GOOL: A Generic OO Language

Jason Costabile
Department of Computing and Software
McMaster University

April 26, 2012

Abstract

Most object-oriented languages seem to have a strong common core, but also seem to be "verbose" in practice, at least with respect to certain classes of programs. But this observation is informal and anecdotal. The aim of this project is to see if some actual empirical evidence could be provided.

The method employed was to reverse engineer an executable language from a collection of OO languages, and demonstrate that this new language was expressive enough to succinctly and accurately express a number of (standard) OO programs. The focus here is explicitly bottom-up, to "discover" patterns. However, since it was humans doing the discovery, the patterns found may have been coloured by individual experiences.

The GOOL tool was the end result of this effort. It consists of an internal/embedded Domain-Specific Language (DSL) to serve as the new OO language, and a code generator which renders DSL programs into a representative sample of OO languages.

The existence of such a language in an accessible and readable form, together with its deterministic transformation into real OO languages, is conjectured to be sufficient evidence for this "core".

Contents

Chapter 1

Executive Summary

The goal of this project is to demonstrate that programming languages of the object-oriented (OO) paradigm share some central concepts. That is, there exists a common "core" among OO languages, made up of a set of fundamental features and patterns.

We have shown this by designing a new programming language that explicitly models this core. The components and features of the language are those which we have found to be shared among traditional OO languages. It has sufficient scope and functionality to accommodate full implementations of simple programs.

To prove that the language really is representative of mainstream OO programming languages, we created a software tool in Haskell, called GOOL. Development began with modification of an existing tool (used to generate code for story-management modules of video games), since many of its components were suitable for reuse and extension.

GOOL implements our language using Haskell data structures, and can perform a "translation" on code from this language to any of six traditional OO languages. The output languages are meant to be a representative sample. The translation process is very straightforward. Each output language has a "translation dictionary" associated with it, and every component of our language can be looked up in this dictionary.

We posit that, since compilable code in many different programming

languages can be "trivially" generated from a single generic representation, these languages must therefore have a core set of common features, and this core is accurately represented by our language.

Chapter 2

Introduction

Object-oriented languages are extremely popular [LLC11]. Even languages that are not strictly object-oriented still support fundamental OO features and allow for OO programming. Throughout all the various implementations, there is a certain general set of core ideas that can be abstracted away from any one specific language. This is made up of a certain set of patterns that form the "heart" of the OO style in general, at least as implemented in mainstream languages.

This project seeks to both demonstrate (empirically) the existence of this core, and to test its boundaries. It strives to find just how much of the set of core ideas can be abstracted out by reverse engineering mainstream languages.

An appealing way to accomplish this is to create a separate language which is somehow representative of the original OO languages, and anchor it to mainstream OO programming through some tangible link to compilable OO code. The link must be substantial and direct enough to prove that the language is, indeed, representative of identified OO patterns. If these criteria are met, the resulting system would suffice to demonstrate the existence of the aforementioned core, as comprised by the patterns represented in the language.

This serves to summarize the ultimate goal of this project. A more specific account of what it aims to accomplish can be found in Chapter 3.

It seemed logical to develop a software tool that could realize this concept. This would be an obvious way to provide practical, concrete evidence. Based on the requirements, a tool could be designed to explicitly demonstrate the aforementioned core.

We proposed a specific methodology for this, describing the tool to be designed and developed. After some deliberation and refinement, we decided to create a new object-oriented programming language, in the form of an embedded domain-specific language (see Chapter 5 for an explanation of domain-specific languages). This would allow for the most direct and straightforward representation of the core. The "link" (acting as proof that this language really is representative of OO programming) would be the tool's ability to translate these programs into traditional OO code, using several mainstream programming languages. The generated code must be a direct translation from the input code, and it must follow the same basic patterns in all output languages. Again, this seemed to be the most logical and straightforward way to prove a connection. The tool was named GOOL: a Generic Object-Oriented Language.

In order to jumpstart development, a previously completed tool was reused as a starting point. This tool, called SAGA, was created for a very different purpose (generation of story manager modules for video game development), but included several components, as well as the basic framework, that would be needed for GOOL. One of its components was a system for code generation to various languages. It also defined an intermediate representation language (known as AbstractCode), which could be developed and expanded into the "core" language that GOOL would use. Thus, it was an obvious choice. Section 8.1 provides a more complete description of SAGA.

The first major phase of GOOL's development consisted of generalizing SAGA's story-specific facilities and transforming it into a tool that could produce general-purpose code. Once this was accomplished, development efforts were focused more directly on the requirements of this project.

Some time was given to making various general improvements. These improvements were largely technical and organizational, primarily benefitting the understandability and maintainability of the code, with regards to

the tool's new purpose.

Another phase of development focused on preparation of the AbstractCode structure for a DSL. AbstractCode is the root node of a large algebraic datatype which represents "code" as an abstract syntax tree. Since it was previously only an intermediate representation, intended to be maintained and used directly only by the developers of SAGA, a fair amount of effort needed to be put into making it suitable for use by a wider audience. The main goal here was to make the DSL code look and feel like a general purpose OO language. As an internal DSL, the code still needed to suit the rules of the host language (Haskell in our case), but it was found that its resemblance to traditional code could be vastly increased through various methods. Chief among these was the implementation of a large number of "shortcut" functions. These generally represent a sequence of AbstractCode data constructors in a single keyword, allowing the programmer to greatly reduce the verbosity of an AbstractCode program. For example, a statement to print a literal string with a line break:

```
PrintState True string . litString
```

can now be abbreviated to simply:

```
printStrLn
```

Additionally, a number of custom unary and binary operators were added to automate the implementation of common expressions and statements. These are similarly used to eliminate the need for typing out clumsy or ugly sequences of datatype constructors in order to represent a simple idea. For instance, a statement to increment the value of a numeric variable:

```
AssignState $ PlusEquals (Var "x") (Var "y")
```

can also be written as:

```
"x" &+= Var "y"
```

Symbolic operators were implemented for expressions that typically use such in general purpose languages, like equality comparisons, or arithmetic operations, as shown above. These, again, served to further the goal of making AbstractCode seem like a conventional language.

In this way, much of the structural overhead of the language could be abstracted out. This task involved the identification of many common or significant patterns in the code (such as the statements shown above). This proved to be an ongoing process, continuing throughout development as more constructs were added to the language and complexity increased.

As an introductory example, the following is a piece of code written entirely with AbstractCode primitives. It implements a simple class that represents a geometric circle. Analogous code in traditional languages will be presented shortly afterwards, for comparison and reference.

```
circleClass :: Class
circleClass =
    let modName = circleClassName
        radius = "radius"
        diam = "diameter"
        modVars = [
            StateVar radius Private (Base Float) 0]
    in Class modName Nothing Public modVars [
        Method modName Public (Construct modName)
            [StateParam "rad" $ Base Float]
            [ Block [
                AssignState $ Assign
                  (ObjVar Self (Var radius)) (Var "rad")
            ], Block [
                PrintState False (Base String) $
                  Lit $ LitStr "Circle created with radius ",
                PrintState True (Base Float) $
                  ObjVar Self (Var radius)
            ] ],
        Method "getDiameter" Public (MState $ Base Float) []
            [ Block [
                DeclState $ VarDecDef diam (Base Float)
                  (Expr $ BinaryExpr
                    (Var radius) Multiply (Lit $ LitFloat 2.0)),
                RetState $ Ret $ Var diam
            ] ],
        Method "getArea" Public (MState $ Base Float) [] $ oneLiner $
            RetState $ Ret $
              Expr $ BinaryExpr (Lit $ LitFloat 3.14159)
                                Multiply
                                (Expr $ BinaryExpr
                                  (Var radius) Power (Lit $ LitInt 2))
    ]
```

The equivalent code written in GOOL (i.e. employing GOOL's shortcut functions and operators):

```
circleClass :: Class
circleClass =
    let modName = circleClassName
        radius = "radius"
        diam = "diameter"
        modVars = [
            privVar neverDel float radius]
    in pubClass modName noParent modVars [
        pubFunc (Construct modName) modName
            [param "rad" float]
            [ Block [
                Self$->(Var radius) &=. "rad"
            ], Block [
                printStr "Circle created with radius ",
                printLn float $ Self$->(Var radius)
            ] ],
        pubFunc (MState float) "getDiameter" []
            [ Block [
                varDecDef diam float (Var radius #* litFloat 2.0),
                return $ Var diam
            ] ],
        pubFunc (MState float) "getArea" [] $ oneLiner $
                return $ litFloat 3.14159 #* (Var radius #^ litInt 2)
    ]
```

Now, the output code that the GOOL program generates from (either of) the previous code samples. The C++ header:

```
class Circle {
        public:
            Circle(float rad);
            float getDiameter();
            float getArea();
            ~Circle();

        private:
            float radius;
};
```

And the C++ source file:

```
Circle::Circle(float rad) {
    radius = rad;
```

```
    std::cout << "Circle created with radius ";
    std::cout << radius << std::endl;
}

float Circle::getDiameter() {
    float diameter = radius * 2.0;
    return diameter;
}

float Circle::getArea() {
    return 3.14159 * (pow(radius, 2));
}

Circle::~Circle() {
}
```

Finally, the equivalent output code in Python:

```
class Circle:
    def __init__(self, rad):
        self.radius = rad

        sys.stdout.write(str("Circle created with radius "))
        print(self.radius)

    def getDiameter(self):
        diameter = radius * 2.0
        return diameter

    def getArea(self):
        return 3.14159 * (radius ** 2)
```

The GOOL syntax seen here is explained in Section 7.3, and more example programs can be found in Section 8.5.

Perhaps the majority of development effort was spent on adding features to the new DSL. Since SAGA had a much more specific purpose, there were a number of common language features that did not yet exist in AbstractCode, as they were never needed for the story engine implementation. The need for several of these, such as simple arithmetic, or a statement to print to the console, was immediately apparent, in order for the DSL to be relatable to mainstream languages. Subsequently, an effective, but simple, methodology for choosing new features to add was designed: examples of small OO programs were chosen and implemented in AbstractCode. In this way, it was

natural to discover features used in the examples that would be necessary or helpful to complete their DSL implementations, but that were not yet supported by AbstractCode. Additionally, these examples would support the existence of the link between the AbstractCode core and traditional OO languages.

Other, larger-scale, additions were made, in the form of supplementary output languages. The modular design of SAGA and GOOL made it relatively simple to attach additional language generation modules, referred to as renderers. Each renderer needed to know how to generate output code for each AbstractCode primitive (and often also for special combinations of primitives) in its particular language, except where this output overlapped with that of an existing renderer (which does come up frequently, for some languages more than others). Therefore, each new renderer increased the effort required to add a new AbstractCode feature, and vice versa. As a result, doing either was rarely a trivial task.

GOOL's development process, added language features, and example AbstractCode implementations are all discussed in a technical and detailed capacity in Chapter 8.

It should be noted that there have been various attempts to create a "semantic core" OO language in the past (for example, Featherweight Java [IPW01]). However, this project aims to identify a more syntactic core, up to obvious isomorphism of syntax (while maintaining semantic equivalence).

This document will give a detailed explanation of the software tool that resulted from this project, the domain-specific language that it defines, and the story of how it was designed and developed. The chosen goals and requirements of the project are listed in Chapter 3. The methodology used to meet these requirements is discussed in Chapter 4. Subsequently, some important background information regarding Domain Specific Languages (DSLs) and the structure of OO languages is given in Chapter 5 and Chapter 6, respectively. Finally, the actual development of the software can be discussed. Section 8.1 explains where development was started and the state of the original tool that was gradually overhauled to become GOOL. Chapter 7 describes the ultimate state of the GOOL program at the culmination

of this project. Once this is understood, Chapter 8 falls back to fill in the gaps in detail, giving a chronological and much more thorough account of GOOL's development.

Chapter 3

Goals

The central goal of this project is to pragmatically demonstrate the existence of a "common core" among OO languages by explicitly exhibiting it. By core, we neither mean the intersection of the semantics of all languages (this would likely be too small), nor the union (too large and unwieldly). We mean a set of abstract features which are either present or trivially encodable in a sufficiently large set of (mainstream) object-oriented languages.

Our task, then, is to discover this set of abstract features. The most obvious place to start would be to take a set of textbooks on programming languages which cover a wide range of languages (such as [TN02, Seb01, FWH01, Sco09, Cla00]) to find such a set. But we were curious to see if that set would be the same as the set that we would find if we tried to build abstract features by recognizing common patterns in the languages themselves.

Of course, there is a certain minimum set of features of an OO language which must be present, and this is discussed in Chapter 6.

