Framework for Composition of Domain Specific Languages and the Effect of Composition on Re-use of Translation Rules

Ludvig Kihlman

A thesis submitted for the degree of **Doctor of Philosophy**

School of Computer Science and Electronic Engineering (CSEE)

University of Essex

August 2021

Abstract

Domain-Specific Languages (DSLs) are programming languages that have been designed to be used to solve problems in a specific domain. They provide constructs that are high-level and domain-specific to make it easier to implement solutions in the given domain. They frequently also limit the language to the domain, avoiding general purpose constructs.

One of the main reasons for using a DSL is to reduce the amount of work required for implementing new programs. To make the use of DSLs feasible, the cost of developing a new DSL for a domain has to be less than the total amount of cost saved by having the DSL. Thus, reducing the cost of developing new DSLs means that introducing DSLs becomes feasible in more situations. One way of reducing costs is to use composition techniques, where new languages are created from existing ones. This includes defining new language constructs in terms of existing ones, combining the constructs from one or more existing languages, and redefining existing constructs.

We present a framework for composing languages on the abstract level and discuss to which degree one can ensure that languages produced by the composition language are valid. In particular, we look at how translation rules for translating from a composed language to a General Purpose Programming Language (GPL) are affected by the composition. That is, to which degree can a language composed from other languages reuse the translation rules of the languages it is composed from. We use a patience game suite as a case-study to show how our composition techniques can be used and demonstrate the short-comings of the techniques. We also show how a tool for composing languages can be created using DSLs produced by composition. The implementations are all in Java.

Contents

1 Introduction **1**

1.1 What is a DSL? 3

1.2 Research 5

1.3 Thesis Outline 7

2 Composition and translation of Domain-Specific Languages **9**

2.1 Introduction 9

2.2 Domains 9

2.3 Languages 12

2.4 Domain-Specific Languages 14

2.4.1 Composition 23

2.4.2 Evaluation 25

2.4.3 Other DSL concerns 28

2.5 Related areas 29

2.5.1 Ontologies 29

2.5.2 Product Families 29

3 Patience Games **31**

3.1 Introduction 31

3.2 Patience 34

3.3 General Design of Patience Suite 35

3.3.1 Software Product Lines and Domains 35
3.3.2 Implementation of Patience Suite 37
3.3.3 Existing Code-bases 37
3.4 Use of Model 43
3.4.1 Host Languages 43
3.4.2 Patience Language 44
3.4.3 Language Expressiveness 47
3.4.4 Language Composition 48
3.4.5 Language Analysis 49
3.4.6 Language Evaluation 50
3.5 Conclusions 51

4 Basic framework for language composition **52**

4.1 Introduction 52
4.1.1 Motivating example 56
4.2 Background 57
4.3 Domain and language 59
4.4 Abstract Language 60
4.4.1 Composition of abstract language 65
4.4.2 Translation 73
4.4.3 Individual composition operations effect on reusability of translation rules 79
4.5 Examples 81
4.5.1 Introduction of constructs 82
4.5.2 Relaxing and restricting 83
4.5.3 Combining and deleting 85
4.5.4 Translation 87
4.6 Limitations and solutions 88

4.7 Conclusion 90

5 Composing Patience Games 91

5.1 Introduction 91

5.2 Background 94

5.3 Patience 95

5.4 Patience Implementation 96

5.5 Abstract Language Model 101

5.6 The language 102

5.6.1 Card Games 105

5.6.2 Cards 106

5.6.3 Games 106

5.7 Translation to Java 108

5.8 Translation to DSL 109

5.8.1 Relation with existing code-base 110

5.9 Discussion 112

6 A Self-Describing Domain-Specific Language for Translations 114

6.1 Introduction 114

6.2 Method 120

6.2.1 Construction of S and H 121

6.2.2 The translation language 124

6.3 Implementation of a translation language 128

6.3.1 Informally checking correctness of translation DSL 130

6.4 Example translation rules 133

6.4.1 Reuse in the construction of T_{TH} 141

6.5 Results and discussion 144

6.6 Conclusion 148

7 Comparison to the State of the Art 150

7.1 Introduction 150

7.2 Background 150

7.3 Comparison 152

7.4 Conclusion 154

8 Conclusion and future work 155

8.1 Introduction 155

8.2 Big picture 155

8.3 Contributions 157

8.4 Shortcomings 158

8.5 Future work 161

References 163

List of Tables

2.1 DSL development phases and patterns according to (Mernik et al., 2005) . 17

3.1 Methods that a Patience Language needs to implement in our model. 43

4.1 To which degree translation rules can be reused under various conditions . . 79

5.1 Concepts in our patience implementation 96

5.2 Methods that a Patience Language needs to implement in our model. 97

5.3 Number of ASTs and rules needed for various methods of translation to GPL. 108

List of Figures

2.1 A very general view of an interpreter, as given in (Aho et al., 2006) 13

2.2 A more detailed view of an interpreter 13

3.1 UML diagram describing the Strategy Pattern 36

3.2 Introducing an interpreter for specifying Concrete Strategies 37

3.3 Feature Diagram for the patience domain based on PySolFC's patience game wizard 38

3.4 UML diagram describing the main parts of a simple patience suite. Some simplifications have been made to better fit the diagram. 39

3.5 UML diagram describing the domain that is relevant for our DSL, including some example patience PF members (Klondike, Golf, and Yukon) 41

3.6 UML diagram describing an interpreted version of the patience game using a DSL. The patience domain from figure 3.5 has been replaced by a package. 42

4.1 A diagram for concrete syntax being directly mapped to semantics, along with domain types that are instantiated to domain objects that can then be referenced by the semantic domain. 53

4.2 A diagram showing concrete syntax being parsed into abstract syntax, which is then executed in the semantic domain. In this case, the concrete syntax is decoupled from the domain types, which instead are referenced to in the abstract syntax. 53

4.3 A diagram showing concrete syntax being parsed into abstract syntax which is then translated into a different abstract syntax, which is then executed in the semantic domain. The domain types also get translated between abstract syntaxes. 53

5.1 UML diagram describing the main parts of a simple patience suite. Some simplifications have been made to better fit the diagram. 97
5.2 UML diagram describing the domain that is relevant for our DSL, including some example patience PF members (Klondike, Golf, and Yukon) 99
5.3 DSL hierarchy for games, card games and patience games being translated to an inheritance hierarchy of GPL classes. 111
5.4 DSL hierarchy for games, card games and patience games being translated to independent GPL classes. 111
5.5 DSL hierarchy for games, card games and patience games being translated to a single GPL class. 113

6.1 A compiler taking a DSL as an input and producing a runnable application as output. 128
6.2 A translator-generator producing a compiler that can take application code as an input and produce a runnable application as output. 129
6.3 Three generations of generator, showing how each part of the generator affect the next generation. 131
6.4 Three generations of generator with an imaginary -1st generator that produces the 0th generator. 133

Glossary

API Application Program Interface. viii, 26, 112

AST Abstract Syntax Tree. viii, 13, 20, 59, 60, 62, 63, 75, 76, 80, 90, 92, 108, 109, 119, 125–127, 137–139, 144, 147

BNF Backus-Naur Form. viii

COTS Commercial, Off The Shelf. viii

DSD Domain-Specific Description. viii

DSDL Domain-Specific Design Language. viii

DSEL Domain-Specific Embedded Language. viii, 3, 4, 55, 56

DSIL Domain-Specific Implementation Language. viii

DSL Domain-Specific Language. i, viii, 1–9, 11, 14, 16–23, 25–30, 52, 54, 55, 57–59, 68, 81, 89, 91–94, 96, 101–110, 112, 114–117, 119, 128–133, 136–139, 141, 142, 145, 147–152, 155–161

DSM Domain-Specific Modelling. viii

DSML Domain-Specific Modelling Language. viii, 19

DSP Domain-Specific Processor. viii

DSSA Domain-Specific Software Architectures. viii, 11

DSVL Domain-Specific Visual Language. viii

EBNF Extended Backus-Naur Form. viii

eLOC effective Lines of Code. viii

FDL Feature Description Language. viii, 12

FODA Feature-Oriented Domain Analysis. viii, 11, 29

GPL General Purpose Programming Language. i, vi, viii, 1, 2, 4–6, 8, 11, 16, 18, 20, 21, 27, 58, 62, 92–94, 96, 104, 108, 112, 114–117, 125, 133, 135, 136, 138, 139, 141, 143, 147–149, 153, 155, 156, 158, 160

IDE Integrated Development Environment. viii, 156

MDE Model-Driven Engineering. viii

ODM Organization Domain Modeling. viii, 11

OOP Object-Oriented Programming. viii, 129

PF Product Family. viii, 2, 29, 30

SLE Software Language Engineering. viii, 155, 156

SPL Software Product-Line. viii, 11, 28, 29

Chapter 1

Introduction

Computers are ubiquitous, as are the programming languages used to create software for them. Programming languages come in many forms, from very low level assembly languages to high-level interpreted languages, such as Python and Ruby. General Purpose Programming Languages (GPLs) such as C/C++ and indeed assembly languages and Python and Ruby are designed to solve any problem that one would want to solve with a computer. Some specialise in efficient and close to the hardware code, such as C/C++ and others on ease of use, such as Python, but they all provide a sense of general purpose computing.

Domain-Specific Languages (DSLs) have been used for some time now as a way of improving software development process. The idea is to have a way of specifying solutions in a language that is easier to use for the particular domain one is working in. This often comes with the constraint that the language is unable or poorly equipped to describe solutions outside the domain.

The motivating example for this research is a scenario where a software development company, is developing a suite of patience games written in a GPL. As they are working on it, they realise that parts of the code are repetitive. Rather than having to keep having to write almost identical code, the software company would like to be able to define the repetitive code once and then reuse it everywhere it is needed. So the company analyses the repeating code and finds that the repetitive code can be defined as a set of methods which are

dependent on a limited set of types. They also find that each method can be implemented using only a few different constructs. They conclude that these parts would be easier to describe in a language custom-made to describe this part of the program, a DSL.

Their argument is that there are parts of the program that share a lot of similarities, but vary slightly. Therefore, they find that it would be easier to use a language that only require the user to specify the differences between the games, rather than having to use a GPL which requires everything to be specified in much more detail.

They have a design of the domain of the program (i.e. a design that specifies what cards, decks, piles and games of patience are) that they want to create a DSL for, but they need to implement the DSL. This would normally include manually writing a parser and interpreter for the language, as well as maintaining and updating that parser and interpreter should the requirement for the DSL change. Rather than doing all this work manually, the company would like to be able to create and maintain the DSL with as little manual input as possible.

There has been a lot of discussion on how to implement DSLs, such as embedding in an existing GPL (Hudak, 1997a), using Product Families (PFs) (van Deursen and Klint, 2002) ontologies (Čeh, Črepinšek, Kosar and Mernik, 2011) and even by analysing user interfaces (Bačíková, 2014). There have also been several tools developed to aid DSL construction, such as (Mernik et al., 2002; Eysholdt and Behrens, 2010; Efftinge et al., 2012). DSL engineering is also the subject of several books, such as (Fowler, 2010; Parr, 2010; Völter, 2013a; Bettini, 2013)

In this thesis we will be investigating DSL composition (Erdweg et al., 2012; Völter, 2013a) and generative programming (Czarnecky, 1998) and in particular, the interaction between the two approaches.

1.1 What is a DSL?

A DSL is a language, usually a programming language (van Deursen et al., 2000; Mernik et al., 2005) though some argue that other, non-programming languages qualify too (Wile, 2001), such as musical notation.

The description given in (Mernik et al., 2005) is “Domain-Specific Languages (DSLs) are languages tailored to specific application domains and offer users more appropriate notations and abstractions.” Note the specification of ‘application domain’, implying that being meant for writing applications are inherent to being a DSL. In contrast, (Wile, 2001) states: “Moreover, DSLs are not necessarily programming languages: they are languages tailored to express something about the solution to a problem.” and “For example, music notation constitutes a DSL for music; it is nearly irrelevant that modern computers can play the "program" represented by the music.”

(Hudak, 1997a) gives intuitions on what DSLs are, such as, “They are more concise”, “They are written more quickly”, “They are easier to maintain”, as well as “They are easier to reason about” and notes “These advantages are the same as those claimed for programs written in conventional high-level languages, so perhaps DSL’s are just very high-level languages?”. They also note “They [programs in a DSL] can be written by non-programmers”. These properties are not shown to be true, but rather stated as an opinion.

(Consel and Marlet, 1998) says “In contrast to GPLs a la Java or C++, a DSL has a narrow application scope and must be readable for domain experts.” and “In general, a domain specific textual program is a concise set of high-level declarations, focusing on what to compute, as opposed to how to compute it.”

We will in general not take a position on where the boundary of DSLs are, however, our work is done with DSLs as programming languages in mind.

Some of the issues with DSLs include the “Tower of Babel" problem (having too many different languages), slower code, and high initial costs. These are all discussed in (Hudak, 1997a), who argue for using Domain-Specific Embedded Languages (DSELs) to improve

some of these issues. DSELs are languages that use the constructs found in (some) GPL languages to create a language contained inside the GPL language, using the GPL language syntax and semantics to execute. Contrast this with External DSLs, that uses an independent interpreter or compiler to produce executable code.

They note: “In summary, the DSEL approach creates a rich infrastructure that:”

1. Allows for rapid DSL design; if nothing else, it can be viewed as a way to prototype a DSL.

2. Facilitates change, whether for experimentation, fault correction, or design evolution.

3. Provides a familiar look and feel, especially for several different DSLs embedded in the same language. In other words, it reduces the size of the Tower of Babel.

4. Facilitates reuse of syntax, semantics, implementation code, software tools, documentation, and other related artefacts.

Since then, there has been a lot of research done, in particular on reducing the initial cost of DSL development, such as: (Nystrom et al., 2003; Kosar et al., 2008; Porubän et al., 2010; Haber et al., 2015; Degueule, 2016).

One related field, in particular for External DSLs, is Generative Programming. (Consel et al., 2005) says about Generative Programming “modeling and implementing system families in such a way that a given system can be automatically generated from specification in one or more textual or graphical domain-specific languages” (paraphrasing (Czarnecki, 2005)) and ties it to DSLs: “When mapping a DSL to GPL, the higher level the DSL is, the more program generation is needed to bridge the gap with the target execution environment. ... Not surprisingly, GPL-translated programs include rather large program templates. The process of generating these templates can be quite complex, relying on various conditions, and requiring a number of instantiations by computing and inserting constants. Without any dedicated tool support, this process can be quite laborious and error-prone.”

1.2 Research

Now we will discuss the main purpose of this thesis. We are looking at creating a framework for specifying DSLs that separates the domain and language parts of DSLs. Splitting up DSLs by domain and language is relatively rare, but not unheard of. For example, in defining DSLs, (Bilitchenko et al., 2011; Borodin et al., 2015; Burgy et al., 2005) all talk about domain as something separate to the language, although not in any kind of formal manner. Many authors do not specifically talk about the domain and language separately, such as (Bravenboer and Visser, 2004; Consel et al., 2005; Diamond and Boyd, 2016; Dinkelaker et al., 2010). This does not mean that the authors do not consider there to be a distinction, they just do not discuss it in relation to their work.

Our work explore the how language composition (Erdweg et al., 2012) can be used for DSL development using abstract syntax and translation from DSL to GPL as a way of providing semantics. We will examine some of the different ways in which this has been achieved before, and then discuss how language composition can be used to create language families.

The composition techniques we will investigate are:

- introduction – introducing a construct to the language
- definition – Defining new constructs based on existing constructs
- sub-typing/restriction – create a new construct by restricting an existing construct
- super-typing/relaxation – create a new construct by loosening restrictions on an existing construct
- deletion – removing a construct from the language
- combination – combining constructs from two or more languages

These are quite different from Erdweg's (Erdweg et al., 2012) extension, restriction, unification, and self-extension, and we will argue for why we prefer our constructs in the context of producing language families.

We also show how composition of abstract languages affect the reusability of translation rules, that translate from the composed language to some existing host language. That is, we investigate to which degree translation rules for the input languages to the composition can be reused with the output language.

We will discuss abstract language and concrete language as two separate entities, to allow an abstract language to have several different concrete languages defined for it.

The way we are going to evaluate our methodology is by providing proof of concept DSLs that have been developed using our methodology. The two domains we will provide DSLs for are patience games and a meta-language for translating DSLs using our techniques.

This research was approached as follows:

Defining a composition and translation framework for DSLs. This consists of defining a set of composition operations on abstract language types, as well as defining a translation operation for translating composed language types into another language.

Present the condition under which translation rules can be reused under composition. We discuss under which operations the translation rules are guaranteed to produce valid output and when we may be able to reuse translation rules.

Verify these effects in two cases:

1. DSL Translation DSL. We show that using our composition and translation framework we can define a language for translating other DSLs into host languages. We then show how this language can define the translation rules for itself in order to make it executable.

2. Patience DSL. We define a DSL for defining Patience games. We show how the patience DSL can be composed from existing DSLs as well as GPLs. We show how much reuse is possible depending on the way the language has been composed and whether it is composed from DSLs or GPLs.

The main contribution of this thesis are:

1. **Exploring Patience Games as a toy example for DSL development.** Patience games are used throughout the thesis to provide examples of how the composition

and translation operations work. We do also consider it a good domain to use when exploring DSL development, especially composition and we have therefore gone into some depth describing the pros and cons of using it as an example domain.

2. **A formal specification of composition and translation for abstract DSLs.** Previous work on DSL composition have focused more on concrete syntax, but we argue that working on the abstract syntax has several benefits, including ease of reuse. Existing work also tends to be in the form of tools, rather than formal mathematics, which makes generalising about the work harder.

3. **Exploration of when and how reuse of translation rules is possible under our composition framework.** We show how our approach can be used to reason about when reuse of translation rules is possible and when new rules have to be created.

1.3 Thesis Outline

The thesis can roughly be broken up in three parts. First, there are two background chapters to set the scene. Then in the middle, there are three chapters defining and demonstrating the composition and translation framework, followed by two chapters of reflection on the work.

- **Chapter 2** is the first background chapter and it serves as a way of breaking down the relevant topics within the DSL research area, discussing the current state of the art in DSL research.

- **Chapter 3** argues for why the Patience game domain is a good test case for DSL engineering and describe the domain, both for the purpose of serving as an example in later discussion of DSL development and to allow it to be used in other DSL research.

- **Chapter 4** describes our framework for language composition and translations. This chapter gives formal definitions for our model of abstract languages, as well as definition of composition operations and translation rules.

- **Chapter 5** describes the composition of a DSL for patience games. We provide an example implementation based on the domain as described in chapter 3. We also show how composition and translation is affected by use of other DSLs, such as game description DSLs or use of GPLs, such as Java.

- **Chapter 6** describes in more detail the implementation of a DSL capable of creating translators for arbitrary DSLs using composition and translation. This DSL is then used to define itself and we discuss the effect of further composition on the original DSL on the translation rules defined in it.

- **Chapter 7** compares our approach to the other approaches to DSL development out there, especially DSL composition and code generation approaches. This builds on the state of the art discussed in chapter 2.

- **Chapter 8** gives the conclusion of the work, a general discussion on the benefits of this work and the shortcomings, as well as future direction of the work and interesting avenues to explore around composition and translation.

Chapter 2

Composition and translation of Domain-Specific Languages

2.1 Introduction

This chapter will give an initial intuition of what is meant with domain and language in the context of Domain-Specific Languages (DSLs) as well as implementation approaches to languages.

Let us first consider how domains and computer languages are generally defined. There are, as with DSLs, no formal definition of either (though individual programming languages are usually formally defined, but we are discussing the general idea of a language, not specific languages). The idea with this section is to give an intuition of domains and languages, not provide a definite definition.

2.2 Domains

In (Harsu, 2002), Maarit Harsu gives a couple of intuitions of what a domain is. For example, Harsu cites (Schmid, 2000), who breaks down domains in four different areas: business, collection of problems (problem domain), collection of applications (solution domain), and

area of knowledge with common terminology. Schmid also provides the following definition: “A domain is defined by the objects, operations, and relationships that domain experts perceive to be important for developing systems for a certain area of functionality.” Harsu also cites (Tracz, 1994) who defines a domain as: “A domain is defined by a set of ‘common’ problems or functions that applications in that domain can solve/do (hence the term ‘application domain’). Also, a domain is typically characterised by a common jargon or ontology for describing problems or issues that applications in it address”.

Prieto-Díaz, Rubén (1990) defines a domain as: “Domain: In a broad context it is ‘a sphere of activity or interest: field’ [Webster]. In the context of software engineering it is most often understood as an application area, a field for which software systems are developed.” and gives as examples: “... airline reservation systems, payroll systems, communication and control systems, spread sheets, numerical control.”. They also note that: “Domains can be broad like banking or narrow like arithmetic operations.” Kang et al. (1990) makes a similar point: “A domain does not necessarily have to occur at a specific level of software granularity, such as that of a system, Computer Software Component (CSC), or Computer Software Configuration Item (CSCI).” and “Rather, a domain is a more general concept which may be stretched to apply to most any potential class of systems. This class is referred to as the target domain, which may have both higher-level domains to which it belongs and sub-domains within it.”. They then gives the example “... different instances of the same type of system (such as window management systems or relational database management systems) can be grouped together to define a domain.” and notes: “In a similar way a domain of data structures could be identified which would be at a much lower level than that of entire systems, but could still constitute a target domain in its own right.”

Simos et al. (1995) definition of domain is similar to (Prieto-Díaz, Rubén, 1990): “A software domain is defined here as an abstraction that groups a set of software systems or functional areas within systems according to a domain definition shared by a community of stake-holders.”

According to the above definitions, a domain can be seen as a collection of things that experts in the field perceive as needed to develop programs for some area of functionality (which is somewhat circular, a domain is defined by the things needed for describing the domain). A domain could also be seen as a set of problems that are solved by a certain solution (again, circular).

For our purposes, we can provide our own, slightly simpler view of what a domain is. In general, a domain consist of things and relations between the things. For the domain to be coherent, all things have to have some relation to other things in the domain. If two sets of things in the domain share no relations, it makes more sense to view them as two distinct domains.

In Object-Oriented terms, it would be convenient to define domains as classes defining things and with attributes and method defining relationships between them.

In the literature, it is common to include domain analysis and domain specification as part of the DSL development process (Mernik et al., 2005; Čeh, Črepinšek, Kosar and Mernik, 2011; Bačíková, 2014). While having a well-defined domain is certainly important for DSL production, it seems like domain specification is not a specific problem for DSLs. In many cases, the domain may well have been defined for other purposes (such as General Purpose Programming Language (GPL) implementations of the domain), and the DSL can just re-use that specification. This is particularly true if one focuses on DSLs meant specifically for aiding software development of existing Software Product-Lines (SPLs), as we are going to be doing in this thesis.

Some often used methods for creating and defining domains are Feature-Oriented Domain Analysis (FODA) (Kang et al., 1990), Organization Domain Modeling (ODM) (Simos, 1995), and Domain-Specific Software Architectures (DSSA) (Taylor et al., 1995) all identified by (Harsu, 2002) as the most well known methods for domain analysis. Since then, it appears that FODA has taken over as the go-to method for domain analysis in the field of DSLs (in so far as any method is used).

To describe domains, Feature Description Language (FDL) is often used (originally proposed by (van Deursen and Klint, 2002)). Other ways of specifying domains is to use ontology specification languages, such as OWL (W3C, 2012) as is the case in (Čeh, Črepinšek, Kosar, Mernik, Henriques, Pereira, Cruz and Oliveira, 2011).

2.3 Languages

Programming languages are a way to for humans to describe program implementations to machines and other humans. For example, (Aho et al., 2006) says "Programming Languages are notations for describing computations to people and to machines." In (Tucker and Noonan, 2007), Tucker and Noonan compares programming languages to natural languages: "Like natural languages, programming languages facilitate the expression and communication of ideas between people." and makes a similar statement to Aho et al. "... programming languages also enable the communication of ideas between people and computing machines." Interestingly, Tucker also mentions expressiveness, in regard to programming and natural languages: "... programming languages have a narrower expressive domain than natural languages". Tucker somewhat sidesteps the issue of what expressiveness means by qualifying with 'expressive *domain*'. Parr has a slightly more pragmatic definition: "A language is just a set of valid sentences" (Parr, 2010).

The general takeaway message is that languages are used to describe things. That is, one would use a language to describe a domain, for example. In fact, the only way to express a domain so that other people and machines can understand is through a common language. This means that it is fairly easy to confuse the idea of a domain with the description of the domain, as the only way we can ever communicate about a domain is by using some agreed upon common language. Domains certainly exist without a language describing them, but the only way we can define what a certain domain is, is by using a language.

It should be noted that languages can be seen as 'things' and relations. So one could say

Figure 2.1: A very general view of an interpreter, as given in (Aho et al., 2006)

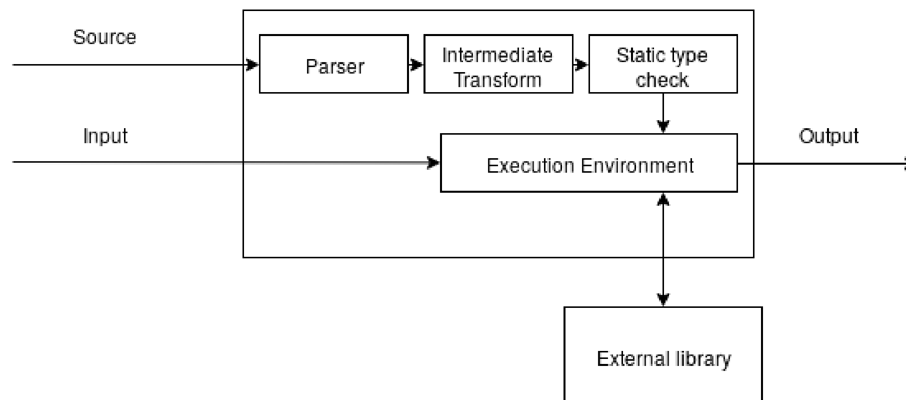

Figure 2.2: A more detailed view of an interpreter

there is a 'domain of languages', which consist of all the things that make up languages. In programming languages, we would expect this domain to consist of things like if-statements, loops, names, and assignments, all of which help to define what a program should do when it is executed. When we describe domains in an Object-Oriented fashion, our language needs constructs to describe classes and attributes.

In figure 2.1 we show a very general representation of an interpreter (as shown in (Aho et al., 2006)). In figure 2.2 we show a more detailed version of one way of implementing an interpreter. It shows four stages of the interpretation process. First, the source-code is parsed and turned into some intermediate representation (usually an Abstract Syntax Tree (AST)). Then the interpreter transform that into some internal representation. That internal representation is then checked for consistency, before the execution can start. In real interpreters, these steps may not be neatly compartmentalised like is shown in this figure.

It is also common to separate between abstract language and concrete language. Abstract language describes the semantics of the language, and the concrete language describe the

syntax of the language (Kleppe, 2008).

To elaborate, the abstract language consist of the operations the language can perform, and the concrete language consist of how we actually write sentences in the language. In other terms, the abstract language define the domain that the language is capable of operating in, and the concrete language defines how we express the operations. In other words, the abstract language defines the semantics of the language, whereas the concrete language defines the syntax.

The distinction between abstract and concrete programming languages is also convenient when creating an implementation that can execute sentences in a given language. The concrete definition of the language is used to construct a parser, that turns concrete sentences into abstract sentences. The definition of the abstract language is used to define how those sentences are executed.

A further convenience is that by separating the two, we can specify several different syntaxes for the same language, without having to specify the semantics several times. This is, however, not something we will be focusing on in this document.

2.4 Domain-Specific Languages

DSLs are languages that are tailored to a specific domain (Mernik et al., 2005).

The general consensus seem to be that a DSL is a language that describes solutions in a particular domain. For example, (Fowler, 2010) defines it as "a computer programming language of limited expressiveness focused on a particular domain" and (Thibault, 1998; Thibault et al., 1999) gives an informal definition as "a DSL [is] a language that is specific (i.e. restricted) to a particular application domain." (Mernik et al., 2005) writes "Domain-Specific Languages (DSLs) are languages tailored to specific application domains and offer users more appropriate notations and abstractions." What an application domain is in this example is not defined. (van Deursen et al., 2000) gives a similar definition: "a programming

language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain." (Hudak, 1997b) simply states that a DSL is "tailored to particular application domains."

van Deursen and Klint (1998); Kosar et al. (2010); Oliveira et al. (2009); Pereira et al. (2008) all reference one of the above definitions instead of giving their own.

Some of the authors seem to think that it is the 'littleness' of the language that makes it domain-specific (e.g. Walton (1998) who says that DSL is "A small, usually declarative, language expressive over the distinguishing characteristics of a set of programs in a particular problem domain") whereas most of the authors emphasizes that it needs to adhere to some domain to be a DSL (as shown above).

COBOL, FORTRAN are specified as GPLs by some (van Deursen et al., 2000) and DSL by some (Sprinkle et al., 2009). Prolog is similarly controversial as (Wile, 2001) unambiguously put it in the GPL section, but (Hudak, 1997b) lists it as a DSL (along with VB). Hudak also include Tcl/Tk in the list of DSLs, while (Mernik et al., 2005) notes that their DSLness is arguable.

Both van Deursen et al. (2000) ("Our definition inherits the vagueness of one of its defining terms: problem domain. Rather than attempting to define this volatile notion as well, we list and categorize a number of domains for which DSLs have actually been built ...") and Mernik et al. (2005) ("We will not give a definition of what constitutes an application domain and what does not") explicitly decline to define what a domain is, which seem to be the crux of what makes a DSL a DSL instead of a GPL. Presumably they either like the idea of having some level of ambiguity of what a DSL is, or they do not think that it is worth the trouble to try to give a concrete definition.

Mernik does go on to say "Leaving matters of definition aside, it is natural to think of DSLs in terms of a gradual scale with very specialized DSLs such as BNF on the left and GPLs such as C++ on the right ... Clearly, domain-specificity is a matter of degree.". He

quotes Jones (1996)'s Function Point Languages Table as an example of such a scale.

DSLs are often classified into different types based on their characteristics. For example, Fowler (2010) divides DSLs into external, internal and language workbench, depending on whether they are implemented to be read from external files, embedded in GPL code, or created using a special toolkit for language creation. On the other hand, Völter (2013a) only recognises internal and external. In (Mernik et al., 2005) several more classifications are provided (see table 2.1). Unlike Fowler, Mernik classify DSLs not only by how they are implemented, but arranges classifications based on steps in the production process. Mernik splits the classifications into decision, analysis, design, and implementation, representing the different steps in the process of producing a DSL. Each step then has further classifications. For the implementation step (which corresponds to Fowler's classifications), Mernik provides seven different classifications: Interpreter, compiler/application generator, preprocessor, embedding, extensible compiler/interpreter, commercial off-the-shelf (COTS), and hybrid. The two first would be classified as external in Fowler's system, while the second to fourth would be internal and COTS would be classified as language workbench. Hybrid does not fit in into Fowler's system, as it can be a combination of several of the others. An earlier attempt at classifying DSLs was published in Spinellis (2001) and Mernik provides a mapping between Spinellis and the ones Mernik provides.

Interestingly, the term 'expressiveness' is used in two distinct ways in the DSL literature. Sometimes it is used to mean how much can be expressed in how little code (similar to 'succinctness' as described in (Grohe and Schweikardt, 2003)), and other times it is used to mean how much can be expressed total in the code. By the first definition, DSLs tend to be more expressive than GPLs, by the second definition DSLs tend to be less expressive. For example, Sonnenberg et al. (2011) writes "It [DSLs] offers expressive power through appropriate notations and abstractions focused on – and usually restricted to – a particular problem domain" (Referencing (van Deursen et al., 2000)). Kollar and Chodarev (2010) similarly states "Domain-specific languages (DSL) are languages much more expressive

Development Phase	Pattern
Decision	Notation
	AVOPT
	Task automation
	Product line
	Data structure representation
	Data structure traversal
	System front-end
	Interaction
	GUI construction
Analysis	Informal
	Formal
	Extract from code
Design	Language exploitation
	Language invention
	Informal
	Formal
Implementation	Interpreter
	Compiler/application generator
	Pre-processor
	Embedding
	Extensible compiler/interpreter
	COTS
	Hybrid

Table 2.1: DSL development phases and patterns according to (Mernik et al., 2005)

and easy to use in its application domain by providing notations and constructs tailored toward this domain" (citing (Mernik et al., 2005)). In Mernik (2012) Mernik keeps the same definition: "Broad applicability often results in suboptimal expressiveness in any particular application domain, hence the motivation for domain-specific languages, which sacrifice generality in exchange for enhanced expressiveness in a particular domain." All of these use expressiveness to mean the amount that can be said in as little code as possible.

On the other hand, Hermans et al. (2009) clearly uses the second definition when writing "Using a DSL, domain specific features can be implemented compactly, however, the language is specific to that domain and limits the possible scenarios that can be expressed." (discussing expressiveness as a success factor in using DSLs) and "To measure the expressiveness of ACA.NET we asked the subjects how often they had to deny a customer a feature,

because it could not have been implemented with ACA.NET (Q17) or how often they had to write extra code to implement a feature (Q18). Answers to both questions are given by a five-point Likert scale ranging" (discussing measurement of impact from expressiveness) Interestingly, van Deursen is a co-author of this paper, even though van Deursen use the first definition in both (van Deursen and Klint, 1998; van Deursen et al., 2000). Hermans is not alone to interpret 'expressiveness' this way. Thielscher (2010) writes "In this paper, we address the fundamental limitation of existing GDL to be confined to deterministic games with complete information about the game state. To this end, we develop an extension of GDL that is both simple and elegant yet expressive enough to allow to formalise the rules of arbitrary (discrete and finite) n-player games with randomness and incomplete state knowledge.". Schmitt also seem to agree with this definitions in Schmitt et al. (2014): "As external DSLs introduce a completely new syntax and semantics, in general, they are more flexible and expressive than internal ones at the cost of a higher design effort."

Several approaches to aiding the production of DSLs have been proposed. For example, (van Deursen and Klint, 2002; Čeh, Črepinšek, Kosar and Mernik, 2011) explore how languages can be developed from feature models. However, as noted in (Völter and Visser, 2011), while discussing feature models in their relationship with Product Line Engineering (PLE), feature models are limited in their expressivity (in the sense of not as much can be expressed). Their argument is that DSLs would be better for describing variability in PLE. This of course only make sense if the DSLs are not themselves created using feature models. They therefore provide an extended feature modelling formalism to overcome this. Further, (Bačíková et al., 2013) notes: "However, the assumption for their solution is the existence of an ontology designed for the given specific domain. Which is actually an equally difficult problem compared to finding an existing DSL for the given specific domain." They go on to point out that there exist an abundance of GPL libraries already, which can be used as a basis for the modelling the domain of a DSL. They then show how User Interfaces (UIs) could be used to generate DSLs.

