and `retN` code combinators) is used.

The idea of a CPS function with global state is captured in the `monad` type:

```
type ('p,'v) monad = 's -> ('s -> 'v -> 'w) -> 'w
  constraint 'p = <state : 's; answer : 'w; ..>
```

If something is of type `monad`, it will take as arguments a state of type `'s` and a continuation of type `'s -> 'v -> 'w` and this will lead to a final result of type `'w`. This parametrized type takes two parameters; `'p` is a constraint on the state and/or the result, while `'v` is the value type resulting from this function that is passed into the continuation. Type `'w` can be the result of more CPS functions than just the next one.

Expanding our example above to accommodate for `'s`, `one`'s definition would look like `let one s k = k s .< 1 >.`. The initial continuation would be `let k0 _ v = v`, with `_` being the ignored state. Its type is `'s -> 'v -> 'w`. Piecing this together, one can surmise that the type of `one` must indeed be in the form of the type given above; `'s -> ('s -> int code -> 'w) -> 'w`, alternatively shown by the compiler as:

```
(<state : 's; answer : 'w; ..>, int code) monad
```

Two specialized versions of the `monad` type have been created during the generator's development process for convenience and readability: `cmonad` and `lm`. These will be explained below.

cmonad

For convenience, a more specific form of the `monad` type was created; it accepts and returns `abstract` values. Almost all functions within the generator will be returning code in some fashion, making this an oft-used type.

```
type ('pc,'p) cmonad_constraint = unit
  constraint
    'p = <state : 's list; answer : 'w abstract>
  constraint
    'pc = <answer : 'w; state : 's; ..>
```

```
type ('pc,'v) cmonad = ('p, 'v abstract) monad
  constraint _ = ('pc,'p) cmonad_constraint
```

The two parameters are very similar to those of `monad`. However, `'pc` is related to `'p` by `cmonad_constraint`. Note that this is only for readability; one could move the constraints from `cmonad_constraint` to `cmonad` to achieve the same result. The types of `state` and `answer` in `'pc` become more restricted in `'p`, where they become encapsulated in `list` and `abstract` respectively. Ultimately, the monad type should look like this:

```
's list -> 's list -> 'v abstract -> 'w abstract -> 'w abstract
```

lm

Another more specific parametric type called `lm` was created for convenience when dealing with an array in the state.

```
type 'x stor_constraint = unit
  constraint 'x = _ tarr
type ('pc, 'v) lm = ('pc, 'v) cmonad
  constraint _ = 'pc stor_constraint
```

Similar to the `cmonad` type, `'pc` is restricted to the polymorphic type `tarr`. This type is defined as follows:

```
type 'a tarr = 'a
  constraint 'a = < state : 'b ; .. >
  constraint 'b = [> `TArr of sstate ]
```

Which leads to `(_ tarr, 'v) cmonad`, in turn leading to `(<state: [`Tarr of sstate];..>, 'v)` resulting in the conclusion below:

```
[`TArr of sstate] list -> [`TArr of sstate] list -> 'v abstract
 ↪ -> 'w abstract -> 'w abstract
```

5.1.3 Code Combinators

A code combinator is a function that takes one or more code fragments and uses those to produce another fragment. A program generator in this case produces a program by combining code fragments into one large one. From the point of view of the generator, these functions produce results of type `abstract`, as specified by /ge-based-kwic/coderep.mli. The GE code came with a set of combinators, but a number had to be added to implement the desired functionality. A simple example would be `Idx.add`, which is defined as follows:

```
let add a b = .< .~a + .~b >.
```

It takes two fragments (which have to be of type `int code`), and splices them into a larger fragment that will add them together once executed. An equivalent set of delayed execution (as described above) combinators exist; its result type is `unit -> int`, but these are both seen as `int abstract`. Code fragments can be *monadic*, which means they are aware of global variables through state s and the generation of code is sequential because CPS is employed with continuations k. The `seq` and `seqM` combinators illustrates this nicely:

```
let seq a b = .< begin .~a ; .~b end >.

let seqM a b =
  fun s k -> k s .< begin .~(a s k0) ; .~(b s k0) end >.
```

Below are some brief explanations of the more interesting combinators.

`let!` This combinator is used as follows:

```
let! patt = expr1 in expr2
```

One can choose any name for `patt`, and `expr1` and `expr2` are of type `monad`. The above is syntactic sugar for:

```
let! expr1 (fun patt -> expr2)
```

Our definition of `let!` is:

```
let (let!) (m : ('p,'v) monad) (f : 'v -> ('p,'u) monad) :
 ↪  ('p,'u) monad = fun s k -> m s (fun s' b -> f b s' k)
```

So given an `m` of type `monad` and a function `f`, another `monad` will be produced. This will have the same constraints on state and result, but the value the continuation takes can be different. It also takes a state `s` and a continuation `k`, and finally applies the `monad` `m` to the state `s` and the newly created nameless function that takes a state `s'` and a value `b`, applying `f` to `b`, `s'` and `k`. This allows `monad` `m` to produce a value `b` and optionally modify state `s'`, which the remainder of the function (after the '`in`') embodied in `f` will then use, along with the modified state and the continuation passed to `let!`.

`genrecloop` The recursive loop combinator. It is defined as follows:

```
let genrecloop gen start = fun s k ->
  k s .<let rec loop j = .~(gen .<loop>. .<j>. s k0) in
↪  loop .~start>.
```

It takes arguments named `gen` and `rtarg` in addition to the usual CPS ones. It then creates a recursive function called `loop`, which takes `j` as its argument and executes `gen` with `loop` and `j` as its arguments, finally calling it such that `j`=`rtarg`. So a `gen` function would have to have two arguments, namely one for `loop` and one for the actual argument(s). It can then call the former to recurse in a way that terminates, and possibly use the latter to determine whether to recurse.

`ret` Defined as follows:

```
let ret (a :'v) : ('p,'v) monad = fun s k -> k s a
```

This ensures that a value of any type can be used in monadic code by calling the given continuation on the given state and the value.