To show that the core really is present in multiple mainstream OO languages, we decided that the most convincing "proof" would consist of translating what we identify as the core into compilable, runnable code in these same OO languages. Furthermore, the translation should be "trivial", in the sense that it can be performed directly and unambiguously. In the next chapter, we expand on what we mean by this.

It is very important to us that we find a core which works in practice. In other words, our principal measure for whether we have achieved our goals will be if we can take a set of fairly standard OO programs, write them in our core language, and automatically translate them to reasonably idiomatic programs in a set of mainstream OO languages.

Chapter 4

Methodology

In order to accomplish the project's goals, a specific method was utilized: we created a language that (we hope) represents the OO core, and implemented it as an embedded DSL.

4.1 General Methodology

The primary goal of the project is to find a representation for the "common core", consisting of patterns found among OO languages. The first — and perhaps most important — step, would be, then, to decide on what these patterns are, and what actually constitutes a "pattern". The process that we employed for this is described in Section 4.2.

With the set of essential patterns in hand, we could set about creating a representation. The most explicit and direct method would be to assemble this representation in the form of a new, generic OO language. Furthermore, the most direct method to prove that our core permeates mainstream languages would be to show that a straightforward translation exists. The constraints and criteria selected for the language are listed in Section 4.4.

To facilitate this process, a software program could be created that would accept programs in the generic language, and "translate" them to traditional languages. Requirements for this software are listed in Section 4.5, and a description of the translation process can be found in Section 4.3.

4.2 Discovering Patterns

Beyond simply implementing the software, the language that makes up the core representation had to be designed. In order to do this, the core needed to be "discovered". This was accomplished by "reverse engineering" mainstream OO languages to find patterns at the heart of OO programming.

A "pattern", in this context, refers to any language feature that can be found in common among multiple languages. These patterns were discovered with simple methods.

The first method was suitable for the most obvious and numerous sort of pattern to be represented in the DSL — the simple, fundamental features (for example, all OO languages make use of variables, conditionals, and loops). Knowledge of programming was applied to brainstorm a list of features that would be necessary for any basic program functionality. Naturally, these would need to be part of the core, as per the project goal to have a functional and practical representation.

In most languages, these sort of patterns would use a nearly identical structure. An "if" conditional, for instance, always tests a boolean expression and uses the result to determine whether a code block should be executed. Also, there is a provision for more possible branches, in the form of "else" and "else if" blocks (some languages may include a primitive for the "else if" parts, like Python's "elif", whereas some languages would have to form this structure by simply chaining together "if-else" statements. However, the functionality, and the underlying pattern, is identical either way, and thus this form of the "if" conditional is a valid pattern for our core).

Other patterns may bear less of a syntactic resemblance, but are nevertheless valid. An example of this is collection containers. The most common container to use for collections in some languages is an array. In some it is a list. In addition, there are frequently other options that serve similar purposes (e.g. Vector; ArrayList). However, in the context of this project, they are extensions of the same pattern. In GOOL's DSL, the differences are abstracted out and a single, representative "List" primitive is used instead.

To identify patterns that were less obvious or more esoteric, further

methods were used. GOOL's DSL should be capable of expressing simple and "usual" programs, to better satisfy the project goals described in Chapter 3. Therefore, some representative example programs were written in GOOL. During this process, roadblocks were sometimes encountered when GOOL was missing certain feature that were needed. At these points, a new pattern had to be found and implemented as a feature in GOOL.

For instance, the State example program represents a common Java design pattern (see 8.5.1). To produce the desired functionality, some of the classes that comprise this example had to inherit from another class. GOOL's DSL did not yet allow for inheritance, but all of the considered mainstream languages supported it in some form. Thus, an important OO pattern was naturally rediscovered. Examples of other patterns identified in this way include various arithmetic operations, list-element modification, and printing to the console.

4.3 Printing Code

As discussed in Section 4.5, GOOL translates code from the DSL to mainstream languages. This translation is, by design, very direct and straightforward — it may be more appropriate to refer to the process as simply "pretty-printing" the code in a particular language. This can be taken as a measure of the "distance" between GOOL's DSL and mainstream languages.

This was done to meet one of the project's goals (see Chapter 3). In particular, the renderer does not perform any inference, and so all relevant details must already be present in the AbstractCode AST representation.

Therefore, the method for printing code is also straightforward: for each output language, GOOL has a translation "dictionary" that explicitly describes how to pretty-print each primitive of the DSL. This dictionary is consulted for every component of a GOOL program that is to be printed.

See Section 7.4 for a more detailed look at the code generation process.

4.4 Language Requirements

The language created for this project must meet several criteria in order to accurately and closely represent the OO core. The following is a list of requirements that we designed for the DSL.

- The language can be easily written and read by a programmer of common skill and experience. It should be fairly natural for the programmer to write code in this language.
- It must accomodate a typical OO structure (what constitutes this structure will be discussed in detail in Chapter 6), and it must provide, at least at a basic level, the fundamental features associated with OO languages and programming languages in general.
- The language must include only such features as could be expected from a typical OO language.
- The scope of the language should be sufficient to fully represent simple common or "usual" programs.
- It must be a generic representation. The structure of the language does not necessarily conform to the structure of any particular existing language.

4.5 Software Requirements

As mentioned previously, the goals of Chapter 3 were accomplished by creating a piece of software. This software had to be specifically designed towards these goals. This section will describe what criteria were required of the software and its DSL in order to ensure that they were met.

- The software will demonstrate the "common core" of OO languages explicitly. It will generate object-oriented code in several different languages from a single representation.

- The software should accept a code representation in the form of the OO language described in Section 4.4.
- The process of translation to all languages must be straightforward and unambiguous.
- The software must transform written generic code into corresponding mainstream object-oriented code in multiple languages.
- Additionally, the output code must be formatted in such a way as to facilitate reading and interpretation by humans. It must follow the usual conventions for whitespace and indentation of written code.
- When compiled and run, the output code should produce semantically equivalent results across all supported languages.
- The software should generate this output code in several different languages which support the object-oriented paradigm, in order to exemplify the patterns which underly it. When rendering code to a language that supports additional paradigms, the object-oriented style should always be preferred.
- Common/popular languages should be chosen as the output languages, as it is desirable to maximize the proportion of programmers that can easily comprehend and use the output code.
- The code rendering system must facilitate extension with supplementary rendering languages.
- In addition, the software should be well-designed with respect to software engineering principles and software quality metrics, including modularity, comprehendability, and maintainability.

We decided that software which meets all of these criteria will suit the aim of the project, and accomplish the overarching goals.

Chapter 5

Domain-Specific Languages (DSLs)

Before discussing technical details, it would be beneficial for the reader to have some background knowledge of important concepts. Thus, we briefly digress to provide an introduction to DSLs, and an explanation of how one is used in this project.

SAGA (described in Section 8.1) also made prominent use of a DSL. This chapter in particular is based largely on the report compiled by its author [Bey11].

5.1 What is a DSL?

A DSL is a language of limited scope which is intended to solve a specific set of problems. This set is the domain of the language. DSLs are used to efficiently and concisely express their respective domains. DSLs are typically not Turing complete, and thus they are generally not used to define entire systems like general-purpose languages.

DSL design requires a thorough understanding of the domain in question. This is because there must be some automated process that can deterministically transform a DSL program into some desired representation. Additionally, since DSLs are frequently used by domain experts to express

domain-specific ideas, the DSL should incorporate the domain's standard representation of problems.

The purpose of employing a DSL is usually to simplify the programming involved in creating a solution for a domain-specific problem — if it can be accomplished just as easily in a general-purpose language, then the effort involved in creating the DSL may not be worthwhile. Using a DSL should generally increase productivity. Sometimes, however, this is not the case, and the main benefits of the DSL lie elsewhere. These might include increased quality of software with respect to various metrics such as reliability, understandability, maintainability, testability, and efficiency. Additionally, the DSL could facilitate implementation of ideas by domain experts who are not programmers.

DSLs may employ various programming styles. Some DSLs are "solution-oriented". This means that the form of the desired solution is described algorithmically. Others are "problem-oriented", meaning that the code is used to describe only what the program should accomplish, but not how to do it. These styles are analagous to, respectively, the imperative and declarative paradigms of general purpose languages. The declarative paradigm is used by many DSLs, and as such they are oriented towards describing the problem. This is often because DSLs are intended to be simpler to program in (within their own domain), and the declarative style is usually appropriate for this purpose.

One primary goal of a DSL is to greatly simplify the expression of common tasks within the domain, with respect to how these tasks would be expressed in conventional languages. Another goal is to reduce the difficulty of implementing complex domain problems.

5.2 Internal vs. External DSLs

Internal/embedded DSLs can be distinguished from external DSLs. Each type of DSL has benefits and drawbacks in various scenarios. Choosing between an internal and an external DSL requires consideration of several factors.

A DSL that is embedded within some general purpose host language is referred to as an internal DSL. Programming is done in the host language, but with the help of some integrated constructs so that the DSL code is essentially a "new" language within the host language, without being entirely distinct.

The effort required to create an internal DSL is relatively low, since the code can be processed by the host language's compiler. Additionally, the developer of the DSL can take advantage of the host language's established capabilities. One drawback of an internal DSL is that, since it must be written within the constraints of the host language's syntax and grammar, its expressiveness may be limited by the rules of the host's parser. The severity of this limitation can vary greatly between host languages. Another potential problem is that the domain experts will be programming in a (host) language with which they might not be familiar.

External DSLs are not embedded in a host language. As such, they will not be processed by an existing compiler, and the syntax and grammar can be freely customized. This will generally allow for a more natural interface with the DSL, making it easier for domain experts to express ideas. However, this type of DSL will be more difficult to implement, as a compiler must be created for it. This can be a complex task, as the compiler must parse the code, possibly create an intermediate representation, and then generate output code.

5.3 Project Domain

As can be inferred from previous chapters, the domain relevant to this project is that of object-oriented languages in general. The result is a rather special case of a DSL, whose domain is comparable to general purpose languages, and whose "domain experts" are programmers. It is still very much a DSL, as it is specialized to favour typical OO problems, and its capabilities are limited to those most fundamentally required in typical OO languages, while still being Turing-complete.

An internal DSL was chosen to reduce the complexity of development,

and expedite the process. As this is primarily an academic exercise, we decided that an internal DSL would suffice to accomplish the goals of the project and demonstrate the desired patterns. Since the intended users are programmers, ease of development could be prioritized over giving the DSL a fully customized syntax, facilitating the implementation of more advanced features and patterns.

The choice of host language is discussed in Section 7.1.

Chapter 6

Concepts of an OO Language

At this point, before GOOL's DSL is discussed in detail, the reader should understand what was taken to be the fundamental structure of a "typical" OO language. This structure drove much of the design of the language. As such, a brief dissection of this structure is provided here.

Note that this is a description of the structure as specifically implemented in this project's DSL — obvious concepts which are ignored (e.g. the *protected* scope in Section 6.2) are simply not supported by the language at this time.

6.1 Classes and Objects

A class is an OO construct which is a template for instances of itself [MSD12]. Instances are *objects* of a class, and the class, in that sense, is often called the *type* of these objects. In general, a class must be instantiated as an object before any of its contents can be accessed or used.

A class definition can contain a local set of variables, called instance variables. The class can also include its own set of methods (see Section 6.3). All of these members are used to associate a certain state and behaviour with objects of that class. Each member is associated with a certain scope (see Section 6.2).

Multiple objects can be declared as instances of the same class. In this

case, each object maintains its own independent collection of data. Modifying the values of the instance variables of one object will not change the variables of any other.

Within a method of a class, *this* or *self* refers to the current object — i.e., the object whose method is being called.

There may also be a hierarchy to classes. A class can "inherit" from another class. If class D inherits from class C, then C is called the *parent* or *superclass* of D, and D is a *child* or *subclass* of C. Inheritance enables code reuse and sharing. Subclasses inherit the methods and variables of their parents.

Any program that creates a variable (object) of type C is called a *client* of class C.

At a high level, an object-oriented program can be viewed simply as a set of interacting objects [TN02].

6.2 Scope

The scope of a method or variable is its level of visibility with respect to its class.

If a variable or method is declared *public*, it will be visible outside of the class. That is, that variable or method can be accessed by any clients of the class.

If a variable or method is declared *private*, it will only be visible within the class. It can be accessed by other methods of the class, but not by clients or subclass objects [TN02].

6.3 Methods

A method is a special function that is a part of an object. It can take a certain set of arguments as input. It may also return some value of a particular type (if not, its return type is "void"). The argument types and return type make up the method's *signature*. In a sense, a method represents an action that the object can perform [Sla08].