Parser generators such as Yacc (Johnson, 1975) and ANTLR (Parr and Quong, 1995) can be used to create DSLs, however, they require quite a good understanding of Language Engineering and are thus not easy to use for developers who are not previously familiar with how to create languages. If one would be interested in using them, there are books such as (Fowler, 2010) and (Parr, 2009) that describe in some detail how to create a DSL from scratch, using a variety of techniques, including parser generators.

There have been quite a few general papers on how to design DSLs. For example, (Luoma et al., 2004) provides a summary of the experiences of Domain-Specific Modelling Language (DSML) practitioners at MetaCase. It describes four approaches to DSML development and how those have been applied to a variety of different domains, and reflects on the results of applying each approach.

(Sprinkle et al., 2009) is a paper discussing when it is feasible to introduce DSLs to a project. It discusses what the prerequisites are (such as existence of a well-defined domain, availability of domain experts and similar) and what the trade-offs are (effort to maintain the DSL vs increased productivity and similar).

Karsai et al. (2014) identifies 26 design guidelines for DSLs in 5 different categories:

1. Language Purpose
2. Language Realization
3. Language Content
4. Concrete Syntax
5. Abstract Syntax

Note though that they give the caveat: “Please be aware that the subsequently discussed guidelines sometimes are in conflict with each other and the language developer sometimes has to balance them accordingly. Additionally, semantics is explicitly not listed as a separate step as it should be part of the entire development process and therefore has an influence on all of the categories above.”

There are also several books on the implementation of DSLs, such as (Fowler, 2010) who gives general advice on how to implement DSLs and (Bettini, 2013) that discuss how to implement DSLs in Xtext and Xtend, which are frameworks for the Eclipse IDE.

Some noteable DSLs, whose development has been described in detail in the literature, include the Graphics Adaptor Language (GAL) (Thibault et al., 1999), which is a DSL for specifying video device drivers and WebDSL (Visser, 2008), which is a DSL for specifying web applications. Other interesting DSLs whose development is documented in the literature include The Tree Processing Language (TPL) (Papegaaij, 2007), which is a language for processing trees, specifically ASTs. Dhouib et al. (2012) describes RobotML, a DSL for specifying robots. Note that DSL for specifying robot behaviour is a common toy example in the DSL literature (Pereira et al., 2008; Wu et al., 2009; Čeh, Črepinšek, Kosar, Mernik, Henriques, Pereira, Cruz and Oliveira, 2011; Mernik, 2013), but those languages are not related to (Dhouib et al., 2012).

Several different tools and techniques for creating DSLs have been discussed in the literature. Early tools created for DSL development include the Jakarta Tool Suite (JTS) and Bali (Batory et al., 1998) that builds on the GenVoca model of generators (Batory and Geraci, 1997). JTS essentially provides additional operations to Java (and potentially other languages) for processing code. In particular, it adds functionality to do lisp-style quoting of code, turning Java snippets into ASTs. Bali is a tool for specifying grammars that can be turned into Java code. JTS is implemented using Bali. JTS has since been superseded by the AHEAD Tool Suite (ATS) (Batory, 2004) which provides similar functionality.

LISA (Mernik et al., 2002; Mernik, 2013) is another tool for creating DSLs and GPLs using an interactive environment and attribute grammars.

Porubän et al. (2009, 2010) introduces YAJCo, a parser generator based on annotation. According to Porubän et al., what sets YAJCo appart from other parser generators is that it focuses on the abstract syntax instead of the concrete syntax. YAJCo is implemented as annotations on Java methods and classes. YAJCo has received several updates since the

original paper, such as (Lakatos and Porubän, 2013; Chodarev et al., 2014) both introducing different forms of composition to the original tool.

Erdweg et al. (2011) presents SugarJ, a tool for adding domain-specific syntax into Java programs using 'sugar' libraries. The tool allows users to import in a java-file libraries specifying DSLs that then allow the user to use the specified DSL in the java-file. The tool then 'desugars' the files, generating plain java-code from the DSL code.

Dinkelaker et al. (2013) describe a tool for incrementally adding concrete syntax to embedded DSLs, a similar use-case as (Erdweg et al., 2011), but without the explicit use of libraries and with support of different host-languages (Dinkelaker et al. reference Groovy and Java as supported languages, with notes for how to implement in other languages). Another difference Dinkelaker et al. claims is that their approach allow concrete syntax to be added incrementally, whereas they claim other approaches require the full syntax to be provided immediately. Dinkelaker et al.'s tool is designed so that the backend is useable without any concrete syntax being defined.

There have been several reviews of the literature, both informal and formal mapping studies. An early paper that discusses the literature is (van Deursen et al., 2000), which describes the general field at the time and includes an annotated bibliography covering what van Deursen et al. thought was the most important publications on DSLs at the time. (Wile, 2001) is a review paper documenting some of the concerns in supporting DSL development. It includes a review of various state-of-the-art approaches (at the time).

Spinellis (2001) reviews different design patterns for DSL development. The work was significantly extended by (Mernik et al., 2005), which not only reviews design patterns, but patterns for all stages of DSL development. An overview of (Mernik et al., 2005)'s patterns was shown in table 2.1. In a similar vein, (Oliveira et al., 2009) reviews the advantages and disadvantage of DSLs (as compared to GPLs), as discussed in the literature.

There have also been some Systematic Mapping Studies on DSLs published, such as (do Nascimento et al., 2012) and (Kosar et al., 2016).

do Nascimento et al. (2012) surveyed the literature with the following questions in mind:

- **Q1**. Which techniques, methods and/or processes are used while working with DSLs, i.e. creation, application, evolution and extension of DSLs?
- **Q2**. Which DSLs have been created and are available for use or are described in some type of publication?
- **Q3**. In which domains are these DSLs being used?
- **Q4**. Which tools are used for the development and usage of DSLs and how such tools support those activities?

in 1440 papers on DSLs published up until 2011.

Kosar et al. (2016) surveyed the literature with the following questions in mind:

- **RQ1**: What has been the research space of the literature within the field of DSLs since the survey paper on DSLs [45] was published 10 years ago?
- **RQ2**: What have been the trends and demographics of the literature within the field of DSLs after the survey on DSLs [45] was published 10 years ago?

where [45] refers to (Mernik et al., 2005) (Mernik is also a co-author on (Kosar et al., 2016)). Kosar et al. found 1153 papers published between 2006 and 2012 (both inclusive).

2.4.1 Composition

Composition of DSLs is a technique for creating DSLs where new DSLs are built from existing languages using composition operations. A common classification for different composition operations is given in (Erdweg et al., 2012) who identifies the following types of composition:

- Language extension – extending an existing language with new constructs
  - Restriction – a special type of extension, where the extension restricts the new language
- Language unification – combining two languages to form a new language
- Self extension – languages that are capable of defining their own extensions
- Composition of above – a combination of the above composition types

The goal of Erdweg's classifications "... is to provide precise terminology for language composition that enables effective communication on language composition and can serve as a basis for comparing existing and future language-development systems."

Erdweg's classifications were developed as part of the work for SugarJ (Erdweg et al., 2011), a tool for embedding DSLs into Java and have been used in several different tools and frameworks, for example (Mernik, 2013) where the classifications are applied to the language construction framework LISA (Mernik et al., 2002). It has also been applied to YAJCo, a parser generator that work with abstract syntax in (Chodarev et al., 2014).

In (Degueule et al., 2015), Deguele presents Melange, a meta-language for modular and reusable DSLs. They use the operators merge, slice and inherit, which they note are similar to Erdweg's extension, restriction, and merging respectively. Note that Erdweg considers restriction to be a special case of extension, while Deguele have them as clearly separated concepts.

Erdweg's composition classifications are also used in Degueule's PhD thesis (Degueule, 2016) on DSL composition.

Besides Erdwegs classifications, there are several other competing classifications. In Lakatos and Porubän (2013), the following classifications are used:

- extension (full language is used and new concepts added)
- specialization (reuse of only some of the concepts of the language)
- insertion (code from one language is used inside other language)
  - direct/embedding (full parts of code are used)
  - referencing (only identifiers from other language are used)

Like Erdweg, Lakatos uses extension to mean that a language get extended with new constructs, but that is the only classification that is the same. Lakatos sees specialisation as its own operation, whereas the same operation (called restriction) in Erdweg is viewed as a special case of extension. Erdweg also does not differentiate between extension and insertion and Lakatos does not define anything similar to Self extension and extension composition.

In (Cazzola and Vacchi, 2016) a composition technique using Scala Traits to compose parsers is presented. The author does not specifically work in any composition classification framework, but does note that some of the work would fall under Erdweg's language extension and unification.

Other composition techniques that reference Erdweg include Diekmann and Tratt (2013) who uses language boxes to mitigate the problem with clashes, where the introduction of a new grammar rule causes the grammar as a whole to be ambiguous when composing. Language boxes are a way of viewing a sentence in an embedded language as a single token in the host language. Diekmann and Tratt's approach requires a special editor that keeps track of where the language boxes begin and end.

In (Chodarev and Kollar, 2016) an extensible host language that can be used for composition is presented. It identifies language extension and unification, and composition of extensions as the operations supported by the framework.

In the work on MontiCore Haber et al. (2015) identifies all of Erdweg's operations to be supported, except for self-extension.

There are also several DSL composition approaches that either precedes Erdweg or were published at the same time as Erdweg and therefore do not reference them.

In Dinkelaker et al. (2010), a technique for composing Embedded DSLs is explored. There are no classification of composition operations included, though.

Völter's self-published book (Völter, 2013a) describes DSLs in-depth. The book also covers composition of DSLs, but the composition is different to Erdweg's and does not reference them either. Völter identifies:

- Referencing – concepts in a language references concepts in another language
- Extension – a language is based on another language, but includes new concepts
  - Restriction – like Erdweg, Völter sees restriction as a special case of extension
- Reuse – a language reuses concepts in another language through indirection
- Embedding – a language is embedded in another language

Völter also explores these in earlier papers Völter and Solomatov (2010); Völter (2013b)

2.4.2 Evaluation

Evaluation of DSLs and DSL development processes is important to be able to determine if a particular DSL or DSL development process actually provides the benefits that they claim they provide.

There has not been a lot of quantitative evaluations in the field of DSLs and the one studies that have been made have mostly been on individual DSLs, rather than DSLs in general or DSL development tools. This means that as we want to evaluate our methodology and tool, we do not have a lot of previous work to build on. We will in this section, however, go through the state-of-the-art in evaluating DSLs and their tools, to then propose two ways we are going to use to evaluate our approach empirically.

As noted in (Gabriel, Pedro and Goulão, Miguel and Amaral, Vasco, 2011), DSLs are not often evaluated by the engineers that create them. They review 36 (out of 242 considered) papers published between 2001 and 2008 to determine how DSLs have been assessed and what the results have been. They found only 5 of the papers discussed a quantitative or qualitative evaluation of DSLs.

Since this study was done, several more papers on the topic has been written. For example, (Kosar et al., 2009) discusses the effect of domain-specific notation on program understanding. They do it by comparing the DOT diagram drawing language (Koutsofios et al., 1991) with a Application Program Interface (API) for constructing the DOT diagrams within C. They evaluated the two approaches, asking users to perform certain tasks using each approach. The tasks were split into three different categories, learn, perceive, and evolve. Each of these categories have several tasks for the users to complete, 11 total. The complete set of tasks are shown below.

- Learn
  - Q1 Select syntactically correct statements.
  - Q2 Select program statements with no sense (unreasonable).
  - Q3 Select valid program with the given result.
- Perceive
  - Q4 Select the correct result for the given program.
  - Q5 Identify language constructs.
  - Q6 Select program with the same result.
  - Q7 Select the correct meaning for the new language construct.
  - Q8 Identify language constructs in the program with comments.

- Evolve
  - Q9 Expand the program with new functionality.
  - Q10 Remove functionality from the program.
  - Q11 Change functionality from the program.

Kosar et al. (2010, 2012) describe two additional studies that used different domains but same approach and analysed the result of all three studies. They found that subjects were significantly more likely to choose correct answers for DSLs compared to GPLs.

Barišić et al. (2011) performs a similar comparison between a DSL and a GPL library. The DSL they were evaluating was Pheasant (PHysicist's EAsy Analysis Tool) (Amaral et al., 2003), "a declarative domain specific visual query language for HEP [High-Energy Physics] data analysis" comparing it with C++. The participants were given four queries to implement in each language, and then asked how well they thought they performed. They measured time to train the subject in each language, amount of correct (or almost correct) answers and time to complete the test. People who knew C++ from before got all the answers 'essentially' correct for both languages, but 'uninformed' participants did tend to get the questions more correct when using Pheasant. The time to complete the test was also shorter with Pheasant, for all participants.

Kahraman and Bilgen (2015) provides a long and in-depth description of how to evaluate DSLs, covering multiple different aspects of DSLs such as usability, reliability, maintainability and many more.

That is a very short summary of some of the work in evaluating DSLs using usability methods. There are also some work on evaluating DSLs based only on their observable properties, such as number of Lines of Code (LOC) for a particular implementation in different languages (Zeng et al., 2006; Merilinna and Pärssinen, 2007) and proposals for a variety of other quality attributes to compare DSL on (Power and Malloy, 2004; Črepinšek et al., 2010).

2.4.3 Other DSL concerns

One DSL research area is language families (Völter, 2008; Zschaler et al., 2010; Kühn et al., 2015). A language family is a set of languages that are related to each other somehow. Völter (2008) describes how a family of languages for software architecture description can be developed. Zschaler et al. (2010) presents the general idea of language families, an approach to create language families using Domain-Specific Metamodelling Language (DSM2L) as well as presenting three language families that have been implemented using DSM2L. Kühn et al. (2015) proposes a bottom-up approach to automatically extract a feature model from a set of language components. They motivate the usefulness of their work by commenting on how the recent rise in constructing languages from components (such as through composition) has made a Software Product-Line (SPL) view of languages sensible: “ As a result, the programming language becomes a family of programming languages created by a language product line (LPL)”

Globalisation of DSLs Cheng et al. (2015) is an area of DSL research that focuses on how to manage several DSLs that interact with each other. The paper (Bryant et al., 2015) discusses the concerns related to DSL interaction, such as how languages can be related to each other:

- Being variants of each other
- Being different versions of each other
- Having different viewpoints of the same domain
- Being composed of each other or from common components

Recent work on globalisation include (Ali, 2020) which introduces Perspectives for Multi-Language Systems (PML) to “... facilitates consistency and reuse of an existing language potentially across other languages and software systems.”.

2.5 Related areas

2.5.1 Ontologies

DSLs have been combined with ontologies. For example, (Walter and Ebert, 2009) describes how a DSL for description of network devices (BEDSL) and FODA (Kang et al., 1990) was combined with the ontology language OWL 2 (Motik et al., 2009) to each provide their own view of the domain (this would be viewpoint relation as explained in the previous section). The concept of using ontologies in this way has since been developed in (Walter et al., 2009, 2014)

In (Čeh, Črepinšek, Kosar and Mernik, 2011; Čeh, Črepinšek, Kosar, Mernik, Henriques, Pereira, Cruz and Oliveira, 2011) tools and theory for creating DSLs based on ontologies in OWL (Lacy, 2005) is presented. The tool reads in an OWL specification and applies a set of rules to it to produce a grammar for a DSL based on it. A similar tool is also presented in (Fonseca et al., 2014) that builds on the same concept but improves it by automating some of the steps.

2.5.2 Product Families

In (Parnas, 1976) Product Families (PFs) are described thus “We consider a set of programs to constitute a family, whenever it is, worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members.” A PF is in other words a set of programs that share a considerable amount of commonality with only some variation between the programs. This implies that it is possible to save a lot of effort by implementing the common parts of all programs in the family once, and then for each specific member of the family implement the variable parts. PFs are also sometimes called SPL (in fact, this seems to be the more common usage).

The relationship between DSLs and PFs has been explored on numerous occasions in

the DSL literature, both as a way of using DSLs to specify PFs (Völter and Visser, 2011; Acher et al., 2013) and using PFs to aid DSL production (Batory, Johnson, MacDonald and von Heeder, 2002; White et al., 2009; Rossel Cid, 2013). The relationship has also been noted in other works that are not specifically about DSLs or PFs (Chen and Babar, 2011; Wu et al., 2009; Vacchi et al., 2013).

PFs are not the primary interest in this project, however, we do find that viewing DSLs in terms of PFs makes it easier to describe some features of DSLs. As has been hinted at, a PF is a collection of (software) products that share a large part in common with each other. Products in a PF are usually called members of that PF (Weiss and Lai, 1999). One can then view a DSL as being an implementation language for members of a given PF. That is, DSLs specify individual members of a given PF. One can argue about whether all DSLs fit this definition; for example, do Make-scripts define members of a PF for compilation-pipelines or if this is pushing the definition of PFs?

Viewing DSLs as implementation languages for members of PFs also gives an intuitive notion of what it means for a language to be specific to a domain. A DSL is specific to the domain described of the PF that the DSL specifies members for.

The idea of commonality and variability that is central to PFs (Weiss and Lai, 1999; Vacchi et al., 2013) is also useful when discussing DSLs. In general, commonality of a DSL can be seen as the 'domain' part of the DSL, whereas variability is part of the 'language' of a DSL. That is the parts of a PF that is common roughly specifies the domain that the DSL works in. The language uses those common parts to construct descriptions of members of the PF. Thus, the language works in the variable part of the PF, given the common parts in the domain.

Chapter 3

Patience Games

3.1 Introduction

Domain-Specific Languages (DSLs) have been studied for some time now as a way to improve the development process by providing an easier way to specify solutions within a specific problem domain (Mernik et al., 2005; Fowler, 2010; Kosar et al., 2016). The common reasons given for using DSLs are that they make development easier by making code easier to comprehend and by reducing the amount of code that needs to be written, as well as letting non-programmers use them.

When discussing the development of DSLs, it is useful to be able to refer to some model domain to help clarify new ideas with concrete examples.

One such common model is the Robot language (Pereira et al., 2008; Wu et al., 2009; Čeh, Črepinšek, Kosar, Mernik, Henriques, Pereira, Cruz and Oliveira, 2011; Mernik, 2013). It does not appear to be an explicit model for trying out DSL techniques on as much as a convenient domain to use in a variety of situations, as everyone has a good idea of what a robot is and what it is expected to do. The language is also more of a toy language, that one would not use for real-world robots.

Another common example is state machines (Crane and Dingel, 2005; Fowler, 2010; Zdun, 2010). These are also well known and therefore one can focus on explaining the

problem and suggested solution, rather than explaining the example environment used to showcase the problem. State machines are quite general and can be used in conjunction with pretty much any domain as a way of managing flow of control. On their own they thus lack a certain specificity that one might want to have when discussing languages specific to a certain domain.

This chapter presents a different model that can be used to discuss ideas surrounding the creation of DSLs. It uses the domain of patience games. The advantage of the patience domain over the previously discussed domains, is that the languages produced for this model can be used to describe real-world human-playable games. The domain of patience games is also quite specific and it is relatively easy to tell what is and is not a patience game.

To the best of our knowledge, patience has not been used to in the DSL literature before. Patience has been studied in other contexts though, especially the complexity of various patience games (Yan et al., 2005; Longpré and McKenzie, 2009; Bjarnason et al., 2009). We think that patience makes a good model for discussing various uses of DSLs, due to some of the properties of patience games.

There are many different patience games, and they all share quite a bit in common. Thus, one might want to have a language specific to specifying patience games. In other words, there is a domain of patience, and in order to describe members of that domain (i.e. patience games), one might want a language that is limited to the patience domains that makes it easier to specify the games.

Besides being a domain that have the properties one would usually look for when considering creating a DSL, there are also other features of patience that make it interesting to study.

For example, one can view the patience domain from different levels of generality. From something very specific, like only considering Klondike-like patience games, to considering all patience games, to considering card games in general (and beyond!). This feature might be used to measure the effect of ever more specific languages on the ease at which

games are implemented. It is also useful when considering language composition (Erdweg et al., 2012); re-using languages for specifying general card-games to create languages for specifying patience games specifically (as an example).

Using DSL to help analysis of domains is also one suggested reason to implement DSLs (Thibault, 1998). There are at least two different things one might want to analyse in the patience domain: the complexity of individual patience games and solving a given patience game deal. Thus, patience games could serve as a way of showing different ways of using DSLs for domain analysis.

To simplify the model, we are assuming a scenario where there is a pre-existing code-base that the DSL have to interact with. We will discuss the design of this code-base in more details later in the thesis.

The reason we have chosen to use a pre-existing code-base for this model is for one that it is a likely real-world scenario, where one wants to provide a DSL for an existing code-base. Secondly, it provides a natural execution platform for any DSLs developed, and thirdly, when considering two alternative patience DSLs we do not have to worry about potential confounding variables introduced from the environment.

The purpose of this chapter is not to show solutions to any of the issues discussed, but rather to present the features that our particular patience model has, that can be used to discuss concerns of DSL development.

We will start the chapter by discussing what patience is, in section 3.2. Then we will go through the design of our patience suite in section 3.3. After that we will discuss DSL issues we believe the patience model can be useful in covering in section 3.4. Finally, we will provide some concluding remarks in section 3.5

This chapter is an extension of the paper (Kihlman, 2017), presented at the 9th Computer Science & Electronic Engineering Conference in Colchester, 2017.

3.2 Patience

One of the first books on patience published in English is Lady Cadogan's Illustrated Games of Patience (Cadogan, 1874). She describes 24 different patience games, together with pictures to show the layout of the games. Interestingly, the games described by Lady Cadogan seem somewhat less formulaic than what one commonly see in modern patience suites.

Gibson (1993) defines patience games as "... any card game played by one person who usually deals out cards and then assembles them in special groups according to established rules."

Morehead and Mott-Smith (2001) describes well over 100 patience games, in quite some detail.

Patience games have one or more decks, and consist of one or more groups of piles, usually called foundation, tableau, waste and reserves. For games that have a foundation, the goal is usually to move all cards on to the foundation piles, in a game specific order. The tableau is used to move around cards until one can move them up to the foundation. When there are waste pile(s), they are used to deal new cards from the deck to. When there are no waste piles(s), cards are dealt straight to the tableau or reserve. Usually, it is not possible for the player to move any cards onto the waste. It may be possible to turn the waste back into the deck in some games (possibly for a limited amount of times). The reserve are piles that one can temporarily move any card to (though usually only one card per pile). In some games, the whole deck is dealt at the beginning of the game, and no further dealing can happen during the game. One of the most popular patience games, Klondike, is an example of a game using foundation, tableau, waste and a deck that deals to the waste. FreeCell, on the other hand, have a foundation, tableau and reserve piles, and the deck is completely dealt to the tableau at the beginning of the game. Some games allow the movement of builds (i.e. a subset of a pile where all cards are built according to the rules for that pile).

The different pile groups usually have different rules for how they are built. The foundation and tableau piles are usually built by either same suit, same colour, alternating colours,

or different suit, in ascending or descending value order (or both). Empty piles can usually either take an ace or a king, or any card. It is common for the foundation to be built in the reverse order (and by different rules for suit) to the tableau (but this is not always the case). The waste and reserves can usually not be built on, with the waste getting its cards from the deck, and the reserves only being able to contain one card each.

Not all games have a foundation to build on. In some games, such as Pyramid, the goal is to discard all cards. Cards are discarded by matching two (or more) cards, for example by suit, value or (as is the case in Pyramid) by having the total of the value add up to a certain number. Matching can either be done by selecting two adjacent cards for removal, or by arbitrarily moving one card on top of another. In these types of games, cards are often dealt straight to the tableau, requiring the player to first discard the top card, before being able to play the lower cards.

3.3 General Design of Patience Suite

3.3.1 Software Product Lines and Domains

There is a close relationship between Software Product Lines (SPL) and DSLs. For example, DSLs are often used to specify SPLs (Völter and Visser, 2011; Acher et al., 2013) and SPLs can be used to aid the production of DSL (Batory, Johnson, MacDonald and von Heeder, 2002; White et al., 2009; Rossel Cid, 2013).

Here we are going to talk about the relation between SPLs and DSLs as one where the DSL is used to specify *members* of a particular SPL (not to be confused with specifying SPLs themselves). In a simplistic way, the SPL takes the role of the *domain* part of the DSL and the DSL takes the role of the variability specification in the SPL.

There are many ways one can view the implementation of an SPL, but for our purposes, we are going to assume that the common parts of the SPL has been implemented in a General Purpose Language (GPL) and is part of a pre-existing code-base. That is, all the parts that

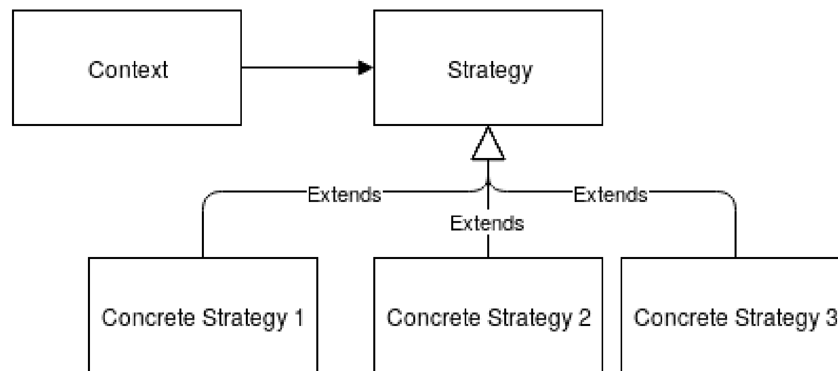

Figure 3.1: UML diagram describing the Strategy Pattern

members of the SPL have in common are implemented in an existing GPL code-base. We also assume that the variability in the SPL is defined using the strategy pattern (Gamma et al., 1994), as described in (Greenfield and Short, 2004, pp. 342). The idea is to have every point of variability defined in terms of the strategy pattern (as shown in figure 3.1). The strategy pattern consist of a class (the context), that delegates some of its logic onto an abstract class (the strategy). That way, the same context can be instantiated with different logic (concrete strategies) without changing the code of the context class. The task of creating a concrete product that is part of the SPL is then reduced to choosing particular concrete strategies, either from a collection of pre-existing strategies or by implementing a new one. If we see SPLs this way, we can introduce a DSL for providing implementations for the concrete strategies. This would then make the DSL an implementation language for products of a particular SPL.

If we use an interpreter to execute the DSLs, we can introduce it as shown in figure 3.2. Interpreted Strategies implements the Abstract Strategy as normal, but rather than executing GPL-code, it passes on any messages to the inherited methods to the interpreter, which can then execute the relevant code as defined in the DSL. The interpreter also needs to interact with the rest of the code-base/SPL. This pattern does not necessarily need to be used, for example, Domain-Specific Embedded Languages (DSEL) approaches would not need an interpreter, and external DSLs could be compiled instead of interpreted. We discuss the interpreter approach here to show how an interpreted external DSL could be implemented,

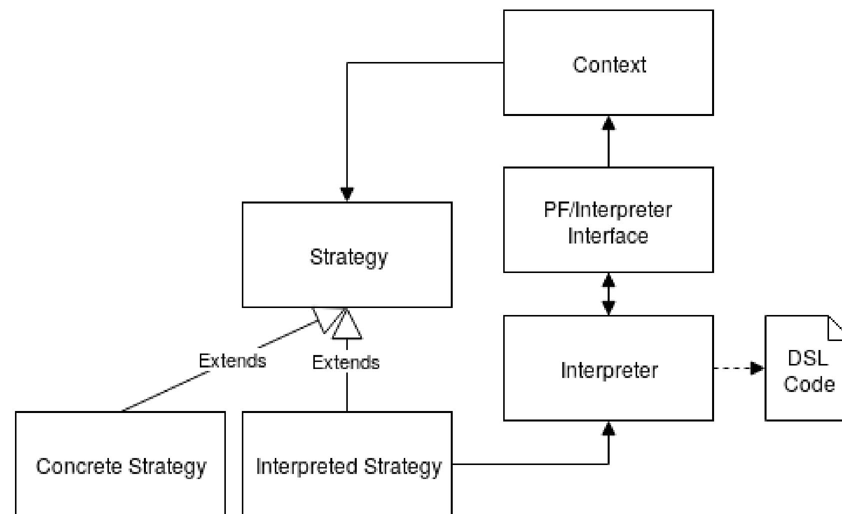

Figure 3.2: Introducing an interpreter for specifying Concrete Strategies

without insisting that all DSLs are implemented this way. Later we will discuss these options in more details.

Our implementation of the core classes have been done in Java.

3.3.2 Implementation of Patience Suite

3.3.3 Existing Code-bases

There exist quite a few patience suites (i.e. programs that allow users to play several different patience games). Some examples of open source patience suites include XPat (Eißfeldt and Bischoff, n.d.), Ace-of-Penguins (Delorie, 2012) KPat (Team, n.d.), Aisleriot (Gnome, n.d.), and PySolFC (PySolFC, n.d.). XPat, Ace-of-Penguins, and Aisleriot are all written in C. KPat is written in C++ using KDE's graphical libraries. PySolFC uses Python. The only one of these that come close to using DSLs is Aisleriot, which uses SCM (scheme) as an external scripting language to define the rules for each game. PySolFC provides a graphical 'wizard' to define rules for new games, although it is fairly limiting.

We could use one of these code-bases as a model, however, as none of them were designed for DSLs, they do not necessarily make the best model environment for DSL construction. Instead, we will provide a simple, purpose built patience suite environment.

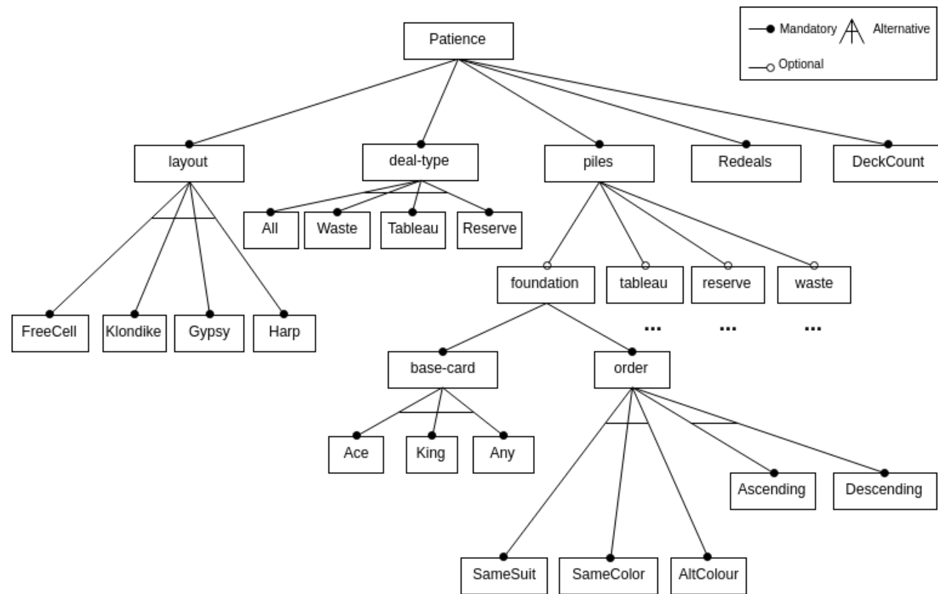

Figure 3.3: Feature Diagram for the patience domain based on PySolFC's patience game wizard

Future work may look into working with one or more of these patience suites, to evaluate how much effort it would take to convert a DSL written for one suite to work with another, but this is beyond the scope of this chapter and the model we are discussing here.

This patience suite was originally developed for showing a particular way of implementing external DSLs, as described in (Kihlman, 2015).

Our patience suite consist of a couple of core classes. One abstract class that define the rules of individual games, a view class that displays the game, as well as control code to let the user interact with the game. Of these, only the class that defines the rules is relevant for the creation of the DSL; The view and controller are the same for all patience games, and individual games are independent of the working of them.

Here we define the type of things that exist in the domain of patience games, as well as examine the relationships between the things. In figure 3.4 we show a very simple design for a patience suite described in a UML class diagram. It consist of cards, piles, groups of piles, and decks, as well as an abstract patience game class and a view class. In the terms of SPLs, the abstract class defines the variability of the domain, and everything else is part of the commonality of the patience SPL. This is in line with how (Greenfield and Short, 2004,

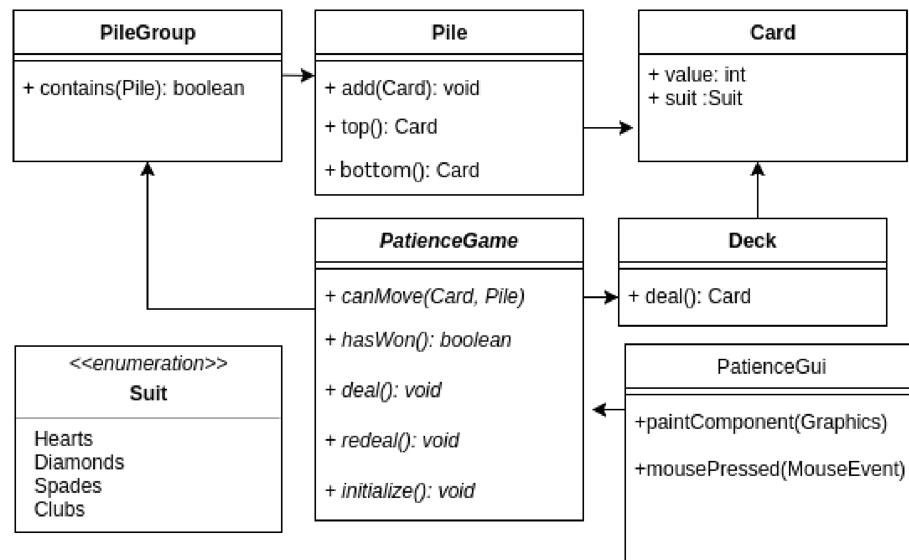

Figure 3.4: UML diagram describing the main parts of a simple patience suite. Some simplifications have been made to better fit the diagram.

pp. 342) suggest managing variability, through the use of the strategy pattern (Gamma et al., 1994).

Gamma et al. describe the intent of the strategy pattern as “Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.” The pattern consist of context, strategy, and concrete strategies. The context is the client using the strategy. The strategy defines an interface for the client to use, and the concrete strategies define the various implementations of the strategy.

In our example, the patience view represents the context, the abstract patience game class represents be the strategy, and patience game implementations (concrete sub-classes to the abstract patience game class) represents the concrete strategies.

So, in the view of SPLs, the patience SPL is defined by the class-diagram in figure 3.4 and individual members are defined by providing a concrete sub-class to the abstract patience game class.

If we consider DSLs as being implementation languages for members of specific SPLs, then a DSL for patience games would consist of a language for specifying sub-classes of the abstract patience game class.

In figure 3.5, we show the domain for the patience DSL, including some concrete strategies (Klondike, Golf, and Yukon). We have removed the view class, as it does not affect the implementation of the strategies and thus is not needed in the DSL.

The domain consist of cards, piles, group of piles and deck(s) of cards. Cards have suit and value, as well as colour (which is completely dependent on suit). Piles are collections of cards. They support adding one or more cards on top and removing one or more cards from the top. Groups of piles are collections of piles. Their purpose is to allow specifying different rules for different set of piles. Decks of cards support dealing a card for the top, shuffling, and repopulating the deck from a pile.