`retN` Defined as follows:

```
let retN (a : 'v code) :
 (<answer: 'w code; ..>, 'v code) monad
  = fun s k -> .<let t = .~a in .~(k s .<t>.)>.
```

This might appear to do the same thing as `ret` at first glance, but it only accepts code and generates a `let` statement, making the argument explicitly named in the resulting code. This ensures that the code passed in as the argument is only executed once instead of multiple times, which his important when dealing with side-effects such as when modifying an array.

Passing On a Modified State: `seqMS`

When modifying the state, the resulting new state needs to be visible to subsequent code. For example, this happens when changing elements in an immutable data type such as `list` contained directly in the state instead of having a mutable data type such as an `array` or a reference (`ref`). In the former case, a new `list` is created every time something changes, and this `list` must replace the previous one in the state. However, in the latter two cases of `array` and `ref`, the state remains unchanged because it is the same `array` or `ref` as before, the difference being the thing that the `ref` is pointing to or the element the array now contains. The initial sequential combinator, seqM, simply passed both fragments the same state. In fact, it seems like only let! passes on the potentially modified state. After finding a temporarily satisfactory workaround in the form of an explicit continuation and numerous unsuccessful attempts as well as shelving the problem for a while, the feat of creating a true sequential combinator was finally accomplished. This combinator ultimately simply leverages let!.

```
let liftM2 f a b =
  let! v1 = a in
  let! v2 = b in
  ret (f v1 v2)

let seqMS a b = liftM2 seq a b
```

5.2 Challenges

A brief overview of some of the greater challenges encountered.

5.2.1 OUnit2 Integration With MetaOCaml and (GNU) Make

OUnit2 is a framework that aims to make unit testing easier when using OCaml. Modelled after JUnit, it allows one to write tests that include a comparison to the expected result, combine these into one or more test suites and ultimately run any number of these test suites. Its use is not complicated with plain OCaml; one simply employs `ocamlfind` to include the appropriate library. Using a Makefile with MetaOCaml is also straight-forward. However, combining the three presented its own technical challenges. An early solution can be found in /ocaml-kwic/. However, the simpler solution was to use the `OCAMLFIND_COMMANDS` environment variable, and set `ocamlc` to `metaocamlc`. This allows one to use `ocamlfind` with MetaOCaml as one would with regular OCaml. The Makefile in /ge-based-kwic/ creates test-suite binaries and subsequently executes them.

5.2.2 Debugging

When using type aliases like `cmonad`, `lm`, and so on, type mismatch errors often report a mixture of these; for example, it might expand the actual type out to the level of `StateCPSMonad.monad`, but leave the analog in the expected type as `lm`. This requires the reader to translate these on the fly, which quickly becomes a major hurdle as functions grow more complex, especially when taking monads as arguments. A question posted to the OCaml mailing list (along with a minimal traversal example) did not yield any effective way of improving this situation. Fortunately, certain errors occur frequently and become easily recognizable for those who have worked long enough with the code, but this remains a poor substitute for more readable error messages.

Another issue frequently is encountered is failing tests without an obvious reason why. OUnit2 was not designed with MetaOCaml in mind but it is generic enough to facilitate the printing of the code created for a particular test when it fails. This makes it as easy as finding bugs if unit tests fail when using regular OCaml.

5.3 Implementation Details

5.3.1 Refinement vs. Usage

A module (in the sense of decomposing KWIC) such as the storage module can have different requirements to satisfy. These can be reflected in the interface of a module, which is called a signature. For use by the input module, it only needs to hold lines. However, the sorting module also needs the lines to be comparable in a way that they can be ordered. Therefore, various OCaml signatures have been created and subsequently combined into larger ones which represent what the KWIC modules expect. In this work, two ways are used to combine signatures: using the `include` keyword, or using `functor`s that create signatures inside themselves. The former represents *refinement*; adding detail to an interface. The latter leaves the choice of `functor` argument until the module signature is used to specify a module, which embodies *usage*. The reason for this is the difference in the moment at which the signatures are combined. For refinement, it is done when the generator is being written, and changes require one to modify the generator itself. Functors on the other hand combine signatures when the modules are assembled into a KWIC implementation, which happens when the user generates the desired configuration, or when the modules are finally 'used'.

5.3.2 Encapsulating What a Word Is: The Value Module

This is the simplest part of the implementation: hiding the design decision of what exactly a word consists of. As outlined in the analysis above, this knowledge should be contained within the storage module because it is unnecessary for the functionality of the other modules. A word has at least a type. This is captured in the following signature:

```
module type Value = sig type value end
```

Because some storage implementations require their data structure to be initialized (e.g. arrays), a default value is sometimes needed. A slight variation on the previous signature captures this:

```
module type ValueWithDefault = sig
  include Value
  val default : value abstract
end
```

When one wants to sort a storage, its elements must be comparable in some way for an order to be determined. Currently, a comparison function can be part of the definition of what a word is:

```
module type ComparableValue = sig
  include Value
  val compare : value abstract -> value abstract -> (_, bool)
  ↪  cmonad
end
```

This comparison function signature specifies that it has to take two abstract values, which could be two pieces of code containing something of the `value` type, and produce a `cmonad` that supplies an abstract Boolean value to the subsequent computation. It should be noted that a separate comparison module will likely be necessary later on, in a fashion similar to the quicksort example in Listing 5. Combining element type and comparison operation is only sufficient during development for simplicity.

Combining these two additions looks as follows:

```
module type ComparableValueWithDefault = sig
  include ValueWithDefault
  include ComparableValue with type value := value
end
```

Note that this is refinement because of the `include` keyword.

An implementation often used for testing is listed below:

```
module IntV = struct
  type value = int
  let default = Idx.zero
  let compare x y = ret (Idx.less x y)
end
```

This conforms to the `ComparableWithDefault`. `Idx` is a module specified by /ge-based-kwic/coderep.mli that provides some basic integer operations and constants such as zero and one.