A special method without a return type, called a constructor, is used to initialize a new object of its class. Execution of the constructor performs memory allocation for the object, and can initialize the instance variables. A constructor is typically not mandatory; if one is not explicitly provided in a class definition, then a simple one (with no parameters) will be implicitly generated. For example, in Java, an object of type C would be initialized in the following way:

```
C x = new C(<arguments>);
```

This would call the class constructor and pass it the specified arguments. Any public method or variable of x can now be referenced by the client code.

Communication among objects is achieved by calling each others' methods — this is called *message passing*. The calling object passes the appropriate parameters for the desired method and waits for a response from the callee, in the form of a return value [TN02].

6.4 Statements

Any block of code is composed of statements. A statement is the smallest element of code which actually performs some action. *Simple* statements may involve assignment to a variable, or calling a function. *Compound* statements can contain other statements (for example, an "if" statement can contain one or more blocks of simpler statements to execute when certain conditions hold) [Net11].

Control structures are statements, construction patterns, rules, or combinators which affect the flow of execution. These include conditionals (like the "if" or "switch" statements) and loops ("while", "for", "forEach").

6.5 Expressions

An expression is some arrangement of constants, variables, function calls, and operators which is well-constructed with respect to the rules of the

particular language, and can be evaluated (e.g., `5 / 0` cannot generally be evaluated). An expression can be evaluated to a value [Inc00].

6.6 Values

A value is considered to be an object or an expression of the language which is in normal form (i.e. it cannot be evaluated any further). Values inhabit types [Mit96].

6.7 Base Types

Base types are the simplest types natively supported by a language. Variables of a base type are not objects; they simply hold a value of the relevant type. In the case of GOOL's DSL, these are integers, floats, characters, strings, and booleans.

6.8 Compound Types

Compound types are constructed from simpler types (e.g., the type of a certain list object could be List of Lists of Integers). In GOOL, the only compound types are lists and iterators.

Chapter 7

The GOOL Program

This chapter details the final state of the GOOL software tool and the AbstractCode language.

7.1 Overview

The GOOL program is a code generator. The programmer writes DSL code as an AbstractCode data structure, and GOOL can print equivalent code in one of the supported output languages.

When run, it first parses a text configuration file and uses this to set some user-defined options in an internal data structure, and to determine in which language to render code. It then passes an AbstractCode data structure (containing the DSL implementation) and any relevant settings to the rendering module of the chosen language. From there, the DSL code structure is evaluated recursively to determine how each component should be rendered, and source files are created containing the resulting code.

GOOL is implemented entirely in Haskell [web11]. Haskell is a purely functional language with several advantages for this type of program. GOOL makes frequent use of higher-order functions, particularly when choosing a function to render a given term in AbstractCode. Haskell handles these in a natural manner which is wholly integrated with the way it handles other sorts of functions. Haskell is also excellent at tree processing (which is what

ASTs are), and is thus an ideal choice for implementing DSLs.

Also, Haskell has a very streamlined syntax which encourages superior readability [Dro09]. There is something of a learning curve for programmers who are inexperienced with functional languages, but once this has been surmounted, concise, clear, and effective code can be written with ease. Haskell provides powerful constructs, like folding and pattern matching, which are particularly effective in facilitating GOOL's implementation. Language definitions in the style of, for example, the Lambda Calculus, naturally lend themselves to functional languages like Haskell (as functional languages are generally, at their cores, augmented forms of the Lambda Calculus [Mig]), which helps to increase understandability and decrease the complexity of designing a system around such a language.

7.2 Currently Supported Output Languages

Renderers for six languages exist in the current version of GOOL. These languages are C#, C++, Java, Objective-C, Python, and Lua.

We chose C#, C++, and Java because they are some of the most popular general purpose languages [LLC11], and are particularly popular among object-oriented languages. This relates back to the requirement about usefulness of the output code. Additionally, this provided a good basis for a representative sample of OO programming. As an added benefit, C# and Java are very similar in style and syntax — which means that their renderers are also very similar. It usually required little additional effort to render a feature in one language once it had been implemented in the other.

We chose Objective-C for its popularity [LLC11], and its current relevance as the primary language of the Cocoa API, used for Mac OS X and iOS (operating system of the iPhone and iPad) programming [Inc11].

Python was chosen for its popularity as well [LLC11], but also because it is a multi-paradigm language that supports object-oriented programming and is not always used to write OO programs. The idea here was to show that, if Python code could be generated alongside other, more "pure" OO languages, the patterns demonstrated would be inherent to OO program-

ming and not just OO languages.

Lua may seem like a strange choice, as it does not have any built-in support for the concept of classes and objects [Lit11]. However, it is possible to simulate this feature using Lua's tables [Lit11, Ier96]. It is even possible to implement OO concepts like inheritance with some effort and overhead [Ier96]. Lua is supported by GOOL in order to take Python's idea much further — if the same common core of an OO language can be found and implemented in this feigned OO environment, then it is likely to be a legitimate observation.

7.3 AbstractCode

AbstractCode is, as mentioned above, an abstract syntax tree representing the GOOL DSL. The language definition is comprised of a multi-tiered data structure, which, at its top-level, encapsulates a complete software program. This program is comprised of modules. Each module is made up of member variables and transformations (methods). The decomposition continues like this down to the level of literals and variables.

The full definition of the AbstractCode language is given in Section A.5. Here, the most central primitives of the language will be shown and explained.

```
data Class = Class {
               className :: Label,
               parentName :: Maybe Label,
               classScope :: Scope,
               classVars :: [StateVar],
               classMethods :: [Method]}
data StateVar = StateVar Label Scope StateType Int
data Method = Method Label Scope MethodType [Parameter] Body
```

A **Class** in GOOL is equivalent to a class in Java or other mainstream OO languages. The "parentName" field allows a Class to inherit from another class by providing its name/label. The member variables and methods of this class are contained in the lists "classVars" and "classMethods", respectively.

Hence, **StateVars** are class variables and **Methods** are just class methods. A StateVar requires a name, scope, and type. The integer field is for the "deletion priority", explained in 8.4.1. A Method requires a name, scope, type, list of parameters, and body of code.

```
data BaseType = Boolean | Integer | Float | Character | String
data ListType = Static | Dynamic
data StateType = List ListType StateType
               | Base BaseType
               | Iterator StateType
               | EnumType Label
               | Type Label
data MethodType = MState StateType
                | Void
                | Construct Label
```

The type of a state variable can fall into a few different categories, defined by **StateType**. If it is a list, it must specify whether the list is static or dynamic, and the type of its elements. Base types include the simple or standard primitive types. An iterator must specify the types of the elements it iterates over. EnumTypes and "custom" Types are selected by providing the name of the desired type. These are typically intended for miscellaneous types defined elsewhere in the same program. For example, if a programmer has defined a class C, and then later wants to create an object of type C, this object could be declared as:

```
StateVar "c_obj" Public (Type "C") 0
```

This would be equivalent to a declaration in C++ or Java as follows:

```
public C c_obj
```

MethodTypes define the possible types for a GOOL Method. Methods have a few more possibilities than variables do, so they require a separate set of types (of which the StateTypes are a subset, since a Method can return a value of any StateType). Unlike StateVars, they can be Void, meaning that they do not return a value. They can also be Constructors of a particular class, which are considered special Method types in GOOL.

```
type Body = [Block]
data Block = Block [Statement]
```

```
data Statement = AssignState Assignment
               | DeclState Declaration
               | CondState Conditional
               | IterState Iteration
               | JumpState Jump
               | RetState Return
               | ValState Value
               | CommentState Comment
               | FreeState Value
               | PrintState {newLine :: Bool, valType :: StateType,
                             printVal :: Value}
               | ExceptState Exception
               | PatternState Pattern
```

These primitives begin to represent the code on a lower level. A **Statement** is a single instruction or control structure. A **Block** is just a series of Statements, and a **Body** is just a group of Blocks (Bodies are only used for aesthetic purposes — so that related Blocks of code can be split up).

Statements are very important to the language as a whole, since they are where the programmer specifies what is actually to be done. They encapsulate a variety of purposes.

JumpStates are used for loop control statements: Break and Continue (with their traditional meanings).

RetStates are used for returning values from method calls.

ValStates are used to just evaluate values or expressions, without assignment (typically for their side effects).

There is also a Statement included for comments (CommentState), which would not normally be included in a language definition alongside functional primitives. This illustrates an interesting point about the limitations of internal DSLs. In the case of AbstractCode, an internal DSL, any information that should be carried through to the printed output must be specified within one of the Haskell data structures that comprise a program representation. Thus, comments had to be incorporated into the syntax of AbstractCode.

FreeStates are used to free memory by deallocating variables.

PrintStates are used for printing to the console.

ExceptStates are used for throwing and handling exceptions. They are described further in Section 8.2.

PatternStates are explained in Section 8.3.

The other Statement types will be demonstrated shortly.

```
data Assignment = Assign Value Value
                | PlusEquals Value Value
                | PlusPlus Value
```

Assignment Statements are used to assign a value to a variable. For example,

```
Assign (Var "x") (Lit $ LitInt 5)
```

is equivalent to the Java assignment:

```
x = 5
```

The Var and LitInt primitives will be shown later.

Specialized addition or incremental Assignment operations are also provided for convenience. The corresponding Minus operators are not included in AbstractCode, but are supported by GOOL with operators `&-=` and `&~-` (`&--`, as may have been expected, is not used because "`--`" starts a comment in Haskell code, and thus could not be used as part of an operator in AbstractCode — this is representative of one drawback of internal DSLs).

```
data Declaration = VarDec Label StateType
                 | ListDec ListType Label StateType Int
                 | ListDecValues ListType Label StateType [Value]
                 | VarDecDef Label StateType Value
                 | ObjDecDef Label StateType Value
                 | ConstDecDef Label Literal
```

Declaration Statements are used to create new variables, lists, objects, or constants. The "Dec" versions just declare, while the "DecDef" versions allow the programmer to declare and initialize to a certain value at the same time.

```
data Conditional = If [(Value, Body)] Body
                 | Switch Value [(Literal, Body)] Body
```

The **Conditional** Statements use a list of (Value, Body) pairs to allow for an unlimited number of conditional branches. The Value serves as the condition to check, and is expected to evaluate to a boolean. The final Body

of code is used as the "default" branch, executed when none of the others hold true. For example,

```
CondState $
    If [( Expr $ BinaryExpr (Var "x") Less (Lit $ LitInt 5),
          [ Block [ RetState $ Ret (Var "x") ] ] ),
        ( Expr $ BinaryExpr (Var "x") Greater (Lit $ LitInt 7),
          [ Block [ RetState $ Ret (Lit $ LitInt (-1)) ] ] )
       ] [ Block [ RetState $ Ret (Lit $ LitInt 0) ] ]
```

Is equivalent to:

```
if (x < 5) return x;
else if (x > 7) return -1;
else return 0;
```

To demonstrate the language in its purest form, the code examples above use only AbstractCode primitives, and do not utilize the many shortcut functions defined by GOOL. While GOOL is still more verbose than a typical language, use of these shortcuts can greatly reduce the overhead and simplify programming. For example, the following code is equivalent to the last example:

```
ifCond [( Var "x" ?< litInt 5,
          oneLiner $ return (Var "x") ),
        ( Var "x" ?> litInt 7,
          oneLiner $ return (litInt (-1)) )
       ] (oneLiner $ return (litInt 0))
```

A few examples of these can be seen in 8.4.4. Shortcut functions are also demonstrated in a more practical way through example implementations in Section 8.5.

```
data Iteration = For {initState :: Statement, guard :: Value,
                      update :: Statement, forBody :: Body}
               | ForEach Label Value Body
               | While Value Body
```

Iteration Statements are loops. AbstractCode supports For, ForEach, and While loops, all of which implement their traditional behaviour. Note that the initialization and update components of the For loop are just Statements. It is up to the programmer to use meaningful Statements here, e.g.

a loop index declaration for "initState", and an assignment that increments the index for "update".

```
data Value = EnumElement {enumName :: Label, elemName :: Label}
           | EnumVar Label
           | Expr Expression
           | FuncApp Label [Value]
           | ObjAccess Value Function
           | Lit Literal
           | Const Label
           | Self
           | StateObj StateType [Value]
           | ObjVar Value Value
           | Var Label
           | ListVar Label StateType
           | Arg Int
           | Input
```

Values make up the most common low-level component of AbstractCode. The definition encompasses not only the traditional things that would be called values in a programming language (like literals) but also terms that would evaluate to a traditional value (like expressions or function calls). It essentially includes any component that can ultimately be assigned to a variable.