Finally, there is the class defining the rules. Each rule is implemented as an abstract method, to signify that individual patience games need to provide their own definition for these rules. The rules that the class should define is: rules for how a card can be moved on to a pile, moving a group of cards onto a pile, whether the game is in a win-state, the game's initial setup, dealing new cards, and what to do when the deck is empty (i.e. how to do a re-deal). We ignore having a rule for a losing state, as it in general is quite hard to compute whether a given game is in a losing state or not.

If one uses a interpreter, a DSL specified member of the patience domain may look something similar to figure 3.6. This shows a concrete sub-class for the abstract patience game class. This sub-class is responsible for implementing all domain members implemented using the DSL. To do this, it uses the interpreter class that is capable of reading and executing the DSL code. A compiler would output separate classes for each domain member, in the same way as a non-DSL implementation would do (as is shown in figure 3.5). A DSEL could, for example, be implemented by providing a special concrete strategy builder that is used to define a concrete strategy through standard DSEL techniques (Fowler, 2010).

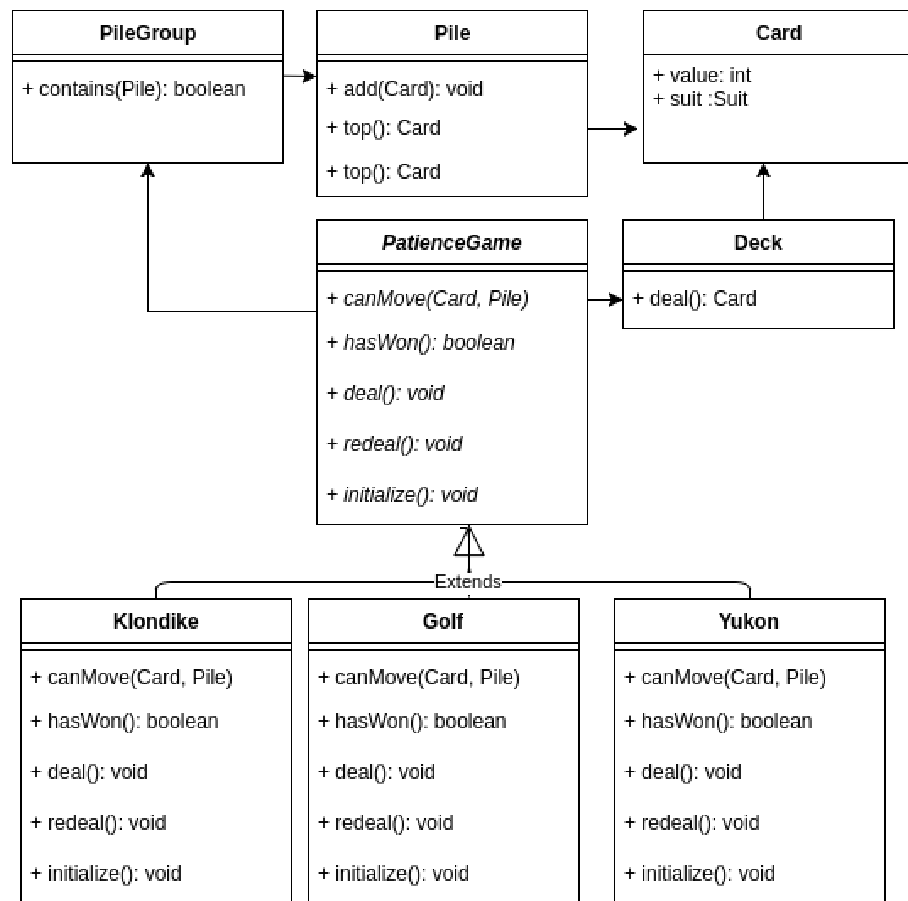

Figure 3.5: UML diagram describing the domain that is relevant for our DSL, including some example patience PF members (Klondike, Golf, and Yukon)

So, what we want to do is create a sub-class for the abstract patience game class that interfaces with an interpreter which is capable of executing DSL code statements. Now, we need to define the type of statements that the interpreter can execute, in order for the sub-class to be able to correctly implement a member of the domain.

The methods that need to be implemented are described in table 3.1. We could support other possible methods, such as hasLost (are we in a losing state?), or shouldTurnCard (should a given card be turned face-up/face-down?), but calculating whether a game is in a losing state can be quite hard, and may likely require a general purpose language to work. Most patience games also follow a simple rule for whether a card should be turned; If it is on the top of the pile, it should face up, and cards that face-up, should stay face-up. Thus, there is not much point to force users to implement either method. A possible alternative would

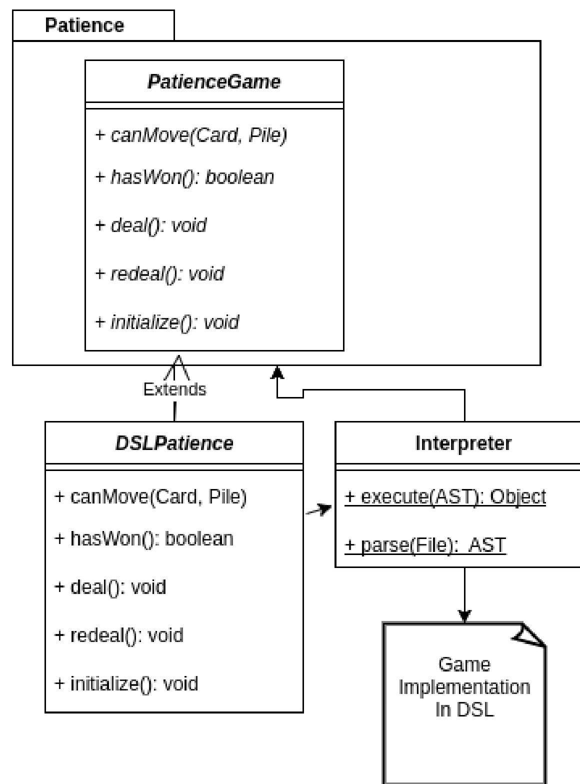

Figure 3.6: UML diagram describing an interpreted version of the patience game using a DSL. The patience domain from figure 3.5 has been replaced by a package.

be to let the methods have default implementations, and then allow users to override the defaults for patience games that have an easier way of determining losing state, or games that have a different rule for turning cards. By default, the hasLost method would return false, and the shouldTurn method would return as mentioned above.

Beyond the methods, each patience game also have to define what pile groups it uses (such as foundation, waste, tableau) and how many piles are in each group. These can be implemented as attributes of the concrete strategies.

One issue when it comes to creating DSL for a code-base such as the one in this chapter is that the code-base itself may not be flexible enough to allow all patience game to be implemented. In this case, it does not necessarily matter how expressive the DSL is, if the underlying framework does not allow specifying certain rules. One obvious limitation of our code-base is that there is no way to specify how many decks that are available, and the game logic assumes that there is only one. The underlying code-base might thus have to be

Operation	Description
canMove(Card, Pile) canMove(Pile, Pile)	Can a card or a pile be moved on top of another pile?
hasWon()	Is the game in a win-state?
deal()	Deal a card, taking a card/-cards of the deck and putting it on relevant piles.
redeal()	If available, reset the deck with unused cards.
initialise()	Initialise the game, dealing the first cards to the correct piles

Table 3.1: Methods that a Patience Language needs to implement in our model.

updated if any such issues arise.

3.4 Use of Model

This section will go through how the model is intended to be used and what considerations there are when using it.

3.4.1 Host Languages

Host languages are languages that the DSL is using for its implementation (Mernik et al., 2005). This is most often discussed in the context of DSELs, where the host language is the language that the DSL is embedded in, but has also been used to describe languages used in generative approaches (Zdun and Strembeck, 2009). This subsection goes through some intuitions about considerations when choosing a host language.

Since we are specifically talking about a case where there is an existing code- base, we have to have a way of interacting with the code-base. In the case of using an interpreter as discussed above, host languages are rather irrelevant, as long as there is a parser that can turn DSL code into a format the interpreter can understand, and the interpreter itself is able

to interact with the code-base.

For compiled languages, the only real option is to compile down to a binary compatible language the code-base is written in.

As with compiled languages, embedded languages need the host language to be compatible with the language the code-base is used in. For example, Scala could be used to define a DSL, if the code-base is written in Java. This can be convenient, as while it is quite possible to define DSELs in Java, languages such as Scala offers a lot more options for the implementation of the DSEL, by providing a more powerful reflection framework (Schmitt et al., 2014).

The above discusses using GPLs as host languages, but what if one wants to use an existing DSL as a host language? For example, what if one wanted to use a Game Description Language (like (Love et al., 2008)) or a Card Game Description Language (like (Font et al., 2013))?

As discussed, there are several patience suites, written in a variety of languages. The question of how one can port a DSL written to work with a host language for one suite to make it work with another host language in another suite is essentially a question of how one can switch out the host language while keeping the DSL the same.

The central questions for choosing host language in our model is 'to which degree is it possible to select host language freely, and to which degree our we stuck with the language that the code-base is written in' and 'what can be done to integrate an otherwise incompatible host language'.

3.4.2 Patience Language

There are some existing DSLs for defining games in general (Love et al., 2008; Thielscher, 2010) and for card games in particular (Font et al., 2013) (though none of these seem to identify their own work as DSLs), but they are not aimed towards single player games and include a lot of generality that is not required for the specific case of patience games.

Now let us talk about the kind of languages we want to create in order to define members of the patience domain. As mentioned, in our example, this means creating concrete subclasses to the abstract patience game class. A patience language thus needs to provide a way of implementing each method in the abstract patience game class. The language also need a way of specifying the pile groups, which can be seen as attributes of the patience game class.

In listing 1 we show how the canMove method might be implemented in our code-base language, Java. While our DSLs do not strictly speaking have to compile down to Java (as discussed above), assuming that the DSL represents something equivalent to what is presented in the listing makes it easier to discuss the considerations for when creating DSLs for implementing the Patience Strategy.

Listing 1 An example of how the canMove method might be implemented in Java.

```
boolean canMove(Card card, Pile pile) {
  if ( foundation.contains(pile) ) {
    if ( pile.isEmpty() ) {
     return card.value == 1;
    } else {
     return card.value == pile.top().value + 1 &&
            card.colour == pile.top().colour;
    }
  }
  if ( tableau.contains(pile) ) {
    if ( pile.isEmpty() ) {
     return card.value == 13;
    } else {
     return card.value == pile.top().value - 1 &&
            card.colour != pile.top().colour;
    }
  }
  //...
}
```

Regardless of how expressive one creates the DSL, it should be able to define similar logic to what is shown in 1. So, `foundation order-by increasing-value` should be translated (at least conceptually) to something like `if (foundation.contains(pile))`

`return card.value == pile.top().value + 1;` The most complicated part in this example is to ensure that the right variables are referenced in the translated version (that is, that foundation, card, and pile exists and actually refer to the things we want them to refer to). For more complicated examples, other concerns show up as well. If the DSL we have created allow the rules for individual pile groups to be defined in parts, such as `foundation order-by increasing-value ; foundation base-card Ace`, then we do not want code such as:

```
if ( foundation.contains(pile) )
    return card.value == pile.top().value + 1;
if ( foundation.contains(pile) )
    if (pile.isEmpty() ) return card.value == 13;
```

to be generated, as it the base-card condition will never fire. Issues such as these are in general covered in the generative programming literature (Czarnecki and Eisenecker, 2000).

Based on the feature diagram in figure 3.3, one could create an abstract language looking something like what is shown in listing 2. This is somewhat inspired by the work of (Čeh, Črepinšek, Kosar and Mernik, 2011).

Listing 2 A small sample of the abstract grammar for a DSL extracted from 3.3

```
Deck-Count ::= <INT> ;
Deal-Type ::= All | Waste | Tableau | Reserves | Spider ;
Number-of-Redeals ::= 0 | 1 | 2 | 3 | Unlimited ;
Foundation-Order ::= SameSuit | AltColor | SameColor | ... ;
Foundation-Base :order:= Ace | King | Any ;
Tableau-Order ::= SameSuit | AltColor | SameColor | ... ;
Tableau-Base ::= Ace | King | Any ;
Reserve-Order ::= SameSuit | AltColor | SameColor | ... ;
Reserve-Base :order:= Ace | King | Any ;
Waste-Order ::= SameSuit | AltColor | SameColor | ... ;
Waste-Base :order:= Ace | King | Any ;
```

An example of how one might implement Klondike in a concrete language based of the abstract grammar is shown in listing 3.

Listing 3 An example of what an implementation might look like if it was implemented in a DSL based of the feature diagram presented in figure 3.3

```
deal to waste;
foundation has base-card Ace, order by SameSuit, Ascending;
tableau has base-card King, order by AltColour, Descending;
deck-count is 1;
redeals is unlimited;
```

There is of course a wide variety of what a patience DSL might look like. Our model only requires that the language can interact with the code-base in some way. Thus, many different versions of patience languages can be written and examined using our model.

3.4.3 Language Expressiveness

As alluded to in the previous section, DSLs can be designed with various levels of expressiveness. For example, we might create a DSL that allow all the usual GPL constructs, such as if-statements, loops and assignment, but only allow them to be used for implementing the given methods in the PatienceGame class (that is, no new methods/classes can be defined). This would allow for a lot of potential ways of implementing any given patience game, but would not provide much of the usual benefits of DSLs, such as domain-specific terminology.

For it to be more DSL-like, one would like to introduce domain-specific constructs. Instead of using if-statements and loops, one could use statements such as `foundation ordered-by card.value == pile.top().value - 1` or even more specific, `foundation ordered-by decreasing-value and alternating-colors`.

We can be even more specific. As mentioned above, PySolFC patience suite allow the user to define their own patience games using a simple GUI 'wizard'. As shown in (Bačíková et al., 2013), GUIs like the one in PySolFC can be used to extract DSLs. This would create a very simple DSL, as there is no expression language involved. Each rule can simply have one of a limited set of alternative values. This makes it very straightforward to create new patience games, but at the expense of not allowing as many alternatives.

One question that can be explored using our patience models is what kind of constructs are useful for the patience language? That is, what are good constructs, that support the goal of creating easy to write DSLs for the patience domain. This should be weighed up against not making the DSL so constrained that some games cannot be described at all. For example, the grammar in listing 2 could not be used to specify patience games where the goal is to match cards instead of build on the foundations.

3.4.4 Language Composition

Besides the topics discussed above, our patience model also have some features where language composition techniques (Erdweg et al., 2012; Mernik, 2013) could be useful. In short, language composition is about combining existing languages and language-features to form new languages. For example, one might extend an existing language with a new feature, or combine two distinct languages into one.

In the case of our patience language, there are several parts where these kinds of operations would be useful. We already discussed how one might want to use an existing game specification language to define the patience language, but this is not the only case where composition could be used.

For example, one might want to create sub-languages of the patience language, that can only specify certain variation of a particular patience game (for example, Klondike has a lot of different variations).

One might also look in the other direction. Instead of defining more specific languages, can one take the patience language and make it more general, thus creating a card specification language (such as the one described in (Font et al., 2013)). This is almost certainly not possible without also changing the code-base, so it is not likely to be useful in situations where one is not able to do so. However, if the code-base can be changed (maybe itself through composition (Batory, Lopez-Herrejon and Martin, 2002)), then it might be a way of bottom-up creation of ever more complex languages.

And if we can provide more specific languages and more general languages, then we should be able to use composition to turn the patience language to a language for specifying two-player card games (for example). That is, one can first use composition to extend the patience games to create a language that can specify any card games, and then restrict that language to only be able to specify two-player card games.

The questions then are how can one provide more specific languages for variations of particular patience games? And how can one extend a patience language to be used as a more general language? Another possible question is what does need changing in the code-base in order for it to support more general language?

3.4.5 Language Analysis

One benefit of DSLs is that it can make analysis of the code easier. As stated in (Thibault, 1998): “A DSL has a syntax which ... is restricted in a way that enables automatic program analysis and the further reuse of program design.” So what are some of the things we might want to analyse in the domain of patience games?

One common feature that patience suites have is the existence of automatic solvers. To what extent is it possible to take a patience game written in a patience DSL and automatically provide a solver for the game? To what extent can a solver use different DSLs that all define the same patience game? Finally, if it is not possible to create a solver from the patience DSL code on its own, what additional rules would be needed/useful for creating a solver, and how do they relate to the original patience DSL?

Another area where analysis of a patience DSL might be of interest is in measuring the complexity of patience games. The complexity of Klondike has been discussed in (Longpré and McKenzie, 2009; Bjarnason et al., 2009). To what extent is it possible to determine the complexity of patience games given a description in a patience DSL. This is likely to be quite hard, as (Yan et al., 2005) says “It is one of the embarrassments of applied mathematics that we cannot determine the odds of winning the common game of solitaire.”, as quoted in

(Bjarnason et al., 2009).

A related, but slightly different analysis problem is finding non-playable games. Games that are either too hard/impossible to solve, or games that are too easy to solve, or have some other feature that makes them not very interesting to play. This could aid patience game developers by notifying them that the rules they specified are not playable.

Lastly, analysis can be used to find bugs in the implementation of particular games. There are of course many ways of debugging code, but in this context we are in particular interested in features of the patience domain that provides ways of debugging that is specific to patience games.

3.4.6 Language Evaluation

Evaluation of DSLs is also an important part of DSL development (Gabriel, Pedro and Goulão, Miguel and Amaral, Vasco, 2011; Kosar et al., 2012). The most common way of evaluating a DSL is by using a usability study. This is done by having subjects perform tasks using different DSLs and/or GPLs and measuring the success rate for each task. For example, (Kosar et al., 2009) measure the subjects' ability to learn, perceive and evolve programs written in different languages (but for the same domain). This can of course be applied to the patience domains as well, using our model as a framework for different DSLs to be built on. In (Kosar et al., 2008), rather than measuring DSLs, methods for constructing DSLs were measured. The patience model could also be used for this kind of study, measuring different approaches to developing patience DSLs.

Beyond these, there are two other ways in which DSLs can be measured in the patience model, related to the expressiveness of the DSLs. One way of measuring the quality of a DSL is by measuring how many correct patience games a given DSL can implement. The easiest way of determining this is through the use of a standard list of well-known patience games that the DSLs ought to be able to implement. Such a list can be obtained from either a rule-book (such as (Gibson, 1993; Morehead and Mott-Smith, 2001)) or through existing

patience suites (such as KPat (Team, n.d.), Aisleriot (Gnome, n.d.)). There are well over 100 patience games described in these various sources, though, so creating and maintaining such a list could be quite cumbersome, and might thus not be worth the effort, unless one can show that there would be a significant advantage in using such an approach.

The second way to measure the DSLs is by measuring how many non-patience games that can be implemented. This is not as an important feature of a DSL, as the fact that one can implement things that are not patience games, does not necessarily impact on the quality of the DSL. It is also harder to measure, as one cannot simply create a list of all implementations that are not patience games. It is therefore not likely to be a fruitful area of exploration.

3.5 Conclusions

We have in this chapter presented a simple model to use when considering DSL composition. We have shown the outline of an implementation of a patience suite that is used as a base for the model. The aim with creating this model is to provide a common, executable framework for showing how different DSL development techniques can be used. These ranges from questions about host-languages to the use of composition as a way of creating DSL, as well as questions regarding using DSLs to analyse domains and evaluation of DSLs. The main aim is to provide a platform on which future work can rest and we hope that this model will be able to aid future research into the areas of DSL development discussed in this article.

The patience model has already been used once to discuss a development technique for DSLs (Kihlman, 2015).

Chapter 4

Basic framework for language composition

4.1 Introduction

This chapter presents a basic framework for language composition and how it can be used to define Domain-Specific Languages (DSLs). It is to a large degree inspired by the work in (Erdweg et al., 2012) as well as subsequent work such as (Mernik, 2013; Völter, 2013a; Chodarev et al., 2014; Degueule et al., 2015).

Language composition consists of creating new languages by composing them from existing languages. That is, a new language is created by combining elements of existing languages using a set of predefined composition operations. The benefit of using language composition is that creating a new language can in general be difficult and language composition makes the process easier by allowing existing language features to be reused in new languages.

The framework we present here specifically deals with composition of abstract languages, as well as translation of one abstract language to another. We view languages as consisting of concrete syntax, abstract syntax, and semantics. The concrete syntax consists

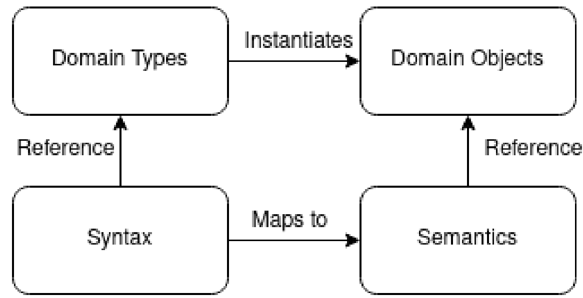

Figure 4.1: A diagram for concrete syntax being directly mapped to semantics, along with domain types that are instantiated to domain objects that can then be referenced by the semantic domain.

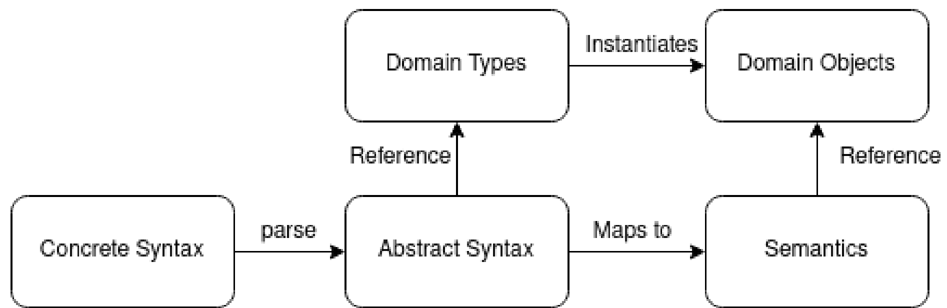

Figure 4.2: A diagram showing concrete syntax being parsed into abstract syntax, which is then executed in the semantic domain. In this case, the concrete syntax is decoupled from the domain types, which instead are referenced to in the abstract syntax.

of the grammar that defines valid (concrete) sentences in the language. The abstract syntax describes a structure of the language which contains only the information required to distinguish two statements in the language (unlike concrete syntax, which usually have so called 'syntactic sugar' associated with it). The semantics describes the meaning of the abstract syntax structures.

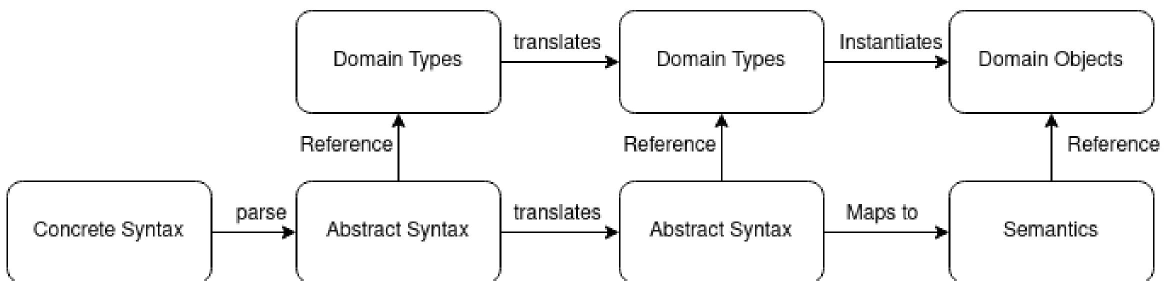

Figure 4.3: A diagram showing concrete syntax being parsed into abstract syntax which is then translated into a different abstract syntax, which is then executed in the semantic domain. The domain types also get translated between abstract syntaxes.

Since we work with DSLs; languages that are specific to particular domains, we also need to consider how the domain is modelled in all of this. In general, we take a pragmatic approach to how we model the domain, and view it in terms of types or classes and instantiations of those types or classes. In some examples, in particularly in this chapter, we also use mathematical sets to describe domains. In general, our approach should be fairly agnostic to how the domain is modelled, as long as it is possible to have well-defined subtypes and supertypes of members in the domain.

In figure 4.1 we show how our language and domain model would work in the case of a direct translation from concrete syntax to a semantic domain. Here, the domain types are declared in the syntactic side and their instantiations are referred to in the semantic domain.

As we work with abstract language, we need to add the abstract syntax in between concrete syntax and semantics. This is shown in figure 4.2. In this case, the declarations and references to the domain types is moved to the abstract language, as the concrete language only serves as a translation layer between the human input and the abstract representation in the machine that processes the code. Other than that, the system works the same.

Lastly, as we work with translations of the abstract language, we show in figure 4.3 how a translation works in our system. Here, we have two different abstract syntaxes, one coming from the concrete syntax, and one being defined by the semantics, with a translation in between them. When we translate between abstract languages, we may also have to translate the domain types in some manner. The domain types that one abstract language works on may be different in the abstract language that the first language is translated into. The second language might, for example, use a more complex typing system. This is not necessarily the case, though, and the types in the first language might well be completely compatible with the types in the second, without translation. For the purpose of this chapter we will assume this is the case. We do think it is important for future work to point out the possible need for domain translation as well as language translation.

One of the benefits of using an abstract language in between the concrete syntax and

the semantic domain is that we can then provide multiple concrete syntaxes for the same abstract syntax, without worrying about changing the semantics. It also allows us to provide multiple semantic definitions of the abstract language without worrying about the concrete syntax. This means that once you have an executable language (that is, one with defined semantics), that language can be used with multiple different concrete syntaxes which will only require the mapping to the abstract language. Then, if a new semantics is added, all defined concrete syntaxes are immediately executable in the new semantic domain.

Another benefit of abstract language is that it is easier to compose than concrete syntax, as it is easier to ensure unambiguousness in abstract language than in concrete language. Thus, the effect of operations on abstract language is easier to reason about than operations on concrete language. To create the full language, one will eventually need to define the concrete syntax though, and when doing so similar pitfalls in regard to ensuring unambiguousness as when composing. Our approach thus relies on the hope that defining a concrete syntax after composition of abstract language is easier than ensuring unambiguousness during composition of concrete languages. We do not show that this is the case, though.

Another issue that needs clarification is the difference between compiler and interpreter. This work is partly about translating between abstract languages, which would generally be considered compiling or transpiling. Compiled DSLs are one among several ways to implement DSLs, as given in for example (Mernik et al., 2005). According to (Kosar et al., 2016), who base their classifications on (Mernik et al., 2005), found that around 28% of papers published between 2006 and 2013 (which is the years the study looked at) were discussing compiled languages. Around 8% were interpreted languages, 15% were pre-processor languages and around 34% where Domain-Specific Embedded Languages (DSELs).

We argue that our approach, though closest to compiled DSLs, can be used for interpreted languages just as well, and with some extra work can also work with pre-processor languages. It could also be used to model DSELs, however, as one of the reason for using

embedded languages is to avoid using external tools (Hudak, 1997a), using our framework would probably be counterproductive (a DSEL modelled in our framework would be equivalent to a pre-processor language that consists only of host to host translations).

We will here discuss compiled and interpreted languages, and will later on discuss pre-processor languages and comment on DSELs. The distinction between compilers and interpreters is nowadays somewhat blurred, especially with the introduction of Just-In-Time (JIT) compilers into interpreters, and so it can be a bit hard to discuss what difference each approach makes. We discussed the use of an executable language to define the semantics and for our purposes it does not matter whether the executable language is one that is interpreted, or one that compiles into machine code. As such, the distinction between them does not matter.

4.1.1 Motivating example

Imagine that we have an execution environment E_h, capable of executing sentences in some language L_h, and we have a language L_1 that is, at least in part, composed from L_h, how much extra effort do we need in order to make L_1 executable in E_h?

The way we make L_1 executable in E_h is through translating sentences in L_1 into sentences in L_h. Thus, what we want to do is minimise the effort to create a translator T_{1h} which translates sentences in L_1 into sentences in L_h. Inevitably, there are many ways of translating sentences in one language to another language, and each way may require a different amount of effort. We determine whether less effort is needed is by considering the case when there is no known relation between L_1 and L_h, and we thus have to translate everything. We then consider something to require less effort, if we can provide a way to reduce the amount of translation without knowing anything about the original translation rules. Thus, we only consider simplifications that do not require any knowledge of the original translation rules. Thus, we know for certain that our translation requires less effort than any original translation, regardless of how simple or complex that translation is.

4.2 Background

There exist several DSL composition frameworks, such as (Erdweg et al., 2012; Völter, 2013a; Lakatos and Porubän, 2013; Degueule et al., 2015). They tend to discuss composition of concrete language, but some, like (Chodarev et al., 2014), discuss composition of abstract language. For a more in-depth discussion on the different approaches, see chapter 2, subsection 2.4.1.

We will mainly compare our work to Erdweg et al., who defines the following composition operations:

- Language extension – extending an existing language with new constructs
  - Restriction – a special type of extension, where the extension restricts the new language
- Language unification – combining two languages to form a new language
- Self extension – languages that are capable of defining their own extensions
- Composition of above – a combination of the above composition types

The main operations we will discuss are given in the list below, though we will also mention some more general operations in subsection 4.4.1.1.

- Introduction
  - Declaring the structure of a construct.
- Restriction of construct
  - i.e. sub-typing an existing construct
- Relaxation of construct
  - i.e. super-typing an existing construct
- Deletion
  - i.e. creating new language by removing an existing construct

- Combination
  - The union of two languages
- Translation
  - Defining the meaning of a construct in terms of other constructs.

Of these, the first three are on individual constructs, and the last three on languages as a whole. Note that the combination operation is similar to Erdweg's language unification and language extension classifications. Restriction in our case share the name with Erdweg's restriction, but Erdweg defines restriction as a special case of extension and Erdweg's classifications tend to be more on the language as a whole than on individual parts of the language. Relaxation is similar to extension, but again, our relaxation operation is for specific constructs whereas Erdweg's extension is on a language as a whole.

One type of composition that Erdweg describes is self-extension. This is not possible within our framework, as we view each operation as creating a new language, thus the idea of being able to extend itself makes no sense (as the output of the extension would be a new language, not the old language with a new operation). A similar argument can be made about Erdweg's extension composition, though as long as one sees the output of an extension composition as a new language, that is like the old language, but with an extension, the classification could apply to our operations as well.

Erdweg's focus is also more on concrete syntax than abstract syntax (though there is no distinction being made in Erdweg's work, and the classifications can be applied to either) whereas we specifically target abstract syntax.

While we discuss these operations as in the terms of DSLs, there is no reason they could not also apply to General Purpose Programming Languages (GPLs).

4.3 Domain and language

We will in this chapter distinguish between domain and language. There is no absolute definition of what a domain is and what a language is in the context of DSLs, and to some degree it does not affect the general discussion on DSLs. In our case, however, we see a benefit in distinguishing the two.

Schmid (2000) provides the following definition for a domain: "A domain is defined by the objects, operations, and relationships that domain experts perceive to be important for developing systems for a certain area of functionality."

In general, domains are made up of 'things' and relations between the things. Languages on the other hand describe those things and their relations. To complicate this a bit, languages are themselves part of the 'domain of languages' and thus some languages describe the domain of languages and thus (potentially) describe themselves. Languages that work in the domain of languages are called meta-languages.

A stumbling block when thinking about the distinction between domains and languages is that we cannot discuss domains without a language to describe the things in the domain. Take for example this '1'. It represents the concept of 'one', but is not itself that concept, just a way of representing that concept.

We see domains as a set of types, and their relations are, for example, functions that operate on those types. Languages are then represented by a special set of types, which we call language constructs. Instances of these types are Abstract Syntax Trees (ASTs). Language constructs are either atomic constructs, or compound constructs consisting of one or more member constructs. Thus, language constructs only refer to other language constructs.

The way language constructs describe domain types is through side-effects. So, for example, the sentence `class ClassName {...}` would as a side-effect introduce a domain type by the name 'ClassName' in the current scope. Note that 'class' is a compound language construct, and 'ClassName' is an atomic language construct. The ellipsis represents

the body of the type, also specified using different language constructs.

Thus, domain types are not directly referred to, but can be determined from the side-effects that the language constructs have.

Another common side-effect that language constructs have is the 'evaluates-to' side-effects. This represents the case when a language construct produces as a side-effect a value of a domain-type, that may then be operated on by other language constructs.

A meta-language would in this logic be a language where the side-effects also refers to language constructs. So, a language that has constructs that evaluates to other language constructs.

In this chapter we only look at composition and translation of language types, though similar logic could be applied to domain types as well. For this reason, we will only briefly discuss side-effects and how they fit in with the rest of the framework.

4.4 Abstract Language

For our approach, we view abstract languages as being made up of a set of abstract language constructs. The language constructs are made up of the name of the construct and the definition of the structure of the constructs. This is like the irregular heterogeneous AST combined with homogeneous AST as described in (Parr, 2010). This is either a set of members with names and types, or a list with the type of elements, or a type for a single value. Some constructs can be root constructs, which are constructs that appear at the root of the ASTs that these constructs describe.

For the language to be consistent, there has to be at least one construct from which one can reach all the others, either by having one of the members/list-type directly refer to the language construct type, or indirectly through sub-typing. Language construct types that are not reachable can not be part of any sentence and are thus pointless. There cannot be any references to non-existing constructs either. A direct reference is when a construct is being

explicitly specified in another construct. Indirect references are constructs that are subtypes of constructs that are directly referenced.

A complete language is a language that is consistent and has at least one root construct from which all other constructs are reachable. If a language is consistent, but not complete, then it is a language fragment. Language fragments are not executable, but can be used to build up more complex and executable languages.

Another type of language is a language that contains a construct using 'Any', which have no subtypes without the 'Any' specifier. The 'Any' specifier is used to allow the specification of language constructs where the structure of the construct does not matter. The 'Any' specifier is the supertype of all language structures and does not have an instantiation. It is useful for specifying expressions, for example $E := Any \rightarrow x$ would mean that subtypes of E contain all language constructs that evaluate to x, regardless of their structure. A complete language that has such a specifier that has no subtypes without the specifier can not be executed, as there is at least one construct referenced that can not be instantiated, as it has no specified structure. Such languages can be useful for providing general structures, without providing all the concrete operations. The simplest of such languages is a general expression language $Expr := Any \rightarrow Value$ where $Expr$ is a root construct and value is any value in any domain.

The subtyping rules for the constructs are similar to subtyping rules for record types. We use nominal subtyping, as constructs have meaning beyond their structure and using only structural subtyping would thus be misleading in some instances, where two constructs that have the same structure mean quite different things.

The rules for *x is a subtype of y*, where x and y are language constructs, are:

- x and y are value constructs and x is declared to be a subtype of y.
- x and y are list constructs and the list type of x is a subtype of the list type of y
- x and y are member constructs and they have the same members and one or more of the members of x are subtypes of corresponding members in y.