5.3.3 Implementing a Storage Module

A storage module stores elements, but this implied simplicity is misleading because it is by far the most complex module within most decompositions containing one or more storage modules. Parnas stated that adding a storage module to the KWIC decomposition would encapsulate design decisions related to the format in which data is stored. Not all architectures described in section 3.1 have an explicit storage module, but this is interpreted as the storage module being a passthrough module. This ensures a single interface will need to be adhered to when accessing data. Unfortunately, it is difficult to answer the question of what this interface ultimately should contain. Having a few implementations to generalize over is often helpful and can be faster than simply reasoning about it.

Interface

As stated before, the concept of interfaces maps to OCaml's notion of signatures. Logically, one would create a signature for the storage module, and then create one or more modules that conform to that signature. However, as previously noted, the distinction between the notions of refinement and usage can be reflected in the code. Therefore, instead of creating multiple overlapping signatures for each of the three cases described in the initial analysis, the end result was a number of signatures that could be combined as needed.

Basics Starting with the basics, one would intuitively define types for the container and for its elements. Unfortunately, as discussed in the analysis, there are also element descriptors; in the case of `array`, these are indices. Furthermore, one could conceive the need for description of a partial container, such as when using a sorting algorithm that embodies a "divide and conquer" strategy. Note that the type of the container and the types of the element and container descriptors are dependent on the data structure the storage module implements. However, the element type can be parametrized. We therefore end up with the following:

```
module type EltDesc = sig type eltdesc end

module type ValueContainer = sig
  type ctnr (*actual container*)
  type ctnrdesc (*description of possibly partial ctnr*)
end
```

The element type will be defined later.

There is also a need for a type that describes the internal state of the storage module. This is then stored in the resulting implementation's global state, which is described in subsection 5.1.2. In the case of a mutable data structure, this type would be equivalent to the `ctnr` type. When the state remains unused, such as when generating an implementation that is functional rather than imperative, it would be `unit`. This type is defined in the `TArr` signature in Listing 15, along with the `lm` monad type explaiend in section 5.1.2.

Creating and Populating a Container The first thing that comes to mind is populating a container. This is indeed necessary for both the input module and testing purposes. The first signature describing a storage module is

```
module type TArr = sig
  type sstate (* internal state of storage module *)
  type 'a tarr = 'a
    constraint 'a = < state : 'b ; .. >
    constraint 'b = [> `TArr of sstate abstract ]

  type 'x stor_constraint = unit
    constraint 'x = _ tarr
  type ('pc, 'v) lm = ('pc, 'v) cmonad
    constraint _ = 'pc stor_constraint
end
```

Listing 15: Definition of the TArr signature

therefore called `CreateContainer`, shown in Listing 16. This is a functor that

```
module type TArr = sig
  type sstate (* internal state of storage module *)
  type 'a tarr = 'a
    constraint 'a = < state : 'b ; .. >
    constraint 'b = [> `TArr of sstate abstract ]

  type 'x stor_constraint = unit
    constraint 'x = _ tarr
  type ('pc, 'v) lm = ('pc, 'v) cmonad
    constraint _ = 'pc stor_constraint
end
```

Listing 16: CreateContainer signature

takes a module specifying a type, which is then assigned to the element type `elt`. Note how the `EltDesc` module is not included; this signature does not specify any function that manipulates elements other than `add`, which has no need for element descriptors. Storage containers also need to be initialized; this can be as simple as returning an empty list when the data structure is a `list`. However, when using an `array`, the initialization requires a fixed number of elements, at which point the initarg type becomes important. The res type is another type dependent on whether an imperative or functional implementation is being generated; if the goal of the module is imperative, it will be unit, otherwise whatever is provided (typically ctnr). This type can would be unit for a list, but int when using an array. The fin and desc functions return the actual container and a description of the (whole) container respectively.

Using a Container In a similar fashion, a signature called UseContainer is defined in Listing 17. The absence of ValueContainer is possible due to

```
module UseContainer (V : Value) = struct
  module type Sig = sig
    include EltDesc
    include TArr
    type elt = V.value

    val access : eltdesc abstract -> (_, elt) lm
  end
end
```

Listing 17: UseContainer signature

careful definition of the eltdesc type. For a list, this is an element itself; for an array there is a state from which we can retrieve the element that the eltdesc describes. It is quite conceivable that a container must be provided in some cases in order to access elements. This is indeed the case in the RotatableContainer signature in Listing 18.

Specific Use: Rotation/Shifting This signature also defines the rotate function one would expect it to. This function is (implicitly) expected to produce a 'rotated' copy of a container by moving the first element it contains

```
dule RotatableContainer (V : Value) = struct
module type Sig = sig
  include CreateContainer(V).Sig
  include EltDesc

  val access : ctnr res abstract -> eltdesc abstract -> (_, elt)
  ↪  lm
  val rotate : ctnr res abstract -> (_, ctnr res) lm
end
d
```

Listing 18: RotatableContainer signature

to the end of the container.

Two pairs of rotation functions have been written; both for arrays and for lists, one using only minimal built-in functions and the other using them much more to implement rotation in as few lines as possible. From this, the procedure common to all four can be deduced[2].

1. Take the first element
2. Shift all other elements forward
3. Append the element from the first step

These steps are turned into three functions. For the list-based approach, the second step is the identity function because the first element is removed from the list, whereas the array needs to move all other elements. The result of this can be seen in /ge-based-kwic/separate/rotates-ml. This work did not require much adaptation; the final type became quite similar to what it already was.

```
  val rotate : ctnr res abstract -> (_, ctnr res) lm
```

[2]The design language from Curutan 2013

It takes a container, a state and a continuation (which takes a rotated container), which then produces a result.