One thing to note here is how different kinds of variables have separate primitives for referencing. In a mainstream language, a variable that represents a list or an enumeration element can typically be referred to in the same way as, say, an integer variable. However, when GOOL prints code in a particular language, it sometimes needs to make simple adjustments that cannot be explicitly specified in the AbstractCode program (since, by design, it performs no inference during translation). Take, for example, enumeration elements. In some output languages, these are really just named integers. In others, like Java, they are a separate entity. Hence, if an AbstractCode program attempts to reference an EnumVar like an integer (say, using it as a list index), then the generated Java code must explicitly cast the variable to an integer, whereas the generated C++ code does not need to do this. So GOOL's Java-printing module must know to insert the casting during code generation. Since GOOL truly just pretty-prints the code, and does

not perform any sort of complex inference, the nature of the variable must be explicitly denoted to facilitate this step. Thus, we have an EnumVar primitive to allow for differentiation. See 8.4.6 for an account of a similar situation with ListVars.

```
data Literal = LitBool Bool | LitInt Int | LitFloat Float
             | LitChar Char | LitStr String
```

Literals wrap Haskell literals (of the base types) to allow them to be used within an AbstractCode program, as seen in the previous examples in this section.

A few additional AbstractCode features that were added during development are discussed in Section 8.2.

7.4 Code Generation

Code generation is performed by a set of Haskell "language renderer" modules. One module is required for each output language. These modules define how AbstractCode is translated to their respective languages.

There is also a primary code generation module called LanguageRenderer. This module defines the Config data structure, which serves as an explicit dictionary for a specific language — i.e., it contains a set of functions which define a direct translation from each component of the AbstractCode language to the target language. Each specific language renderer must populate the fields of the Config structure with these "translation" or rendering functions for the various components of AbstractCode. Additionally, there are several fields with simpler types for certain keywords and properties of the language which do not require full rendering functions. The Config structure completely defines how AbstractCode programs should be rendered in a given language.

Listing 7.1: Partial Haskell Definition of the Config Data Structure

```
data Config = Config {
    renderCode :: [Label] -> AbstractCode -> Code,

    argsList :: Doc,
```

```
    bitArray :: Doc,
    commentStart :: Doc,
    ext :: Label,
    include :: Label -> Doc,
    inherit :: Doc,
    iterForEachLabel :: Doc,
    iterInLabel :: Doc,
    ...

    top :: FileType -> Label -> Doc,
    body :: FileType -> Label -> [Class] -> Doc,
    bottom :: FileType -> Doc,

    assignDoc :: Assignment -> Doc,
    bodyDoc :: Body -> Doc,
    conditionalDoc :: Conditional -> Doc,
    declarationDoc :: Declaration -> Doc,
    exprDoc :: Expression -> Doc,
    funcDoc :: Function -> Doc,
    iterationDoc :: Iteration -> Doc,
    litDoc :: Literal -> Doc,
    classDoc :: FileType -> Label -> Class -> Doc,
    scopeDoc :: Scope -> Doc,
    statementDoc :: StatementLocation -> Statement -> Doc,
    valueDoc :: Value -> Doc,
    ...
}
```

The Doc type here is used by the HughesPJ pretty printer, and represents an assembled document or piece of text that can be written to an output file [Ter12]. All generated code is formatted with appropriate spacing and line breaks to make it human-readable.

The other major component of the LanguageRenderer module is made up of default implementations for every necessary rendering function. Each rendered language has some components that are represented the same way in another language. Java and C#, for example, can use identical implementations for the majority of their rendering functions. Even for very dissimilar languages, like Lua, many elements, such as lists of parameters or function calls, would overlap with those found in other languages. Thus it was a natural decision to have a pool of rendering functions that can be shared amongst the individual renderers. These shared implementations help to

avoid a great deal of unnecessary effort and repeated code. Some functions have secondary or tertiary shared implementations, when there is more than one implementation that can be shared amongst two or more languages.

Often, however, a language's way of representing a certain component does not overlap with any others. In these cases, the language renderer must provide its Config structure with a custom implementation of the matching type signature.

The naming convention that has been adopted for rendering functions is as follows: the Config-structure field for a function is suffixed with just "Doc" (e.g. "conditionalDoc"). LanguageRenderer's default implementation ends with DocD ("conditionalDocD"). Apostrophes are appended for any secondary and tertiary shared implementations ("conditionalDocD′", "conditionalDocD″"), as is the standard Haskell convention for denoting a modified version of a function [Lip11]. Language-specific implementations end with "Doc′" ("conditionalDoc′"); again, a nod to the Haskell convention.

Some representative rendering functions will be shown below, as examples. Section 7.3 can be referenced for the meaning of any AbstractCode primitives used below.

```
assignDocD :: Config -> Assignment -> Doc
assignDocD c (Assign v1 v2) = valueDoc c v1 <+> text "=" <+>
        valueDoc c v2
assignDocD c (PlusEquals v1 v2) = valueDoc c v1 <+> text "+=" <+>
        valueDoc c v2
assignDocD c (PlusPlus v) = valueDoc c v  <> text "++"
```

This is the default implementation of a rendering function for Assignment statements. This function is particularly straightforward. Haskell's pattern matching is used to differentiate among the possible Assignment constructors. Some of the pretty printer's functions are used to form the returned Doc: the "text" function isomorphically transforms a String into a Doc, the "<>" operator concatenates two Docs, and "<+>" concatenates and inserts a space [Ter12]. Thus, a simple assignment would have the printed form `value1 = value2`, a "PlusEquals" assignment would have the form `value1 += value2`, and a "PlusPlus" assignment would look like `++value`.

```
bodyDocD :: Config -> Body -> Doc
```

```
bodyDocD c bs = vibmap (blockDoc c) blocks
    where blocks = filter (\b -> not $ isEmpty $ blockDoc c b) bs

blockDocD :: Config -> Block -> Doc
blockDocD c (Block ss) = vmap (statementDoc c NoLoop) statements
    where docOf s = statementDoc c NoLoop s
          notNullStatement s = (not $ isEmpty $ docOf s)
              && (render (docOf s) /= render (end c NoLoop))
          statements = filter notNullStatement ss
```

These functions are responsible for pretty-printing Bodies and Blocks of code, respectively. Since a Body is just a collection of Blocks, the "vibmap" combinator is used to intersperse line breaks between all Blocks. Any empty Blocks are filtered out and ignored. The blockDocD function prints a Block as a sequence of Statements — again, filtering out blank Statements.

```
conditionalDocD :: Config -> Conditional -> Doc
conditionalDocD c (If (t:ts) elseBody) =
    let ifSect (v, b) = vcat [
            text "if" <+> parens (valueDoc c v) <+> ifBodyStart c,
            oneTab $ bodyDoc c b,
            blockEnd c]
        elseIfSect (v, b) = vcat [
            elseIf c <+> parens (valueDoc c v) <+> ifBodyStart c,
            oneTab $ bodyDoc c b,
            blockEnd c]
        elseSect = if null elseBody then empty else vcat [
            text "else" <+> ifBodyStart c,
            oneTab $ bodyDoc c elseBody,
            blockEnd c]
    in vcat [
        ifSect t,
        vmap elseIfSect ts,
        elseSect]
conditionalDocD _ (If [] _) = error "If with no body encountered"
```

This is the portion of the default Conditional-rendering function that deals with If conditionals (the Switch part uses a similar approach).

Components like "ifBodyStart", "blockEnd", and "elseIf" are defined in the Config structure for each language. By having simple keywords specified separately, we can avoid having to rewrite the conditionalDoc function for a certain language just because, say, other languages use "else if" while this one uses "elif". The same pattern still holds, albeit with slight differences

in syntax.

The first (Value, Body) pair is taken to be the "if" branch, and any subsequent pairs are "else if" branches. Each "if" or "else if" branch uses a similar format:

```
[else] if (condition) {
        <code Block to execute if condition holds>
}
```

All branches (including the "else" branch, as long as it is not empty) are concatenated to form the final Doc.

The second pattern-matching case catches If Conditionals with empty (Value, Body) lists. These are invalid, since an "if" statement must check at least one condition, so an error is thrown and the printing process fails.

```
statementDocD :: Config -> StatementLocation -> Statement -> Doc
statementDocD c loc (AssignState s) = assignDoc c s <> end c loc
statementDocD c loc (DeclState s) = declarationDoc c s <> end c loc
statementDocD c _ (CondState s) = conditionalDoc c s
statementDocD c _ (IterState s) = iterationDoc c s
statementDocD c loc (JumpState s) = jump s <> end c loc
statementDocD c loc (RetState s) = retDoc c s <> end c loc
statementDocD c loc (ValState s) = valueDoc c s <> end c loc
statementDocD c _ (CommentState s) = comment c s
statementDocD c loc (FreeState v) = text "delete" <+> valueDoc c v <>
        end c loc
statementDocD c loc (PrintState newLn t v) = printDoc c newLn t v <>
        end c loc
statementDocD c loc (ExceptState e) = exceptionDoc c e <> end c loc
statementDocD c loc (PatternState p) = patternDoc c p <> end c loc
```

This function makes more use of Haskell's pattern-matching. A different implementation is required for each Statement type. In most cases, the statementDocD function just refers to a more specific rendering function for that kind of Statement and applies the end-of-statement symbol (a semicolon, for languages that use such a symbol).

The interesting point in this case is that, since it is so general, every output language uses this version of the Statement-rendering function. It is illustrative of a recurring theme in the code-generation system: refer back to the language-specific Config structure wherever possible, so as to maximize generality.

7.5 Configuration File

A simple configuration file must be provided to GOOL at runtime. Its main purpose is to specify the desired code generation language (this is the only mandatory setting), and to tell GOOL to render one of the included AbstractCode example implementations, if desired. If an example implementation is not specified, GOOL attempts to render the default AbstractCode implementation.

There are a few language-specific options that can be set as well. A default setting is specified for each one in the relevant language renderer. These design choices are provided merely as an example of possibilities; if GOOL's development were to be continued, there are numerous other configuration settings that could and should be supported.

An EBNF listing of the syntax of the configuration file can be found in Section A.4.

The configuration file options are:

- *Generation Language*: the desired output language for code generation. Supported choices are C#, C++, Java, Objective-C, Python, and Lua. This option must be set in the configuration file, or GOOL will not run. All others are elective.

- *ExampleImplementation*: if this option is set, then code for the chosen example will be generated instead of the default implementation module. Available choices are Patterns, QuickSort, State, and StateV2.

- *JavaListType*: the container type to use for lists in pretty-printed Java code. If Java is not the chosen Generation Language, then setting this option will have no effect. Supported list types are ArrayList, LinkedList, and Vector. If this is not set, GOOL will default to using Vectors.

- *CppListType*: the type to use for lists in pretty-printed C++ code. This option will have no effect if C++ is not the chosen Generation

Language. Supported list types are deque and vector. The default choice is vector.

- *ObjCStaticListType*: the type to use for lists in pretty-printed Objective-C code that are declared Static (Dynamic lists must be NSMutableArrays). This option will have no effect if Objective-C is not the chosen Generation Language. Supported list types are NSArray and NSMutableArray. The default choice is NSArray.

Chapter 8

Detailed Development History

At this point, the reader has hopefully gained a reasonable understanding of what GOOL does and how it works. This chapter serves to provide the educated reader with a more detailed summary of what work was done to transition from SAGA to GOOL in its current form, throughout the duration of this project. This includes major changes, additions, and a few of the notable issues that were faced. The account is roughly chronological, though some related efforts have been grouped for the sake of readability.

8.1 Starting Point

In order to properly explain the development of GOOL, it must first be understood where development began.

GOOL, the tool developed to meet the requirements of Chapter 3, was built off of a previous DSL-centric project, known as SAGA (Story as an Acyclic Graph Assembly) [Bey11, BC11]. SAGA implemented a text-based DSL which allowed non-programmers to easily and efficiently create a "story engine" for a video game that could be integrated into real-world game projects. The SAGA program served as a compiler: It parsed a "story description file" (DSL code), created an intermediate representation of the

story engine, and then generated the actual code in a user-selected general purpose language.

As mentioned in Chapter 2, SAGA provided much of the framework needed for GOOL. For example, SAGA already included the necessary facilities for code generation in C#, C++, and Java. The main benefit to starting from SAGA was its intermediate representation language. In SAGA, the intermediate representation was an algebraic data type which effectively modelled the basic structure of source code (i.e. abstract syntax trees).

GOOL's DSL began to take shape from the roots of SAGA's intermediate representation, called AbstractCode.

For a full discussion and report on SAGA, please see [Bey11] and [BC11].

8.2 Added AbstractCode Features

This section will discuss a select few of the primitives that were implemented in the AbstractCode language during development of GOOL.