Constructs can also have side-effects, which are things that happen when the construct is executed, aside from the main effect of execution. These include introduction of new type-references and evaluating to a value. The side-effects are used to model things in languages that are not easily modelled through the structure of the language construct. Each type of side-effect has different rules for how subtyping it works.

To summarise, our model of the abstract language consists of a set of abstract language constructs (or abstract language types) that consist of a name, a structure and side-effects. The instance of these language types are ASTs.

We write the abstract language types as shown in equation 4.1. The side-effect can be labelled, as is shown in equations 4.2, 4.3. The labels that we are going to discuss are global, local, and labels beginning with ':'. The global level means the side-effect affects the entire program, such as inserting a symbol in the global scope, for example a class or function declaration in the global scope in most GPLs. The local label means the side-effect affects the local context. For example, variable declarations in a function. Labels beginning with ':' are internal and only affect the children of the construct in question. An empty ':' (that is a colon on its own) means it affects all children. Leaving out the label gives the special meaning of 'evaluates to'. That is, the side-effect only exists right at the end of the execution of the construct and ceases to exist after the next construct is executed. This is the same as expression evaluation in most languages. The most general of all expressions is simply $Expr := Any \rightarrow Type$ (that is, an expression of any structure that evaluates to some type). All other expressions are subtypes of this $Expr$.

$$Name := Structure \rightarrow Side\text{-}effect \tag{4.1}$$

$$Name := Structure \xrightarrow{global} Side\text{-}effect \tag{4.2}$$

$$Name := Structure \xrightarrow{:lhs} \textit{Side-effect} \tag{4.3}$$

The structure part of the construct is either a labelled set of language types, or a list of ASTs of language type, or without structure. We write this as is shown in equation 4.4.

$$Structure = \begin{cases} \{label_1 : Type_1 ... label_n : Type_n\} \\ \{Type\} \\ \varepsilon \end{cases} \tag{4.4}$$

The side-effects we are going to discuss are either a labelled type, meaning the side-effect inserts a symbol with the label as name and with the specified type in the relevant context, or is simply a type, which is only used for the 'evaluates to' side-effect and it means that the construct evaluates to the given *type*. Note that here *type* refers to a domain type, not a language type, as was the case in the structure.

$$Side-Effect = \begin{cases} label : Type \\ Type \end{cases} \tag{4.5}$$

For example, in equation 4.6 we show an example of how we use this. Here we define a construct E, with two members n of type F and m of type G, which as a side-effect evaluates to a domain type T.

$$E := \{n : F, m : G\} \rightarrow T \tag{4.6}$$

If we are not interested in every part of the construct we simply leave the uninteresting part out, as seen in equations 4.7, 4.8, 4.9, if we are only interested in the name and side-effect, structure and side-effect or name and structure. We can also just specify the individual parts on their own, as E, $\{n : F, m : G\}$, and $\rightarrow T$ (note that the arrow and side-effect type are always together, as they both provide meaning to the side-effect as a whole). If E refers to an

already specified language type, then equation 4.7 should be read as ‘E with the previously defined side-effect, here named T’. Likewise for the equation 4.9.

$$E \to T \tag{4.7}$$

$$\{n : F, m : G\} \to T \tag{4.8}$$

$$E := \{n : F, m : G\} \tag{4.9}$$

We also need to consider some subtyping rules for the language constructs. Two list constructs’ structure are subtypes of each other if their list-types are subtypes of each other, as shown in equation 4.10.

$$\frac{T_1 \leq: T_2}{\{T_1\} \leq: \{T_2\}} \tag{4.10}$$

Two labelled constructs’ structure are subtypes of each other, if they have the same members and they are all (non-strict) subtypes of each other, as shown in equation 4.11.

$$\frac{T_1 \leq: U_1 \ldots T_n \leq: U_n}{\{n_1 : T_1 \ldots n_n : T_n\} \leq: \{n_1 : U_1 \ldots n_n : U_n\}} \tag{4.11}$$

Structureless constructs are always equal in their structure (as they all have none), which means they are always non-strict subtypes of each other.

With the side-effects we are discussing in this chapter, we have two different cases, one for *internal and evaluates to* and one for *global and local side-effects*. The *evaluates to* side-effect is a subtype of another *evaluates to* side-effect if their types are subtypes. This is shown in equation 4.12.

Internal side-effects do not affect the subtyping, since when a construct is replaced, so is its entire internal structure. Thus, only side-effects that affects the surrounding language

construct affects the substitutability of the construct.

$$\frac{T \leq: U}{\rightarrow T \leq: \rightarrow U} \tag{4.12}$$

For global and local side-effects, it is not possible to determine subtypes in general. In fact, we are unaware of any global or local side-effect that can be subtyped.

As discussed earlier, we use nominal subtyping throughout this thesis. One way to look at is to view the name, structure, and side-effect as being three separate parts. The case when the structures and side-effects are subtypes was discussed above. For the names to be subtypes, we have to declare them to be subtypes. Of course, declaring the names to be subtypes only makes sense if the structure and side-effect are also subtypes. In short, a construct is a subtype of another construct if the structure and side-effects are subtypes, and if they have been declared to be subtypes of each other.

For the remainder of this chapter, we will ignore side-effects, as our main concern is on the composition of the structure of the language constructs. To consider the composition of side-effects, we would have to consider the composition of domain-types, which are outside of the scope of this chapter (and thesis as a whole) as it would at the very least double the amount of work needed to do it justice.

4.4.1 Composition of abstract language

When we compose new languages, we want to ensure that the new languages are consistent. We also want to keep track of the sub/super type relations between languages as a whole as well as the individual constructs within the languages being composed.

Equation 4.13 shows the condition for a language to be valid. All constructs in L reference only other constructs in L.

$$\forall e \in L \forall t \in types(e) | t \in L \tag{4.13}$$

Before we continue the discussion, we need to define what we mean by reachable. This is shown in equation 4.14. Here t is reachable from e in the language L, if

- e is equal to t
- t is a subtype of e
- t is reachable from any of the member types of e
- there is a subtype of e from which t is reachable

This is shown in formal notation in equation 4.14.

$$reachable(e,t) := \begin{cases} e = t \\ t \leq: e \\ \exists n \in types(e) | reachable(n,t) \\ \exists f \in L | f \leq: e \wedge reachable(f,t) \end{cases} \tag{4.14}$$

Equation 4.15 shows the condition for a language to be consistent. A language is consistent if all constructs in it are reachable from at least one other construct.

$$\forall e \in L \exists f \in L | e \neq f \wedge reachable(f,e) \tag{4.15}$$

Equation 4.16 shows the condition for a language being complete. A language is complete if all constructs in it are reachable from at least one other construct that is a root construct.

$$\exists e \in L | \text{is_root}(e) \wedge \forall t \in L | reachable(e,t) \tag{4.16}$$

In equations 4.17 and 4.18 we show what we mean by direct and indirect references respectively. A direct reference is a reference where one construct has a member that is the type of the other construct, an indirect reference is when one construct has a member that is

a subtype of the other construct.

$$t \in types(e) \tag{4.17}$$

$$\exists f \in types(e) | t \neq f \wedge t \leq: f \tag{4.18}$$

When we compose our languages, there is a choice between two approaches. We can either first completely specify the structure of the language, and then declare the relations of the language to another language. Or we can take an existing language and use our composition techniques to construct a new language out of the existing one. That is, we allow the declared constructs to be recomposed at a later time if need be, so it is possible to state that an existing construct should be treated as being composed from another construct.

The composition operations can either be used to state that some new construct C' is the result of some composition operation on some set of existing constructs $\{C_1...C_n\}$ or we can say that the existing construct C should be treated as if it was the result of an existing operation on some set of existing constructs $\{C_1...C_n\}$.

The reason we allow both approaches is that we might want to reuse the structure of a composed language, but recompose onto another language, in order to benefit from the properties of the new language. That is, we want to be able to map a single language onto several different domains, while being able to reuse the translation rules for languages that work in those domains. By allowing recomposition of language construct based on different languages, we can reuse the translation rules for those languages.

The first operation we are going to describe is the introduction operation, which was shown above in equation 4.1. This simply declares the name and structure of a language construct. Strictly speaking, it is not a composition operation, but we include it in order to be able to construct languages that are not completely composed from existing languages.

For the language to remain consistent, the constructs referenced in the introduced con-

struct needs to exist in the language that the new construct is being introduced into and the new construct needs to be a root construct or else referenced (directly or indirectly) from an existing construct. The last case is not possible to achieve in one go, so with introduction of non-root constructs, the best we can achieve is eventual consistency.

The next operation is restriction of a construct. This creates a new construct (or recomposes an existing one) by changing the type of some of the members or the list type with a subtype. The new construct is a subtype of the existing construct. A language under restriction, remains consistent if the type being replaced by a subtype is still being referenced from a different construct. Thus, we cannot completely restrict away a construct in the language. Together with the deletion operation, we can achieve eventual consistency in all cases of restriction.

The reason to use restrict is to have constructs that are closer to the intended domain (as opposed to being close to the implementation domain). That is, we want the construct to match the preferred abstraction level of the domain (as described in (Kahraman and Bilgen, 2015)).

Restricting might also make it easier to analyse programs written in the DSL by removing constructs that are difficult to reason about. It might also reduce the error surface, by reducing the amount of things that can produce errors. Another possible benefit might be that it is easier to optimise more constrained constructs. We will not discuss these properties in this thesis, though.

$$E' = E := \{n : E_1\} \cap \{n : E_2\} | E_1 :\geq E_2 \implies E' \leq: E \tag{4.19}$$

In equation 4.19 a new construct E' is created by restricting a construct E with a child ‘n’ of type E_1 is restricted by setting the ‘n’ child to type E_2. Assuming that E_1 is a subtype of E_2 (which is required for the restriction operation), E' will be a subtype of E. List and side-effect restriction works similarly, as shown below. Atomic constructs do not have anything to restrict, so no equation is shown for them. They are simply declared to be subtypes if

needed.

$$E' = E := \{E_1\} \cap \{E_2\} | E_1 :\geq E_2 \implies E' \leq: E \tag{4.20}$$

$$E' = E \to E_1 \cap \to E_2 | E_1 :\geq E_2 \implies E' \leq: E \tag{4.21}$$

Relaxation, which is the reverse of restriction, can be also be done, where the members are replaced by a supertype, which makes the new construct a supertype of the existing one. This can be useful if an existing language is too restricted for what you need it to be. The relaxation composition always produces an inconsistent language, as the new construct can neither be directly referenced (due to being a new construct) nor indirectly referenced (it is a supertype of a construct, but not a subtype of any construct).

The relax operation exists mostly to provide symmetry with the restriction operation, but there may be cases where creating a supertype construct may make sense, for example if one wants to create a common base language from two existing languages that share similarity, but not enough to have one be the base of the other. The common base language can then be used to define all the common parts of the two languages, so that the original languages do not both need to have defined their own versions of these parts.

$$E' = E := \{n : E_1\} \cup \{n : E_2\} | E_1 \leq: E_2 \implies E' :\geq E \tag{4.22}$$

As can be seen in equation 4.22, relaxation works in the same way as restriction, but E_1 has to be a subtype of E_2 and the result is that E' is a supertype of E. The same applies for list and side-effect relaxation.

$$E' = E := \{E_1\} \cup \{E_2\} | E_1 \leq: E_2 \implies E' :\geq E \tag{4.23}$$

$$E' = E \rightarrow E_1 \cup \rightarrow E_2 | E_1 \leq: E_2 \implies E' :\geq E \tag{4.24}$$

The combination operation is used to take two sets of constructs from different languages and combine them into a new language. For the language to remain consistent, the constructs need to, in some way, be able to refer to each other, either directly or indirectly. This might either be due to them both referring to the same existing common language constructs or through the addition of constructs that bind the two languages together.

$$L' = L_1 \uplus L_2 | \forall e \in L' : e \in L_1 \vee e \in L_2 \tag{4.25}$$

We can delete a construct, as long as it is not directly referenced by any of the other constructs. That is, only indirectly referenced constructs can be deleted while keeping the language consistent. Restriction can be used to remove references from a language, which can be used in combination with the delete composition to completely remove a construct from a language, while keeping the language consistent. Note that while the relaxation composition never completely removes a reference, it can be used to create an indirect reference, which can then be removed from the language (assuming no direct references remains).

$$L' = L_1 \ominus E | (\forall e \in L' : e \in L_1) \wedge (\forall e \in L_1 : e = E \vee e \in L') \wedge E \notin L' \tag{4.26}$$

4.4.1.1 Higher/more abstract level

There are some additional operations that we could use, but for various reasons we will not discuss in detail. Here we will discuss some of them.

First, we could have a replace operator (as seen in equation 4.27, with list constructs). This operator creates a new construct by replacing one of the members of an existing construct. If the replaced member is a subtype of the original, it is equivalent to the restrict operator and if it is a supertype it is equivalent to the relax operator. The problem with this

operator is that it does not give us any way of reasoning about the relationship between the original construct and the composed one. It is essentially just a more convenient introduction operator for our purposes.

$$E' = E := \{E_1\} \odot \{E_2\} \tag{4.27}$$

Another type of operators are those that add or delete members to structured constructs. These are shown in equations 4.28 and 4.29. These could be used in our framework, where the add operation creates a subtype and the delete operation creates a supertype, but they were left out for simplicity's sake and due to the fact that reuse of translation rules, while working for the add operator, makes little sense, as any reused translation rule will ignore the added member, which is probably not what is wanted. In other words, more work is needed to for it to make sense to include these operators in our framework.

$$E' = E \oplus \{n' : E_2\} \tag{4.28}$$

$$E' = E \ominus \{n' : E_2\} \tag{4.29}$$

There are also some convenience operators, such as the equality operator that declares two constructs to be the same, as shown in equation 4.30. This implies that anything that can be done with one of the constructs can also be done with the other.

$$E_1 \in L, E_2 \in L : E_1 = E_2 \tag{4.30}$$

The rename operator changes the name of a construct to a new name. This can be modelled using the equality operator and the global replace operator as shown in equation

4.31.

$$E_1 \in L : E_2 = E_1 \wedge E_1 \updownarrow E_2 \tag{4.31}$$

The last operation is a one-line composition, which includes combination, deletion, and composition of individual constructs into one. It is mostly a conceptual operation, the point of it being that the result is a *complete* (or at least *consistent*) language, without having any inconsistent languages 'created' in between. The operation consists of first combining together all existing languages from which the new language will be created, then listing all additional single construct compositions, including equating any constructs that exist in both languages, then renaming anything that needs renaming. At the end, any construct that is not needed gets deleted. This is shown in equation 4.32. We have no general way of guaranteeing that the output language is complete or consistent, but checking it after composition is straightforward.

$$L' = ((L_1 \uplus L_2) : E_y \odot E_x ... E_a \odot E_b) \ominus (E_1 ... E_n) \tag{4.32}$$

Sometimes the restriction and relaxation operations can be used to introduce new constructs. That is, if we have an existing language L in which we want to restrict some constructs in with a construct/set of constructs from another language L_2, we would normally first construct a new language $L' := L \uplus L_2$ and then specify which constructs are restricted with constructs from L_2. When we talk about this kind of combined operations on a higher level, where we don't care about the details of which constructs get restricted/relaxed, we use the restrictions/relaxation operator on the language level. That is, if L_1 and L_2 are languages, the sentence $L_1 \cap L_2$ means construct a new language based on L_1 where some of the constructs are restricted to constructs in L_2. This also implies that the constructs in L_2 do not already exist in L_1. Similarly, if C is a construct, the sentence $L_1 \cap C$ means construct a new language based on L_1 where at least one construct is restricted by C.

4.4.2 Translation

The translation operation consists of a set of rules setting out how to turn constructs in one language into constructs in a target language. This can be done manually for each new language created, but since we are composing languages from already existing ones, we could benefit from reusing the rules used by the existing languages from which our new language is composed. This assumes that there exist rules for translating the existing languages into our chosen target language. Given the existence of the rules, we also have to ensure that they do not conflict when applied to the new composed language.

In general, it is impossible to guarantee that the rules do not conflict, as general translation is not closed under composition. It is, however, possible to constrain the translation to allow us to create a translation operation that is closed under composition.

Under what circumstances can we guarantee that the translation is valid? The translation is valid if all valid sentences in the composed language can be translated into valid sentences in the target language. To determine this, we need to not only have a description of the composed language, but also the target language. We also need to have a way to determine whether the output of the translation rules describes valid sentences in the target language.

We view the translation rules as a set of rules of the format $C ::= T_{expr}$ where C is the language construct in the composed language and T_{expr} is a translation expression, specifying how the construct should be translated. To determine validity, we need to check that each T_{expr} generates a valid sentence in the target language, and that the combination of several different T_{expr} also produces valid sentences. To ensure that the combination of the rules also produces valid sentences, we start with the root construct in the composed language, which needs to produce a root construct in the target language. As our translation rules encode the output type, we can require that only translation rules that produces output that is valid in the context that the translation rule is used are allowed, thus ensuring that all the children of the root construct are translated into valid sentences and that they only occur in places where they are allowed within the root construct.

The rules for determining validity of the output of individual T_{expr} rules is more complicated. A T_{expr} always consists of one root construct, which determine the overall type of the T_{expr} and thus the type of the $C ::= T_{expr}$ rule.

To ensure that the sentences in the target language will type-check under composition, it needs to be possible to fully type-check each T_{expr} without considering any other translation rule. That is, it has to be possible to do all type-checking locally. In general, type-checking tends to be non-local, such as references to types or functions in global scope. As discussed in (Eden et al., 2006), composition is not possible in general when there exist non-local dependencies. Thus, to allow composition in general, only type-checking rules that are local can be allowed in the target language. It may in some cases be possible to circumvent non-local dependencies, but in general type-checking can not be guaranteed when non-local type-checking dependencies exist. Even with non-local type-checking, it may still be possible to translate composed languages while reusing translation rules, but we cannot guarantee that the translated sentences will type-check.

The subtyping rules for translations are similar to function subtypes. A translation rule $C ::= T_{expr}$ is similar to a function $T(C) \rightarrow T_{expr}$. $C_1 ::= T_{1_{expr}}$ is a subtype of $C_2 ::= T_{2_{expr}}$ if C_1 is a supertype of C_2 and $T_{1_{expr}}$ is a subtype of $T_{2_{expr}}$. Note the inverse relationship between C_1 and C_2. Since we normally want the same input to produce the same output, we usually require $T_{1_{expr}}$ to be equal to $T_{2_{expr}}$. While all the logic works the same in either case, if we do not require equality, then there will be some input values that are the same for the subtype and supertype translation, but will get translated to different outputs. This makes it harder to reason about the output, when translation rules get substituted.

Since the argument to the translation rule has the inverse relationship to the rule itself, we can always replace the translation rule for a subtype construct with that of a supertype construct. This makes intuitive sense, since the supertype can take all the values the subtype can take, and so can always provide a translation for any value the subtype can take.

If we have a translation rule for a subtype construct, we can create a rule for the supertype

by creating a translation rule $T_2(s \in C_2) ::= T_1(s), s \in C_1; T'(s), s \notin C_1$. That way we only need to provide the translation for cases where the construct is not in C_1. This also ensures that the same input is translated to the same output in both rules.

The T_{expr} part of the translation rule can take many forms. The simplest is for it to consist of a construct instance (AST) of the output language, with additional expand statements that show where a given member of the input construct should be expanded in the output construct. It could also be more complex, with conditional expansion, or arbitrary expansion with additional helper functions and records to keep track of what has been expanded before and then to expand the next construct bases on some arbitrary decision.

To keep the translation rules for languages compatible, we would ideally want the property $c_1 <: c_2 \Rightarrow T_1(c_1) <: T_2(c_2)$ to always hold. If this property does not hold, it is likely that translation rules that depend on the T_2 translation will break and not produce valid ASTs if c_2 is replaced by c_1 in the input AST. When we combine constructs from many different languages, it is inevitable that at least some translation rules will not conform to this rule, unfortunately.

Before we move on, we will quickly define some functions that we use when describing the rest of the composition framework.

- name(c: Construct) - given a language construct, return the name of the language construct
- members(c: Construct) - return the names of all members in a language construct
- types(c: Construct) - return the types for the members' language construct
- member_types(c: Construct) - return a tuple of names and type for all members in a language construct
- reachable(from: Construct, to: Construct) - determines if the second argument is reachable from the first argument
- is_labelled(c: Construct) - returns true if the language construct is a labelled construct
- is_list(c: Construct) - returns true if the language construct is a list

- is_atom(c: Construct) - returns true if the language construct is an atomic construct
- list_type(c: Construct) - returns the containing type of a list construct
- side_effects(c: Construct) - returns the side-effects
- eval_to(c: Construct) - returns type for the evaluate to side-effect

All of these can also be used with instances of construct (i.e. ASTs).

We now want to show that if a rule for translation exists for a supertype, it can be reused by a subtype without breaking the translated program. This is not possible for a general translation function as shown in equation 4.33, so we shall construct a simple translation expression language that has this property. The type for the translation expression is shown in equation 4.34.

$$T(s \in S) \rightarrow H \tag{4.33}$$

$$E := \begin{cases} \{n_1 : (E_1 \rightarrow H_1) ... n_n : (E_n \rightarrow H_n)\} \rightarrow H & is_labelled(H) \wedge \\ & n_1 ... n_n \in members(H) \wedge \\ & H_{\{n_1\}} :\geq H_1 ... H_{\{n_n\}} :\geq H_n \\ \{E_1 \rightarrow H_1\} \rightarrow H & is_list(H) \wedge \\ & H_l \leq: list_type(H) \\ \varepsilon \rightarrow H & is_atom(H) \\ T'(s \in S) \rightarrow H & S_1 \leq: S_2 \implies T'(S_1) \leq: T'(S_2) \end{cases} \tag{4.34}$$

This is the same as equations 4.35, where E_{label}, E_{list}, E_{ε}, and $E_{general}$ are all subtypes

of E.

$$
\begin{aligned}
E :=& Any \rightarrow H \\
E_{label} :=& \{n_1 : (E_1 \rightarrow H_1)...n_n : (E_n \rightarrow H_n)\} \rightarrow H_{label} \\
E_{list} :=& \{E_1 \rightarrow H_1\} \rightarrow H_{list} \\
E_{\varepsilon} :=& \varepsilon \rightarrow H_{\varepsilon} \\
E_{general} :=& T'(s \in S) \rightarrow H_{general}
\end{aligned}
\tag{4.35}
$$

In both 4.34 and 4.35 we specify that a translation expression is one of

- labelled construct
- list construct
- structureless construct
- a general translation function

Labelled constructs that evaluates to a domain type H, where H is a labelled construct, which contains all the members of the E construct, and the evaluation types for each member in E are subtypes of the corresponding member in H. List constructs evaluate to domain type H, with list type of $E \rightarrow H_l$, where H is a list type construct and H_l is a subtype of the list type of H. Atom types evaluate to H, where H is an atomic construct type. Lastly, the general construct consists of any function that evaluates to a domain type H, as long as the output is a subtype when the input is a subtype.

We will now consider a translation function where each translation rule is a construct in the input language associated with an E construct, where the E construct contain references to the children of each construct in S and the E expression is used to generate sentences in H. We consider the case where the property expressed in equation 4.36 holds.

$$
T(s \in S_1) = E_1 \wedge T(s \in S_2) = E_2 \wedge S_1 \leq: S_2 \implies E_1 = E_2 \tag{4.36}
$$

That is, if the input to T is subsets, then the E construct used is the same. In other words, subtypes share the same rule.

Let us first discuss how the constructs in E are evaluated to produce the output constructs. They are evaluated are by replacing the values of all the children with the result of their evaluation. Since each E construct has the same children as the H construct they are evaluating to and each child of the E constructs evaluates to the type of the corresponding child in the H construct, this will correctly generate the structure of the H construct. If we only had E constructs for labelled, list and atomic constructs, then this would be enough to always correctly generate the structure of the H construct. However, as we want to be able to construct our output constructs based on the structure of the input constructs and as the label, list and atomic E constructs have no way of referencing the structure of the input, we provide the T' function to allow us to construct translations that references the children of the input constructs.

The T function represents all constructs whose evaluates-to side-effect is dependent on the children of the construct being translated. A simple example of such a construct would be a translate construct $translate := child : ID \to H$ which takes the identifier for a child of one of the input constructs and looks up the translation rule for translating the child and applies it to the child and returns the result. So, if we have the input constructs $S_1 := n : S_2$ and $S_2 := \varepsilon$ and the translation rules $S_1 ::= translate(n)$ and $S_2 ::= 1$, then the translation rule for S_1 would translate to the translation of S_2 (which we arbitrarily set to translate to the integer 1). The type the translate construct translates to is dependent on the type that is passed to it.

As we have mentioned, the T function has to have the property that $T(S_1) <: T(S_2)$ when $S_1 <: S_2$. For the translate construct, this is easy to show, as we have restricted us to always use the same translation rule for all subtypes and as the same translation rule will always translate to the same type (or a subtype) for any subtype.

To show that the label, list and atomic translation constructs translates to subtypes when

one of the children is a subtype, consider the simple translation rules used above. If we added two new constructs, S_3 which is a subtype of S_2 and $S_4 = S_1 \cap n : S_3$, making S_4 a subtype of S_1, then the translation rule for S_1 used above would work for S_4 as the translate construct would still pick up the translation rule for S_2 and still translate to the same output. Since the children of subtypes are also subtypes, as long as the T function translates subtypes in S to subtypes in H, translation rules that depend on T will translate to subtypes in H for subtypes in S. For translation rules like $S_n ::= T(S_{child})$, the rule will work for subtypes of S_n per our definition of T. For rules like $S_n ::= \{c_1 : T(S_{child})\}$ the rule will work for subtypes of S_n as $\{c_1 : T(S_{child})\}$ will produce a subtype of the output construct when one of the children is replaced by a subtype. The same is true for the list and the atomic translation constructs. We can add any construct to T, as long as we can show that this property holds, like we showed for the translation construct above.

Note that since the translation function we described above always evaluate to a language construct, the language that implement it would be a meta-language, as described in section 4.3 above.

4.4.3 Individual composition operations effect on reusability of translation rules

Composition technique	Reuse possible
Introduce	No reuse
Restrict	Full reuse
Relax	Partial reuse
Combine	Sometimes reuse
Delete	Full reuse

Table 4.1: To which degree translation rules can be reused under various conditions

Now that we have covered what it means for the translation rules to be valid, we can discuss how the composition operations affect the reuse of translation rules. An overview of this is shown in table 4.1.

For the introduction operation, we always have to create a new translation rule. As the newly declared construct has no relation to any existing construct, we cannot reuse any rules. If we later recompose the construct, we can reuse translation rules based on the composition operation used for recomposing.

For the Restriction operation, we can always reuse, as the Restriction operation creates a subtype of the original construct. Any translation operation that works on the supertype will also work on the subtype.

The relaxation operation creates a supertype. At first it might seem that we need to re-specify the translation rule for relaxed constructs, and we do to some extent. But we only need to specify the parts that are 'new' in the relaxed construct. That is, we can reuse the existing rule for all ASTs that are members of the subtype construct we relaxed. Our translation framework does not support this at this moment, though, and translation rules for relaxed constructs have to be manually specified.

When combining two languages together, we can reuse any existing rules for the two languages we are combining, as long as they do not clash in their output. For example, if $s_1 \in L_1$ and $s_2 \in L_2$ and $s_1 <: s_2$, but $T_1(s_1) <:> T_2(s_2)$ then reusing $T_1(s_1)$ would cause problems in translations where a subtype of the translation $T_2(s_2)$ is expected. In this case we can just drop the T_1 rule and use $T_2(s_1)$ instead. Another issue can arise if the same construct exists in both languages, in which case one of the rules has to be chosen. This might not always be possible if the rules translate to different types.

The Delete operation removes a construct, which means we can simply drop the translation rule for that construct and thus we do not need to create any new rules for languages under deletion composition.

When translating to a language that itself is composed of other languages/language fragments, the newly composed L_1 language might share language fragments with the L_h language we want to translate to. In that case, no translation is needed, but to make this explicit we can provide an id-translation rule, which simply translates the construct to itself.

Similarly, if a construct in L_1 is a subtype of a construct in L_h, we can simply treat it as its supertype and do not need to provide a separate translation rule for it. This can again be done using a special upcast translation rule, that converts the subtype construct instance to a supertype construct instance. A situation like this might occur either if the L_1 language construct is composed from a L_h language construct directly using restriction, or if as with the id example above it is composed from a mutual language fragment.

4.5 Examples

We now give some examples to show how the composition we are going to use an expression language as well as the patience domain. Using expression languages is common in the DSL literature and the suitability of the patience domain as an example domain for discussing DSL was discussed in chapter 3.

The expression language we are going to use consists of operator constructs such as *sum*, *sub* as well as references to literals, such as *NUM*, *INT*. The expression language we use is a fragment languages, that is, it does not have a root construct, and can thus not be used as a language on its own.

For the patience domain, we will look at the specific example of describing a rule for a valid move. Valid moves in patience games consist of the type of pile (a.k.a. pile group) that a card is being moved on, as well as an expression over the value and suit of the card and the content of the pile.

4.5.1 Introduction of constructs

The introduction of constructs is straightforward. The expression language can be defined as follows:

$$expr := Any \rightarrow \text{Value}$$

$$sum := \{lhs : expr,\ rhs : expr\} \rightarrow \text{Value}$$

$$sub := \{lhs : expr,\ rhs : expr\} \rightarrow \text{Value}$$

$$NUM := \varepsilon \rightarrow \mathbb{R}$$

$$INT := \varepsilon \rightarrow \mathbb{Z}$$

These define a general expression as being anything that evaluates to a value. It also defines the constructs sum and sub, which are structured constructs with two members that are expressions and they evaluate to a value. We also have two structureless constructs, NUM and INT, which evaluate to real numbers and integers respectively. Here we assume that *Value* can reference any value in the domain. Each of these constructs can be declared subtypes of the expr construct as they all are structural subtypes (by definition of Any) and their side-effects are also all subtypes (by definition of Value).

The constructs for the move rule in patience can be defined as follows:

$$Rule := \{group : pilegroup, rule : RuleExpr\}$$

$$RuleExpr := Any \rightarrow \mathbb{B}$$

$$pilegroup := \varepsilon \rightarrow PileGroup$$

$$pile := \varepsilon \rightarrow Pile$$

$$card := \varepsilon \rightarrow Card$$

$$In := \{pile : pile, pilegroup : pilegroup\} \rightarrow \mathbb{B}$$

$$Value := \{card : Any \rightarrow Card\} \rightarrow \{2...10, Jack, Queen, King, Ace\}$$

$$Suit := \{card : Any \rightarrow Card\} \rightarrow \{Hearts, Diamonds, Clubs, Spades\}$$

$$Top := \{pile : pile\} \rightarrow Card$$

$$Bottom := \{pile : pile\} \rightarrow Card$$

$$Empty := \{pile : pile\} \rightarrow \mathbb{B}$$

Here we have a move rule, which is a structured construct with an 'group' member that references a pilegroup and a 'rule' member that references a RuleExpr. A RuleExpr is any construct that evaluates to a boolean (shown as $\mathbb{B}$ in the equations) and could as such be replaced with a *bool_expr*. The advantage of having a specific construct for the RuleExpr is that then we can limit which constructs can actually be used to ones that are relevant to moving cards. The pilegroup, pile, and card are structureless constructs that evaluate to PileGroup, Pile, or Card respectively. These are used to refer to the card being moved, the pile the card is moving to, as well as which pilegroup the pile is in. The remaining constructs could be implemented in the domain rather than the language, as functions/methods on the PileGroup, Pile, and Card types. In that case we would need constructs for calling the functions/methods.

Note that this language cannot be instantiated as it uses a construct with *Any* that has no subtypes. Thus, there are no constructs that can be instantiated for the rule member of the Rule construct.

4.5.2 Relaxing and restricting

We can create expression constructs specifically for numbers and integers, by restricting the side-effect of the general expression to reals, integers, and bools respectively.

$$num_expr := expr \cap (\rightarrow \mathbb{R})$$

$$int_expr := num_expr \cap (\rightarrow \mathbb{Z})$$

$$bool_expr := expr \cap (\rightarrow \mathbb{B})$$

Note that the NUM and INT constructs from earlier could be declared subtypes of the num_expr and INT could be declared a subtype of the int_expr construct.

We can create instantiable constructs for the move rule above by restricting the $RuleExpr$ to some specific structure. The $Rule$ for moving cards above already specifies the pile group which contains the pile the card is being moved onto. Now we might want to split the rule member into the cases where the pile is empty and non-empty. This can be done as:

$$PileRule := RuleExpr \cap \{is_empty : RuleExpr, else : RuleExpr\}$$

We might then want to create a new rule where all rules are implemented using the $PileRule$ we defined above:

$$Rule' := Rule \cap \{rule : PileRule\}$$

The relaxing operation $\cup$ works similarly, but in the reverse. So the $expr$ construct could be derived by any of the restricted expression constructs above. For example:

$$expr := num_expr \cup (\rightarrow Value)$$

Similarly, $Rule$ can be derived from $Rule'$ as:

$$Rule := Rule' \cup \{rule : RuleExpr\}$$

As discussed, restricting creates subtypes and relaxing creates supertypes. On its own,

the completeness of a language is closed under the restriction operation. That is, if a language is complete, then using restriction on one of the language constructs yields another language that is also complete. This is due to the fact that the construct being restricted has to be reachable from the root for the original language to be complete, and restriction creates a subtype which creates an indirect reference between the original and the new construct, which means the root construct also reaches the new construct.

On the other hand, a complete language is never closed under the relaxing operator. The relaxing operator (almost) always creates a new construct that is neither directly referenced (as it is new, none of the existing constructs could possibly reference it) nor an indirect reference, as it is a supertype of an existing construct, not a subtype. The only exception is when the construct being relaxed is a root construct that already references all the other constructs. In this case, the new construct can serve as the new root that can reach all other roots, as the original root is reachable through indirect reference and all other constructs are reachable from there. In some other cases it might be possible to declare the new construct a subtype of some other construct already in the language (thus completing the language), but in general there needs to either be a new construct created that is the subtype of an existing construct that directly or indirectly references the relaxed construct, or the constructs need to be relaxed all the way to the root.

4.5.3 Combining and deleting

Combination $\uplus$ is straightforward. If L_{int} is a language for integer expressions, L_{bool} is a language for boolean expression then

$$L_{int} \uplus L_{bool}$$

is a language containing integer and boolean expressions. Note that though they are both expression languages and they both contain constructs that are derived from the *expr* construct, the combine operator does not generate consistent (and therefore not complete) languages.

To make the output of the combine operator consistent, at least one construct from one of the languages has to be declared equal or a supertype to a construct from the other language from which all other constructs in that language can be reached. Alternatively, a new construct could be introduced that fulfils the same purpose. For example:

$$L_{int} \uplus L_{bool} : expr_{int} = expr_{bool}$$

creates a consistent language if $expr_{int}$ is a construct in the L_{int} that can reach all the constructs in L_{int} and $expr_{bool}$ is a construct in the L_{bool} that can reach all the constructs in L_{bool}.