Specific Uses of Containers: Comparing Values The `OrderableContainer` signature in Listing 19 differs from the previous ones by expecting a `ComparableValue`. The implication is that the `compareVals` function uses the `compare` function that is specified by `ComparableValue`, as well as the `access` function of `UseContainer`.

```
module OrderableContainer (V : ComparableValue) = struct
  module type Sig = sig
    include UseContainer(V).Sig

    val compareVals : elt abstract -> elt abstract -> (_, bool)
 ↪  lm
  end
end
```

Listing 19: OrderableContainer signature

Assisting Bubble Sort: Tagging Note that this notion of tag is separate from that in the context of finally tagless Carette, Kiselyov, and Shan 2009. Whereas those were a construct necessary to embed a DSL in a host language, these tags are not even necessarily tags; there is information about the data returned attached to it. Concretely, whether elements in a container have moved since the last sorting iteration in the case of bubble sort.

The `TaggableContainer` signature more interesting in that it includes another signature that is parametrized over the 'backend' being used (either code or direct).

```
module TaggableContainer (V : Value)(T : Tag) = struct
  module type Sig = sig
    include CreateContainer(V).Sig
    include Tags.TaggingA(T)(CODE).Tagged
  end
end
```

Listing 20: TaggableContainer

The reason this is parametrized extensively is the possibility for different implementations; in OCaml, one can communicate this information by using the `option` type, pairing the desired information with a Boolean value into a tuple, and through many other ways. Both ways explicitly mentioned above have been implemented and tested; see /ge-based-kwic/tags.ml, as well as /ge-based-kwic/code.ml and /ge-based-kwic/direct.ml.

Specific Use: Traversal Under the assumption that it operates on one storage, the sorting module needs a storage that provides a traversal function which retrieves and sets elements. The elements must also be comparable to put them in some order, which is made possible by including the `OrderableContainer` signature. In order to compare them, elements need to be retrieved, as specified by the `UseContainer` signature that is included in `OrderableContainer`. These three operations are all part of different signatures, which separate functions into logical groups. There is a `TraversableContainer` signature (shown in Listing 22), which presents traversal functions (as well as a function that keeps traversing until no further changes are made). This signature includes the signature which specifies a comparison function, which in turn includes a signature containing a function to retrieve elements. This is reflected in the `BubbleSortContainer` signature in Listing 23, which is only a wrapper around

```
module TaggingA (TAG : Tag)(T : Abstract.T) = struct
  module type Tagged = sig
    type 'a tag = 'a TAG.tag
    val tag : 'a T.abstract -> bool T.abstract -> 'a tag
↪ T.abstract
    val get_tag : 'a tag T.abstract -> bool T.abstract
    val process : 'a tag T.abstract ->
      ((bool T.abstract -> ('b -> ('x -> 'y -> 'y) -> unit
↪ T.abstract)) option) ->
      ('a T.abstract -> ('b -> ('c -> 'd -> 'd) -> 'e
↪ T.abstract)) ->
      (unit          -> ('b -> ('f -> 'g -> 'g) -> 'e
↪ T.abstract)) ->
                        ('b -> ('b -> 'e T.abstract -> 'h) ->
↪ 'h)
  end
end
```

Listing 21: TaggingA signature

the TraversableContainer.

After unsatisfactory attempts at adding traversal to the existing code directly, a separate implementation was created. This was focused only on unifying the paradigms, foregoing other concerns such as abstracting the container away. The result of this can be found in /ge-based-kwic/separate/traverseExercise.ml. Traversal in this case wraps a loop (whether it be recursive or iterative) around the body of that loop, which is provided as an argument. There is the possibility for data persistent across loop iterations; passing it on as an argument in the recursive case (functional) or storing it in a variable in a while-loop (imperative). The condition for either loop is fixed to ensure all elements are processed. The loop body takes the current element, the loop-persistent data and produces a new value. The traverse function assigns this new value to the current element and passes on the possibly modified loop-persistent data. This work was

```
module TraversableContainer (V : ComparableValue) (L : ValueWrapper) (T : Tag) = struct
  module type Sig = sig
    include OrderableContainer(V).Sig
    include TaggableContainer(V)(T).Sig
      with type elt := elt
      with type sstate := sstate
      with type 'a tarr := 'a tarr
      with type 'x stor_constraint := 'x stor_constraint
      with type ('pc, 'v) lm := ('pc, 'v) lm
    type loopdata = elt L.wrap
    type 'a mres (* monadic result *)
    val traverse :
      ctnrdesc abstract ->
      (eltdesc abstract ->
        loopdata abstract ->
        (<answer: ctnrdesc mres; state: [> `TArr of sstate abstract] as 'a;..>, elt *
 ↪   loopdata) cmonad) ->
      loopdata abstract ->
      (<state: 'a;..>, ctnrdesc) cmonad
    val traverseTwo :
      ctnrdesc abstract ->
      (eltdesc abstract ->
        loopdata abstract ->
        (<answer: ctnrdesc mres; state: [> `TArr of sstate abstract] as 'a;..>, elt) cmonad)
 ↪   option ->
      (eltdesc abstract ->
        eltdesc abstract ->
        loopdata abstract ->
        (<answer: ctnrdesc mres; state: 'a;..>, (elt * elt) * loopdata) cmonad) ->
      loopdata abstract ->
      (<state: 'a;..>, ctnrdesc tag) cmonad
    val iterativeconverge :
    (ctnrdesc abstract ->
    (<answer: ctnrdesc mres; state: 'a;..>, ctnrdesc tag) cmonad) ->
    ctnrdesc abstract ->
    (<state: 'a;..>, ctnrdesc) cmonad
  end
end
```

Listing 22: TraversableContainer signature

subsequently integrated with the existing code, which maintains the information-hiding concerns. The type of the `traverse` function ultimately became what is shown in Listing 24. It essentially takes a container description, a function that modifies elements, initial loop-persistent data, and the resulting `cmonad` (specifying a state and a continuation are also arguments to this function). The function that will become the loop body also returns a `cmonad`. Interestingly, the constraints on both of these must be identical. This is most likely because the state is modified in the imperative implementation, and therefore requires

```
module BubbleSortContainer (V : ComparableValue) (L :
↪  ValueWrapper) (T : Tag) = struct
  module type Sig = sig
    include TraversableContainer(V)(L)(T).Sig
  end
end
```

Listing 23: BubbleSortContainer signature

```
  val traverse :
  ctnrdesc abstract ->
  (eltdesc abstract ->
  loopdata abstract ->
  (<answer: ctnrdesc mres; state: [> `TArr of sstate abstract]
↪  as 'a;..>, value * loopdata) cmonad) ->
  loopdata abstract ->
  (<state: 'a;..>, ctnrdesc) cmonad
```

Listing 24: Function signature of `traverse`

it to be consistent across the `traverse` function.