Listing 8.1: Binary operators, with symbolic versions.

```
data BinaryOp = Equal | NotEqual | Greater
              | GreaterEqual | Less | LessEqual
              | Plus | Minus | Multiply
              | Divide | Power | Modulo

Equal:        v1 ?== v2
NotEqual:     v1 ?!= v2
Greater:      v1 ?> v2
GreaterEqual: v1 ?>= v2
Less:         v1 ?< v2
LessEqual:    v1 ?<= v2

Plus:     v1 #+ v2
Minus:    v1 #- v2
Multiply: v1 #* v2
Divide:   v1 #/ v2
Power:    v1 #^ v2
Modulo:   v1 #% v2
```

We added several binary operators to the language, each with a symbolic shortcut operator. Equal, Less, and LessEqual had already been included in SAGA's version, but the rest are new. Since there was also a unary Not operator, supporting the other logical operators was not strictly necessary. These are an example of a feature that was added because it is expected in any full-featured programming language, for the programmer's convenience.

The arithmetic operators, on the other hand, are representative of features that were added because they are strictly necessary. Mathematical operations are a fundamental component of programming.

Listing 8.2: Two of the primitives for an AbstractCode Module.

```
data Class = Enum {
                 className :: Label,
                 classScope :: Scope,
                 enumElements :: [Label]}
           | MainClass {
                 className :: Label,
                 classVars :: [StateVar],
                 classMethods :: [Method]}
           | ...

data Method = MainMethod Body
                    | ...
```

Usage example (defines an Enum for the days of the week, and in the MainMethod, declares a variable named "today" and initializes it to the Enum element for Tuesday):

```
daysEnum :: Class
dayEnum =
  Enum "Days" Public ["Mon","Tue","Wed","Thur","Fri","Sat","Sun"]

testClass :: Class
testClass = MainClass "Test" [] [
        MainMethod [
            Block [
                varDecDef "today" (EnumType "Days") ("Days" $: "Tue")
            ]
        ]
  ]
```

SAGA only supported one standard Class type. GOOL added Enum Classes as a useful feature. They are not truly necessary (in fact, some of the output languages do not have built-in Enums; in these cases, GOOL simulates them with regular integers). The feature was inspired by SAGA's Story Manager, which would communicate about a specified list of story events using hard-coded strings. This seemed inefficient and ugly, and having an Enum type would have allowed for a better, while still human-readable, solution. We assumed that similar cases could occur in AbstractCode programs.

The MainClass Class and MainMethod Method are somewhat representative of the transformation into a robust, executable language. SAGA's Story Manager was meant to be integrated into larger programs, but GOOL is intended to generate standalone programs. Different languages specify Main in different, specific ways (for example, C++ and Java require a method with a specific name and signature, while Python and Lua programs simply begin execution at code that is not encapsulated within methods or classes), and thus it is necessary to explicitly specify which module and method are the Main ones in AbstractCode. Many components of the language have similar reasons for their existence.

Listing 8.3: AbstractCode exception-throwing and -handling primitives.

```
data Exception = Throw {excMsg :: String}
               | TryCatch {tryBody :: Body, catchBody :: Body}
```

Exceptions were incorporated into AbstractCode because error-handling is an important part of programming. All output languages supported this construct in one form or another, so it seemed natural that AbstractCode should as well. While this approach is very simplistic (TryCatch will only catch exceptions thrown explicitly by an AbstractCode Throw statement, not those that may be thrown when, say, a list is accessed out of bounds; also, there is only one type of exception to throw), it demonstrates the concept well enough to be suitable for the language. Additionally, since different categories of exceptions may be represented in various ways between languages, they would all need to be defined explicitly in AbstractCode in order to be used (but one could certainly imagine defining a set of exception-

type primitives).

8.3 Gang of Four Design Pattern Statements

Listing 8.4: The Pattern Statements of AbstractCode.

```
data Pattern = State StatePattern
             | Strategy StratPattern
             | Observer ObserverPattern
data StatePattern = InitState {fsmName :: Label, initState :: Label}
                  | ChangeState {fsmName :: Label, toState :: Label}
                  | CheckState {fsmName :: Label,
                        cases :: [(Label,Body)], defaultBody :: Body}
data StratPattern = RunStrategy {stratName :: Label,
    strategies :: Strategies, assignResultTo :: Maybe Value}
data Strategies = Strats {strats :: [(Label, Body)],
    returnVal :: Maybe Value}
data ObserverPattern = InitObserverList {observerType :: StateType,
    observers :: [Value]}
```

The Pattern Statements are different from most of the others — they are in no way expected or fundamental. They were added as a more experimental feature. State, Strategy, and Observer are three examples of the Gang of Four design patterns (specifically, they are behavioural patterns) [Jav08], which are a well-known set of patterns that are used solve common OO programming problems.

Even though they do not have directly analogous primitives in the target languages, these patterns can still be implemented rather simply, and do not require any deviation from our goal of maintaining a "trivial" translation process (see 3). In fact, these patterns do not even need to be explicitly translated — all of their rendering functions utilize existing AbstractCode features to create a new implementation. For example, the following Haskell code shows the rendering functions for the Observer Pattern statements.

```
patternDocD c (Observer (InitObserverList t os)) = declarationDoc c $
  ListDecValues Dynamic observerListName t os
patternDocD c (Observer (AddObserver t o)) = valueDoc c $
  obsList $. ListAdd last o
    where obsList = observerListName `listOf` t
          last = obsList $. ListSize
```

```
patternDocD c (Observer (NotifyObservers t fn ps)) = iterationDoc c $
  For initv (Var index ?< (obsList $. ListSize)) ((&++)index) notify
    where obsList = observerListName `listOf` t
          index = "observerIndex"
          initv = varDecDef index (Base Integer) $ litInt 0
          notify = oneLiner $
                     ValState $ (obsList $. at index) $. Func fn ps
```

The Pattern Statements prove that patterns do not need to be an *explicit* part of OO languages in order to be a part of their common core — many patterns can be built from fundamental elements that are implicit to the nature of object-oriented programming. These serve as an illustration of the scope of our language, and thus, the discovered core.

8.4 Changes, Problems, and Improvements

8.4.1 Objective-C

Among the first changes made was the addition of more language renderer modules. SAGA supported only Java, C#, and C++. These languages would not be sufficient for GOOL's goal, due to a lack of both quantity and variety. The Objective-C renderer was produced first, as it was expected that this would be the simplest of the new languages to implement. This perception was, however, due to our initial unfamiliarity with the language. Throughout development, Objective-C was generally the source of more difficulty than any other language, typically when trying to implement a new AbstractCode feature. More than a few of these problems stemmed from memory management. Objects that needed to be explicitly released in Objective-C often did not need this in C++, and vice versa. Thus a feature had to be devised that would allow the "deletion priority" to be explicitly specified in the AbstractCode. This was an unfortunate concession to make, as it is a language-specific parameter, but we deemed it necessary (short of ignoring memory management altogether, which, while possibly still allowing for functional programs, would obviously be incorrect).

It may be enlightening to know of a few representative examples. NSArrays were used as the default "list" type in Objective-C since C-style arrays

would not be sufficient for some of AbstractCode's list-manipulation functions. This choice came with a bit of complexity, as NSArrays cannot hold primitive types. Thus, whenever a primitive is added to a list in an AbstractCode program, the Objective-C rendering must wrap that primitive with an appropriate object (usually an NSNumber) before adding it. Similarly, when reading a list element, the object must first be unwrapped back into a primitive.

Another specific, and curious, example of an Objective-C difficulty was found in dealing with NSNumbers. The rendered implementation would crash when intializing an NSArray with a series of NSNumbers. We found that, when using integers 0 thru 12, the intialization worked as desired. However, using any other number resulted in a crash. We eventually discovered that Objective-C includes some sort of optimization for small integers only. This optimization would somehow prevent a memory issue that would otherwise occur [Ove11]. This assymmetric behaviour was surprising. We eventually solved the problem by using an NSAutoreleasePool to automatically handle the release of the non-optimized NSNumbers.

8.4.2 Python

The Python renderer was added shortly thereafter. In comparison to Objective-C, there were no specific or large categories of problems that came up. Defining a direct translation proved to be relatively straightforward. Some modifications/generalizations did need to be added to AbstractCode. For example, Python needed to know when a statement was referring to a member of "this", as opposed to a locally-defined variable. Thus the "Self" value was added to the language, and all member variables now had to be referenced appropriately.

To illustrate, when a Statement within a class had previously looked like:

```
(Var "x") &= (Var "y") #+ litInt 5
```

Where x and y are class member variables, it would now need to be explicit that x and y are members of `Self`, like so:

```
Self$->(Var "x") &= Self$->(Var "y") #+ litInt 5
```

8.4.3 Structure

At this point, we made some improvements to the structure of the code. Previously, the Config data structure had only contained a few specific options and smaller language features. There was a LanguageRenderer class which defined an assortment of rendering functions that might be used by some of the language-rendering modules. Each specific language renderer would instantiate the LanguageRenderer class and define any supplementary functions that were needed. This was a bit haphazard and inflexible. Some new languages might need more control over rendering than was allowed by the current setup. Thus, we modified the Config structure to serve as an explicit, complete dictionary for a language (this is explained in Section 7.4) — all rendering functions for a language simply had to be referenced in this dictionary; they could be defined or reused as desired. The LanguageRenderer module's purpose was now mainly to define the general Config structure. This resulted in a great deal of repeated code, until LanguageRenderer became the repository for shared rendering functions.

Many of the function implementations were extremely similar between different languages, but had very small differences that would prevent them from being shared. For example, several of the AbstractCode Function statements had the same basic form in most languages (<object>.<function name>(<parameters);), and differed only in the name of the actual function. Specifically, in Java, say, getting the length of a list would look like:

```
list.size();
```

Whereas in Objective-C, we would use:

```
list.count();
```

To avoid this redudancy, we generalized many rendering function implementations. The keyword or component that might differ was added as a field in the Config structure, and then the general implementation referenced this field. In this way, more languages could use the shared/default implementation, and this minimized code repetition. While we undertook a focused effort towards generalization after expanding the scope of the Config

structure, this principle was applied many times throughout development of GOOL, and represents an ongoing process.

We put the naming convention for rendering functions (described in Section 7.4) in place at this point, to help with organization and clarity of the new structure.

8.4.4 Combinators

We then put some work into increasing the language's resemblance to traditional code. SAGA's design had accounted for this to a certain degree, as had subsequent additions during development of GOOL, but there was a fair amount of room for improvement. To this end, symbolic operators were added and standardized. We created more shortcut functions, and generalized some existing ones, or split them into common derivations.

Some illustrative examples follow.

Using only primitives	**Using shortcut functions/operators**
`Expr $ BinaryExpr (Lit $ LitInt 5) Minus (Lit $ LitFloat 3.2)`	`(litInt 5) #- (litFloat 3.2)`
`AssignState $ PlusPlus (Var "i")`	`(&++)"i"`
`ObjAccess (Var "list") (ListAccess $ ObjVar Self (Var "idx"))`	`(Var "list") $. at (Self$->(Var "idx"))`
`If [(Expr $ BinaryExpr (Var "x") Less (Lit $ LitInt 3), [ Block [ PrintState True (Base String) (Lit $ LitString "OK") ] ] )] []`	`ifCond [( (Var "x") ?< (litInt 3), oneLiner $ printStrLn "OK" )] noElse`

This was another process which would continue throughout GOOL's development, requiring some consideration each time more features were added to AbstractCode.

8.4.5 Splitting

Until this point, the AbstractCode definition had remained integrated with SAGA's Story Manager implementation. The program was of an adequate level of complexity, and had provided a suitable testing ground for any changes and additions made thus far. In order to add and test any significant new features, however, it would soon be necessary to take the defining step of detaching the Story Manager from the language. Due to the considerations made in earlier steps, this was much simpler than expected and took little effort — primarily, it involved moving a few blocks of code into a new module. Also, a few datatypes used for story elements were abstracted out, and replaced with generic type-definition constructors (e.g. instead of having the language natively supporting a "NodeTransition" type, the Story Manager code defined an object type with a label of "NodeTransition"). In some cases, modules that should be generic still required the use of some functionality of the story-specific module, but this was soon eliminated.

More important was the creation of "generic mode", which gave GOOL a way to use and render any AbstractCode representation. This had not been possible previously because some extra functionality — namely, the parsing of a story scipt, and the data structures that were subsequently created — was tied in to the representation. Thus, we added a short procedure at the beginning of execution (after the Config file was read and parsed) which decided whether to run GOOL in generic mode or in story mode (essentially, if a Story script was provided, run story mode, else run generic mode). We generalized higher-level code generation instructions to allow for this. Story mode would parse the script as usual and pass the associated data to the dedicated Story Manager implementation module. Generic mode would simply render the implementation found in the default implementation module. This step also did not require significant effort, but was

an important milestone nonetheless.