The delete operator $\ominus$ is straightforward in its use as well.

$$L_{int} \ominus sub$$

would for instance create a new integer expression language without a sub operator from the integer expression language declared above. Complete languages are not closed under the delete operator. The delete operator does create a complete language from a complete language if the construct being deleted is only ever indirectly referenced and does not reference any construct that is not reachable from the root through any other construct other than the one being deleted. If the construct being deleted is directly referenced, then the resulting language references a non-existing construct. If there is a construct that is only reachable through the construct being deleted, then the resulting language ends up with a construct that is no longer reachable from the root.

4.5.4 Translation

The expression language is found in most languages, and can therefore often simply be translated to itself. For example, the sum construct

$$sum := \{lhs : expr,\ rhs : expr\} \rightarrow Value$$

can have a translation rule like

$$sum := \text{(sum \$lhs \$rhs)}$$

Here the left-hand side of the ':=' operator identifies the name of the construct to be translated, and the right-hand side gives the output to be generated. The right-hand side is here in the form of an s-expression, where the members of the construct being translated are represented with a $-sign. Note that the 'sum' on the left-hand side refers to the sum construct in the input language, whereas the sum on the right-hand side refers to the sum construct in the output language. To generate the output, each node with a $ needs to be translated using the appropriate translation rules. The exact way in which this is achieved is beyond the discussion in this chapter, but will be discussed in more detail in chapter 6.

For the patience example, we might have translation rules like

$$Rule' := \text{(If (\$In \$pilegroup \$pile) \$rule))}$$

$$PileRule := \text{(If (\$Empty \$pile) (Return \$empty) (Return \$else)))}$$

to translate move rules into some other language. Here we assume that the output language has an 'If' construct and that $ nodes get translated using an appropriate translation rules. In this example, $In, $Empty, and $pile refers to the In, Empty, and pile constructs discussed above, and $pilegroup, $rule, $empty, and $else refers to the members of the $Rule'$ construct.

The In and Empty constructs might be translated as

$$In := \text{(Call (Member \$pilegroup "contains") \$pile))}$$

$$Empty := \text{(Call (Member \$pile "isEmpty"))}$$

if the output language contains the constructs Call (which calls a function with zero or more arguments) and Member (which accesses a member of an object, in these cases, methods).

4.6 Limitations and solutions

When translating from one language to another, type errors may occur. It is not possible to completely avoid type-errors in composition, as type systems are non-local and it is thus not possible to compose without a possibility of error (Eden et al., 2006). It is in some cases possible to reduce the chances of errors occurring. Our approach allows for arbitrary 'side-effects' of language constructs, which can be used to describe typing behaviour. For example, some language constructs might have the effect of introducing a symbol into the global scope, or the local scope, or in the context of the children of the language construct. This would allow for type-checking the input language. Then we could introduce translation rules for these side-effects, and construct them in such a way that if the input language type-checks correctly, then so will the output language. This wouldn't cover all cases, but would help in many common cases. This could be done for many different typing cases, where side-effects would be introduced in the input language, a type-checker to check that those side-effects are used correctly would also have to be constructed, and finally translation rules that translates the side-effects from input to output language would have to be written.

As mentioned, this work does not cover the composition and translation of side-effects. Future work to cover this would allow for more complex translation of languages, as our framework as presently defined assumes that the side-effects in the input language can sim-

ply be copied over to the output language. As side-effects are the way we refer to (non-language) domain types, composition of side-effects would probably also require a framework for composing domain types. There are various approaches to domain composition but we would probably choose something similar to our approach for language definitions, as we have stated, language is a domain, so language composition should be viewable as a subset of domain composition.

Beyond side-effects and domain composition, we could also introduce translation rule composition. As briefly discussed in previous sections, there are cases where with the current framework we have to rewrite a translation rule completely, even when we could reuse an existing rule for parts of the new rule. This is, for example, the case when we have a new supertype rule created using the relaxing operation. With translation composition, we could compose a new translation rule from the existing rule and then add any additional rules required to handle the supertype construct.

We mention interpreters and compilers as the primary use for this framework, but with some modifications it could apply to pre-processor languages as well. If we have a DSL L_0 and a host language L_h, then a pre-processor language can be constructed as $L_1 = L_0 \uplus L_h$, potentially with some additional ‘glue’ constructs that combines them together. It should then be possible to add the translation rules for L_0 to translation rules that translates L_h to itself, and that should create a compiler that pre-processes L_1 into L_h. A similar argument can be done for embedded DSLs, though in that case it would simply be using L_h, which does not seem meaningful.

4.7 Conclusion

In this chapter we have presented a framework for composing languages from existing languages and shown how composition affects the translation of the composed language into other languages. To do this, we have discussed the differences between domain and languages, concrete syntax and abstract syntax, abstract syntax types as well as translations. We have also discussed other similar approaches and how ours differs from them.

In discussing the composition framework, we also showed how concepts such as consistent languages and language fragments play a role in language composition and ensure that new languages can be translated into existing executable languages. We also showed how meta-languages can be defined within our approach as languages that, when executed, produces a language definition as output.

We have also discussed some of the complications in our approach, such as non-local type-references. While our approach can be shown to translate into structurally correct ASTs, we cannot guarantee that the output type-checks correctly. An approach for improving the translation of type-rules within our framework is a possible future addition. Similar work has already been done in Lorenzen and Erdweg (2016), though they use lambda calculus for their approach and our framework is designed around a procedural paradigm and it is not clear how well lambda calculus would fit into our framework.

Chapter 5

Composing Patience Games

5.1 Introduction

There has been a lot of research into how Domain-Specific Languages (DSLs) can improve the development process by providing an easier way to specify solutions within a specific problem domain Mernik et al. (2005); Fowler (2010); Kosar et al. (2016).

Some of the common reasons given for why using DSLs improve development are that DSLs makes the code easier to understand, they reduce the development time, and makes it easier to write correct programs. (Kosar et al., 2009)

A common problem with adoption of DSLs is that implementing a new DSL takes a lot of time and effort (Cazzola and Vacchi, 2016). In chapter 4 we have presented a technique for making it easier to develop new DSLs from existing languages.

We have discussed how our composition technique can be used to create a language for describing composition and translation in chapter 6 and we have discussed how patience games can be used as a model for discussing DSL development techniques in chapter 3. We now want to show that our composition techniques work on other domains than language creation, and so we will use our composition techniques to create a DSL for the patience domain.

The main thing we want to show with this is that languages other than meta-languages

can be created using our technique. This will include translation of our DSL both into other DSLs and into a General Purpose Programming Language (GPL) (Java). We also talk about how our technique can be used to manage abstraction level. That is, we will show how to increase and decrease the abstraction level of a DSL, which is useful if there already exists a language that either works on too high a level, or a language that is too low level for the required use case. We show this by showing how a DSL for describing card games in general can be made into a sub-language, specific to the patience domain. We also show how one can go in the other direction, creating a more general language from a specific one.

How can we evaluate whether our technique is suited for patience games?

- Based on functionality of the end product (i.e. is the constructed DSL able to actually define patience games)
- based on the amount of code needed to turn the DSL into a GPL
- interfacing with GPL code
  - Amount of glue code required
- Different levels of abstraction in the DSL
  - Range of patience games that can be implemented vs. code required for implementation

Of these, we only use the amount of code needed to turn the DSL into a GPL as a measure, as it is the most straightforward one to use. We count the code by counting the amount of Abstract Syntax Trees (ASTs) that are needed to specify the translation between DSL and GPL. Without modification, this method is not suitable to reason about the difference between a DSL implementation of patience and a GPL implementation. As our goal is to compare different ways of implementing DSL translation rules, not whether the DSL approach is better than a GPL only approach, it is not within the scope of this study. For comparisons between DSL vs. GPL implementations on a variety of domains, see (Kosar et al., 2010, 2012).

One of the things we can do is provide translations to an intermediate language. The reasons one might want to do this is to make the translation rules simpler by not having to translate as far down the abstraction pyramid. It also decouples the translations from the final implementation language. That is, if the intermediate language has translation rules for multiple implementation languages, we can translate down to any of those languages with only translation rules for the intermediate language. We compare this approach to straight translation into a GPL using the above-mentioned method.

It should be noted that even if we already have definitions from our DSL to the GPL, we unfortunately need to rewrite all the translation rules if we employ an intermediate language. There is also no guarantee that the intermediate language will interface with the same code-base that the DSL to the GPL translations interfaced with. As such, the introduction of intermediate languages should only be undertaken if there are long-term advantages to it (as there are no short-term ones, potential long term discussed above). This also implies that there is an advantage early on to recognise and introduce adequate intermediate languages. This gives all the long term advantages of using intermediate languages without the short-term penalty of having to rewrite the rules.

Some of the things one might look for in an intermediate language are:

- Platform independence, that is, a language that has translation rules for multiple GPL languages.
- Compatibility with other DSLs, that is, there are several other DSLs that have local-only translations to the intermediate language.
- Analysability, that is, the intermediate language has some properties that make it easy to analyse.

5.2 Background

As discussed in chapter 3, patience games have some properties that make them an interesting case study for DSL development purposes. We will focus on defining a DSL for specifying (a subset of) patience games and show how these can be translated to a GPL as well as a toy game description language.

There exists some Game Description Languages (like (Love et al., 2008)) as well as Card Game Description Language (like (Font et al., 2013)), but we will use a simpler toy language for the examples provided in this chapter. This is both to simplify the translation rules and due to all of the existing languages being designed for two or more player games.

To the best of our knowledge, there are no existing patience DSLs, not in the published literature nor as part of any existing patience games suite. The closest thing are suites such as Aisleriot (Gnome, n.d.) which uses Scheme to specify the games while the suite itself is written in C and PySol (PySolFC, n.d.), which provides a GUI wizard for specifying custom rules to games (though it is very restrictive).

The patience framework we are going to use was described in chapter 3. It consists of all the classes necessary to describe the domain of patience. The rules for particular patience games are implemented as sub-classes to an abstract class *PatienceGame*. Each type of rule (move rules, win condition, etc) are implemented as methods in the *PatienceGame* class. Thus, to write a DSL for patience games using this framework means writing a language that can be translated into a subclass of *PatienceGame*.

We will also discuss a theoretical game framework for general games to show how patience games could be implemented as a special case of a general game. We do not provide implementation for this, though, as a general game suite would be too complex and out of scope to implement for this work.

We will use the DSL composition framework described in chapter 4 to implement the patience DSL.

5.3 Patience

Gibson (1993) defines patience games as "... any card game played by one person who usually deals out cards and then assembles them in special groups according to established rules.".

In general, patience games tend to consist of one or more decks, and groups of piles, such as foundation, tableau, waste and reserves. In some games, the deck is completely dealt out at the beginning of the game, in other games the player is able to deal from the deck as the game proceeds.

We go through patience in more detail in chapter 3 and will here only sum up the different concepts:

- Deck – one or more card decks from which cards are dealt to one of the pile groups
- Card – a single card, either in the deck or a pile
- Pile – a single pile of cards, belonging to one of the pile groups
- Pile group – a group of piles with distinct rules for how cards can be placed
  - Tableau – where cards are often initially dealt
  - Foundation – where cards are supposed to be moved in the right order to win the game
  - Waste – when present, cards from the deck is dealt to it
  - Reserve – a place to temporary store cards to open up new moves on the tableau
  - Discard pile – place to store cards that are no longer in play
- Build – the rules for how cards can be placed on top of other cards in a given pile group
  - By-Value – cards are built by their value, ascending or descending

Concept	Properties	Examples
card	value, suit, colour	2, Ace, King; diamonds, clubs; red, black
pile	in(), top(), bottom()	in(card, pile); card == pile.top()
pilegroup	in(), pile_at(), tableau, foundation, waste	in(pile, foundation), in(card, tableau.pile_at(1))
deck	deal_to(), shuffle(), fill_from()	deck.deal_to(waste), deck.shuffle(), deck.fill_from(waste)

Table 5.1: Concepts in our patience implementation

- By-Suit – cards are built by their suit (hearts, diamonds, clubs, spades)
- By-Colour – cards are built by their colour (red, black)
- By-Alternating-Colour – cards are built by alternating colours
- Combination of above

5.4 Patience Implementation

To help the discussion on translation from a DSL to a GPL, we will quickly consider how a patience suite (that is, a program that allows the user to play several different patience games within the same program) could be implemented.

Patience games consists of cards, piles, groups of piles, decks and rules for how these interact. A quick rundown of these can be found in table 5.1.

The rules describe how cards are related to the (groups of) piles and decks, as well as what the win condition is etc. The full set of operations can be seen in table 5.2.

Our patience suite consists of a couple of core classes: one abstract class that defines the rules of individual games, a view class that displays the game, as well as control code to let the user interact with the game. Of these, only the class that defines the rules is relevant for the creation of the DSL; The view and controller are the same for all patience games, and individual games are independent of the working of them.

Here we define the type of entities that exist in the domain of patience games, as well as

Operation	Description
canMove(Card, Pile) canMove(Pile, Pile)	Can a card or a pile be moved on top of another pile?
hasWon()	Is the game in a win-state?
deal()	Deal a card, taking a card/cards of the deck and putting it on relevant piles.
redeal()	If available, reset the deck with unused cards.
initialise()	Initialise the game, dealing the first cards to the correct piles

Table 5.2: Methods that a Patience Language needs to implement in our model.

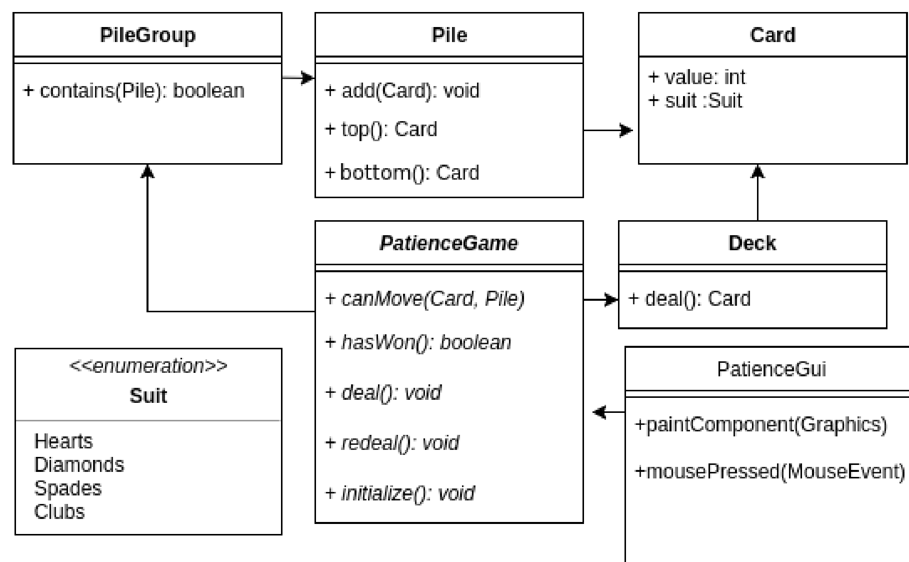

Figure 5.1: UML diagram describing the main parts of a simple patience suite. Some simplifications have been made to better fit the diagram.

examine the relationships between the entities. In figure 5.1 we show a very simple design for a patience suite described in a UML class diagram. It consists of cards, piles, groups of piles, and decks, as well as an abstract patience game class and a view class.

In figure 5.2, we show the domain for the patience DSL, including some concrete strategies (Klondike, Golf, and Yukon). We have removed the view class, as it does not affect the implementation of the strategies and thus is not needed in the DSL.

The domain consists of cards, piles, group of piles and deck(s) of cards. Cards have suit and value, as well as colour (which is completely dependent on suit). The suits are hearts, diamonds, clubs and spades, and the colours are red (hearts, diamonds) and black (clubs, spades). Piles are collections of cards. They support adding one or more cards on top and removing one or more cards from the top. Groups of piles are collections of piles. Their purpose is to allow specification of different rules for different sets of piles. Decks of cards support dealing a card for the top, shuffling, and repopulating the deck from a pile.

Finally, there is the class defining the rules. Each rule is implemented as an abstract method, to signify that individual patience games need to provide their own definition for these rules. The rules that the class should define are: rules for how a card can be moved on to a pile, moving a group of cards onto a pile, whether the game is in a win-state, the game's initial setup, dealing new cards, and what to do when the deck is empty (i.e. how to do a re-deal). We ignore having a rule for a losing state, as it is in general quite hard to compute whether a given game is in a losing state or not.

For the game Klondike, the `canMove(card, pile)` function would be implemented as shown in listing 4.

If the pile is part of the foundation, then if it is empty, the card is movable if it is a King. If the pile is not empty, then the card is movable if the value of the card is one less than the top card on the pile, and the colour of the card is different to the colour of the top card on the pile. For the tableau, the rule is similar, except that empty piles accept Aces, and non-empty requires the card value to be one more than the top of the pile value, and for the suit to be

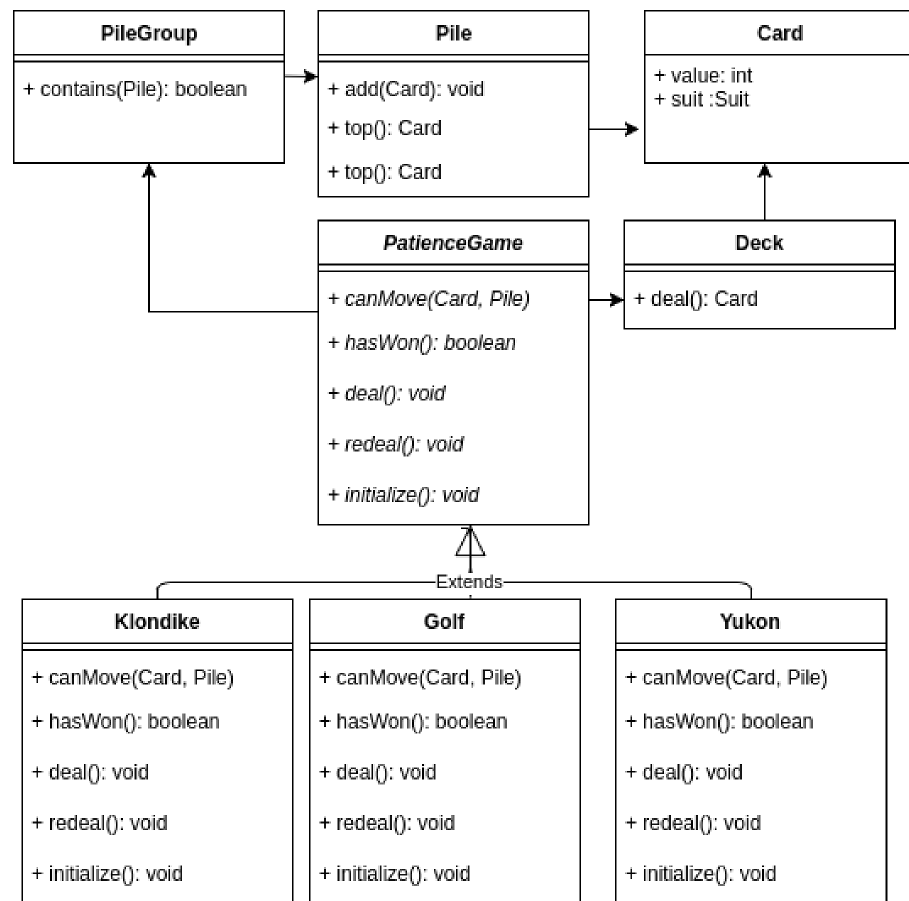

Figure 5.2: UML diagram describing the domain that is relevant for our DSL, including some example patience PF members (Klondike, Golf, and Yukon)

the same. There are no other piles that can be moved to.

The `canMovePile(from, to)` is similar, except all the references to card would be replaced with `from.bottom()`.

The `hasWon()` method for Klondike would look something like what is shown in listing 5.

If any pile does not have a King on the top, then the game is not in a win-condition, else it is. As cards can only be moved onto the foundation if they follow the correct order, we do not need to check if the final piles are correctly ordered, as they cannot be anything else.

Listing 4 Rough Python code for Klondike (partial) canMove function

```
def canMove(card: Card, pile: Pile):
    if pile in foundation:
        if pile.is_empty():
            return card.value == Ace
        else:
            return card.value == pile.top().value + 1 and
                   card.suit == pile.top().suit

    if pile in tableau:
        if pile.is_empty():
            return card.value == King
        else:
            return card.value == pile.top().value - 1 and
                   card.colour != pile.top().colour
    return False
```

Listing 5 Rough Python code for Klondike hasWon function

```
def hasWon():
    for pile in foundation:
        if pile.top().value != King then
            return False
    return True
```

5.5 Abstract Language Model

We presented our framework for composition in chapter 4. Here we will summarise how it works. We model abstract language as a set of constructs that are either labelled constructs, list constructs or structureless/atomic constructs. How we write them are shown in equations 5.1, 5.2, and 5.3 respectively.

$$Name := \{child1 : Type1, ...childN : TypeN\} \rightarrow \textit{Side-effect} \tag{5.1}$$

$$Name := \{Type\} \rightarrow \textit{Side-effect} \tag{5.2}$$

$$Name := \varepsilon \rightarrow \textit{Side-effect} \tag{5.3}$$

The labelled constructs represent constructs with named members, list-constructs represents lists, where the type specifies the kind of constructs the list-construct can contain. The structureless constructs represent leaf-nodes that contain no additional information except their name.

In addition to the above, the constructs can also specify side-effects of their execution, that is, they can specify what happens when they execute. We are in particular interested in the *evaluates-to* side-effect. This specifies that a particular construct, when executed, produces an object of the specified type.

To construct our patience DSL we use language composition. The composition operations we will use are introduce $:=$, restrict $\cap$, relax $\cup$, delete $\ominus$, and combine $\uplus$.

We also need to consider how translation rules behave. For example, when can we substitute one translation rule for another, and how do construct subtypes fit in with this?

A translation rule is essentially a function $T(c : C_1) \rightarrow C_2$, where C_1 is a construct in

our input language and C_2 is a construct in our output language. Thus, translation rules behave like functions in the case of subtyping. So for $T_1(c : C_1) \rightarrow C_2$ to be a subtype of $T_2(d : D_1) \rightarrow D_2$, C_1 (the covariant) needs to be a supertype (or same type) of D_1 and C_2 (the variant) needs to be a subtype (or same type) of D_2.

So, T_1 can replace T_2 if they both translate to the same type, and T_1 takes a construct that is a supertype of the construct taken by T_2. This means that we can reuse the translation rules of supertype constructs when we create a subtype construct, as the translation rule for the supertype construct is a subtype of the translation rule for the subtype construct.

5.6 The language

Let us now discuss a language for describing patience games for the patience suite described above. To be used with the patience suite, the language needs to describe subclasses to *PatienceGame*.

Since we use the DSL composition framework described in chapter 4, we are going to be describing translation rules that translate sentences in the patience DSL to methods in the *PatienceGame* subclass. We are also going to discuss how the other composition techniques can be used to increase reuse.

As shown in listing 6, the language consists of rules for moving cards, moving piles, win condition how to deal cards, how to redeal cards (resetting the deck), and the initial setup.

The move rules can be specified by stating the pile group they apply to, the rule for moving a card to an empty pile in the given pile group and the rule for moving a card to a non-empty pile in the pile group. Rules for moving to piles are similar, except instead of a card, it is a pile that gets moved.

The methods that will implement these in the patience framework are methods taking a Card and a Pile as input and returning a boolean value, and a method taking two Pile objects and returning a boolean respectively.

Listing 6 Patience DSL constructs

```
Patience := {
              name: ID,
              moveRules: {MoveRule},
              movePileRules: {MoveRule},
              winCondition: WinExpr,
              deal: DealExpr,
              redeal: RedealExpr,
              init: InitExpr,
              pilegroups: {PileGroup}
            }
PileGroup := {name: PileGroupId,  piles: INT}
MoveRule := {
               pile: PileID,
               emptyCond: PGBool,
               elseCond: PGBool
            }
WinExpr := PGBool
DealExpr := {to: {PileGroupId}}
RedealExpr := {from: {PileGroupId}}
InitExpr := {InitPileExpr}
InitPileExpr := {
                   piles: PileGroupId,
                   first_pile: INT,
                   last_pile: INT
                }

FlipExpr := RedealExpr ∩ {}
ShuffleExpr := RedealExpr ∩ {}
```

The win condition is specified as a condition on each pile in one or more of the pile groups. Each condition is specified given a pile in the pile group as well as the index of that pile. The win condition is implemented as a method that takes no input and returns a boolean in the framework,

The deal rule specifies where to deal the cards from the deck to. The redeal rule specifies from which pile group the deck should be restocked from, and the initial setup rule specifies how the cards should be dealt initially. They are all implemented as void methods with no arguments in the framework, so the DSL rules can be translated to a method with arbitrary

code, which does not return anything.

Further to this, the language also specifies the available pile groups and how many piles are in each group, as shown in listing 7.

Listing 7 Additional Patience DSL constructs

```
CardExpr := Any → Card
PileExpr := Any → Pile
PGBool := Any → Bool
PGExpr := PGBool
PGCmp := {lhs: PGAttrExpr, rhs: PGAttrExpr}→ Bool
PGInPG := {lhs: PileExpr, rhs: PileGroupId}→ Bool
PGInP := {lhs: CardExpr, rhs: PileExpr}→ Bool
PGAttr := {object: Any, attr: ID} → AnyType

PileAttr := PGAttr ∩ {object: PileExpr}
CardAttr := PGAttr ∩ {object: CardExpr}
PGAnd : PGBool ∩ {lhs: PGBool, rhs: PGBool}
PGOr : PGBool ∩ {lhs: PGBool, rhs: PGBool}
PGNot : PGBool ∩ {value: PGBool}
PGAttrExpr := IntExpr ∩ {lhs: PGAttr, rhs: PGAttr}
PGEq : PGCmp ∩ {}
...
```

At first this looks like it is going to need a lot of translation rules, but with composition we can reuse most of the translation rules. With a couple of exception, they can all be reused from a standard expression languages. For example, the `PGBool` expression can be composed from a standard library for boolean expression using restriction. The subtypes of `PGBool` (`PGAnd`, `PGOr`, `PGNot`) can be composed from standard And, Or, Not expressions. Subtypes of `PGCmp`, representing comparison operators can be constructed similarly, as can subtypes of `PGAttr`, represented by the 'dot' operator in most GPLs. Our framework currently does not have any way of specifying that one construct is a subtype of another, so instead we use the restriction operator $\cap$ with an empty construct to specify subtypes, as is seen in some of the definitions in Listings 6 and 7.

The constructs that might not be possible to are the `PGInPg` and `PGInP`, representing checking if a pile is in a pilegroup and if a card is in a pile respectively. Since this operation

would often involve calling a method on a collection (such as 'contains' in Java), it would not be possible to reuse a translation rule as calls have a completely different structure to the `PGIn*` constructs. In languages such as Python, which does have a separate 'in' operator, these constructs could reuse its translation rule.

We use PGBool here to represent boolean expressions in the patience game. We could also have used BoolExpr to represent general boolean expressions, but we decided to use to use PGBool to highlight that the constructs specifically represent boolean expressions that work on cards, piles and pile groups.

5.6.1 Card Games

Since patience games are a type of card game, we will consider what a general card game DSL might look like. Since card games are very heterogeneous, we will leave the language quite general, and not have any specific constructs in it .

Listing 8 Card game DSL constructs

```
CardGame := {
               name: ID,
               players: NUM,
               moveRules: {MoveRule},
               winCondition: Any → Bool,
               deal: DealExpr,
               init: InitExpr,
             }
MoveRule := Any → Bool
DealExpr := Any
InitExpr := Any
```

A card game, much like a patience game, consists of rules defining valid moves, what the win condition is, how to deal cards, and the initial setup. Unlike patience, general card games also have to deal with multiple players (Patience games also technically have players, but it always only one player, so there is no reason to keep track of players in patience). As games are very heterogeneous, we cannot provide any specific constructs for implementing

dealing, moving, win conditions and initial setup rules. We have to define these as possibly having any implementation. Specific card game families can then restrict these to something that is specific to that card game family's domain.

5.6.2 Cards

Cards can be viewed as their own domain, outside of the games that use them. The main benefit in our case of separating them is that it allows us to separate two slightly different concerns, namely the rules from what the rules operate on. Arguably, the card language specified here is inherently part of the card game language discussed above, as all card games inevitably needs to specify operations on cards. However, it should be noted that as the two languages are currently defined, combining them would result in an inconsistent language, as neither the card nor the card game language refer to each other as currently defined.

The card language consists of ways to specify individual cards, decks of cards and operations on a deck (such as `deal`), Piles (and operations, like `top`, `bottom`), Groups of piles (that is, a set of piles that are closely related somehow). Card expressions, pile expressions, group expressions and deck expressions are small expression languages that consist of all the possible operations on the respective constructs. As mentioned in section 5.6, our framework does have a way of specifying subtypes other than through restrictions. This is seen in the last eight definitions in Listing 9.

5.6.3 Games

We can go one step further, and consider a general (turn based) game DSL, that describe games in general. This is inevitably even more heterogeneous than card games, and thus even less specific construct can exist.

In our language, shown in listing 10, general games consist of players, rules for what players can do during their turn, rules for win condition and rules for how the game is

Listing 9 Card DSL constructs

```
Card := {kind: Kind, value: Value}
Kind := Any
Value := NUM
Deck := {Card}
Pile := {Card}
Group := {Pile}
CardExpr := Any
CardKindExpr := CardExpr ∩→ Kind
CardValueExpr := CardExpr ∩→ Value
PileExpr := Any
PileCardExpr := PileExpr ∩→ Card
GroupExpr := Any
GroupPileExpr := GroupExpr ∩→ Pile
DeckExpr := Any
DeckCardExpr := DeckExpr ∩→ Card
Top := PileExpr ∩ {pile: Pile}
Bottom := PileExpr ∩ {pile: Pile}

Hearts := Kind ∩ {}
Diamonds := Kind ∩ {}
Clubs := Kind ∩ {}
Spades := Kind ∩ {}
Ace := Value ∩ {}
King := Value ∩ {}
Queen := Value ∩ {}
Jack := Value ∩ {}
```

initially setup. These constructs that one would expect in all types of games, without any constructs for specific games. All specific expression constructs have to be composed in.

Note that the Game, Card Game and Patience Game languages are intentionally constructed to resemble each other, roughly how they would be created if they were restricted or relaxed from each other. This is to create an ideal situation for composition, though in the real world it would probably not be feasible to implement the Game and Card Game languages like this.

Listing 10 Game DSL constructs

```
Game := {
               name: ID,
               players: NUM,
               moveRules: {MoveRule},
               winCondition: Any → Bool,
               init: InitExpr,
             }
MoveRule := Any → Bool
InitExpr := Any
```

Translation method	Number of ASTs in translation	Number of Rules
Without reuse	247	29
With subtyping and reuse	228	20

Table 5.3: Number of ASTs and rules needed for various methods of translation to GPL.

5.7 Translation to Java

For translation to Java, we want to translate our DSL code into a class in Java that an existing code base can use to represent the game in some form. We do not try to create a general purpose translation that could be used for any code base. Instead, we describe one particular way of implementing a patience suite in Java and then describe how we can provide a DSL for describing patience games for that particular patience suite. This does mean that our translation rules will only work for this one particular implementation, but modifying the rules to fit other ways of implementing a patience suite should not be too difficult. To reduce the chance of designing the patience suite to the needs of the DSL, the Java code was written before the DSL was designed.

In table 5.3 we show how many AST nodes and translation rules are required to specify the translation from the patience DSL to Java. As can be seen, the number of ASTs is fairly close, while the number of rules is reduced by almost a third. This is due to the fact that a lot of the ASTs in non-reusable rules are used for boilerplate code, whereas the reusable rules often requires only a handful of ASTs. Almost a half of the ASTs appears in the translation

rule for the root AST “Patience".

5.8 Translation to DSL

Though there does exist some Game Description Languages (Love et al., 2008) as well as Card Game Description Languages (Font et al., 2013), we will use our own toy card game DSL and game DSL here for simplicity.

It is worth noting that the Patience DSL is not necessarily a subtype of the Card Game DSL, which in turn is not necessarily a subtype of the Game DSL. It should be, at least theoretically, possible to create such a relationship between the three DSLs, in which case no extra translation rules would have to be written for the Card Game and Patience DSLs to fully translate them.

It may make sense to not have subtype relationships between the DSLs, as it constrains what can be expressed in the Card Game and Patience DSL, it likely requires more effort to ensure everything that needs to be expressible is expressible, and different translations might be needed regardless (as is shown in the next section).

As the card game DSL and the general DSL both only have one construct that does not reference the *Any* construct, there is only one possible construct whose translation rule can be reused (as constructs containing *Any* cannot be instantiated nor have translation rules associated with them). However, the card DSL does have several constructs whose translation rules can be reused. Thus, the card game DSL (when combined with the card DSL) does offer reuse beyond what translating straight to Java does, whereas the game DSL does not. This is a consequence of how we have chosen to model the game and card game DSLs and it could be possible to find constructs that can be instantiated both in general games and in patience games, we just could not come up with any.

5.8.1 Relation with existing code-base

In order for the DSL to be executable, it will have to be translated to an executable language. So that the executable language can interface with the code generated from the DSL, it is convenient to have abstract classes in the code-base that the DSL generate subclasses for (Kihlman, 2015).

When we compose our patience DSL from other DSLs, we get several different options for how to translate the language to its final executable form. In our example we assume we have a patience language composed from a card game language, which in turn is composed from a game language.

In this case we might have constructed our code-base so that we have a base *Game* class, which is extended by a *CardGame* class, which is extended by the *PatienceGame* class. In this case, the DSLs can be translated to the relevant class, as shown in figure 5.3. This is a structure we might get if we know beforehand that we are going to employ a DSL for each of the different domains and so the code-base is structured to take that into account.

It is also possible that there is no relation between the different classes, and that each DSL gets translated to a completely separate class in potentially separate code-bases. This is shown in figure 5.4. This is something that might happen if each code-base is developed separately, and only afterwards DSLs are added and composed from each other.

Lastly, we might only use the *Game* class, and have all the other DSLs generate code for that class. This is shown in figure 5.5. This might happen if the *Game* class is sufficient for card game programs and patience game programs not to need specialised versions of the class.