Unfortunately, the above is not well-suited towards sorting, which often requires comparisons between two elements and may subsequently change both elements' values, as is the case with bubblesort. For this reason, a modified traversal function called `traverseTwo` was created. The chief difference is that it examines two elements at a time, and moves this 'window' by one element for every iteration. However, this adds more complexity; the original function only had to consider the case where the container was empty and the case where there was an element. This function will have to consider the cases where the list is empty, there are two elements, and there is only one element (last in the container). As shown in Listing 25, `traverseTwo` is similar to the original `traverse` function, but with two functions as arguments. Note that the first

```
val traverseTwo :
ctnrdesc abstract ->
(eltdesc abstract ->
loopdata abstract ->
(<answer: ctnrdesc mres; state: [> `TArr of sstate abstract]
↪  as 'a;..>, value) cmonad) option ->
(eltdesc abstract ->
eltdesc abstract ->
loopdata abstract ->
(<answer: ctnrdesc mres; state: 'a;..>, (value * value) *
↪  loopdata) cmonad) ->
loopdata abstract ->
(<state: 'a;..>, ctnrdesc tag) cmonad
```

Listing 25: Function signature of `traverseTwo`

function is an optional argument; one can opt to give a single element the same treatment as an empty container, namely the identity function. This is useful because the single-element case comes after that element has been considered in a two-element case along with the preceding element. It might already have been set at that time, because the two-element case loop body function returns two new values, both of which are assigned.

Another integration issue is the `loopdata` type. One might assume this can simply be defined in the storage module in a fashion similar to `eltdesc` and `ctnrdesc`. However, sorting algorithms have different uses for this data, so the type should depend on the sorting module. Currently, the storage module functors take a module as a parameter that contains a parametrized type that is used to define the `loopdata` type. Through this mechanism, sorting modules specify this type in the storage module. Storage modules provide the type of their elements `elt` to this parametrized type, due to sorting typically having to do with elements. While this seems sufficient for now, more sorting algorithm implementations would provide more insight for creating a more robust interface. Note that this parametrized type could be a record, with the parameter only being one of the fields (or even ignored completely). This allows sorting modules to have full control over the `loopdata` type, while also having option of having it contain elements of the storage they are operating on without having to know their type.

The containers for lines and for words may be the same, but do not need to be as illustrated below.

Separate from the `BubbleSortContainer` signature, a `LineContainer` has also been specified. A line is both a container and an element, because it contains words, but also is a member of a storage container. Lines are expected to be able to shift (or rotate), hence the inclusion of `RotatableContainer`. The storage container must allow for sorting, so `ComparableValue` is being included.

Types Used to Produce Both Functional and Imperative Code

As already briefly mentioned, a module which generates purely functional code should leave the state unused because the data is passed around through function arguments. Conversely, if a module generates imperative code, the state will be used and very little data will be passed around. For this reason, a few parametric types are introduced by the interface. For example, the `'a res` type described above; this is the type of what is passed around by functions such as the `init` or `add` functions. For functional implementations, this should be `'a`, but for imperative implementations, it should be `unit`. Similarly, there is a type `mres`. This type is introduced by the `TraversableContainer` signature, and is used to describe the `answer` part of the `monad` constraint. (As stated above, this is the ultimate result of the computation.) This is used in traversal functions when one of the arguments is a function that produces a monad. The monad produced by the traversal function provides the rest of the computation a value of a certain type; this is the `answer` of the monad that is the result of the function specifying the body of the loop that runs on each element. Like the `res` type, this type is either `unit` or `'a`. For functional implementations, it is `'a` because the returned value is used in the subsequent computation; in the imperative case it should be `unit` because the state is used instead.

Tuples vs. Records

Throughout development, tuples were used; however, the code they generate proved to be suboptimal. Every time a part of a tuple had to be used (often only containing two elements), the `fst` or `snd` functions had to be called, the result of which would have to be enclosed in parentheses before being processed further. This was mitigated partly by generating `let` statements using `retN`, but ultimately records seemed like the preferred alternative. Generated code accessing a record field would be easier to read because it would simply consist of the record name along with the field name. Unfortunately, this decision has not yet propagated throughout the entire generator code.

Implementations of the BubbleSortContainer Signature

As described above, OCaml supports both the imperative and functional paradigms, and the `StateCPSMonad` ensures they can both be used when generating code. Apart from confirming Parnas' idea that the format in which the data is stored is contained within the module, these implementations also show that a unified API between functional and imperative implementations exists.

(Even if it is not always easy to arrive at, often requiring multiple implementations and sometimes starting fresh before integrating with the existing code.) Implementations are always functors that take other modules. This allows for more generic implementations; for example, similar to the signatures, all implementations take a module specifying the type of the element they will contain. They also take a module that specifies the type of the data persistent across loop iterations as discussed above. Furthermore, two modules for the tags denoting whether a container has changed or not; one for the type of the tag, the other for the functions for creating and reading a tag. All implementations are contained in /ge-based-kwic/bubblesort.ml. Unfortunately, they span multiple pages and are therefore not included as listings in this chapter.