8.4.6 Special Cases

Concurrently, a few "special cases" that were coded into the language renderers had to be eliminated. These were typically implementations of a rendering function that would check for a certain statement or label that appeared in the Story Manager implementation, and then render it in a specific way for a specific language.

For instance, the following Value-rendering function from Objective-C demonstrates such a "special case":

```
valueDoc' (ObjAccess v@(Var vlbl) f@(ListAccess _)) =
  if vlbl == "eventData" then
    brackets $ objAccessDoc v f <+> innerFuncAppDoc "boolValue" []
  else objAccessDoc v f
```

This case is just intended to print the accessing of a list element (typically, this would only amount to something like `list[0]`). However, it first checks to see if the name of the list being accessed is the same as a specific hard-coded list name used in the Story Manager implementation, and prints an extra function call if so. This is a symptom of the problem described in 8.4.1, where primitive types had to be wrapped and unwrapped when used in lists. The list "eventData" happened to be the only list that coincided with this issue in the Story Manager, and so the problem was temporarily fixed with the patch shown here.

Obviously, this did not suit the aim of GOOL. These special cases were merely a symptom of a more general case that needed to be accounted for. The vast majority of the work involved in fixing these consisted of identifying the general case. For example, the case above was solved by adding the ListVar primitive to AbstractCode. This is used whenever referencing a variable representing a list — as opposed to Var, which would still be used for non-list variables. ListVar requires a type parameter, and thus knows the type of its list elements (whereas Var only requires the label of the variable). This is required in some places to render different list types in different ways,

such as in the Objective-C NSArray issue described above, where primitive types must be unwrapped every time they are accessed from a list.

The new, more general solution to the given example is shown here:

```
valueDoc' c (ObjAccess v@(ListVar _ t) f@(ListAccess _)) =
  getValueDoc t $ objAccessDoc c v f

getValueDoc :: StateType -> Doc -> Doc
getValueDoc t d = let integer = "integer" in
    case t of EnumType _     -> valFrom integer
              Base Boolean   -> valFrom "bool"
              Base Integer   -> valFrom integer
              Base Float     -> valFrom "float"
              Base Character -> valFrom "char"
              otherwise      -> d
    where valFrom typeName =
        brackets (d <+> text (typeName ++ "Value"))
```

Whenever a ListVar with a base type is accessed, the "unwrapping" function for the relevant type is automatically appended.

It should be noted that a more sophisticated type-driven compiler would be able to infer the relevant information. However, in the case of GOOL, this would go against our goal of having a direct, straightforward translation (see Chapter 3 for project goals).

8.4.7 Lua

Subsequently, we implemented the Lua renderer. Since Lua does not natively support object-oriented code, some work was involved in adapting its renderer to the AbstractCode language. However, this was not as difficult as might be expected, since there has been much previous work on implementing OO in Lua. Once this aspect was working correctly, no specific or complex problems were faced.

8.4.8 Extras

We then implemented a few example programs in AbstractCode. These served multiple purposes — intially, the act of creating them was used as a method of identifying important missing features of AbstractCode. After the

language became suitably functional, the implementation modules of these examples were permanently integrated into GOOL, and can now be rendered into compilable code at any time with a Config-file option. They now serve as demonstrations of GOOL's execution, and as examples of finished AbstractCode programs, suitable for reference as necessary. Additionally, they act as proof of the accomplishment of this project's goals. Some of these examples are discussed in detail in Section 8.5.

We devoted most of the remainder of GOOL's development effort to adding to AbstractCode and making it a more full-featured language. Since SAGA's AbstractCode was an intermediate representation used solely for the Story Manager implementation, it had originally lacked any functionality that the Story Manager did not require. For example, it included a "PlusPlus" statement for incrementing an integer variable by one, but otherwise did not support any sort of algebra. As mentioned previously, GOOL's meta-language would need to include a reasonable level of functionality in order to achieve its goals. Thus, we added a multitude of features to the language. Some, like algebra, were fundamental and necessary; others, like "ForEach" loops or "Switch" statements, were not strictly needed (AbstractCode already included For loops and nested If statements), but made the language more robust and programmer-friendly. A selection of the more interesting additions are discussed in Section 8.2.

8.5 Example Implementations

Note that the examples discussed here are only a selection of those which are complete or relatively polished — many other informal examples were used during development to test various language features. While somewhat simple, one can extrapolate the possibility of larger or more complex programs from these proofs of concept.

8.5.1 State

This example is based on the Gang of Four's State design pattern [Jav08]. The State example demonstrates fundamental AbstractCode features and constructs.

The Controller module is the most important component. Note that the definitions of the terms "accounting", "sales", "management", and "connection" are shown in Listing 8.7 below.

Listing 8.5: AbstractCode implementation of the Controller module.

```
controllerClass :: Class
controllerClass =
    let modName = "Controller"
        acctVar = "acct"
        salesVar = "sales"
        mgmtVar = "manage"
        current = "current"
        open = "open"
        close = "close"
        log = "log"

        modVars = [
            pubVar  alwaysDel accounting acctVar,
            pubVar  alwaysDel sales      salesVar,
            pubVar  alwaysDel management mgmtVar,
            privVar neverDel  connection current]

        setCurrent n = oneLiner $
            Self$->(Var current) &= Self$->(Var n)
        doMethod n = oneLiner $
            ValState $ Self$->(Var current) $. Func n []
    in pubClass modName noParent modVars [
        pubFunc (Construct modName) modName [] [
            Block [
                Self$->(Var acctVar) &= litObj accounting [],
                Self$->(Var salesVar) &= litObj sales [],
                Self$->(Var mgmtVar) &= litObj management [],
                Self$->(Var current) &= Self$->(Var acctVar)
            ]
        ],
        pubFunc Void "makeAccountingConnection" [] $
            setCurrent acctVar,
        pubFunc Void "makeSalesConnection" [] $
            setCurrent salesVar,
```

```
        pubFunc Void "makeManagementConnection" [] $
            setCurrent mgmtVar,
        pubFunc Void open [] $ doMethod open,
        pubFunc Void close [] $ doMethod close,
        pubFunc Void log [] $ doMethod log
    ]
```

For easy comparison, the Java code generated for the Controller module follows.

Listing 8.6: Output Java code for the Controller module.

```
public class Controller {
    public Accounting acct;
    public Sales sales;
    public Management manage;
    private Connection current;
    public Controller() {
        acct = new Accounting();
        sales = new Sales();
        manage = new Management();
        current = acct;
    }

    public void makeAccountingConnection() {
        current = acct;
    }

    public void makeSalesConnection() {
        current = sales;
    }

    public void makeManagementConnection() {
        current = manage;
    }

    public void open() {
        current.open();
    }

    public void close() {
        current.close();
    }

    public void log() {
        current.log();
    }
```

```
}
```

The example also illustrates how the embedded nature of the DSL can be leveraged to produce useful program-specific shortcuts when writing code. Several variable names and types have been assigned to names using the Haskell "let ... in" construct. Thus repeated use of string literals can be avoided. Additionally, repeated code can be generalized with parameters, as seen with the "setCurrent" and "doMethod" combinators above. Shortcut functions can also be defined at a more global level, to be accessible within all modules:

Listing 8.7: Sample shortcut function definitions from State.

```
connectionName, acctName, salesName, mgmtName :: String
connectionName = "Connection"
acctName = "Accounting"
salesName = "Sales"
mgmtName = "Management"

connection, accounting, sales, management :: StateType
connection = Type connectionName
accounting = Type acctName
sales      = Type salesName
management = Type mgmtName
```

The program shown above represents a simple class named "Controller" with four member variables ("acct", "sales", "manage", and "current"). The "$->" operator is used to reference member variables of an object. "&=" is standard assignment, and "$." is used to call methods of an object. Controller does not inherit from another class (indicated by "noParent") and has eight public methods. The first is its constructor, which takes no parameters (indicated by the empty list: []) and consists of one block of code. This block contains four statements, each initializing one of the member variables. The first three are initialized to a new object of type Accounting, Sales, and Management, respectively. The last, "current", defaults to the value of "acct". Looking at the list of members variables ("modVars"), "current" is of type Connection. The Connection class is defined elsewhere as the parent of Accounting, Sales, and Management, thus "current" can hold a value of any of those three types.

The next three methods are used to change the state of the "current" member to either Accounting, Sales, or Management (the "oneLiner" shortcut function indicates that a Body of code consists of only one statement). The final three methods call some methods "open", "close", and "log" on the object held by "current".

These methods are defined in the Accounting class as follows.

Listing 8.8: AbstractCode implementation of the Accounting module.

```
acctClass :: Class
acctClass =
    pubClass acctName (extends connectionName) [] [
        pubFunc Void "open" [] $ oneLiner $
            printStrLn $ "open database for Accounting"
        pubFunc Void "close" [] $ oneLiner $
            printStrLn "close the database"
        pubFunc Void "log" [] $ oneLiner $
            printStrLn "log activities"
    ]
```

Inheritance is indicated by the "extends" parameter, in place of "noParent". Sales and Management are defined analogously. Each method simply prints a string to the console.

See the example module for the full, working implementation (module listing in A.2.3), including all class definitions. To generate code for this example, put the option "ExampleImplementation = StateV2" in the Configuration file.

8.5.2 QuickSort

This example implements an in-place version of the well-known sorting algorithm. Specifically, the algorithm described by [Alg] was referenced.

Listing 8.9: AbstractCode implementation of QuickSort.

```
qsClass :: Class
qsClass =
    let list = "arr" `listOf` int
        partition = "partition"
        recQs = "rec_quicksort"
        arr = "arr"
        list = arr `listOf` int
        left = "left"
        right = "right"
        tmp = "tmp"
        i = "i"
        j = "j"
        pivot = "pivot"
        index = "index"
    in pubClass "QuickSort" noParent [] [
        privFunc (typ int) partition
          [param arr (List Dynamic int), param left int, param right
              int] [
            Block [
                varDecDef i int (Var left),
                varDecDef j int (Var right),
                varDec tmp int,
                varDecDef pivot int $
                    list $. ListAccess (((Var left #+ Var right) #/
                        litFloat 2.0) $. Floor $. Cast int)
            ],
            Block [
                while (Var i ?<= Var j) [ Block [
                    while ((list $. at i) ?< Var pivot) $
                        oneLiner $ (&++)i,
                    while ((list $. at j) ?> Var pivot) $
                        oneLiner $ (&~-)j,
                    ifCond [(Var i ?<= Var j,
                        -- then
                        [ Block [
                            tmp &.= (list $. at i),
                            ValState $
                               list $. ListSet (Var i) (list $. at j),
                            ValState $
                               list $. ListSet (Var j) (Var tmp),
```

```
                        (&++)i,
                        (&~-)j
                    ] ] )
                ] noElse
            ] ]
        ],
        Block [ returnVar i ]
    ],
    privFunc Void recQs
      [param arr (List Dynamic int), param left int, param right
          int] [
        Block [
            varDecDef index int $
                Self $. Func partition [list, Var left, Var right
                    ],
            ifCond [(Var left ?< (Var index #- litInt 1),
                oneLiner $ ValState $ Self $. Func recQs [list,
                    Var left, Var index #- litInt 1])]
                noElse,
            ifCond [(Var index ?< Var right,
                oneLiner $ ValState $ Self $. Func recQs [list,
                    Var index, Var right])]
                noElse
        ] ],
    pubFunc Void "quicksort" [param arr (List Dynamic int)] $
        oneLiner $
        ValState $ Self $. Func recQs [list, litInt 0, (list $.
            ListSize) #- litInt 1]
]
```

This implementation might appear complex to a reader who is unfamiliar with AbstractCode. It may be helpful to compare it to the generated output. In Java:

Listing 8.10: Output Java code for the QuickSort module.

```
public class QuickSort {

    private int partition(Vector<Integer> arr, int left, int right) {
        int i = left;
        int j = right;
        int tmp;
        int pivot = arr.get((int)(Math.floor((left + right) / 2.0)));

        while (i <= j) {
            while (arr.get(i) < pivot) {
```

```
                i++;
            }
            while (arr.get(j) > pivot) {
                j = j - 1;
            }
            if (i <= j) {
                tmp = arr.get(i);
                arr.set(i, arr.get(j));
                arr.set(j, tmp);
                i++;
                j = j - 1;
            }
        }

        return i;
    }

    private void rec_quicksort(Vector<Integer> arr, int left, int
        right) {
        int index = partition(arr, left, right);
        if (left < (index - 1)) {
            rec_quicksort(arr, left, index - 1);
        }
        if (index < right) {
            rec_quicksort(arr, index, right);
        }
    }

    public void quicksort(Vector<Integer> arr) {
        rec_quicksort(arr, 0, arr.size() - 1);
    }
}
```

For further reference, the generated Python code:

Listing 8.11: Output Python code for the QuickSort module.

```
class QuickSort:
    def partition(self, arr, left, right):
        i = left
        j = right
        pivot = arr[int(math.floor((left + right) / 2.0))]

        while (i <= j) :
            while (arr[i] < pivot) :
                i = i + 1
            while (arr[j] > pivot) :
```

```
                j = j - 1
            if (i <= j) :
                tmp = arr[i]
                arr[i] = arr[j]
                arr[j] = tmp
                i = i + 1
                j = j - 1

        return i

    def rec_quicksort(self, arr, left, right):
        index = self.partition(arr, left, right)
        if (left < (index - 1)) :
            self.rec_quicksort(arr, left, index - 1)
        if (index < right) :
            self.rec_quicksort(arr, index, right)

    def quicksort(self, arr):
        self.rec_quicksort(arr, 0, len(arr) - 1)
```

The QuickSort module contains a private Partition method (three parameters, the first being a dynamic list of integers, and the others being left and right bounds; returns an integer), a private Quicksort method for performing the recursive calls (same parameters as Partition; void return type), and a public Quicksort method (the only parameter is a list; void return type) for initiating the sorting process.