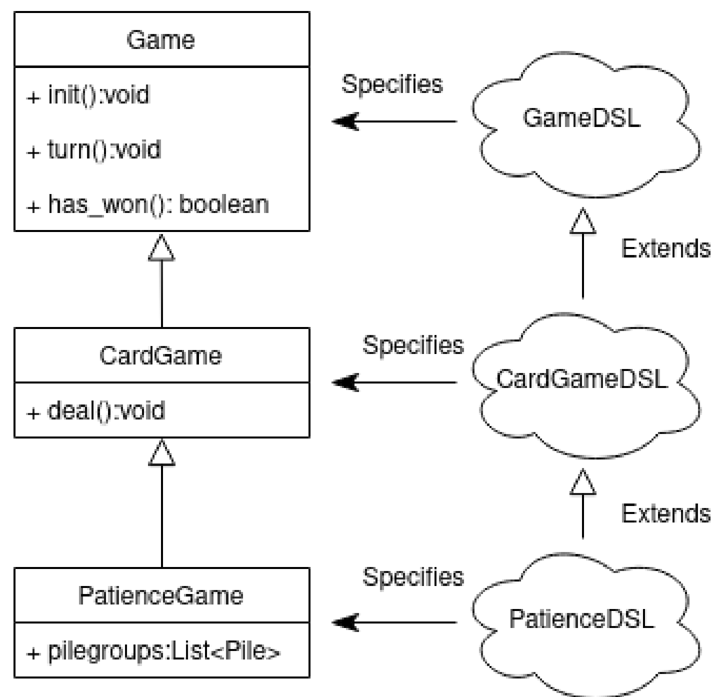

Figure 5.3: DSL hierarchy for games, card games and patience games being translated to an inheritance hierarchy of GPL classes.

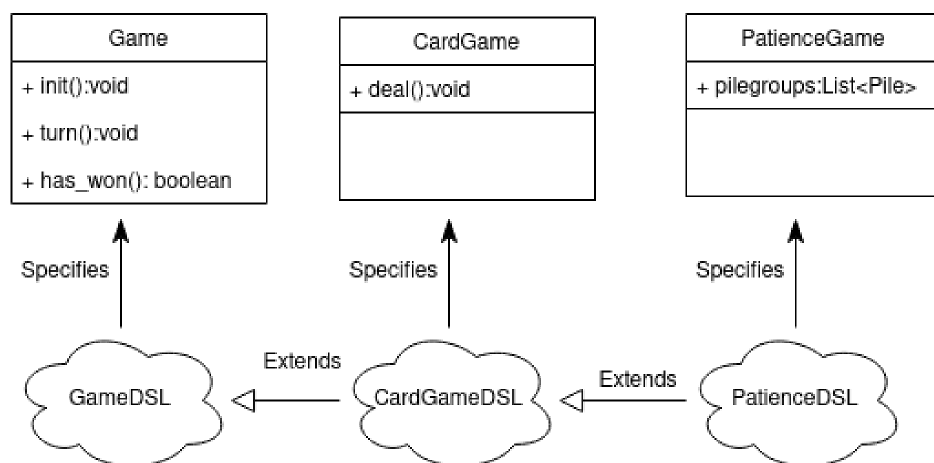

Figure 5.4: DSL hierarchy for games, card games and patience games being translated to independent GPL classes.

5.9 Discussion

In this chapter, we have designed the DSLs to fit into a particular model for how the DSLs will be run eventually. We have assumed that a particular GPL Application Program Interface (API) exists that we will translate our DSL into. This has informed some of the design decisions we have made for our DSL definitions. It would be interesting to see what difference it would make to how we define our DSLs if we had a completely different API, or if we instead of running the DSL code wanted to analyse it. Both in terms of the actual structure of the DSL and how it would affect translation rules.

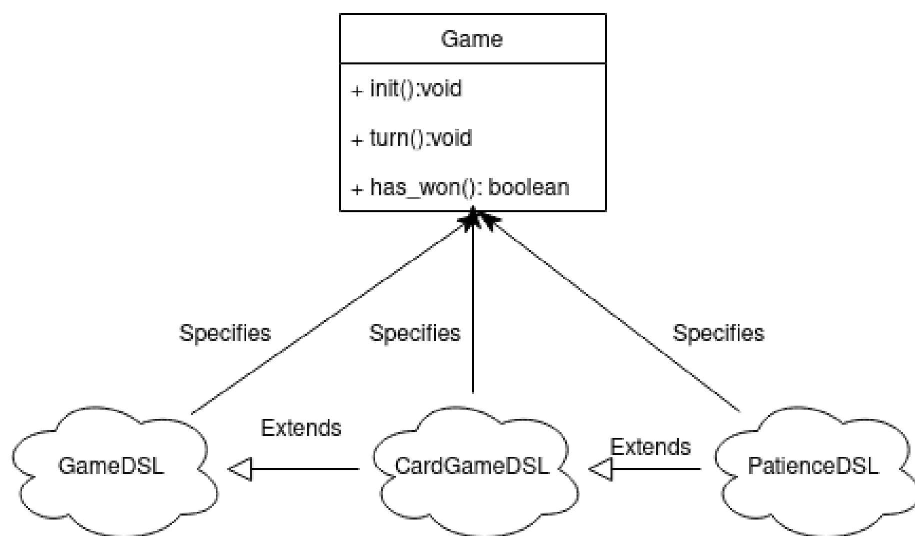

Figure 5.5: DSL hierarchy for games, card games and patience games being translated to a single GPL class.

Chapter 6

A Self-Describing Domain-Specific Language for Translations

6.1 Introduction

This chapter will present the implementation of a translation language that translates the abstract syntax of a source language to the abstract syntax of a host language and discuss the interaction between Domain-Specific Language (DSL) composition and translation between languages. We will also show how this DSL can be used to describe the translation of itself into a General Purpose Programming Language (GPL), thus showing it is self-describing.

DSLs are used, among other reasons, to make it easier and quicker to program by having languages that operates on a higher level than regular GPLs (Kosar et al., 2016). Creating DSLs is complicated though and whether it makes sense to employ a DSL in any particular situation depends on the amount of investment into development and maintenance of the DSL which is required (Walter et al., 2014; Chodarev and Kollar, 2016). One way of reducing the development and maintenance is by using DSL composition (Erdweg et al., 2012; Völter, 2013b).

Mernik et al. (2005) identifies seven different ways of implementing DSLs:

- interpreter
- compiler
- preprocessor
- embedding
- extensible compiler/interpreter
- Commercial Off-The-Shelf (COTS)
- hybrid

The first two are the usual meanings within computer science, the DSL is either interpreted or compiled using a custom-made interpreter or compiler. Embedded DSLs (sometimes called EDSLs or DSELs) are embedded into GPLs, often by utilising reflection and other advanced features of high-level GPLs. Preprocessor DSLs are similar, in that the DSL code is combined with host language code, but requires an external program to preprocess the files, turning the DSL code into host language code. Extensible compiler/interpreter are similar to compiler and interpreter, except they use existing compilers/interpreters that allow extensions to be added to the compiler/interpreter. COTS are pre-existing tools for creating DSLs. Hybrid approaches are combinations of the above. These classifications are fairly common in the literature and were used among other places as in the systematic mapping study presented in (Kosar et al., 2016).

In (Fowler, 2010) on the other hand, DSL implementation is divided into external, internal and language workbench, depending on whether they are implemented to be read from external files, embedded in GPL code, or created using a special toolkit for language creation. In this classification, Mernik et al.'s embedded and preprocessor DSLs would be embedded languages in Fowler's classifications. The (extensible) Compiler/interpreter would usually be classified as external in Fowler's classifications and COTS would likely be classified as language workbench. The hybrid classification in Mernik et al.'s classifications does not really map to anything specific in Fowler's classifications.

We think the work covered in this chapter would most likely be useful as part of compiler/interpreter tools, which as mentioned Fowler would classify as external DSLs. From here on, when we use embedded we use Fowler's usage of the term, and we use external to refer to compiled/interpreted DSLs. Thus, though there are good reasons to separate between embedded (in Mernik et al.'s sense) and preprocessor languages, we group them together like Fowler.

When a new DSL is created, there needs to be a way to execute the new language. Embedded DSLs (besides preprocessor) are inherently executable as they use the language they are embedded in to execute. External DSLs (as well as preprocessor DSLs) need to have an interpreter or compiler that translates the instructions in the DSLs into instructions that are executable.

In embedded DSLs, the language the DSL is embedded in is called the host language (Mernik et al., 2005). We also use *host language* to describe a GPL that an external DSL is translated to, or interpreted in.

There are many different ways of implementing external DSLs, from hand-coded interpreters/compilers to a variety of different tools (Erdweg et al., 2015). Our approach builds on the idea of composition (Erdweg et al., 2012), but unlike Erdweg, whose system works on concrete syntax, we compose on the abstract syntax.

Language composition consists of creating new languages by composing them from existing languages. That is, a new language is created by reusing elements of existing languages. Several different ways of composing DSLs have been discussed in the literature, (Erdweg et al., 2012; Völter, 2013a; Lakatos and Porubän, 2013; Chodarev et al., 2014; Degueule et al., 2015) and we have discussed them in chapter 2, subsection 2.4.1.

In the previous chapter (chapter 4) we discuss a novel way of approaching composition. We define a set of composition operators that work on abstract syntax types. The operations are:

- Introduction

  - Declaring the structure of a construct.

- Restriction of construct
  - i.e. sub-typing an existing construct
- Relaxation of construct
  - i.e. super-typing an existing construct
- Deletion
  - i.e. creating new language by removing an existing construct
- Combination
  - The union of two languages
- Translation
  - Defining the meaning of a construct in terms of other constructs.

The two main questions we want to answer in this chapter are firstly: whether our framework, in particular the translation part of our framework, is powerful enough to describe the translations needed to define its own semantics using an arbitrary GPL as a host language; and secondly to what degree we can reuse existing translation rules when we define our translation language.

In this chapter, we will go through the translation operations in more detail, and show how a translation program can be built using a translation DSL defined using itself by making use of our composition framework. A DSL for translations used to define a translation from itself to a GPL is a difficult concept. The language consists of rules for translating each construct in the DSL into equivalent GPL code, but the DSL constructs it is translating are the same constructs that are used to express the translation. This also provides us with an executable GPL version of the translation DSL, which is capable of turning other translation specifications written in the DSL into executable GPL versions. We use this to provide a

check on the correctness of the implementation, which we will discuss later. We will also show how composition affects translation in the context of translator translation.

Our model of the abstract language consists of a set of abstract language constructs (or abstract language types) that consist of a name, a structure and side-effects. The structure is either a labelled set of member types, a list with a specified list type or without any structure. We write this as $Name := Structure \rightarrow \textit{Side-Effects}$. We show the different types of structure in equation 6.1.

$$\begin{cases} T := \{n_1 : T_1 ... n_n : T_n\} \rightarrow S \\ T := \{T_1\} \rightarrow S \\ T := \varepsilon \rightarrow S \end{cases} \tag{6.1}$$

The side-effects consist of the scope (global, local, evaluation, internal) and a specification of the actual side-effect (for example, introduce symbol of type T, evaluate to type T).

The subtyping rules for lists are shown in equation 6.2 and for labelled constructs are shown in equation 6.3. Structureless constructs are always equal in their structure(lessness), and are therefore always (non-strict) subtypes of each other.

$$\frac{T_1 \leq: T_2}{\{T_1\} \leq: \{T_2\}} \tag{6.2}$$

$$\frac{T_1 \leq: U_1 ... T_n \leq: U_n}{\{n_1 : T_1 ... n_n : T_n\} \leq: \{n_1 : U_1 ... n_n : U_n\}} \tag{6.3}$$

We model translations as shown in equation 6.4.

$$E := \begin{cases} \{n_1 : E ... n_n : E\} \rightarrow H \\ \{E\} \rightarrow H \\ \varepsilon \rightarrow H \\ T'(s \in S) \rightarrow H \end{cases} \tag{6.4}$$

that is, we define a translation language type that evaluates to a host language type and consists of either:

- a labelled construct where all member types are themselves a translation language type which evaluates to a structure construct in the host language
- a list construct where the list type is a translation language type
- a structureless construct
- a set of translation functions from the source language to the host language

We have named the set of translation functions T' to distinguish from the general translation function $T(S) \rightarrow H$, which we implement using the translation language type.

We have shown in chapter 4, subsection 4.4.2, that the translation function T has the property $S_1 \leq: S_2 \implies T(S_1) \leq: T(S_2)$ when $S_1 \leq: S_2 \implies T'(S_1) \leq: T'(S_2)$ is true (that is, we showed that all but T' has this property, as T' is assumed to have this property). In this chapter we introduce some additional functions for T', show that they satisfy the property and then show how we can construct a DSL for translating other DSLs from a source language to a host language and what happens when the translation specification for that language is processed by itself. We also show how composition of the language can facilitate reuse.

So far we have discussed the language types, but we will in this chapter also have to use the instantiations of those types. The instantiation of a language type is an Abstract Syntax Tree (AST). We will use a slightly modified s-expression to write these out. The

modification is to add label:value in each labelled construct, to be able to distinguish between labelled types and list types more easily. So an instantiation of a language type $Add := \{lhs : \mathbb{Z}, rhs : \mathbb{Z}\} \rightarrow \mathbb{Z}$ could be written as `(Add lhs:2 rhs:2)`.

6.2 Method

Our approach to constructing a translation language consists of composing a new language out of three existing languages. The first language is a language consisting of general constructs to aid translation, which we make specific to specify source language translations into the host language. The translation language consists of constructs for the general structure of the translation rules (list of rules, rules) as well as some operations that can be used in the translation expression.

Now, the translation language we are constructing does not use the source and host language directly. Instead, it uses constructs that are based on the source language and constructs based on the host language. This is because we do not need to write sentences in the source or host language when we specify translations, we write sentences that translate from source to host language. The constructs derived from the source language we call S and the constructs derived from the host language we call H. The construction of S and H is fairly straightforward and we will discuss it in the next section.

In general terms, the translation language that translates input sentences in S to output sentences in H can be viewed as the composition $T \cap (S \uplus H)$. That is, the general translation language T is restricted to the combination of languages S and H. For short, we will write this as T_{SH}.

6.2.1 Construction of S and H

There are a couple of different ways we can view the construction of S from the source language.

The naive way is to create the language such that it contains constructs for each type in the source language that evaluates to the corresponding construct in the source language. The attributes for each construct in the source language would be constructs in S that evaluate to the value of that particular attribute.

The problem with this approach is that it completely decouples the constructs from their attributes and would make it possible to refer to attributes of other constructs in translation rules, which would make no sense and could not type-check.

Instead, we can construct S in such a way that every construct in S is a sub-type of the Rule construct in T. The left-hand side would be the respective type in the source language. The attributes would then be symbols in the right-hand side of the rule. An example of this is shown in listing 14.

For example, imagine a source language consisting of arithmetic operations and numbers. So valid sentences might be `(add lhs:1 rhs:2)` and `(sub lhs:2 rhs:5)`. Then we have a host language consisting of push statements and call statements along with identifiers for functions to call as well as numbers that can be pushed. We then get an S that consists of identifiers `add`, `sub`, as well as numbers. Assuming the parameters for `add` and `sub` are `lhs` and `rhs` in both operations, we also get those as identifiers, referencing the respective values provided to the arithmetic operations. For the language H we get the construct `push` that evaluates to the push and call operation in the host language as well as numbers that evaluate to themselves in the host language.

Both the source language and the host language use numbers (NUM) and they mean the same thing in both languages, so the translation rule for NUM is just a rule that generates the original number. For `add` and `sub`, more complicated translation rules are needed.

In listing 11 we show the source language and host language that we want to translate

Listing 11 Definition of source and host language. ID references any identifier, such as ‘add’ or ‘sub’.

```
Source := (ADD := {lhs:NUM, rhs:NUM}) ⊎
          (SUB := {lhs:NUM, rhs:NUM}) ⊎
          NUM
Host := (Push := {value:Any}) ⊎
        (Call := {operator:ID) ⊎
        NUM ⊎
        ID
```

between. Note that the host language would normally have many more constructs, like pop, and a construct for defining new operations. The ID construct represents all possible identifiers and is meant to capture operations defined elsewhere. We only use these four constructs here to avoid unnecessary complexity.

Listing 12 A simple definition of T, the general translation language. T_{Host} references constructs that represent the output language (i.e. host) of the translation.

```
T := (T_Rules := {T_Rule}) ⊎
     (T_Rule := {name: ID, rule: T_expr}) ⊎
     (T_expr := Any → T_Host) ⊎
     (T_Host := Any) ⊎
     ID
```

In listing 12 we show a simple definition of the translation language T. We will discuss some more constructs, but this is a minimal working definition of T. Note that T_{Host} is defined simply as Any. We use T_{Host} to highlight that T_{expr} specifically evaluates to constructs in the host language, without having to specify what the host language is. Without T_{Host}, T_{expr} could evaluate to any constructs, and thus we could not guarantee that the output of the translation is a valid sentence in the host language. For this method to work, all host constructs have to be declared to be subtypes of T_{Host}. For the rest of this chapter, we will assume that this the case.

In listing 13 we show the definition of H for the host language defined in listing 11. As can be seen, each construct in H is identical to the corresponding construct in the host language, except that each child construct is replaced with a T_{expr} and they all evaluate to

Listing 13 Full listing of H for host language. Constructs that have $_{Host}$ are constructs from the host language.

$$
\begin{aligned}
\texttt{H := } & (\texttt{Push:= \{value: T}_{expr}\texttt{\}}\rightarrow\texttt{Push}_{Host}) \uplus \\
& (\texttt{Call := \{operator:T}_{expr}\texttt{\}}\rightarrow\texttt{Call}_{Host}) \uplus \\
& \texttt{NUM := } \varepsilon \rightarrow \texttt{NUM}_{Host} \uplus \\
& \texttt{ID := } \varepsilon \rightarrow \texttt{ID}_{Host}
\end{aligned}
$$

the corresponding construct in the host language.

Listing 14 Full listing of S for source language.

$$
\begin{aligned}
\texttt{S := } & (\texttt{ADD := T}_{Rule} \cap (\texttt{\{name: ADD\}} \\
& \quad \xrightarrow{:rule} (\texttt{lhs: NUM}) \\
& \quad \xrightarrow{:rule} (\texttt{rhs: NUM}))) \uplus \\
& (\texttt{SUB := T}_{Rule} \cap (\texttt{\{name:SUB\}} \\
& \quad \xrightarrow{:rule} (\texttt{lhs: NUM}) \\
& \quad \xrightarrow{:rule} (\texttt{rhs: NUM}))) \uplus \\
& (\texttt{NUM T}_{Rule} \cap \texttt{\{name: NUM\}})
\end{aligned}
$$

In listing 14 we show the definition of S for the source language defined in listing 11. Here we define our `add`, `sub`, and `NUM` constructs as restricted versions of the $\texttt{T}_{Rule}$ construct, with the `name` child set to a static value (`ADD`, `SUB`, and `NUM` respectively) and with the symbols `lhs` and `rhs` being introduced into the `rule` child's scope. Note that our framework does not support restrictions to static values (as will be discussed later) but for now consider the static values as being constructs that only consist of one possible instantiation (`ADD`, `SUB` and `NUM` in our cases).

Another way to view S is not to have it as a language, but just instantiations of the $\texttt{T}_{Rule}$s for each construct in the source language. This would mean not using the static value, but it would also mean that there is no relationship between the source language and the $\texttt{T}_{Rule}$ and we would not be able to introduce the members of the source construct into the `rule` child of the $\texttt{T}_{Rule}$. In our implementation we are forced to use the latter approach because we are not supporting the introduction of static values, but our discussion mostly assumes that the former approach is used.

Listing 15 Example translation in a Lisp-like syntax

```
(ADD rule:(List
    (Push lhs)
    (Push rhs)
    (Call function:add)
  )
)
```

An example of a translation is shown in listing 15, which translates code such as `(Add 1 3)` into `(List (push 1) (push 4) (Call function:add))`. That is, it turns pre-order function calls into stack-machine/post-order calls.

6.2.2 The translation language

The language for specifying these translations can have its semantics defined in itself. That is, one can specify the semantics of T in terms of H using T. This language would consist of the composition $T \cap (T \uplus H)$ or T_{TH}. The output of a translation can then itself be used to translate the specification in T into a specification in H. Technically, we would actually be using $T \cap (T \cap (T_1 \uplus H_1) \uplus H)$ or $T_{T_{T_1H_1}H}$ as the source language is itself the full translation language including source and host constructs (though it does not need to be the exact same source and host languages). Note that no further composition or subscripting of T_1 is necessary; we only need to provide translations for the constructs in T_1 and H_1. For simplicity's sake, we will use $T \cap (T \uplus H)$ or T_{TH} from here on, though, as this distinction only matters when the translation (or host) language used to describe the translations is different from the translation (or host) language being passed into the translator. In our case, the translation language and host language remains the same to allow passing the definition of the translator to itself to translate. As such, the distinction does not matter in this chapter.

For our translation language we are going to introduce some new constructs that are subtypes of the $\texttt{T}_{expr}$ construct. Most of them we are only going to go through quickly here and only use the most important ones throughout the remainder of the chapter. The

important ones are:

- $expand := \{ast : ID\} \rightarrow Host$
- $AST := \{name : ID, children : \{AST\}\} \rightarrow \{Host\}$

The *expand* construct is used to expand an AST in the source language to a construct in the host language. The *AST* construct is used to construct an AST in the host language. Since the only identifiers that are available in our language are ones introduced by the side-effect in S, the *expand* construct is essentially limited to expanding children of the construct in S whose rule is currently being considered. It is important to note that the *AST* operator produces code in the host language that represents an AST, but the AST it represents can be arbitrary. This means that the host language must be able to represent ASTs somehow, which should not be a problem for GPLs. If a translation to a host language *Host* is being defined, then we need to check that each use of *AST* defines a valid AST in *Host*. Thus, the name child of the *AST* construct is technically limited to only identifiers naming constructs in *Host*, and the children must correspond to the child-types of the respective constructs named by the name child. This is not something our framework can specify at this time, but it is possible to define the translation rules so that this is still ensured.

Additionally, we have the following constructs that can make it easier to write the translation rules. Here $\mathbb{S}$, $\mathbb{B}$, $\mathbb{D}$, refer to strings, booleans, and arbitrary domain types respectively.

- $template := \{ast : ID \rightarrow Source, template : T_{expr}\} \rightarrow Host$
  - a construct that takes an identifier for a source constructs and then uses the T_{expr} to provide an alternative way of translating this particular source AST rather than using the normal translation rule.
- $concat := \{T_{expr} \rightarrow \{Host\}\} \rightarrow \{Host\}$
  - a construct that takes a list of T_{expr} that evaluates to list types of *Host* constructs and combines them together as one single list of *Host* constructs.

- $concat := \{T_{expr} \rightarrow \mathbb{S}\} \rightarrow \mathbb{S}$
  - similar to above, but combines together a list of strings.
- $name := \varepsilon \rightarrow \mathbb{S}$
  - a construct that evaluates to the name of the current source AST being translated.
- $members := \varepsilon \rightarrow set(\mathbb{S})$
  - a construct that evaluates to the names of the children of the current source AST being translated. This only makes sense for labelled constructs.
- $types := \varepsilon \rightarrow set(\mathbb{D})$
  - a construct that evaluates to the types of the children of the current source AST being translated. This only makes sense for labelled constructs.
- $member_types := \varepsilon \rightarrow set(tuple(\mathbb{S}, \mathbb{D}))$
  - a construct that evaluates to the names and types of the children of the current source AST being translated. This only makes sense for labelled constructs.
- $is_labelled := \varepsilon \rightarrow \mathbb{B}$
  - evaluates to true if the current AST being considered is a labelled construct.
- $is_list := \varepsilon \rightarrow \mathbb{B}$
  - evaluates to true if the current AST being considered is a list construct.
- $is_atom := \varepsilon \rightarrow \mathbb{B}$
  - evaluates to true if the current AST being considered is a structureless construct.
- $list_type := \varepsilon \rightarrow \mathbb{D}$
  - a construct that evaluates the type of the current source AST being translated. This only makes sense for list constructs.
- $side_effects := \varepsilon \rightarrow set(tuple(\mathbb{S}, \mathbb{D}))$

  - a construct that evaluates to the side-effects of the children of the current source AST being translated.

- $eval_to := \varepsilon \rightarrow \mathbb{D}$
  - a construct that evaluates to the *evaluates-to* side-effect of the children of the current source AST being translated.

Note that constructs that do not evaluate to constructs in $Host$ cannot be subtypes of T_{expr} and can therefore not be used directly in a translation rule. There needs to be a construct that evaluates to constructs in $Host$ that has a child-type compatible with the respective *evaluates-to* types for these constructs to be used. These might be more general constructs such as conditionals, comparisons and loops, or some other construct-specific translations.

For this chapter, we are not interested in a general discussion on translating from S to H, but rather how to translate from T to H. To do this, we need to write translation rules for T to H in our language T_{TH}. Thus, we need to define translation rules for the constructs $\texttt{T}_{Rules}$, $\texttt{T}_{Rule}$, `expand`, `AST` and the constructs in H. This is shown in listing 16.

Listing 16 Expansions rules we need to define in T_{TH}

```
(T_Rules
  (T_T_Rules rule:(expansion-in-T_TH))
  (T_T_Rule rule:(expansion-in-T_TH))
  (T_expand rule:(expansion-in-T_TH))
  (T_AST rule:(expansion-in-T_TH))
  (H_... rule:(AST ...))
)
```

Here the `H_...` represent all the various constructs in H. As they all have the same structure, an AST with “name" child set to the name of the relevant H construct and the child names the same as the ones in the H construct, and their value an expansion of the corresponding child AST, we do not show them here.

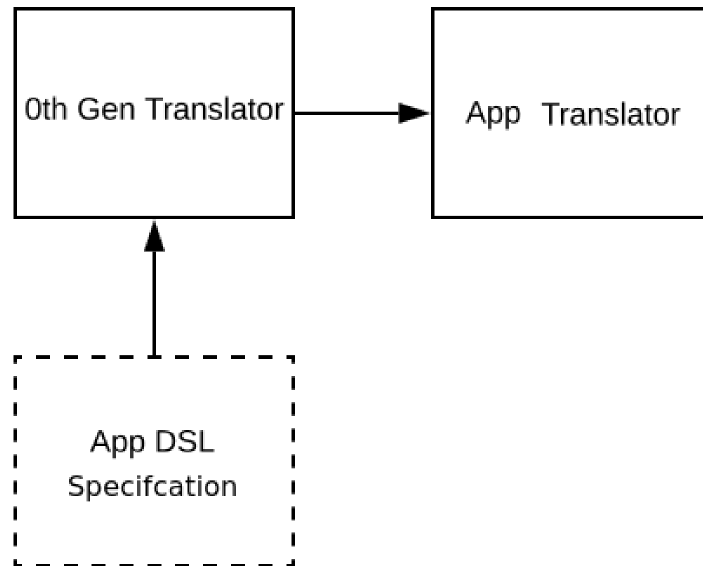

Figure 6.1: A compiler taking a DSL as an input and producing a runnable application as output.

6.3 Implementation of a translation language

To make code executable, we need to pass it through a compiler (or interpreter) that then produces executable code (and/or run it in one go). This is shown in figure 6.1. In our case, the 'App DSL' is the specification of a translator. This will output a new translator that can then be used to translate other (or the same) DSL translation specification. This is shown in figure 6.2. Here, the 0th generation translator is a manually constructed translator, while the 1st generation translator is generated from the translator DSL. The next generation is then a translator for some arbitrary application DSL.

As the translator is itself a type of translator that translates between specifications for how to translate between S and T into a program that implements these translations, we can pass the specifications for how to translate between the specifications and the translator as input to the 0th generation translator and produce a 1st generation translator that takes the same specifications and produces translators from them (i.e. we have recreated the 0th generation). This can of course be done forever, producing new (nearly) identical generators. There are some interesting features to consider when supplying the translator with the specification for itself in the first few generations.

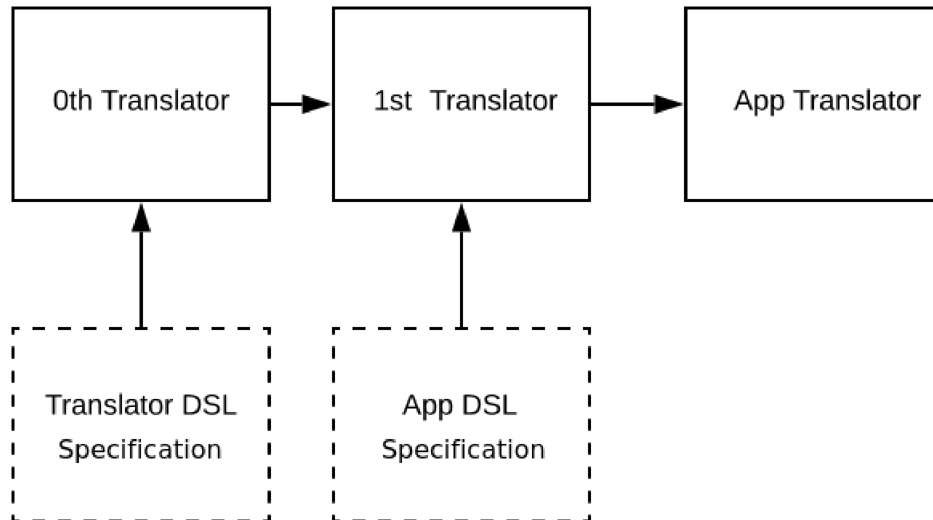

Figure 6.2: A translator-generator producing a compiler that can take application code as an input and produce a runnable application as output.

In the discussion that follows, we will use the term ‘next translator’ to mean the translator that is produced by the ‘current translator’, that is the translator that we are discussing at the moment. We will also use the term ‘previous translator’ to specify the translator that produced the current translator.

There are endless possibilities for implementations of our translator, but we will go through one sensible approach that should apply in most cases.

If we consider the 0th generation, it will consist of some structural code, such as function definitions, classes (if in an Object-Oriented Programming (OOP) language), if-statements and loops. These do not directly influence the output, they are just the scaffolding that any program needs in order to run.

There are also generative statements that *does* generate code for the next generation, but that code is itself structural. We will call these static generative statements. Finally, there are generative statements that generate code in the next generation depending on the DSL specification input, we will call these dynamic generative statements.

In a well-behaved translator, the static generative statements need to create only structural code. That is, they cannot create any code that generates code. There are two reasons for this, first, the output of a translator should be based on the input translation specifica-

tion. Since the static generative statements are not dependent on the input, the translator that is generated would generate code that is not specified in the translation specification. Not generating the translator according to the specification is generating it wrong.

The other reason that static generative statements cannot generate generative statements is that the output (as well as input) language of a translator is given in the translation DSL specification. Any translator knows only what language it is supposed to output (or take as input), it cannot know what the next generation of translator is supposed to output, without knowing the DSL specification for the next generation. As static generative statements does not have this specification by definition, they can thus not know what language the next generation is supposed to output.

The dynamic generative code is then responsible for all the generative code in the next generator. These can not produce any structural code in the next generator. In the same way as the static generator code does not know the input and output language for the next generator, the dynamic generator code does not know the output language that the current generator translates to. A specification for a translation DSLs (as well as any other DSL) can be passed through any translator that can understand the language the specification is written in, regardless of what language the translator outputs.

Which part is produced by what is shown in figure 6.3.

6.3.1 Informally checking correctness of translation DSL

The scheme presented in the previous section, allow us to create a way of testing that the system works as intended. As is seen in figure 6.3, the 2nd generation is fully dependent on the DSL specification. Generation 1 structure is dependent on the 0th generation's static generative statements, which are not dependent on the DSL specification. The static generative statements in the 1st generation are dependent on the DSL specification though, as they are generated by the 0th generation's dynamic generative statements. In the 2nd generation, the structure is generated by the static generative statements in the 1st generation, which are

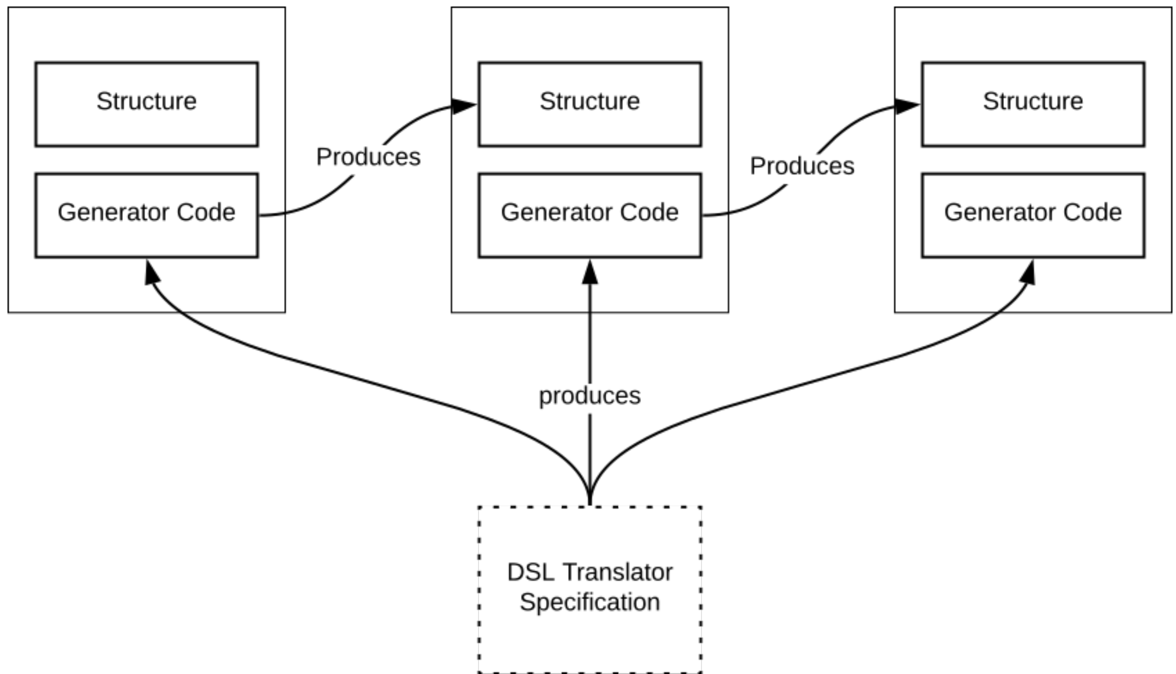

Figure 6.3: Three generations of generator, showing how each part of the generator affect the next generation.

dependent on the DSL specification. The generative statements are similarly dependent on the DSL specification.

Thus, regardless of how the 0th generation is producing code, the 2nd generation will always be the same (given the same DSL specifications), assuming the 0th generation is working correctly. This also means that if the 2nd generation is called with the same DSL specification, it will produce an exact copy of itself (as the 3rd generation is also fully dependent on the input DSL). So, if we generate until the 2nd generation, and then check that the output of the 2nd generation is identical to itself, we can be pretty sure that the 0th generation is producing correctly (as well as subsequent generators) and that the DSL specification is correct.

While it is possible to produce a 0th generation that is capable of generating a 2nd and 3rd generation that are the same, but do not generate according to the specifications, it is unlikely to happen by accident. Any mistakes in either the 0th generation translator or the translator DSL specification are likely to cause the output to not be a valid code, as it requires

a certain amount of precision to generate code that runs. Even if the 0th generation manages to produce a running (but faulty) 1st generation, the 2nd generation is again unlikely to run, and even less likely to produce a copy of itself for the 3rd generation.