ListStorageG2 A functional list-based approach to creating a storage module. Initialization in this context means creating an empty list. This implementation allows for traversal and rotation of the data it contains. The list containing the actual data is passed around between functions as an argument. Most function implementations do not raise questions; the traversal and rotation implementations are longer and look more complicated (requiring a lot more debugging to get right), but they are logical consequences of the signatures described above.

ArrayStorageG An imperative array-based implementation of the storage interface. Initialization means creating an empty array of a given length. The elements will have to contain some value, so this storage module expects a module that not only specifies the element type, but also the default value for these elements. This implementation contains traverse functions, but no rotation function.

Implementation of the LineContainer Signature

ArrayLineG A simplified version of the ArrayStorageG implementation that allows for rotation. This module was a foray into the creation of the shifting module. From the perspective of the sorting module, a storage with comparable elements is sufficient. Whether these are simple elements or containers themselves is irrelevant. However, a shifting module sees lines in a storage, which means there must be some container within a storage to hold the words for individual lines. It has a different function signature for retrieving elements because it is not intended to be stored in the state; instead it is to be stored in an array that is in the state. This implies it has to be passed around as a function argument, which required changes to the element retrieval function. Other than that, it adheres to the `RotatableContainer` and `ComparableValue` signatures.

ListStorageG2 While not fully adherent to the `LineContainer` signature, this module contains comparison and rotation functions. The primary issue at this point is the signature of the `access` function, which currently differs between `RotatableContainer` and `UseContainer` due to the technical limitations encountered during the implementation of `ArrayLineG`.

5.3.4 Sorting Module

As already mentioned above, the algorithm of choice is bubble sort. The primary motivation for this is its simplicity when implementing it. Because of the `traverseTwo` function, the sorting implementation in Listing 26 is very straightforward. It's only a call to `traverseTwo` with only a loop body for

```
module BubbleSortG3 (V:ComparableValue) (T:Tag)(Storage :
↪   BubbleSortContainer(V)(ValOpt)(T).Sig) = struct
  let bsiter arr =
    let loopbody2 eltdesc nextdesc _ =
      let! curr = Storage.access eltdesc in
      let! next = Storage.access nextdesc in
      let! unordered = Storage.compareVals curr next in
      ret (Tuple.tup2 (ifL unordered (Tuple.tup2 next curr)
↪   (Tuple.tup2 curr next)) Maybe.none) in
    Storage.traverseTwo arr None loopbody2 Maybe.none

  let bubblesort ctnr = Storage.iterativeconverge bsiter ctnr
end
```

Listing 26: BubbleSortG3 implementation

the two element case, and the the traversal is repeated until no elements are swapped anymore. (Also see /ge-based-kwic/bubblesort.ml).

5.3.5 Summary

This chapter has shown that a significant portion of the generator has been implemented when considering the six modules Parnas (1972) specified. New architectures did not specify any new ones, as mentioned in chapter 4. Signatures and implementations have been created for the storage and sorting modules. Progress has been made towards a shifting module. Input and output modules as well as the master control module are trivial in comparison. Furthermore, a lot of functionality has been moved to the storage module. Sorting makes use of a traversal function that is part of the storage module, as would the shifting module through a function to rotate lines. To combine modules into a KWIC implementation, the modules can be specified using OCaml's module language.

it would be similar to calling a function, only calling a functor that will result in a module when it is provided with other modules as arguments.S

Numerous design decisions have been encapsulated. For example, storage element type has been moved into a separate module, and everything else is parametrized to deal with any type this module provides. The storage module not only hides the container as Parnas (1972) proposed, but also hides how a line is rotated and how traversal takes place. Traversal is used instead of retrieval and insertion functions because it hides more about the storage module's internal data structure; for example, the notion of indices does not leak out to other modules. While hiding retrieval and insertion functions behind traversal and iteration functions as well as moving the line rotation function to the storage module might seem like countering information hiding, these decisions hide more of the data structure encapsulated in the storage module.

Chapter 6

Conclusions & Future Work

This work challenges some commonly held perceptions on software engineering. It is a first step towards showing that high-level design decisions such as software architecture can be parametrized through generative programming. While software architecture is often perceived to be a design decision that requires developers to start from scratch if changed, this work shows that it is not as unchangeable as typically assumed. It also shows that functional and imperative paradigms are not as detached from each other as is often thought. While they map to each other in theory, this work shows that a single generator can produce implementations in both paradigms. This is done by demonstrating a unified API within the generator, which then can generate code using either paradigm. The above has been illustrated using the KWIC program, which has been analyzed further. This work shows various ways in which Parnas' KWIC example decomposition can be improved to hide more and more design decisions. An umbrella architecture for KWIC has been explored to reflect this. Furthermore, techniques that were not available at the time Parnas 1972 was published such as traversal of containers are shown to hide information better.

6.1 Future Work

The generator presented in this work should be completed to better show the variability in software architecture MSP enables. Another interesting avenue to explore is VanHilst and Notkin 1996, which goes into how module internals should be designed; this would be slightly different from Parnas' high-level decomposition. It would be exceedingly useful if the error messages produced by the OCaml compiler was more aware of type aliasing, and consequently provided more consistent error messages in that regard. A long-term objective would be to expand the collection of paradigms a single generator can produce code in. The current limitation is the number of paradigms OCaml supports. Applying this work to a real-world example would demonstrate the practicality of this work. Formal verification and proving the correctness of the generated code would also be an interesting avenue to pursue.

Appendices

Appendix A

Implementing Specific Architectures

A short overview of preliminary work consisting of implementing KWIC in various languages using various software architectures, progressively encapsulating more design decisions.

Like chapter 5, paths such as /java-kwic/ will be referred to at times; these are directories or files provided in the accompanying archive file found at `https://www.cas.mcmaster.ca/~schaapal/mthesis/code.tar.gz`. The paths are also valid hyperlinks which will open the specific file or directory in a browser.