This example makes use of various comparison operators (beginning with "?") and assignment operators (beginning with "&"). Arithmetic operators (beginning with "#") are also used for the pivot index calculation.

Some notable statements and functions: "varDecDef" declares and initializes a variable, while "varDec" just declares it. "listOf" is used infix to indicate the element type of a ListVar. "litFloat" and "litInt" are used to indicate float and integer literals, respectively. "ifCond" begins an If conditional. "at" is a function called on a ListVar to access an element. "&++" and "&∼-" are incremental and decremental assignment operators. "&.=" is standard assignment, but assumes that the left operand is the label of a Var. "Floor" is the mathetical floor operation, and "Cast" is used for explicit typecasting.

See the example module for the full, working implementation (module listing in A.2.3). To generate code for this example, put the option "ExampleImplementation = QuickSort" in the Configuration file.

It speaks well to the purpose and goals of GOOL that the generated Java code for this example matches the original Java (which the AbstractCode was written against) very closely (other than intended differences, such as how the recursive method was hidden from the module interface).

Chapter 9

Conclusions

The results of this project demonstrate common core of the selected, current OO languages. The fact that several disparate languages with OO elements can be generated equally well from a single, non-trivial, reasonably concise DSL implementation suggests that the DSL must exemplify many patterns that are shared between all languages.

Designing the transformation from AbstractCode to real object-oriented code was not, in fact, particularly difficult. A fair amount of programming effort was certainly required, but the design and decisions involved were mostly straightforward. This, of course, could have been a much more complex task, had the DSL not been designed accordingly.

The most complex aspect of GOOL's development was the design of AbstractCode. The language needed to balance conciseness and simplicity with robustness and sufficient generality to support all of the output languages. At times, this was a challenge. For example, C++ and Objective-C both require explicit destructors for objects. Additionally, the pieces of memory that need to be released in these destructors often differ between the two languages. Therefore, GOOL needed to be able to differentiate between variables which must be explicitly freed in only C++, only Objective-C, both, and neither. Ultimately, the problem was solved by introducing a manually-specified "deletion priority" factor to each member variable of a class — resulting in a small (but still unfortunate) increase in complexity,

yet preserving the generality of the DSL, as well as its expandibility to additional future output languages. There were any number of similar situations throughout GOOL's development, where one of GOOL's qualities had to be prioritized over others.

Fortunately, in many situations, these sorts of decisions could be smoothed over by creating helper functions to abstract over the most common cases. This process proved to be beneficial, as the resulting DSL code can be very simple and flexible, while still providing important language-specific details. In the end, the design decisions made for the DSL meant that implementation of the code generation procedures would be relatively natural.

Many relevant and interesting ideas became apparent during development to support our central goal. For instance, as touched upon in Section 7.4 the Java and C# code-rendering modules use nearly identical implementations. There are a few small syntactic differences, to be sure (e.g. C# must use "Count" for list sizes, where Java uses "size"; Java must use a "String.equals()" method to check for equality in strings, while C# can use the regular == operator). But by and large, the C# and Java "dictionaries" are evidently quite similar (at least up to the level that is considered by our DSL).

In fact, all the language renderers can share some portion of their implementations. Even the most "out-of-place" language that we considered (Lua) shares a sizeable amount of rendering functions. We have, for instance, the Block, Body, and Assignment default rendering functions — all seen in Section 7.3 — shared between all output languages. In part, this is due to the generalized design of the rendering modules. But at a higher level, this can only be possible because of inherent, common patterns between the languages. A short distance from the generic language of AbstractCode implies a small syntactic/semantic measure of "deviation" from a certain central core. This directly supports the point we set out to prove.

The GOOL program is likely not appropriate for use in real-world scenarios. Few realistic use cases exist, and, while many features have been implemented in the language, it is not mature enough to be used for serious, complex applications. GOOL was intended to be a purely academic exercise;

a methodology to search for, discover, and demonstrate a set of patterns. This goal has been achieved to an extent that is more than sufficient; however, a great deal of further development should be carried out to discover just how far the common core extends.

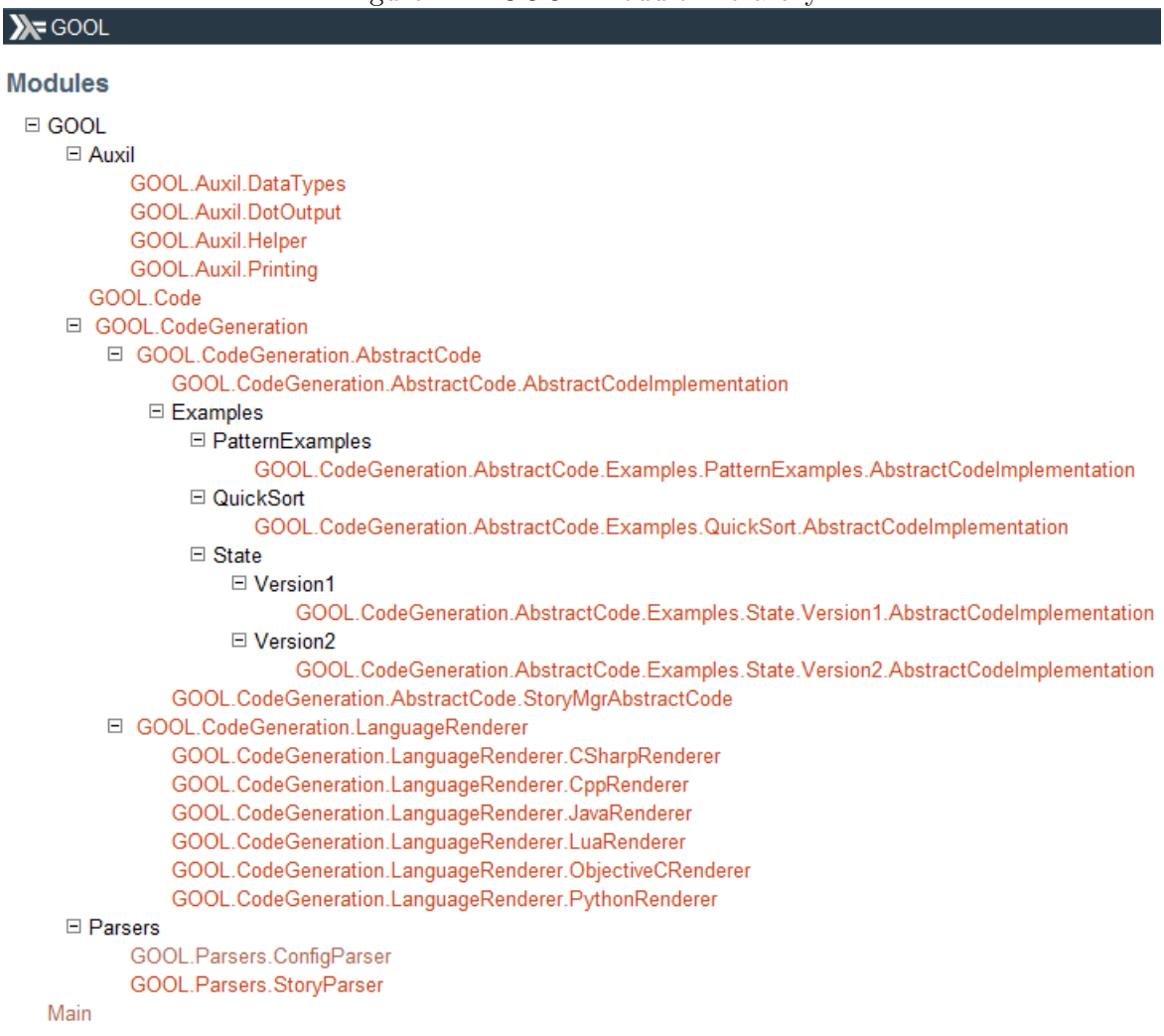
Appendix A

Module Information

A.1 Module Hierarchy

The following module hierarchy was produced by Haddock, a tool used to automatically generate documentation from specially annotated Haskell source code [MW10].

Figure A.1: GOOL Module Hierarchy



A.2 High-level Module Descriptions

A.2.1 GOOL.Auxil

GOOL.Auxil.DataTypes

Defines some data types that are used globally.

GOOL.Auxil.DotOutput

This is a SAGA-specific module (see Section 8.1). It is not used in GOOL's Generic Mode. Since the Story Mode functionality is out of scope for this report, such modules will not be discussed here.

GOOL.Auxil.Helper

Contains several helper functions for formatting output, typically used by the renderer modules. This includes, for example, functions for transforming arbitrary strings into valid variable names, and for handling lists of renderable AbstractCode elements.

GOOL.Auxil.Printing

Contains various useful functions for printing which are not provided by the pretty-printing library.

A.2.2 GOOL.Code

GOOL.Code

Defines the Code data type. A Code structure wraps generated code documents together with their expected file names.

A.2.3 GOOL.CodeGeneration

GOOL.CodeGeneration

Implements the high-level process of producing a Code structure from an AbstractCode, and creating the output files.

GOOL.CodeGeneration.AbstractCode

Defines the structure of the AbstractCode datatype, and all of the components of the AbstractCode language. Also contains many shortcut functions and symbolic operators which serve to replace cumbersome and/or common patterns in AbstractCode programming.

GOOL.CodeGeneration.AbstractCode.AbstractCodeImplementation

Contains a program written in AbstractCode. This is the 'default' implementation; i.e. if GOOL is not instructed to render a specific pre-existing example, this is the implementation that will be rendered.

GOOL.CodeGeneration.AbstractCode.Examples.PatternExamples.AbstractCodeImplementation

Contains one of the example AbstractCode implementations. This example simply demonstrates the functionality of some of the "PatternState" statements of AbstractCode. These statements implement some of the well-known "Gang of Four" OO design patterns at a high level.

GOOL.CodeGeneration.AbstractCode.Examples.QuickSort.AbstractCodeImplementation

Contains one of the example AbstractCode implementations. This example implements QuickSort, and performs a few test sorts.

GOOL.CodeGeneration.AbstractCode.Examples.State.Version1.AbstractCodeImplementation

Contains one of the example AbstractCode implementations. This example demonstrates a simple State Machine program.

GOOL.CodeGeneration.AbstractCode.Examples.State.Version2.AbstractCodeImplementation

Contains one of the example AbstractCode implementations. This example demonstrates the same State Machine program as in Version 1, but uses inheritance to make the implementation more concise.

GOOL.CodeGeneration.AbstractCode.StoryMgrAbstractCode

This is a SAGA-specific module.

GOOL.CodeGeneration.AbstractCode.LanguageRenderer

Defines the structure of the Config datatype, which is meant to contain all rendering functions for a specific language, and thus acts as an explicit

dictionary for the language. This module also contains a set of default implementations for the rendering functions.

GOOL.CodeGeneration.AbstractCode.LanguageRenderer.CSharpRenderer

Contains the logic to render C# code from an AbstractCode program.

GOOL.CodeGeneration.AbstractCode.LanguageRenderer.CppRenderer

Contains the logic to render C++ code from an AbstractCode program.

GOOL.CodeGeneration.AbstractCode.LanguageRenderer.JavaRenderer

Contains the logic to render Java code from an AbstractCode program.