The one exception to this is if the 0th generation is a Quine, that is, a program that reproduces itself when run. In that case, the 2nd and 3rd generation will be the same, but the translator will be absolutely useless in translating translator specifications. From our own experiences, we can say that it is possible to accidentally produce a Quine as a 0th generation, if one does not pay enough attention when writing the logic for how the dynamic generative statements generate the next generation.

It is possible that some features that are not needed when reproducing itself get lost in translation. To check this, the different translators should be supplied with various other DSL specifications for other languages. In this case, they should all produce translators that behave the same, and the 1st translator and subsequent translators should produce identical code. Here we can't simply rely on the fact that the code behaves the same to verify if the translation worked, we also have to check that the generated translator works as expected. Otherwise, all the 0th translator could do is to generate a translator that always produces the same output regardless of input and the test would still pass.

The 0th generation can be produced through manual coding, or from some other type of translator-translator. This is shown in figure 6.4. The -1st generation is any translator-translator, including a purely imagined one (if the 0th translator is coded manually). This also means we can check translator-translators that translates to different languages. If we have a translator-translator that translates to language H, we can check it by providing it with the specifications for translating to language G, then treat the output as the 0th generation and run the test as described above. We can then redo the test by passing the specification for a translator translating to H to the 0th generation and redo the test on the output. This provides some confidence that both the specification for translating to G and to H are correct.

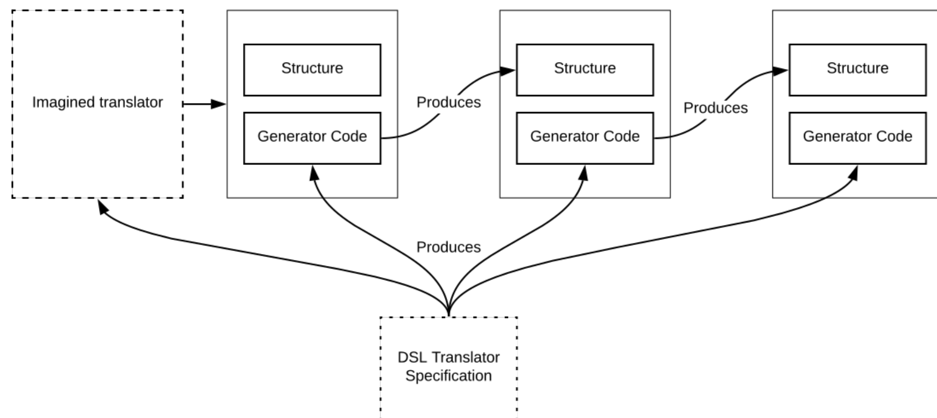

Figure 6.4: Three generations of generator with an imaginary -1st generator that produces the 0th generator.

6.4 Example translation rules

As translation rules for real world GPLs tend to require a fair bit of code, we will here describe what a translation of a translation DSL specification would look like using a simplified toy GPL, let us call it P for pseudo. The participating languages are thus T, P, and the translation specification DSL. If we name the H language for P H_P, then our translator language would be T_{TH_P} (or more precisely $T_{T_{TH_P}H_P}$). We thus need to provide translations for all constructs in T_{TH_P} into constructs in P. Using our translation specification DSL that we described earlier in this chapter, we do this by implementing a T_{Rule} for each construct in T_{TH_P}, with the rule for how to translate it specified using the T_{expr} in H_P and T.

Because the input language is the same language as the rules are written in, it can become a bit difficult to tell the difference between the two. In the examples below, we have prepended the input language constructs with a "`T_`" to make it easier to tell them apart.

Listing 17 Expansion rules we need to define in T_{TH}

```
(T_Rules
    (T_T_Rules rule:(Function name:do_expand params:(List ast)
           code:(List
             (expand ast:$0)
             ...
             (expand ast:$n)
    )))
    (T_T_Rule rule:(If cond:(Eq lhs:(Dot lhs:ast rhs:type)
                                  rhs:$type)
           code:(Return expand($rule))
    ))
    (T_expand rule:(Call
           fun:do_expand
           args:(List
                (Call fun:(Dot lhs:ast rhs:get)
                      args:(List $ast))
    )))
    (T_$ID rule:(Call
           fun:(Dot lhs:ast rhs:get)
           args:(List <ID>)
    ))
    (T_AST rule:(AST
           type:"Call",
           children:(List
             "AST",
             AST("List", (List
                  (expand ast:$0)
                  ...
                  (expand ast:$n)
             ))
           )
    ))
    (T_If rule:(AST type:If children:(List
           (expand ast:$cond)
           (expand ast:$body)
    ))
    (T_Return rule:(AST type:Return children:(List
           (expand ast:$value)
    ))
    ...
)
```

Listing 18 Expansion rule translations in GPL, first half

```
function do_expand(ast) {
    if ast.type == "T_TRules":
        return AST("Function", "do_expand", ["ast"], [
                    do_expand(ast.get(0)),
                    ...
                    do_expand(ast.get(n)),
                 ])
    if ast.type == "T_TRule":
        return AST("If, AST("Eq",
                            AST("Dot", "ast", "type"),
                            AST("Call",
                                  AST("Dot", "ast", "get"),
                                     ["type"]
                            )
                         ),
                         [AST("Return",
                              do_expand(ast.get("rule"))
                         )]
               )
    if ast.type == "T_expand":
        return AST("Call",
                    "do_expand",
                    [AST("Call",
                         AST("Dot", "ast", "get"),
                         [ast.get("ast")]
                    )]
               )
    if ast.type == "T_$ID":
        return AST("Call",
                    AST("Dot", "ast", "get"),
                    [ast.value]
               )
```

Listing 19 Expansions rule translations in GPL, continued

```
      if ast.type == "T_AST":
        return AST("Call", ["AST",  [
                       ast.get("type"),
                       AST("List",
                          do_expand(ast.get(0))
                          ...
                          do_expand(ast.get(n))
                       )
              ]])
      if ast.type == "T_If":
        return AST("Call", ["AST", [
                      "If",
                      AST("List",
                          do_expand(ast.get("cond")),
                          do_expand(ast.get("body"))
                      )
               ]])
      if ast.type == "T_Return":
        return AST("Call", ["AST", [
                      "Return",
                      AST("List",
                         do_expand(ast.get("value"))
                      )
               ]])
    ...
}
```

In listing 17 we show the rules for translating our translator written in the translation DSL. In listings 18, 19 we show what the output would be when supplied to a translator for translating the translation DSL specifications into our toy GPL. In the translation DSL, we use Function to represent functions in the GPL, Dot represent member access (e.g. ast.type, Eq represents equality test ("=="), List represents a list of something, Call represent function calls, and Return represents return statements. We use $<ID> in order to refer to symbols introduced by side-effects. In order to keep these examples at a reasonable length, some additional simplifications have been made. In particular, in places where loops would be required, we simply used the first and last element, with an ellipsis in between.

As can be seen, especially in listing 19 the translated rules can become quite large, which is why we show simplified code in this chapter.

There are a couple of things we want to point out before we go through how the translation is made. First, the rules for `T_$ID` should ideally be governed by side-effect translations. The constructs that the `T_$ID` translates are all introduced through side-effects and so it would be cleaner to have separate rules for how to translate such side-effects. The rule as it stands essentially says that an identifier in the input language corresponds to a member access of the AST being translated in the output language.

Another thing to clarify here is the expected type of all the different translation rules. All the rules translates to sentences in the output language, but the type of sentences can be split up into three main types. The translation rules for constructs in T translates into structural code. That is, they translate to arbitrary sentences in P, except for statements that generate code. The exception has to be manually enforced, as there is technically nothing preventing a generative statement from being specified in the translation rule.

Translation rules for H_P translate into static generative statements that generate sentences in P. Remember, constructs in H_P are all constructs that generate constructs in P, as per our definition in section 6.2.1. This has to remain true both in the input language and the output language, which is why the translated rule is always a generative statement that generates an AST corresponding to the type of the P_H construct being translated.

Finally, the T_AST construct translates to a dynamic generative statement, that can generate arbitrary output code in any language. This is the statement that allows the user to specify generative statements in the translation DSL specification to create translation rules for the output language. As can be seen in listing 17, the translation rule for `T_Return`, representing the *Return* construct in H_P uses an AST statement for its specification. Likewise, every other construct in H_P will have a similar translation rule; an `AST` instance with the type set to the corresponding type and with the children of the `AST` being each of the children of the construct expanded.

In short, sentences in T generate structural code, sentences in H_P generate code that generate sentences in P and the AST construct generates code that generates arbitrary code. This needs to be reflected in the translation rules for each of the constructs.

Let us go through how the translation rules in listing 17 would get translated by the translator in listings 18, 19 by examining how a couple of the rules would get translated. In this explanation, we will use generation 0 to refer to the initial GPL translator that is translating the translator DSL specification into generation 1 GPL code.

First, all the rules in the translator DSL specification are contained in an AST of type T_T_{Rules}. When the generation 0 translator sees this AST, it outputs an AST representing a function that takes an AST as input, where each of the statements in the body are the translation of the rules that make up the translation DSL specification.

Each one of these rules is represented by an AST of type T_T_{Rule}. When the translator sees the T_T_{Rule} AST, it translates it to an AST representing an if-statement. The if-statement condition checks if the type of the *ast* variable that generation 1 will be translating matches the type of the T_T_{Rule} that generation 0 is currently translating. The body of the if-statement consists of an AST representing a return-statement, where the value of the return-statement is the translation of the T_T_{Rule} ASTs 'rule' child. As such, the T_T_{Rule} statements are translated into guarded commands (Dijkstra, 1975).

This is a point where it is easy to get confused, as there are two different 'ast' being referred to here. The first one is the 'ast' that is the parameter given to the translator function in the generation 1 translator that we are generating, which could be any type of AST. The other 'ast' refers to the AST that is currently being translated which, if we are at this point in the code, is always a T_T_{Rule} AST. Note that all the constructs that start with `T_` (in our example) are of type T_T_{Rule}, as described in 6.2.1.

When translating the 'rule' child of a T_T_{Rule} AST, we call `do_expand` of generation 0 to generate an AST representing code that generates ASTs according to the specifications in the 'rule' child. If the 'rule' child contains a `If` AST, for example, then the `do_expand`

will go to the relevant guarded command and return the AST specified there. In this case it would reach the if-statement for the `T_If` AST shown in listing 19. As can be seen, even though the if-statement is a fairly simple construct, the translation rule is quite complex.

The rule consists of an AST representing a call to an `AST` function, where the first parameter is "If" (i.e. the name of the AST to be created) and the second parameter is a List AST containing the translation of the ‘cond‘ and ‘body’ children. It is thus an AST representing the creation of an AST for an if-statement in the output language. If this was the if-statement specified for the T_T_{Rule}, then the ‘body’ child would contain a return-statement, which would be translated in a similar manner to the if-statement, as shown in listing 19.

The `expand` ASTs that appear in the various rules are translated as an AST representing a call to a ‘do_expand’ function, with the specified child of the generation 1 ast as a parameter. This ‘do_expand’ function is the generation 1 function that translates ASTs, which is the same function that is generated by the $\texttt{T_T}_{Rules}$ translation specified in listing 18.

The last rule we will look at is the one for translating the *AST* construct. It might be tempting to think that the *AST* construct should be translated as `AST(ast.get("type"), [do_expand(ast.get (0))...do_expand(ast.get(n))])`, but that would generate a generation 1 translator with code written in the output language of generation 1, which is (potentially) different from the language generation 1 is implemented in (as generation 1 is written in the output language of generation 0 and there is no guarantee that they will be the same). That is why the *AST* construct is being translated to a AST representing a construction of an AST in generation 1. That is, the translation of the *AST* construct produces an AST that represents an AST, rather than an AST that represents arbitrary code. As mentioned above, the *AST* construct is thus generating dynamic generative code.

There are a couple of ways that the translation can go wrong. Firstly, one might accidentally view the code in the ‘rule’ child of the translation rules in the translation DSLs as being GPL code rather than being code that generates the GPL code. For example, in the

rule:

```
(T_T_{Rule} rule:(If cond:(Eq lhs:(Dot lhs:ast rhs:type)
                                  rhs:$type)
                     code:(Return (expand $rule))))
```

the translation could accidentally be viewed as a literal if-statement in the output language, generating the code:

```
if ast.type == "T_T_{Rule}":
     return if ast.type == ast.get("type")
                 return do_expand(ast.get("node"))
```

which is both semantically and syntactically (at least in most languages) wrong. It is also possible to go wrong in the other direction, generating all code as ASTs of ASTs of code. The previous rule would in this case be generated as

```
AST("If", AST("Eq", AST("Dot", ["ast", "type"]), "T_T_{Rule}")
      AST("Return", AST("AST", ["If",
            AST("AST", ["Eq", AST("Dot", ["ast", "type"]),
                  AST("Call", [AST("Dot",
                                    ["ast", "get"]),
                                        "type"])]),
            AST("AST", ["Return", AST("Call",
                           ["do_expand", AST("Call",
                              [AST("Dot",  ["ast", "get"]),
                                     "node"])
                           ])
            ])
      ]))
)
```

This may be syntactically valid in languages that allow function calls in the global scope, but it is not what we want semantically, as it does not produce a function we can call to translate languages.

Beyond these two types of errors, any mistakes in the generative statements (or the translation DSL specification) would of course also produce erroneous output. The errors might appear either in the next generation or two generations ahead of the current generation of the GPL translator, depending on whether the error is in the code that produces static statements or generative statements. The errors can either be present in the translation DSL specification (in which case it will appear in two or three generations) or the generation 0 GPL translator.

6.4.1 Reuse in the construction of T_{TH}

Listing 20 Translating to functions

```
function do_expand(ast: T_T_Rules) {
    return AST("List",
                do_expand(ast.children[0]),
                ...
                do_expand(ast.children[n])
          )
}
function do_expand(ast: T_T_Rule) {
    return AST("Function", "do_expand",
        [AST("Param", "ast", ast.get("type"))], [
        AST("Return", do_expand(ast.get("rule")))
    ])
}
...
```

Let us consider how translation rules can be reused to reduce the number of rules that need to be written for any one translation DSL. Firstly, the output language rules can both be generated automatically from the output language, and are also always the same regardless of input language/translation specific language constructs. As such, the output language

translation constructs need only be generated once and can then be reused in all translation DSLs.

For the other constructs, reuse is a bit more complicated. For example, we could view the rule construct as a restricted if-statement. The first issue we encounter is that we have called the children of the rule construct "type" and "rule", whereas an if-statement would have children called something like "condition" and "block" (or "true_block", "false_block" if it is an if-else statement). Our framework does not have a facility for restricting constructs with differently named children, though we could simply rename the children of the rule construct to make them match the if-statement. It is not a neat solution, but it is what we are forced to do if we want to reuse the translation rule for if-statements.

The next issue is that the "type" child would need to be of type BoolExpr and the "rule" would need to be of type List of Statements. The "type" child can simply be declared as a subtype of BoolExpr, as long as it is declared with the evaluates-to boolean value side-effect (which is not necessarily an intuitive way of declaring the "type", after all, the "type" child is supposed to be referencing a construct in the input language, not a boolean value). Similarly, the "rule" child can be declared as a list of statements, as long as the constructs that generate the output language are also declared as statements, as well as any translation specific constructs that are to be used in the "rule" child. Note that it is still necessary to provide translation rules for the constructs used in the "type" child and the "rule" child, as neither BoolExpr construct nor statements have any general translation rules.

Some languages support function overloading, where the multiple functions share the same name, but have different arguments and implementation. In such languages, we can construct our rules as functions that each handle a different type of construct, as seen in listing 20. In this scenario, we could see rules as a restriction of a function. In this case, the same issue with different child names occurs, but we also have another issue. The function has a name associated with it, and the rule construct has no equivalent child. This could be mitigated by having a composition that allowed the insertion of static children to yield a

subtype relation between two constructs. Unfortunately, our composition framework does not have such a composition and it is thus not possible to restrict a function as a way of reuse for the rule construct. It should be noted, though, that if we were able to do the restriction, the "type" child of the rule construct would be referencing an input AST type rather than a boolean as is the case for the if-statement. This suggests that functions might be a better approximation of the rule constructs, besides the missing "name" child.

We can do a similar thing with the root construct, Expansions. The expansions construct can either be viewed as a list construct consisting of the rules for the individual translations or a labelled construct, with additional children containing, for example, information about intended input and output languages and a specific child containing all the rules. The labelled construct will yield the same issues with misnamed and missing children, but it also causes another issue. As the Expansions construct is a root construct, it needs to be translated into a root construct in the output language, but a function on its own is rarely considered a root construct in GPLs. In GPLs the root tends to be a list of statements, which of course could be a list containing just one function declaration, but this is structurally very different from a function declaration on its own.

The list construct would be directly, without changes, restricted from a root list of statements construct, though this only make sense if the rules are translated as functions (or other constructs allowed in the global scope) as described above. If the rules were translated to if-statements, reusing the translation rule for *list of statements* would generate code with nothing but if-statements in the global scope (which, granted, is valid code in some languages).

Note that the above only matters when we want to reuse existing translation rules, and none of the mentioned issues prevents a manually created translation rule from being created.

The expand construct can be restricted from a function call operator that calls the `do_expand` function. Here again, the structure of the expand needs to have an equivalent structure

to the call construct for this to work. The child specifying the AST to be expanded also needs to be an expression that evaluates to an AST. This should not be an issue, as that is exactly what is expected for the expand construct.

It should be noted that when we talk about reusing existing translation rules, we are not talking about the rules for the H language, but rather the constructs from the output language of the current translator.

As is apparent by now, reuse through restriction is not straightforward, at least not for our example in this chapter. Reuse through combination on the other hand allows us to reuse the entirety of the H-language in all translators that use it. There are however issues that arises from using combination to facilitate reuse. If we wanted to reuse translation constructs from another language, they would have to respect the naming conventions used in the translation for the Expansions and Expansion construct, such as the name of the `do_expand` function and the variable `ast`. Some of this can be mitigated by properly using side-effects and side-effect translations as discussed in chapter 4, it is not always possible to ensure that no clashes occur, as it is in general a non-local concern (Eden et al., 2006).

6.5 Results and discussion

We have shown how our translation framework can be used to define the semantics of itself using translations and discussed some of the implications of doing so. We have also shown how using composition techniques can reduce the number of translation rules that are required to be implemented for any one translator, which makes the construction of new translators more feasible.

We have also shown the effect of passing a translator specification to a translator over several generations and how this can be used to reason about the correctness of the implementation. The test for correctness we presented is very much a required but not sufficient test for whether a translator (and translator specification) is correctly implemented. The test

relies on the fact that after two generations, the generated code is wholly dependent on the translator specification, and no artefacts from the original translator should remain. Thus, after two generations, the code that is generated should always be identical for the same translator specification. This test fails to be useful if the original translator is a quine (or generates a quine). If the original translator either is a quine or generates a quine, then the test will pass, even though the translator itself is useless for anything other than generating itself. Accidentally creating a quine is possible if too much of the original translator is hard-coded, completely removing input-dependant code-generation from it. It is straightforward to check that the translator is not producing quines by passing it a non-translator DSL specification and ensuring that the generated translator correctly translates the non-translator DSL (and does not translate the translator DSL). Other errors in either the original translator or the translator specification will in general cause errors, with the exception of errors that are specifically designed to pass this test, something we do not worry about. Note that this test, in the best of cases, only covers the constructs used by the translator specification, which may not be all the constructs provided by the translator DSL. Thus, it is important to test the translator in different ways as well, such as by translating DSLs with well known behaviour to ensure that the translator translates these correctly. These DSLs can be translated both by the 0th generation translator and the 2nd generation translator, for extra assurance that the translator translates correctly.

Another result we found was that in practice (at least in the case of the translator DSL), reuse through restriction is tricky. The main issues found are structural mismatches between constructs, missing child constructs and mismatches between root constructs. One possible explanation for these issues is that there is too large an abstraction distance between the input and output language, making it difficult to reconcile them through restriction. Some of these issues can be mitigated with further expansion of the composition framework, in particular by adding support for restriction facilitated by renaming of child constructs and restriction facilitated by adding static children to the restricted construct. In the former case,

we essentially create a new construct, that has the same child-types, but with child-names matching our preferred construct to restrict from. The exact names of child constructs have some arbitrariness to them, as they serve more as mnemonics for the programmer than any semantic purpose in the language. This means that a lot of time allowing renaming of child constructs is quite sensible, assuming that the languages being worked on are already finalised, and it is therefore not possible to change the original construct's definition. There are of course many cases where two constructs are modelling completely different things, and in those cases it makes little sense to force the children to have the same name, as reusing the translation for both constructs will likely lead to the wrong code being generated.

In the case of adding static children, the aim is to fill in missing child constructs with a static value, in order to be able to reuse translation rules that require that child construct to exist. The motivation for such a composition operator is that in certain cases, a child construct in a construct subtype becomes redundant, as all instances of the construct have the same value for that child. For example, a construct that only models people called 'Pete' does not require a 'name' child construct, as it will be the same for all instances. On the other hand, a construct modelling people more generally would require the 'name' child. In this case, we would need to add the 'name' child back in so that we can reuse the translation rule for the more general construct (assuming the translation depends on the 'name' child being present).

Note that it is possible that the effort required to reconcile mismatching construct might actually be greater than simply writing a new translation rule from scratch. Unfortunately, we are not at this time able to reason about the effort each approach requires, as the framework we are using does not support either of these composition operations.

One place where the composition framework reduces the number of translation rules needed is in using the combination operator to combine the general translation constructs (T) with constructs derived from the input language (S) and the output language (H). The H language tends to be very large, as it contains a construct for each construct in the output

language, which is often a GPL. Through combination, the translation rules for H can be reused in any translator specification that translates to the output language from which H is derived. Other translation frameworks often get around having to specify the rules for the output language by allowing arbitrary strings to represent the output. The benefit of composing the translation language in our way is that we can type-check the rules and guarantee that the generated code is valid (as long as there are no non-local references, anyway). We thus get the benefit of early detection of errors in the translation without adding a lot of extra work.

Finally, a note on the `AST` construct first described in subsection 6.2.2 and referenced throughout this document. As described in section 6.3, an important property of the translation language is that static generative statements only generates structural code and that all code for generative statements in the output translator are fully defined in the translation DSL the translator is translating. That is, on its own, a translator should only generate structural code, no generative code, and all generative code is defined only in the input DSL. After finishing this chapter, we noticed that the `AST` construct could not adhere to this property. In order for the `AST` construct to function properly, it needs to generate not just structural code, but also generate generative code. The generative code it has to generate is code that construct an AST in the output translator. The effect of this is that the output translator gets locked into the output language it can translate to. In a correct translator, the output translator should be able to translate from any input language, to any output language as long as the translation rules are written in the correct translation DSL. With the `AST` construct, the output can only ever be in one language, whichever language the statements for constructing ASTs are written in.

In our case, where we repeatedly feed the generated translator with its own translation definition, it does not matter, as the input and output language remains the same. This is why the problem with the `AST` construct was not noticed earlier. The `AST` construct was introduced to simplify the rules for translating host language constructs and it is still possible

to achieve the same result, it just requires more code to be written.

6.6 Conclusion

In this chapter we have discussed how translations in our framework work in detail and we have shown how a translation DSL can be created using our composition framework and translations. We have also shown some of the issues that arise when trying to reuse translation rules using our composition framework.

At the start of this chapter, we said there are two questions we want to answer: whether our frame-work is powerful enough to describe the translations needed to define its own semantics using an arbitrary GPL as a host language; and to what degree we can reuse existing translation rules when we define our translation language.

The first question, whether we can define the translation part of our framework in itself, is clear from the fact that we did just that in this chapter. In this chapter we showed the implementation by explaining a selection of the operations in the translation framework in detail and then more broadly explained the rest of the operations. The framework has also been fully implemented in Java, to verify that it works with real programming languages as well as the pseudo-language used in this chapter.

For the second question, the answer is more complicated. We showed that our framework is unable to reuse translation rules for several of the constructs we discussed. The solutions for some of these inadequacies is to introduce more operations to our framework. We briefly discussed two possible operations, one for adding static child-nodes to the constructs, and one for renaming child-nodes. These additions would, however, not solve all the problems with reuse that we uncovered. One possible reason for the inability to reuse constructs in the host-language could be that the source-language and host-language have too big an abstraction gap between them.

The one case where reuse was possible without modification is for the combination

operator. By separating out the different parts of the translation language, we can reuse the constructs that produce the constructs of the output language. It is worth noting that these constructs can only really be used for other translation languages that translates to the same output language, but it does mean that it is possible to create translators with different input languages but the same output language more easily. It should also be noted that the reuse through combination works due to none of the translation rules for the constructs having any non-local dependencies. Any reference to symbols outside of the translation rule could potentially cause conflicts. This issue will always exist as in general it is impossible to use composition in the presence of global constraints (Eden et al., 2006).

In the future, it would be interesting to continue investigating the effects of composition on translations of translation DSLs, but using several different GPL host languages and cross-compiling between them. This could provide more robust ways of validating translation DSL implementation using the method described in section 6.3.1.

This chapter only covers the translation part of our framework and in the future it would be good to cover the entire framework. This would include a language for defining the structure of a language using composition and translation rules for turning sentences in that language into sentences in a GPL. This way of having multiple DSLs all working together is called globalisation and presents with it some additional challenges (Cheng et al., 2015).

All in all, this chapter shows that complicated self-referential systems can be built using our framework for composition and translation of DSLs, while also showing some of the short-comings and potential improvements to the system.

Chapter 7

Comparison to the State of the Art

7.1 Introduction

In order to reason about the contributions of this thesis, we will in this chapter review what other similar work has been published in the literature and how our approach compares to the existing literature.

7.2 Background

Composition has been used for constructing new Domain-Specific Languages (DSLs) for well over a decade now (Cleenewerck, 2003; Estublier et al., 2005). The goal is to be able to easily create new DSLs by reusing existing DSLs.

Several different techniques for composing DSLs have been suggested, such as Aspect Oriented Programming techniques (Estublier et al., 2005; Dinkelaker et al., 2010), using inheritance (Ghosh, 2010; Völter and Solomatov, 2010; Mernik, 2013), functional/-monad composition (Ghosh, 2010), Traits (Cazzola and Vacchi, 2016), as well as manual approaches (Erdweg et al., 2011). For the most part, these techniques work on the concrete syntax of the language, often with the abstract syntax and semantics being tightly coupled and not considered on their own. This is not always the case, though, with for example

(Chodarev et al., 2014) discussing composition of the abstract syntax.

In terms of the type of composition operations that can be used to construct DSLs, (Erdweg et al., 2012) gives a popular set of definitions:

- Language extension – A language is extended with additional concepts
- Language unification – Two languages are combined to form a third one
- Self extension – A language that is able to extend itself
- Extension composition – A combination of the above

These definitions are used by (Mernik, 2013) to discuss how the language workbench LISA (Mernik et al., 2002) handles the different types of composition. In (Chodarev et al., 2014) show how they can be mapped to composition of abstract syntax, and (Cazzola and Vacchi, 2016) also refer to these definitions when discussing the types of compositions supported by their tool.

Erdweg's definitions are not formally defined, which make it easier to use them to describe the operations in existing DSL composition framework, but makes it harder to reason about the consequences of applying them when constructing DSLs.

Besides Erdweg's classifications, there are several other competing classifications. In Lakatos and Porubän (2013), the following classifications are used:

- extension (full language is used and new concepts added)
- specialization (reuse of only some of the concepts of the language)
- insertion (code from one language is used inside other language)
  - direct/embedding (full parts of code are used)
  - referencing (only identifiers from other language are used)

(Völter, 2013a) describes composition of DSLs which is different to Erdweg's. Völter identifies

- Referencing – concepts in a language references concepts in another language

- Extension – a language is based on another language, but includes new concepts
  - Restriction – like Erdweg, Völter sees restriction as a special case of extension
- Reuse – a language reuses concepts in another language through indirection
- Embedding – a language is embedded in another language

Some of the problems with composition include grammars not being closed under composition (Erdweg et al., 2012; Chodarev and Kollar, 2016), and non-local dependencies in generated code (Cleenewerck, 2003; Dinkelaker et al., 2013). Erdweg et al. also discusses the complications in implementing language unification, and ascribes it to the lack of common backend for the languages. Finally, when code is generated, it is preferable if the generated code type-checks and is compilable, as the user of the DSL will otherwise be presented with unintelligible errors from a language different to the one they are implementing their code in. This is addressed in (Lorenzen and Erdweg, 2013, 2016) using lambda calculus.

7.3 Comparison

Our work touches on all of these problems in our attempt to provide a formal technique for composing DSLs and investigate composition's effect on the reuse of translation rules from the composed language to some pre-existing language. We specifically look at the case when there exists a language L' that has been composed from some existing languages $L_{1..n}$ and the languages $L_{1..n}$ all have rules for how they are to be translated into some host-language L_h. The question is to which degree the translation rules can be reused in order to translate sentences in L' into valid sentences in L_h.

As we specifically work with abstract syntax, we only provide some short thoughts on how one might deal with composition of grammars in the context of our composition technique. One of our main focuses in this work is to which extent translation rules can be reused. As part of this, we show how translation rules that have only local dependencies can safely be reused. We also discuss some of the ways that some non-local dependencies can

be mitigated. As far as we know, such in-depth discussion on the effect of composition on the reusability of translation rules has not been published before, and we believe that our approach provides a useful understanding of the limits of compositional techniques abilities to allow for reuse of code.

In our work we also separate out concerns in ways that are less common in the literature. We separate between concrete and abstract syntax, which as (Chodarev et al., 2014) notes is not too common in the field. Unlike, for example (Erdweg et al., 2012), we view composed languages as completely new languages. Whereas Erdweg et al. speaks about extending a language by using composition, we would view the end result of composition as a new and separate language. We also separate between the abstract syntax which we view as the (abstract) structure of the language, and the semantics of the language, which is in our case provided by the translation rules. Thus, the composition techniques we define work on the abstract language, and our goal is then to map pre-existing semantic definitions back into the newly created language.

The problem with directly comparing our approach to existing approaches is that the majority of existing approaches are often tightly coupled with an existing tool, framework or General Purpose Programming Language (GPL), whereas our approach is based on mathematical constructs. The advantage of using existing technology is that the technology usually comes with existing features that enhance the development approach. The downside is that the approach then only works with one specific tool.

As far as we are aware, there has not been any earlier work on examining the extent to which composition affects translation of languages, whether on the concrete syntax or abstract syntax level.

7.4 Conclusion

While there has been a lot of work on composition so far, our approach differs from other approaches by being based in mathematical constructs and not being tied to any particular tool or language. Ours is also the only work we know of that has specifically investigated the effect composition has on translation of languages.

Chapter 8

Conclusion and future work

8.1 Introduction

In this thesis, we have presented a framework for composition of Domain-Specific Languages (DSLs) and discussed the way composition affects the reuse of translation rules when translating a composed DSL into a General Purpose Programming Language (GPL).

It builds on previous work on DSL composition, as presented in (Erdweg et al., 2012; Völter, 2013a) among others. Our work focuses on composing abstract languages using a novel approach and showing how our approach can be used to reason about reuse, in particular reuse of translation rules for translating a composed DSL into a GPL.

Our work includes the definition of our framework for composing abstract languages, the definition of our translation framework between languages defined using our composition framework and then shows the framework in use, as well as evaluates how much reuse can be achieved within the context of translation rules.

8.2 Big picture

The work discussed in this thesis covers only a small part of DSL development. The larger field it works in is Software Language Engineering (SLE), the field of creating software lan-

guages (Kleppe, 2008). The field of SLE is large and our work is mostly confined to a small portion of it, of defining structure and meaning of abstract languages through composition and translation. The work is thus limited both in the implementation method used and the specific part of languages we work with.

There are numerous different ways of implementing DSLs (as well as GPLs) and the aim of composition as an implementation strategy is to leverage reusable language components to simplify the construction of new languages (Chodarev and Kollar, 2016).

Software languages are often split up into concrete syntax, abstract syntax, and semantics (Kleppe, 2008), (Völter, 2013a, p. 26), though sometimes it is only split up into (concrete) syntax and semantics (Turner, 2017).

Translation is one of four common ways of specifying the semantics of a language, as given by (Kleppe, 2008): denotational, pragmatic, translational, and operational.

Outside of implementation, DSL development phases includes decision, analysis, design, and implementation as given by (Mernik et al., 2005) and domain engineering, design, implementation, testing, deployment, evolution, recovery and retirement as given in (Barišić et al., 2012).

There are also other concerns surrounding the DSL development process, such as implementing common Integrated Development Environment (IDE) features such as error highlighting, help texts, syntax highlighting, and debuggers (Zdun and Strembeck, 2009).

As mentioned, our work is specifically about implementing the abstract syntax of languages using composition, as well as the translation of such languages into other languages. In the future, our work could be extended to include concrete syntax, as part of turning the framework into a full-featured tool for DSL development. As it stands today, the framework's main purpose is to investigate what is theoretically possible when using composition and translation to create an abstract syntax for a DSL.

One of the benefits of working on the abstract syntax is that it removes the complexities of context-free grammars. This lets us focus on ensuring that the composition and translation

generates structurally correct code without having to consider the effects on the concrete grammar. For a complete language composition framework, it is necessary to consider concrete grammar too, but as we are interested in only a specific subset of the issues involved in composing languages, not worrying about grammars lets us focus on the parts that are of interest to us.

8.3 Contributions

Our main contribution in this thesis is the work around a framework for composing DSLs using formal methods. In particular, we investigate how the composition technique interacts with translation rules between languages. That is, we investigate to which degree translation rules for a language can be reused for a language composed from the original language.

There are several frameworks that uses composition to ease DSL development, such as (Dinkelaker et al., 2010; Völter and Solomatov, 2010; Erdweg et al., 2012; Mernik, 2013; Chodarev et al., 2014; Cazzola and Vacchi, 2016) but none of them take a formal approach, nor do they discuss translation of languages in any depth. For a longer discussion on the similarities and differences between our approach and other approaches, see chapter 7.

What sets our approach apart from other existing approaches is that the existing approaches we have reviewed are tied to specific tools, while the approach we present is in the form of mathematical constructs that could be used in a variety of tools. As far as we know, no other approach allows for judgements about subtypes in the way that our work does. As such, other approaches do not allow for judgements on relationships between composed languages to the degree that our approach allows. Due to constraints on the research, only some of the effects of these relationships were explored, specifically the effect that composition has on reuse of translation rules. As far as we can tell, doing the same with any of the existing tools would not be possible, as our analysis hinges on judgements on subtypes, which other tools do not explicitly give.

To test the power of our approach, we show how our composition technique can be used to describe itself. That is, how we can create a DSL for specifying composition and translation rules, and how that DSL then can be used to describe itself. We show that the approach works by repeatedly feeding its own description to itself to create a copy of itself and verifying that each new iteration works the same.

We also show that our composition technique can be used to create a DSL for patience games. We used this to show how our technique can reduce the amount of work needed to define a translation from a newly composed DSL to an existing DSL or GPL. We also compared the translation to GPL to translation to DSLs in a similar domain. We verified the patience DSL by showing it can be used to describe a wide range of games.