A.1 First Steps

One of the first things done after reading Parnas 1972 was creating implementations of the two KWIC software architectures described. Having created implementations would strengthen understanding of the architectures and provide a starting point in terms of having architectures to generate. This was done in Haskell because this was judged to be a good language for achieving the ultimate goal of writing a generator that generates multiple architectures. Implementing architectures gives a concrete goal to work towards when creating a generator, since it provides a working instance of the program to be generated. Properly implementing the architecture ensured it was a learning experience on the programming side as well.

```haskell
-- First version of KWIC as seen in Parnas' "On the Criteria To
↪  Be Used in Decomposing Systems into Modules"

-- helper function
rotate (x:xs) = xs ++ [x]

-- use in GHCi
kwic xxs = [Data.List.sort $ take (length xs) $ iterate rotate
↪  xs | xs <- xxs ]
```

Listing 27: KWIC, the Haskell way (a pipe-and-filter software architecture)

While it is easy enough to get something working, it is considerably more difficult to create code that does the expected task exactly as Parnas specified. The intuitive step is to divide the program into functions (since this is a functional programming language after all), and to take advantage of built-in constructs and libraries. This resulted in the code seen in Listing 27. Note that this is not exactly what Parnas specified; its result will be a list of sorted lists, one sub-list for each line and all its rotated permutations. However, this deviance allows for incrementally provided input; the way Parnas described KWIC implicitly assumes all input is provided once, since sorting takes all lines and rotations into account. This is especially unfortunate in the case of a pipe-and-filter architecture because the output would wait on the sorting call, which depends on all input being processed, negating one of the greater benefits of this architecture.

```haskell
import Data.List

-- helper functions
rotate :: [a] -> [a]
rotate [] = []
rotate (x:xs) = xs ++ [x]

merge :: [[a]] -> [a]
merge [] = []
merge ([]:xxs) = merge xxs
merge ((x:xs):xxs) = x:(merge ((xs):xxs))

-- use in GHCi
kwic :: Ord a => [[a]] -> [[a]]
kwic xxs = sort $ merge [take (length xs) $ iterate rotate xs |
 ↪  xs <- xxs ]
```

Listing 28: KWIC (as specified by Parnas), the Haskell way (a pipe-and-filter software architecture)

As seen in Listing 28, adding a merge function to combine all the sub-lists of lines with their respective rotated versions into one before sorting provides the expected result.

However, Parnas assumes mutable state. This is understandable given the publication date, but it is not as easy to achieve in Haskell as it is in other (presumably more imperative) languages. Haskell strives to be a pure functional language, encapsulating everything that produces side-effects in monads. Therefore, one of the first challenges was becoming familiar with and using a mutable data structure called MArray. With it, many other challenges arose; understanding monads, kinds, and more.

A.2 Dataflow-style KWIC

In order to write a version of KWIC that is closer to Parnas' flowchart-derived (less desirable) modularization, every module became its own file, manipulating a mutable data structure (an MArray in this case). Unlike above, input needs to be handled appropriately instead of supposedly being passed into the program by external means (such as the programmer supplying it as arguments when calling the main function). The result can be viewed in /dataflow-kwic/.

A.3 A First Attempt at Information-Hiding-style KWIC

Using typeclasses, functions are specified in a very general way. This can be seen as analogous to interfaces in Java for example. These functions can then be implemented in many ways, at which point design decisions are made. For example, from the typeclasses it is not immediately apparent that an MArray is or could be used. When making an instance of one of these typeclasses, one may decide to use an MArray as shown in /hiding-kwic/, or one may choose to use another data structure that fits the type signature. Some modules show implementations using a simple list instead. Also note the Line typeclass and corresponding instance, signifying that this too has become something that can change – another design choice. The use of four-letter words as the testing input is purely historical; the implementation handles words of arbitrary length.

A.4 Foray Into Phantom Types

Phantom types allow the type system to hold more information, and appeared to be useful to hold the state of the line storage, such as 'Ready', 'Shifted', and 'Sorted'. However, while this was achievable when using a simple list as the container for lines and words, it became apparent that combining this with typeclasses and the MArray or MVar mutable (and therefore monadic) data types posed significant type system challenges. It is for this reason that a temporary switch to Java was made. The results of this attempt can be found in /phantom-kwic/.

A.5 Information Hiding KWIC in Java

A more careful decomposition of KWIC has been implemented in Java. Haskell's typeclasses map to Java's interfaces and abstract classes for this purpose. Multiple interchangeable implementations for various modules have been created. This can be seen in /java-kwic/ and /java-ii-kwic/.

A.6 Return to Haskell

The progress in Java was ported to Haskell. However, mutability proved to be a stumbling block once again. The partial results of this effort can be found in /haskell-from-java-kwic/. Due to the complexity of mutability in Haskell, MetaOCaml became the new language of choice for the generator. A first attempt is listed in /ocaml-kwic/. The final result can be found in /ge-based-kwic.

Bibliography

Carette, Jacques (2006). “Gaussian Elimination: A case study in efficient genericity with MetaOCaml”. In: *Science of Computer Programming.* Special Issue on the First MetaOCaml Workshop 2004 62.1, pp. 3–24. ISSN: 0167-6423. URL: `http://www.sciencedirect.com/science/article/pii/S0167642306000712` (visited on 11/09/2015) (cit. on pp. 4, 34, 38, 43, 47).

Carette, Jacques, Mustafa Elsheikh, and Spencer Smith (2011). “A Generative Geometric Kernel”. In: *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation.* PEPM ’11. New York, NY, USA: ACM, pp. 53–62. ISBN: 978-1-4503-0485-6. URL: `http://doi.acm.org/10.1145/1929501.1929510` (visited on 09/13/2016) (cit. on p. 33).

Carette, Jacques and Oleg Kiselyov (2011). “Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code”. In: *Science of Computer Programming.* Special Issue on Generative Programming and Component Engineering (Selected Papers from GPCE 2004/2005) 76.5, pp. 349–375. ISSN: 0167-6423. URL: `http://www.sciencedirect.com/science/article/pii/S016764230800110X` (visited on 09/07/2016) (cit. on pp. 4, 34, 38).