GOOL.CodeGeneration.AbstractCode.LanguageRenderer.LuaRenderer

Contains the logic to render Lua code from an AbstractCode program.

GOOL.CodeGeneration.AbstractCode.LanguageRenderer.ObjectiveCRenderer

Contains the logic to render Objective-C code from an AbstractCode program.

GOOL.CodeGeneration.AbstractCode.LanguageRenderer.PythonRenderer

Contains the logic to render Python code from an AbstractCode program.

A.2.4 GOOL.Parsers

GOOL.Parsers.ConfigParser

Defines the parser for the GOOL configuration file. See Section 7.5 for details.

GOOL.Parsers.StoryParser

This is a SAGA-specific module.

A.2.5 Main

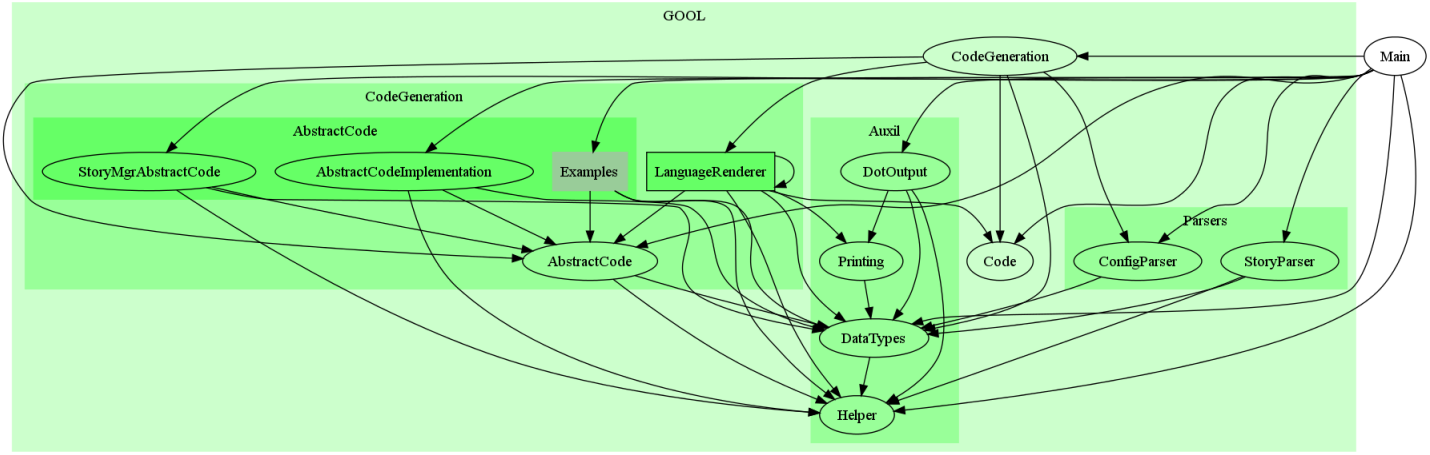
The entry point for execution of GOOL. This module is responsible initiates and directs the I/O processes: reading and parsing the configuration file, code generation, and printing any operational feedback for the user.

A.3 Module Dependency Diagram

Dependencies between the modules of GOOL are illustrated below. The graph was produced by graphmod, a tool used to visually represent module dependencies of Haskell programs [Dia10].

Note that the LanguageRenderer and Examples groups have been collapsed for clarity.

Figure A.2: GOOL Module Dependency Diagram



A.4 Configuration File Syntax

Listing A.1: EBNF Grammar of the Configuration File

```
config = opt_whitespace, 'Generation', whitespace, 'Language',
         whitespace, '=', whitespace, language, [ eximp ],
         [ javalist ], [ cpplist ], [ objclist ],
         opt_whitespace ;
```

```
language = 'C#' | 'C++' | 'Java' | 'Objective-C'
         | 'Python' | 'Lua' ;

whitespace = whitespace_char, { whitespace_char } ;
whitespace_char = ? any white space character ? ;
opt_whitespace = [ whitespace ] ;

eximp = whitespace, 'ExampleImplementation', whitespace, '=',
        whitespace, example ;
javalist = whitespace, 'JavaListType', whitespace, '=',
           whitespace, javalistype ;
cpplist = whitespace, 'CppListType', whitespace, '=',
          whitespace, cpplistype ;
objclist = whitespace, 'ObjCStaticListType', whitespace, '=',
           whitespace, objclisttype ;
example = 'Patterns' | 'QuickSort' | 'State' | 'StateV2' ;
javalisttype = 'ArrayList' | 'LinkedList' | 'Vector' ;
cpplisttype = 'deque' | 'vector' ;
objclisttype = 'NSArray' | 'NSMutableArray' ;
```

A.5 AbstractCode Language Definition

Listing A.2: Haskell Definition of the AbstractCode DSL

```
data AbstractCode = AbsCode Package
data Package = Pack Label [Class]
data Class = Class {
                classname :: Label,
                parentName :: Maybe Label,
                classScope :: Scope,
                classVars :: [StateVar],
                classMethods :: [Method]}
           | Enum {
                classname :: Label,
                classScope :: Scope,
                enumElements :: [Label]}
           | MainClass {
                classname :: Label,
                classVars :: [StateVar],
                classMethods :: [Method]}
```

```
data StateVar = StateVar Label Scope StateType Int
data Method =
        Method Label Scope MethodType [Parameter] Body
      | GetMethod Label MethodType
      | SetMethod Label Parameter
      | MainMethod Body
data Scope = Private | Public
data Parameter = StateParam Label StateType
               | FuncParam Label MethodType [Parameter]
type Body = [Block]
data Block = Block [Statement]
data Statement = AssignState Assignment
               | DeclState Declaration
               | CondState Conditional
               | IterState Iteration
               | JumpState Jump
               | RetState Return
               | ValState Value
               | CommentState Comment
               | FreeState Value
               | PrintState {newLine :: Bool, valType :: StateType,
                   printVal :: Value}
               | ExceptState Exception
               | PatternState Pattern
data Assignment = Assign Value Value
                | PlusEquals Value Value
                | PlusPlus Value
data Declaration = VarDec Label StateType
                 | ListDec ListType Label StateType Int
                 | ListDecValues ListType Label StateType [Value]
                 | VarDecDef Label StateType Value
                 | ObjDecDef Label StateType Value
                 | ConstDecDef Label Literal
data Conditional = If [(Value, Body)] Body
                 | Switch Value [(Literal, Body)] Body
data Iteration =
        For {initState :: Statement, guard :: Value, update ::
            Statement, forBody :: Body}
      | ForEach Label Value Body
      | While Value Body
data Jump = Break | Continue
data Exception = Throw {excMsg :: String}
               | TryCatch {tryBody :: Body, catchBody :: Body}
data Return = Ret Value
data Function = Func {funcName :: Label, funcParams :: [Value]}
              | Cast StateType
              | Get Label
```

```
              | Set Label Value
              | IndexOf Value
              | ListSize
              | ListAccess Value
              | ListAdd {index :: Value, addVal :: Value}
              | ListSet {index :: Value, setVal :: Value}
              | ListPopulate Value StateType
              | IterBegin | IterEnd
              | Floor | Ceiling
data MethodType = MState StateType
                | Void
                | Construct Label
data StateType = List ListType StateType
               | Base BaseType
               | Iterator StateType
               | EnumType Label
               | Type Label
data ListType = Static | Dynamic
data BaseType = Boolean | Integer | Float | Character | String
data Value = EnumElement {enumName :: Label, elemName :: Label}
           | EnumVar Label
           | Expr Expression
           | FuncApp Label [Value]
           | Lit Literal
           | ObjAccess Value Function
           | StateObj StateType [Value]
           | Self
           | Var Label
           | ObjVar Value Value
           | ListVar Label StateType
           | Const Label
           | Arg Int
           | Input
data Literal = LitBool Bool
             | LitInt Int
             | LitFloat Float
             | LitChar Char
             | LitStr String
data Expression = UnaryExpr UnaryOp Value
                | BinaryExpr Value BinaryOp Value
data UnaryOp = Negate | SquareRoot | Abs
             | Not
data BinaryOp = Equal | NotEqual
              | Greater | GreaterEqual
              | Less | LessEqual
              | Plus | Minus
              | Multiply | Divide
```

```
              | Power | Modulo
data Pattern = State StatePattern
             | Strategy StratPattern
             | Observer ObserverPattern
data StatePattern = InitState {fsmName :: Label, initialState :: Label
    }
                  | ChangeState {fsmName :: Label, toState :: Label}
                  | CheckState {fsmName :: Label, cases :: [(Label,
                      Body)], defaultBody :: Body}
data StratPattern = RunStrategy {stratName :: Label, strategies ::
    Strategies, assignResultTo :: Maybe Value}
data Strategies = Strats {strats :: [(Label, Body)], returnVal ::
    Maybe Value}
data ObserverPattern = InitObserverList {observerType :: StateType,
    observers :: [Value]}
                     | AddObserver {observerType :: StateType,
                         observer :: Value}
                     | NotifyObservers {observerType :: StateType,
                         receiveFunc :: Label, notifyParams :: [Value
                         ]}
```

Bibliography

[Alg] Algolist. Algorithms and Data Structures - Quicksort. `http://www.algolist.net/Algorithms/Sorting/Quicksort`.

[BC11] Lucas Beyak and Jacques Carette. SAGA: A DSL for Story Management. In *DSL 2011: IFIP Working Conference on Domain-Specific Languages*, volume 66, pages 48–67, September 2011.

[Bey11] Lucas Beyak. SAGA: A Story Scripting Tool for Video Game Development. Technical report, McMaster University - Department of Computing and Software, April 2011.

[Cla00] Robert G. Clark. *Comparative Programming Languages*. Addison Wesley, 3rd edition, November 2000.

[Dia10] Iavor S. Diatchki. HackageDB: graphmod-1.2.3. `http://hackage.haskell.org/package/graphmod`, August 2010.

[Dro09] Sadek Drobi. Lennart Augustsson on DSLs Written in Haskell. `http://www.infoq.com/interviews/DSL-Haskell-Lennart-Augustsson`, February 2009.

[FWH01] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press, 2nd edition, January 2001.

[Ier96] Roberto Ierusalimschy. *Programming in Lua*. Lua.org, 2nd edition, March 1996.

[Inc00] Sun Microsystems Inc. Java Language Specification, Second Edition - Expressions. http://java.sun.com/docs/books/jls/second_edition/html/expressions.doc.html, 2000.

[Inc11] Apple Inc. The Objective-C Programming Language. https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html, October 2011.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3), May 2001.

[Jav08] Javacamp. Java Design Patterns At a Glance. http://www.javacamp.org/designPattern/, September 2008.

[Lip11] Miran Lipovača. *Learn You a Haskell for Great Good!* No Starch Press, April 2011.

[Lit11] Steve Litt. Lua OOP. http://www.troubleshooters.com/codecorn/lua/luaoop.htm, 2011.

[LLC11] DedaSys LLC. Programming Language Popularity. http://langpop.com/, April 2011.

[Mig] Matt Might. 7 lines of code, 3 minutes: Implement a programming language from scratch. http://matt.might.net/articles/implementing-a-programming-language/.

[Mit96] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.

[MSD12] MSDN. Classes (C# Programming Guide). http://msdn.microsoft.com/en-us/library/x9afc042.aspx, 2012.

[MW10] Simon Marlow and David Waern. Haddock User Guide. http://www.haskell.org/haddock/doc/html/index.html, 2010.

[Net11] The C++ Resources Network. Structure of a Program. http://www.cplusplus.com/doc/tutorial/program_structure/, 2011.

[Ove11] Stack Overflow. NSNumber numberWithInt crashing on numbers >= 13. http://stackoverflow.com/questions/7165735/nsnumber-numberwithint-crashing-on-numbers-13, August 2011.

[Sco09] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 3rd edition, April 2009.

[Seb01] Robert W. Sebesta. *Concepts of Programming Languages*. Addison Wesley, 5th edition, July 2001.

[Sla08] Mark Slagell. Ruby User's Guide - Methods. http://www.rubyist.net/~slagell/ruby/methods.html, 2008.

[Ter12] David Terei. HackageDB: Text.PrettyPrint.HughesPJ. http://hackage.haskell.org/packages/archive/pretty/latest/doc/html/Text-PrettyPrint-HughesPJ.html, 2012.

[TN02] Allen Tucker and Robert Noonan. *Programming Languages: Principles and Paradigms*. McGraw-Hill, 2002.

[web11] The Haskell Programming Language. http://www.haskell.org/haskellwiki/Haskell, July 2011.