While this thesis only explores a small part of the potential uses of our framework, it does provide a solid ground for future research.

8.4 Shortcomings

In chapter 3 we introduced the patience domain and showed how it can be used to reason about various concerns that arise when developing DSLs. While the chapter does provide a good enough base for working with the patience domain, there are still several improvements that could be made. One of the shortcomings is the lack of any proper formal domain analysis, such as described in (Kang et al., 1990; Simos, 1995). The chapter does provide a definition of the domain using a Feature Diagram (Kang et al., 1990) based on the patience construction 'wizard' that is included in the patience game suite PySolFC (PySolFC, n.d.), but it is very constrained. The design of the patience GPL code-base presented in chapter 3 could also be improved, as it currently does not support specifying the layout of the piles and it does not quite support all types of patience games (such as games with multiple decks). Finally, it would be good to have an ontology of patience games, describing a wide range of different common patience games that could be used to verify that a patience DSL is able to

describe a wide range of patience games.

In chapter 4 we introduced our DSL composition framework for abstract syntax. It describes the operators in our composition framework, the rules for subtyping, as well as how we translate from one language to another and under which circumstances translation rules from one language can be reused in another. The chapter presents side-effects as part of the model of abstract languages we use. We did not go into how these side-effects work when composing and translating, though. One of the consequences of this is that proper type-checking of input and output languages in translators is not possible, as a lot of the type information is specified in the side-effects in the context of our framework.

In chapter 6 we showed how our framework can be used to create a self-describing translator DSL. As we do not discuss side-effects and domains in detail in chapter 4, we cannot discuss how to handle translation of these in chapter 6. This also means that our tool does not perform any type-checking of the translation rules. This short-coming is particularly problematic, as one of the main benefits of our approach is that it allows to type-check the translation rules and determine (with some caveats) whether the output of applying the translations will type-check. This only works with properly implemented side-effects, though. The tool also only works with translations, not compositions. That is, there is a DSL for specifying translations between already composed languages, but no DSL for specifying the composition itself. Work is underway to provide the composition DSL, but it is not yet ready to be published.

In chapter 5 we showed how our framework can be used in creating a DSL for the patience domain. One of the short-comings of chapter 5 is that we use our own intermediate DSL languages when discussing the possibility of using intermediate languages. While this makes it easier to showcase the potential for using intermediate languages, by allowing us to ignore some of the complexities found in existing DSLs, it does mean that some of the conclusions drawn are not necessarily applicable to using existing DSLs.

Another short-coming in that we do not much discuss how appropriate the DSL we con-

structed for the patience domain is for the purpose of describing patience games. What it means to be good at describing a certain domain (such as the patience domain) is a bit difficult to define, but a good start is that it should require less effort than some other common approach (such as using a GPL) and the percentage of valid patience games it can implement. Measuring effort is not completely straightforward and depends on a variety of variables, such as the approach we compare to, the implementation details of the domain, and what experience the person making the effort has. Measuring the percentage of valid patience games it can implement is not easy either, as what counts as a valid patience game is not well-defined. It can be rephrased as how many patience games out of a list of known patience games can be defined, which is the best we can do without a formal definition of what a patience game is.

Lastly, we will discuss some general short-comings. The current version of the composition techniques is constrained to only work well with local translations. Outside of this, it cannot guarantee that a set of translation rules produces sensible results (even with properly implemented side-effects). The approach has also so far only been tested on itself and the patience games domain.

There is no quantitative tests of our approach, this is in part because creating a good quality quantitative test takes a lot of resources, and they were not available. On top of that, our work is focusing on the abstract syntax, and most of the quantitative testing that is done in the DSL is done on concrete syntax. So we would have to set up a completely new test framework, which would have meant extra resources and no way to compare to previous work.

Finally, the composition technique works only on the abstract syntax, whereas it is usually the concrete syntax that one wants to construct, as without a concrete syntax, there is no way to actually write sentences in the language. In this work we have overcome this by using a lisp-like syntax whenever we needed to specify sentences using the abstract syntax.

8.5 Future work

For the future, the composition techniques should be extended with side-effect composition and domain composition. This would involve investigating the type of side-effects that are commonly found in languages, and provide composition and translation rules for those side-effects. The domain composition would involve rules for creating domain types in such a way that it becomes easy to translate a newly constructed domain type into an existing language. Both of these are dependent on the output language, so any work on this front will likely have to provide a way to translate differently depending on the output language.

The tool presented in chapter 6 needs to be completed with support for specifying compositions and for that to be integrated with the translations. This work is currently underway and will hopefully be publishable soon. To truly complete the tool, it also needs to implement side-effects and domains, as described in the previous paragraph.

How to integrate the composition of the abstract language with concrete languages is also an interesting question we have not yet answered. Ideally, when a new abstract language is constructed, a concrete language could be derived from existing concrete language definitions. However, as (Chodarev and Kollar, 2016) notes, concrete languages are not closed under composition, so it is not in general possible to completely automatically generate the grammars from the abstract language definition. Ideally, the creation of the concrete grammar would work similar to how our work handles translation rules, where some parts can be automatically derived from the abstract language composition, and some parts would have to be manually specified. Crucially, there needs to be a way of determining whether any particular composition requires new grammar rules to be specified or if they allow for existing ones to be reused.

There are also several other concerns that are generally considered to be part of a DSL construction environment, such as tools for debugging languages in the constructed DSL, translations into artefacts other than runnable code (such as generation of highlighting rules for the DSL), as well as analysis tools in general for the DSLs . Our system supports none

of this right now, but it would be interesting to see how this could be integrated with the composition techniques. That is, in the same way that we have investigated how translation behaves under composition, we could investigate how debugging/analysis behaves under composition.

List of References

Acher, M., Collet, P., Lahire, P. and France, R. B.: 2013, Familiar: A domain-specific language for large scale management of feature models, *Science of Computer Programming* **78**(6), 657–681.

Aho, A., Lam, M., Sethi, R. and Ullman, J.: 2006, Compilers: Principles, techniques and tools.

Ali, H.: 2020, Multi-language systems based on perspectives to promote modularity, reusability, and consistency, *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, Association for Computing Machinery, New York, NY, USA.
URL: *https://doi.org/10.1145/3417990.3419489*

Amaral, V., Helmer, S. and Moerkotte, G.: 2003, A visual query language for hep analysis, *2003 IEEE Nuclear Science Symposium. Conference Record (IEEE Cat. No.03CH37515)*, Vol. 2, pp. 829–833 Vol.2.

Bačíková, M.: 2014, Domain analysis of graphical user interfaces of software systems, *Information Sciences and Technologies* **6**(4), 17.

Bačíková, M., Poruban, J. and Lakatos, D.: 2013, Defining Domain Language of Graphical User Interfaces, *2nd Symposium on Languages, Applications and Technologies*, Dagstuhl Publishing, pp. 187–202.

Barišić, A., Amaral, V., Goulão, M. and Barroca, B.: 2011, Quality in use of domain-specific languages: A case study, *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '11, ACM, New York, NY, USA, pp. 65–72.
URL: *http://doi.acm.org/10.1145/2089155.2089170*

Barišić, A., Amaral, V., Goulao, M. and Barroca, B.: 2012, How to reach a usable dsl? moving toward a systematic evaluation, *Electronic Communications of the EASST* **50**.

Batory, D.: 2004, Feature-oriented programming and the ahead tool suite, *Proceedings. 26th International Conference on Software Engineering*, IEEE, pp. 702–703.

Batory, D. and Geraci, B. J.: 1997, Composition validation and subjectivity in genvoca generators, *IEEE Transactions on Software Engineering* **23**(2), 67–82.

Batory, D., Johnson, C., MacDonald, B. and von Heeder, D.: 2002, Achieving extensibility through product-lines and domain-specific languages: A case study, *ACM Trans. Softw. Eng. Methodol.* **11**(2), 191–214.
URL: *http://doi.acm.org/10.1145/505145.505147*

Batory, D., Lofaso, B. and Smaragdakis, Y.: 1998, Jts: Tools for implementing domain-specific languages, *Fifth International Conference on Software Reuse*, pp. 143–153.

Batory, D., Lopez-Herrejon, R. E. and Martin, J. P.: 2002, Generating product-lines of product-families, *Proceedings 17th IEEE International Conference on Automated Software Engineering*, pp. 81–92.

Bettini, L.: 2013, *Implementing Domain-Specific Languages with Xtext and Xtend*, Packt Publishing Ltd.

Bilitchenko, L., Liu, A., Cheung, S., Weeding, E., Xia, B., Leguia, M., Anderson, J. C. and Densmore, D.: 2011, Eugene a domain specific language for specifying and constraining

synthetic biological parts, devices, and systems, *PLOS ONE* **6**(4), 112.
URL: *http://dx.doi.org/10.1371/journal.pone.0018882*

Bjarnason, R., Fern, A. and Tadepalli, P.: 2009, Lower bounding klondike solitaire with monte-carlo planning., *ICAPS*.

Borodin, A., Kiselev, Y., Mirvoda, S. and Porshnev, S.: 2015, *On Design of Domain-Specific Query Language for the Metallurgical Industry*, Springer International Publishing, Cham, pp. 505–515.

Bravenboer, M. and Visser, E.: 2004, Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions, *ACM SIGPLAN Notices*, Vol. 39, ACM, pp. 365–383.

Bryant, B., Jézéquel, J.-M., L ammel, R., Mernik, M., Schindler, M., Steinmann, F., Tolvanen, J.-P., Vallecillo, A. and V olter, M.: 2015, *Globalized Domain Specific Language Engineering*, Springer International Publishing, Cham, pp. 43–69.

Burgy, L., Consel, C., Latry, F., Lawall, J., Palix, N. and Réveillere, L.: 2005, Telephony software engineering: A domain-specific approach, *Technical report*, Research Report RR-5548, INRIA, Bordeaux, France.

Cadogan, A.: 1874, *Illustrated Games of Patience*, Sampson Low, Marston, Low, and Searle.

Cazzola, W. and Vacchi, E.: 2016, Language components for modular dsls using traits, *Computer Languages, Systems & Structures* **45**, 16 – 34.
URL: *http://www.sciencedirect.com/science/article/pii/S1477842415300208*

Čeh, I., Črepinšek, M., Kosar, T. and Mernik, M.: 2011, Ontology driven development of domain-specific languages, *Computer Science and Information Systems* **8**(2), 317–342.

Čeh, I., Črepinšek, M., Kosar, T., Mernik, M., Henriques, P., Pereira, M. J. a., Cruz, D. and Oliveira, N.: 2011, Tool-supported building of dsls from owl ontologies, *INForum'11, Simpósio de Informática (CoRTA'11 track).*

Chen, L. and Babar, M. A.: 2011, A systematic review of evaluation of variability management approaches in software product lines, *Information and Software Technology* **53**(4), 344–362. Special section: Software Engineering track of the 24th Annual Symposium on Applied Computing Software Engineering track of the 24th Annual Symposium on Applied Computing.
URL: *http://www.sciencedirect.com/science/article/pii/S0950584910002223*

Cheng, B. H. C., Combemale, B., France, Robert B.and Jézéquel, J.-M. and Rumpe, B.: 2015, *On the Globalization of Domain-Specific Languages*, Springer International Publishing, Cham, pp. 1–6.

Chodarev, S. and Kollar, J.: 2016, Extensible host language for domain-specific languages, *Computing and Informatics* **35**(1), 84–110.

Chodarev, S., Lakatoš, D., Porubän, J. and Kollár, J.: 2014, Abstract syntax driven approach for language composition, *Open Computer Science* **4**(3), 107–117.

Cleenewerck, T.: 2003, Component-based dsl development, *in* F. Pfenning and Y. Smaragdakis (eds), *Generative Programming and Component Engineering*, Vol. 2830 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 245–264.
URL: *http://dx.doi.org/10.1007/978-3-540-39815-8_15*

Consel, C., Latry, F., Réveillere, L. and Cointe, P.: 2005, A generative programming approach to developing dsl compilers, *Generative Programming and Component Engineering*, Springer, pp. 29–46.

Consel, C. and Marlet, R.: 1998, Architecture software using: a methodology for language development, *Principles of Declarative Programming*, Springer, pp. 170–194.

Crane, M. L. and Dingel, J.: 2005, Uml vs. classical vs. rhapsody statecharts: Not all models are created equal, *MoDELS*, Vol. 3713, Springer, pp. 97–112.

Črepinšek, M., Kosar, T., Mernik, M., Cervelle, J., Forax, R. and Rousell, G.: 2010, On automata and language based grammar metrics, *Computer Science and Information Systems* **7**(2), 309–329.

Czarnecki, K.: 2005, Overview of generative software development, *Unconventional Programming Paradigms*, Springer, pp. 326–341.

Czarnecki, K. and Eisenecker, U. W.: 2000, Generative programming, *Edited by G. Goos, J. Hartmanis, and J. van Leeuwen* p. 15.

Czarnecky, K.: 1998, *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*, PhD thesis, PhD thesis, Technische Universit at Ilmenau, Department of Computer Science.

Degueule, T.: 2016, *Composition and Interoperability for External Domain-Specific Language Engineering*, PhD thesis, Universite de Rennes.

Degueule, T., Combemale, B., Blouin, A., Barais, O. and Jézéquel, J.-M.: 2015, Melange: A meta-language for modular and reusable development of dsls, *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2015, ACM, New York, NY, USA, pp. 25–36.
URL: *http://doi.acm.org/10.1145/2814251.2814252*

Delorie, D.: 2012, Ace of penguins.
URL: *http://www.delorie.com/store/ace/*

Dhouib, S., Kchir, S., Stinckwich, S., Ziadi, T. and Ziane, M.: 2012, *RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications*, Springer Berlin

Heidelberg, Berlin, Heidelberg, pp. 149–160.
URL: *http://dx.doi.org/10.1007/978-3-642-34327-8_16*

Diamond, S. and Boyd, S.: 2016, Cvxpy: A python-embedded modeling language for convex optimization, *Journal of Machine Learning Research* **17**(83), 1–5.

Diekmann, L. and Tratt, L.: 2013, Parsing composed grammars with language boxes, *Workshop on Scalable Language Specifications*.

Dijkstra, E. W.: 1975, Guarded commands, nondeterminacy and formal derivation of programs, *Commun. ACM* **18**(8), 453457.
URL: *https://doi.org/10.1145/360933.360975*

Dinkelaker, T., Eichberg, M. and Mezini, M.: 2010, An architecture for composing embedded domain-specific languages, *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, ACM, New York, NY, USA, pp. 49–60.
URL: *http://doi.acm.org/10.1145/1739230.1739237*

Dinkelaker, T., Eichberg, M. and Mezini, M.: 2013, Incremental concrete syntax for embedded languages with support for separate compilation, *Science of Computer Programming* **78**(6), 615–632. Special section: The Programming Languages track at the 26th {ACM} Symposium on Applied Computing (SAC 2011) & Special section on Agent-oriented Design Methods and Programming Techniques for Distributed Computing in Dynamic and Complex Environments.
URL: *http://www.sciencedirect.com/science/article/pii/S0167642312002134*

do Nascimento, L. M., Viana, D. L., Silveira Neto, P. A. M., Martins, D. A. O., Garcia1, V. C. and Meira, S. R. L.: 2012, A systematic mapping study on domain-specific languages, *The Seventh International Conference on Software Engineering Advances (ICSEA 2012)*, pp. 179–187.

Eden, A. H., Hirshfeld, Y. and Kazman, R.: 2006, Abstraction classes in software design, *IEE Proceedings-Software* **153**(4), 163–182.

Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W. and Hanus, M.: 2012, Xbase: Implementing domain-specific languages for java, *SIGPLAN Not.* **48**(3), 112–121.
URL: *http://doi.acm.org/10.1145/2480361.2371419*

Eißfeldt, H. and Bischoff, M.: n.d., Xpat.

Erdweg, S., Giarrusso, P. G. and Rendel, T.: 2012, Language composition untangled, *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, LDTA '12, ACM, ACM, New York, NY, USA, pp. 7:1–7:8.
URL: *http://doi.acm.org/10.1145/2427048.2427055*

Erdweg, S., Rendel, T., Kästner, C. and Ostermann, K.: 2011, Sugarj: Library-based syntactic language extensibility, *SIGPLAN Not.* **46**(10), 391–406.
URL: *http://doi.acm.org/10.1145/2076021.2048099*

Erdweg, S., van der Storm, T., Völter, M., Tratt, L., Bosman, R., Cook, W. R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, a., Molina, P. J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V., Visser, E., Vlist, K. v. d., Wachsmuth, G. and Woning, J. v. d.: 2015, Evaluating and comparing language workbenches, *Computer Languages, Systems & Structures* **44**, 24 – 47. Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014).
URL: *http://www.sciencedirect.com/science/article/pii/S1477842415000573*

Estublier, J., Vega, G. and Ionita, A. D.: 2005, Composing domain-specific languages for wide-scope software engineering applications, *in* L. Briand and C. Williams (eds), *Model Driven Engineering Languages and Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 69–83.

Eysholdt, M. and Behrens, H.: 2010, Xtext: Implement your language faster than the quick and dirty way, *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, ACM, New York, NY, USA, pp. 307–309.
URL: *http://doi.acm.org/10.1145/1869542.1869625*

Fonseca, J., Pereira, M. and Henriques, P.: 2014, Converting Ontologies into DSLs, *3rd Symposium on Languages Technologies and Applications*, Dagstuhl Publishing, pp. 0–7.

Font, J. M., Mahlmann, T., Manrique, D. and Togelius, J.: 2013, *A card game description language*, Vol. 7835 of *Lecture Notes in Computer Science*, Springer.
URL: *http://dx.doi.org/10.1007/978-3-642-37192-9_26*

Fowler, M.: 2010, *Domain-Specific Languages*, Addison Wesley.

Gabriel, Pedro and Goulão, Miguel and Amaral, Vasco: 2011, Do software languages engineers evaluate their languages?, *CoRR* **abs/1109.6794**.
URL: *http://arxiv.org/abs/1109.6794*

Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: 1994, *Design patterns: elements of reusable object-oriented software*, Pearson Education.

Ghosh, D.: 2010, *DSLs in Action*, 1st edn, Manning Publications Co., Greenwich, CT, USA.

Gibson, W. B.: 1993, *Hoyle's Modern Encyclopedia of Card Games*, Selecta Book.

Gnome: n.d., Aisleriot.
URL: *https://wiki.gnome.org/action/show/Apps/Aisleriot*

Greenfield, J. and Short, K.: 2004, *Software Factories*, Wiley Publishing, Inc.

Grohe, M. and Schweikardt, N.: 2003, Comparing the succinctness of monadic query languages over finite trees, *International Workshop on Computer Science Logic*, Springer, pp. 226–240.

Haber, A., Look, M., Perez, A. N., Nazari, P. M. S., Rumpe, B., Völkel, S. and Wortmann, A.: 2015, Integration of heterogeneous modeling languages via extensible and composable language components, *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pp. 19–31.

Harsu, M.: 2002, *A survey on domain engineering*, Citeseer.

Hermans, F., Pinzger, M. and van Deursen, A.: 2009, Domain-specific languages in practice: A user study on the success factors, *Model driven engineering languages and systems*, Springer, pp. 423–437.

Hudak, P.: 1997a, Domain-specific languages, *Handbook of Programming Languages* **3**, 39–60.

Hudak, P.: 1997b, Domain-specific languages, *Handbook of Programming Languages* **3**, 39–60.

Johnson, S. C.: 1975, *Yacc: Yet another compiler-compiler*, Vol. 32, Bell Laboratories Murray Hill, NJ.

Jones, C.: 1996, Programming languages table, release 8.2, *Software Productivity Research, Burlington, MA* .

Kahraman, G. and Bilgen, S.: 2015, A framework for qualitative assessment of domain-specific languages, *Software & Systems Modeling* **14**(4), 1505–1526.
URL: *http://dx.doi.org/10.1007/s10270-013-0387-8*

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E. and Peterson, A. S.: 1990, Feature-oriented domain analysis (foda) feasibility study, *Technical report*, DTIC Document.

Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M. and Völkel, S.: 2014, Design guidelines for domain specific languages, *arXiv preprint arXiv:1409.2378*

abs/1409.2378.
URL: *http://arxiv.org/abs/1409.2378*

Kihlman, L.: 2015, Producing domain-specific languages from strategy patterns, *Computer Science and Electronic Engineering Conference (CEEC), 2015 7th*, pp. 9–12.

Kihlman, L.: 2017, A test model for domain-specific language developement, *Computer Science and Electronic Engineering Conference (CEEC), 2017 9th*.

Kleppe, A.: 2008, *Software Language Engineering*, Pearson Education, Inc.

Kollar, J. and Chodarev, S.: 2010, Extensible approach to dsl development, *Journal of Information, Control and Management Systems* **8**(3).

Kosar, T., Bohra, S. and Mernik, M.: 2016, Domain-specific languages: A systematic mapping study, *Information and Software Technology* **71**, 77 – 91.
URL: *http://www.sciencedirect.com/science/article/pii/S0950584915001858*

Kosar, T., Martı, P. E., Barrientos, P. A., Mernik, M. et al.: 2008, A preliminary study on various implementation approaches of domain-specific language, *Information and Software Technology* **50**(5), 390–405.
URL: *http://www.sciencedirect.com/science/article/pii/S0950584907000419*

Kosar, T., Mernik, M. and Carver, J. C.: 2012, Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments, *Empirical software engineering* **17**(3), 276–304.

Kosar, T., Mernik, M., Crepinsek, M., Henriques, P., da Cruz, D., Pereira, M. and Oliveira, N.: 2009, Influence of domain-specific notation to program understanding, *Computer Science and Information Technology, 2009. IMCSIT '09. International Multiconference on*, pp. 675–682.

Kosar, T., Oliveira, N., Mernik, M., Pereira, V. J. a. M., Črepinšek, M., Da, C. D. and Henriques, R. P.: 2010, Comparing general-purpose and domain-specific languages: An empirical study, *Computer Science and Information Systems* **7**(2), 247–264.

Koutsofios, E., North, S. et al.: 1991, Drawing graphs with dot, *Technical report*, Technical Report 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill, NJ.

Kühn, T., Cazzola, W. and Olivares, D. M.: 2015, Choosy and picky: Configuration of language product lines, *Proceedings of the 19th International Conference on Software Product Line*, SPLC '15, ACM, New York, NY, USA, pp. 71–80.
URL: *http://doi.acm.org/10.1145/2791060.2791092*

Lacy, L. W.: 2005, *OWL: Representing information using the web ontology language*, Trafford Publishing.

Lakatos, D. and Porubän, J.: 2013, Patterns for composition of domain-specific languages, *Journal of Computer Science and Control Systems* **6**(1), 62.

Longpré, L. and McKenzie, P.: 2009, The complexity of solitaire, *Theoretical Computer Science* **410**(50), 5252–5260. Mathematical Foundations of Computer Science (MFCS 2007).
URL: *http://www.sciencedirect.com/science/article/pii/S0304397509006100*

Lorenzen, F. and Erdweg, S.: 2013, Modular and automated type-soundness verification for language extensions, *SIGPLAN Not.* **48**(9), 331–342.
URL: *http://doi.acm.org/10.1145/2544174.2500596*

Lorenzen, F. and Erdweg, S.: 2016, Sound type-dependent syntactic language extension, *SIGPLAN Not.* **51**(1), 204–216.
URL: *http://doi.acm.org/10.1145/2914770.2837644*

Love, N., Hinrichs, T., Haley, D., Schkufza, E. and Genesereth, M.: 2008, General game playing: Game description language specification.

Luoma, J., Kelly, S. and Tolvanen, J.-P.: 2004, Defining domain-specific modeling languages: Collected experiences, *4 th Workshop on Domain-Specific Modeling*.

Merilinna, J. and Pärssinen, J.: 2007, Comparison between different abstraction level programming: Experiment definition and initial results, *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM07), Montréal, Candada*.

Mernik, M.: 2012, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments: Recent Developments*, IGI Global.

Mernik, M.: 2013, An object-oriented approach to language compositions for software language engineering, *Journal of Systems and Software* **86**(9), 2451–2464.
URL: *http://www.sciencedirect.com/science/article/pii/S0164121213001271*

Mernik, M., Heering, J. and Sloane, A. M.: 2005, When and How to Develop Domain-Specific Languages., *ACM Computing Surveys* **37**(4), 316–344.

Mernik, M., Lenič, M., Avdičauševič, E. and Žumer, V.: 2002, Lisa: An interactive environment for programming language development, *International Conference on Compiler Construction*, Springer, pp. 1–4.

Morehead, A. and Mott-Smith, G.: 2001, *The Complete book of Solitaire and Patience*, foulsham.

Motik, B., Patel-Schneider, P. F., Parsia, B., Bock, C., Fokoue, A., Haase, P., Hoekstra, R., Horrocks, I., Ruttenberg, A., Sattler, U. et al.: 2009, Owl 2 web ontology language: Structural specification and functional-style syntax, *W3C recommendation* **27**(65), 159.

Nystrom, N., Clarkson, M. R. and Myers, A. C.: 2003, Polyglot: An extensible compiler framework for java, *International Conference on Compiler Construction*, Springer, pp. 138–152.

Oliveira, N., Pereira, M. J. a., Henriques, P. and Cruz, D.: 2009, Domain specific languages: A theoretical survey, *Proceedings of the 3rd Compilers, Programming Languages, Related Technologies and Applications (CoRTA'2009)*, Faculdade de Ciências da Universidade de Lisboa, pp. 35–46.

Papegaaij, E.: 2007, *The tree processing language: Defining the structure and behaviour of a tree*, Master's thesis, University of Twente.
URL: *http://essay.utwente.nl/705/*

Parnas, D.: 1976, On the design and development of program families, *Software Engineering, IEEE Transactions on* **SE-2**(1), 1–9.

Parr, T.: 2009, *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*, 1st edn, Pragmatic Bookshelf.

Parr, T.: 2010, *Language implementation patterns: create your own domain-specific and general programming languages*, Pragmatic Bookshelf.

Parr, T. J. and Quong, R. W.: 1995, Antlr: A predicated-ll(k) parser generator, *Software: Practice and Experience* **25**(7), 789–810.
URL: *http://dx.doi.org/10.1002/spe.4380250705*

Pereira, M. J. a. V., Mernik, M., Henriques, P. R. et al.: 2008, Program comprehension for domain-specific languages, *Computer Science and Information Systems* **5**(2), 1–17.

Porubän, J., Forgáč, M. and Sabo, M.: 2009, Annotation based parser generator, *2009 International Multiconference on Computer Science and Information Technology*, IEEE, pp. 707–714.

Porubän, J., Sabo, M., Kollár, J. and Mernik, M.: 2010, Abstract syntax driven language development: Defining language semantics through aspects, *Proceedings of the International Workshop on Formalization of Modeling Languages*, pp. 1–5.

Power, J. F. and Malloy, B. A.: 2004, A metrics suite for grammar-based software, *Journal of Software Maintenance and Evolution: Research and Practice* **16**(6), 405–426. This is the postprint version of the published article, which is available at DOI: 10.1002/smr.v16:6.
URL: *http://eprints.maynoothuniversity.ie/6419/*

Prieto-Díaz, Rubén: 1990, Domain analysis: An introduction, *SIGSOFT Softw. Eng. Notes* **15**(2), 47–54.
URL: *http://doi.acm.org/10.1145/382296.382703*

PySolFC: n.d., Pysolfc.
URL: *http://pysolfc.sourceforge.net/*

Rossel Cid, P. O.: 2013, *Software product line model for the meshing tool domain*, PhD thesis, Universidad de Chile.

Schmid, K.: 2000, Scoping software product lines, *Software Product Lines*, Springer, pp. 513–532.

Schmitt, C., Kuckuk, S., Kostler, H., Hannig, F. and Teich, J.: 2014, An evaluation of domain-specific language technologies for code generation, *Computational Science and Its Applications (ICCSA), 2014 14th International Conference on*, pp. 18–26.

Simos, M. A.: 1995, Organization domain modeling (odm): Formalizing the core domain modeling life cycle, *Proceedings of the 1995 Symposium on Software Reusability*, SSR '95, ACM, New York, NY, USA, pp. 196–205.
URL: *http://doi.acm.org/10.1145/211782.211845*

Simos, M., Creps, R., Klingler, C. and Lavine, L.: 1995, Software technology for adaptable reliable systems (stars). organization domain modeling (odm) guidebook, version 1.0., *Technical report*, DTIC Document.

Sonnenberg, C., Huemer, C., Hofreiter, B., Mayrhofer, D. and Braccini, A.: 2011, The rea-dsl: A domain specific modeling language for business models, *Proceedings of the 23rd*

International Conference on Advanced Information Systems Engineering, CAiSE'11, Springer-Verlag, Berlin, Heidelberg, pp. 252–266.
URL: *http://dl.acm.org/citation.cfm?id=2026716.2026743*

Spinellis, D.: 2001, Notable design patterns for domain-specific languages, *Journal of Systems and Software* **56**(1), 91–99.
URL: *http://www.sciencedirect.com/science/article/pii/S0164121200000893*

Sprinkle, J., Mernik, M., Tolvanen, J. and Spinellis, D.: 2009, Guest editors' introduction: What kinds of nails need a domain-specific hammer?, *Software, IEEE* **26**(4), 15–18.

Taylor, R. N., Tracz, W. and Coglianese, L.: 1995, Software development using domain-specific software architectures: Cdrl a011—a curriculum module in the sei style, *SIGSOFT Softw. Eng. Notes* **20**(5), 27–38.
URL: *http://doi.acm.org/10.1145/217030.217034*

Team, K.: n.d., Kpat.
URL: *https://games.kde.org/game.php?game=kpat*

Thibault, S.: 1998, *Domain-specific languages: Conception, implementation and application*, PhD thesis, PhD thesis, IRISA/University of Rennes 1.

Thibault, S. A., Marlet, R. and Consel, C.: 1999, Domain-specific languages: from design to implementation application to video device drivers generation, *Software Engineering, IEEE Transactions on* **25**(3), 363–377.

Thielscher, M.: 2010, A general game description language for incomplete information games., *AAAI*, Vol. 10, Citeseer, pp. 994–999.

Tracz, W.: 1994, Domain-specific software architecture (dssa) frequently asked questions (faq), *ACM SIGSOFT Software Engineering Notes* **19**(2), 52–56.

Tucker, A. B. and Noonan, R. E.: 2007, *Programming Languages: Principles and Paradigms*, 2nd edn, McGraw-Hill.

Turner, R.: 2017, The Philosophy of Computer Science.
URL: *https://stanford.library.sydney.edu.au/entries/computer-science/*

Vacchi, E., Cazzola, W., Pillay, S. and Combemale, B.: 2013, Variability support in domain-specific language development, *in* M. Erwig, R. F. Paige and E. van Wyk (eds), *Software Language Engineering*, Vol. 8225 of *Lecture Notes in Computer Science*, Springer International Publishing, pp. 76–95.
URL: *http://dx.doi.org/10.1007/978-3-319-02654-1_5*

van Deursen, A. and Klint, P.: 1998, Little Languages: Little Maintenance?, *Journal of Software Maintenance* **10**, 75–92.

van Deursen, A. and Klint, P.: 2002, Domain-Specific Language Design Requires Feature Descriptions, *Computing and Information Technology* **10**(1), 1–17.

van Deursen, A., Klint, P. and Visser, J.: 2000, Domain-specific languages: An annotated bibliography., *Sigplan Notices* **35**(6), 26–36.

Visser, E.: 2008, WebDSL: A Case Study in Domain-Specific Language Engineering., *Lecture Notes in Computer Science*, Vol. 5235, Springer, pp. 291–373.

Völter, M.: 2008, A family of languages for architecture description, *8th OOPSLA Workshop on Domain-Specific Modeling (DSM)*, pp. 86–93.

Völter, M.: 2013a, *DSL Engineering*, CreateSpace Independent Publishing Platform.
URL: *http://dslbook.org*

Völter, M.: 2013b, *Language and IDE Modularization and Composition with MPS*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 383–430.
URL: *http://dx.doi.org/10.1007/978-3-642-35992-7_11*

Völter, M. and Solomatov, K.: 2010, Language modularization and composition with projectional language workbenches illustrated with mps, *Software Language Engineering, SLE* **16**.

Völter, M. and Visser, E.: 2011, Product line engineering using domain-specific languages, *Software Product Line Conference (SPLC), 2011 15th International*, pp. 70–79.

W3C: 2012, OWL 2 Web Ontology Language Document Overview (Second Edition).
URL: *http://www.w3.org/TR/owl2-overview/*

Walter, T. and Ebert, J.: 2009, Combining dsls and ontologies using metamodel integration, *IFIP Working Conference on Domain-Specific Languages*, Springer, pp. 148–169.

Walter, T., Parreiras, F. S. and Staab, S.: 2014, An ontology-based framework for domain-specific modeling, *Software & Systems Modeling* **13**(1), 83–108.

Walter, T., Silva Parreiras, F. and Staab, S.: 2009, OntoDSL: An Ontology-Based Framework for Domain-Specific Languages, *in* A. Schürr and B. Selic (eds), *Model Driven Engineering Languages and Systems: 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings*, Vol. 5795 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 408–422.
URL: *http://dx.doi.org/10.1007/978-3-642-04425-0_32*

Walton, L. A.: 1998, Generating the parser-pretty-printer isomorphism.

Weiss, D. M. and Lai, C. T. R.: 1999, *Software Product-Line Engineering: A Family-Based Software Development Process*, Addison Wesley.

White, J., Hill, J. H., Gray, J., Tambe, S., Gokhale, A. S. and Schmidt, D. C.: 2009, Improving domain-specific language reuse with software product line techniques, *IEEE Software* **26**(4), 47–53.

Wile, D. S.: 2001, Supporting the DSL Spectrum, *CIT. Journal of computing and information technology* **9**(4), 263–287.

Wu, H., Gray, J. and Mernik, M.: 2009, Unit testing for domain-specific languages, *in* W. Taha (ed.), *Domain-Specific Languages*, Vol. 5658 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 125–147.
URL: *http://dx.doi.org/10.1007/978-3-642-03034-5_7*

Yan, X., Diaconis, P., Rusmevichientong, P. and Roy, B. V.: 2005, Solitaire: Man versus machine, *Advances in Neural Information Processing Systems*, pp. 1553–1560.

Zdun, U.: 2010, A dsl toolkit for deferring architectural decisions in dsl-based software design, *Information and Software Technology* **52**(7), 733–748.

Zdun, U. and Strembeck, M.: 2009, Reusable architectural decisions for dsl design, *14th European Conference on Pattern Languages of Programs (EuroPLoP)*.

Zeng, J., Mitchell, C. and Edwards, S. A.: 2006, A domain-specific language for generating dataflow analyzers, *Electronic Notes in Theoretical Computer Science* **164**(2), 103–119.

Zschaler, S., Kolovos, D. S., Drivalos, N., Paige, R. F. and Rashid, A.: 2010, *Domain-Specific Metamodelling Languages for Software Language Engineering*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 334–353.