Carette, Jacques, Oleg Kiselyov, and Chung-Chieh Shan (2009). "Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages". In: *Journal of Functional Programming* 19.5, pp. 509–543. ISSN: 1469-7653. URL: `http://journals.cambridge.org/article_S0956796809007205` (visited on 09/02/2016) (cit. on pp. 4, 34, 48, 67).

Curutan, Bianca (2013). "CPCG: A Cross-Paradigm Code Generator". MA thesis. McMaster University (cit. on pp. 3, 4, 34, 35, 66).

Czarnecki, Krzysztof and Ulrich W. Eisenecker (2000). *Generative Programming* (cit. on p. 9).

Elsheikh, Mustafa (2010). "A Generative Approach to Meshing Geometry". MA thesis. McMaster University. 116 pp. URL: `http://macsphere.mcmaster.ca/handle/11375/9913` (visited on 06/09/2016) (cit. on pp. 33, 34).

Fabri, Andreas et al. (2000). "On the design of CGAL a computational geometry algorithms library". In: *Softw. Pract. Exper.* 30.11, pp. 1167–1202. ISSN: 0038-0644 (cit. on p. 33).

Garlan, David and Mary Shaw (1994). *An Introduction to Software Architecture.* Pittsburgh, PA, USA: Carnegie Mellon University (cit. on pp. 1, 12–14, 17–24, 38).

Kiselyov, Oleg (2015). *BER MetaOCaml.* Version N102. URL: `http://okmij.org/ftp/ML/MetaOCaml.html` (visited on 08/06/2016) (cit. on p. 28).

Kiselyov, Oleg, Kedar N. Swadi, and Walid Taha (2004). "A methodology for generating verified combinatorial circuits". In: *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software.* Pisa, Italy: ACM, pp. 249–258. ISBN: 1-58113-860-1 (cit. on p. 35).

Larjani, Pouya (2013). "Software Specialization as Applied to Computational Algebra". PhD thesis. McMaster University. URL: `http://macsphere.mcmaster.ca/handle/11375/13006` (visited on 01/24/2016) (cit. on pp. 29, 34, 39, 43, 49).

Mehlhorn, Kurt and Stefan Näher (1999). *LEDA: a platform for combinatorial and geometric computing.* New York, NY, USA: Cambridge University Press. ISBN: 0-521-56329-1 (cit. on p. 33).

Minsky, Yaron, Anil Madhavapeddy, and Jason Hickey (2013). *Real World OCaml: Functional Programming for the Masses.* Google-Books-ID: nab-tAQAAQBAJ. "O'Reilly Media, Inc." 509 pp. ISBN: 978-1-4493-2476-6 (cit. on p. 47).

Parnas, David Lorge (1972). "On the Criteria to Be Used in Decomposing Systems into Modules". In: *Commun. ACM* 15.12, pp. 1053–1058. ISSN: 0001-0782. URL: `http://doi.acm.org/10.1145/361598.361623` (visited on 07/20/2015) (cit. on pp. 1, 2, 4, 7, 9, 12, 21, 22, 38, 42, 77–79, 82).

— (1976). "On the Design and Development of Program Families". In: *IEEE Transactions on Software Engineering* SE-2.1, pp. 1–9. ISSN: 0098-5589. URL: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1702332` (cit. on p. 9).

Schorn, Peter (1991). "Implementing the XYZ GeoBench: A Programming Environment for Geometric Algorithms". In: *CG '91: Proceedings of the International Workshop on Computational Geometry =- Methods, Algorithms and Applications.* London, UK: Springer-Verlag, pp. 187–202. ISBN: 3-540-54891-2 (cit. on p. 34).

Simpson, R. Bruce (1999). “Isolating Geometry in Mesh Programming”. In: *Proc. of the 8th Int'l Meshing Roundtable*. South Lake Tahoe, California, pp. 45–54 (cit. on p. 34).

Swadi, Kedar N. et al. (2006). “A monadic approach for avoiding code duplication when staging memoized functions”. In: *PEPM*. Ed. by John Hatcliff and Frank Tip. ACM, pp. 160–169. ISBN: 1-59593-196-1 (cit. on p. 35).

Szymczak, Daniel (2014). “Generating Learning Algorithms: Hidden Markov Models as a Case Study”. MA thesis. McMaster University. URL: `http://macsphere.mcmaster.ca/handle/11375/14101` (visited on 06/09/2016) (cit. on p. 34).

Taha, Walid (2004). “A Gentle Introduction to Multi-stage Programming”. In: *Domain-Specific Program Generation*. Ed. by Christian Lengauer et al. Lecture Notes in Computer Science 3016. DOI: 10.1007/978-3-540-25935-0_3. Springer Berlin Heidelberg, pp. 30–50. ISBN: 978-3-540-22119-7 978-3-540-25935-0. URL: `http://link.springer.com/chapter/10.1007/978-3-540-25935-0_3` (visited on 01/29/2016) (cit. on pp. xi, 28, 30, 31, 47).

VanHilst, Michael and David Notkin (1996). “Decoupling Change from Design”. In: *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*. SIGSOFT '96. New York, NY, USA: ACM, pp. 58–69. ISBN: 978-0-89791-797-1. URL: `http://doi.acm.org/10.1145/239098.239109` (visited on 07/20/2015) (cit. on p. 80).

Veldhuizen, Todd L. (1998). “Arrays in Blitz++”. In: *ISCOPE '98: Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*. London, UK: Springer-Verlag, pp. 223–230. ISBN: 3-540-65387-2 (cit. on p. 33).

van Vliet, Hans (2000). *Software Engineering: Principles and Practice*. Second Edition. Wiley (cit. on pp. 1, 8, 19–24).

Whaley, R. Clint, Antoine Petitet, and Jack J. Dongarra (2001). "Automated Empirical Optimization of Software and the ATLAS Project". In: *Parallel Computing* 27.1–2, pp. 3–35 (cit. on p. 33).