University Of Bergen

C♭ : An Intermediate Representation Language for the Purpose of Software Migration to Java and C♯

Hussam Yousif

June 1st 2018

Master Thesis

raincode LABS

compiler experts

Supervised by
Anya Helene Bagge
Johan Fabry
Vadim Zaytsev

Abstract

In this thesis we will present the design and implementation of an intermediate representation language created for the purpose of software migration to Java and C♯. Furthermore we will examine a set of transformations performed on programs of this language, followed by the discussion of a set of programs which perform semantic analysis. Finally we will examine the testing framework built around this language. The result of this thesis is a language built as much as feasible within the intersection of Java and C♯. Other results include the aforementioned program transformations, semantic analysis and testing framework.

ACKNOWLEDGMENT

I want to thank my parents for their continued support. I'd also like to thank the three key people in this thesis: Anya Helene Bagge, Johan Fabry and Vadim Zaytsev. I'd like to thank them as mentors and as friends. I am truly grateful for everything, I enjoyed every and each conversation we shared. Furthermore I'd like to thank everyone at Raincode for being so friendly and helpful. Finally I'd like to thank everyone at the University of Bergen's Algorithms group.

To my loving parents and Arsène Wenger.

CONTENTS

CHAPTER

ONE

INTRODUCTION

Throughout this thesis we will explore the domain of meta programming, source to source conversion, program transformation and language design, testing and implementation. We will do all of that by exploring the C♭ project. This chapter will introduce the problem, our main tool and the solution. Furthermore we will ask a question which will be answered thoughout this thesis.

Every so often a box like this will appear. These are mainly used for discussion of design decisions, clarifications or to give some further context into the topic at hand.

This thesis assumes an understanding of imperative and object-oriented programming on the part of the reader.

1.1 Legacy software

As technologies advance in computer science the idea of migrating old software to newer technologies will be an integral part of the future. For instance COBOL is still in wide use in the industry [8]. Finding developers willing and capable of programming in these old languages may prove to be quite the challenge. Due to such reasons it may be desirable for corporations to migrate their software from some old language to a more modern language. However these systems are often quite large. Rewriting from scratch may simply not be feasible due to the sheer amount of work required. An alternative is needed, a way to migrate old software into newer technologies without manually re-writing the code.

1.2 Program transformation

Ideally we would like to migrate software from older technologies to newer ones via the press of a button. Programmers are lazy, rewriting a program that already exists is... *boring!* Thus we would like to create a program which does the rewriting for us!

The natural tool of choice is program transformation. Program transformation is a metaprogramming technique where some transformation is applied to source code such that a new program is generated. Program transformation is nothing new, compilers use it when optimizing programs. In fact, compiler optimization of programs is an interesting example of program transformation due to the requirement that the input program and the output program contain the same behavior.

In this project we are interested in preserving the result of a program. Suppose you have a program p. If we migrate p to some other language via program transformation producing p' we are interested in preserving the result of the execution of p. Meaning that the execution of p and p' should yield the same output on the same input.

Our first task is to design a platform which can transform programs to some modern language. We want to use program transformation for this platform. Therefore we wish to create what we call the serializer, also often called a transpiler. Those are programs that map source code to source code where the output program is in a different programming language. We will consider other program transformations than serializers, but the serializer is one of the central parts of this thesis.

Finally we should consider the code generated from program transformations. Generated code is usually of low quality. This is one of the weaknesses of program transformations. In our case, C♭ is a practical tool which is meant for the industry, thus we would want to minimize bad code. Perhaps we can use program transformations to improve the code quality?

Question 1. *Can we use program transformation to improve the quality of the serializer output?*

1.3 Observation: Java and C♯ are similar

Java and C♯ are two of the more popular languages in the industry [26], furthermore a key observation can be made about the two languages. They are very similar, at least at first sight. Thus the question arise:

Question 2. *Is it feasible to design and implement a language entirely contained in the intersection of Java and C♯?*

If we could design a language around the intersection of Java and C♯ then we could use that language to serialize to Java and C♯. This language needs to be modern and advanced as it is going to be used in the industry. The language should support object-oriented programming as it is the main programming paradigm of C♯ and Java. Furthermore it should contain a standard library such that we can transform uses of standard library in the input program to reasonably equivalent behavior in the output program. This standard library should be designed in such a way that it is easy to extend. We don't know what sort of software we are going to transform, or even what language. Therefore we should be able to adapt our intersection language to the functionality of the input program.

1.4 C♭

The conclusion was an intermediate representation language as the foundation. By foundation we are referring to a framework which our serialization process will be built around. We call our language C♭. C♭ cannot be compiled directly, but uses the aforementioned serializer to transform a program to Java or C♯. The compiler of the target language is then used in order to execute the program.

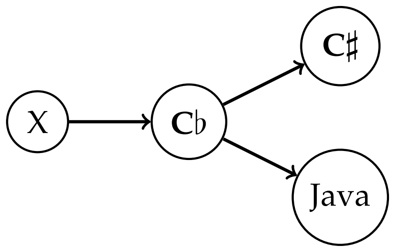


Figure 1.1: Serializing a program form language X to Java or C♯

After fully developing C♭ we should have the platform to transform programs to other languages. Suppose we are given a PL/I program, the use case should then be the following:

We're given a program in PL/I which we wish to migrate to Java or C♯. First we create a serializer from PL/I to C♭. A serializer which targets C♭ needs to be developed for every source language. After acquiring a serializer which targets C♭ we should generate a C♭ program from the PL/I program. From this point on, the rest of the work should be done by the press of a button using the C♭ to Java or C♯ serializer.

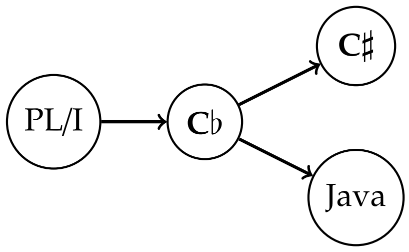


Figure 1.2: A client wants to migrate a PL/I program to Java or C♯

Before that, we need to design C♭. We want to map constructs from C♭ into constructs in Java and C♯. The easiest way to do that is by designing C♭ such that for every construct in C♭, there is an equivalent construct in Java and C♯. Thus, C♭ should be contained in the *intersection* of Java and C♯.

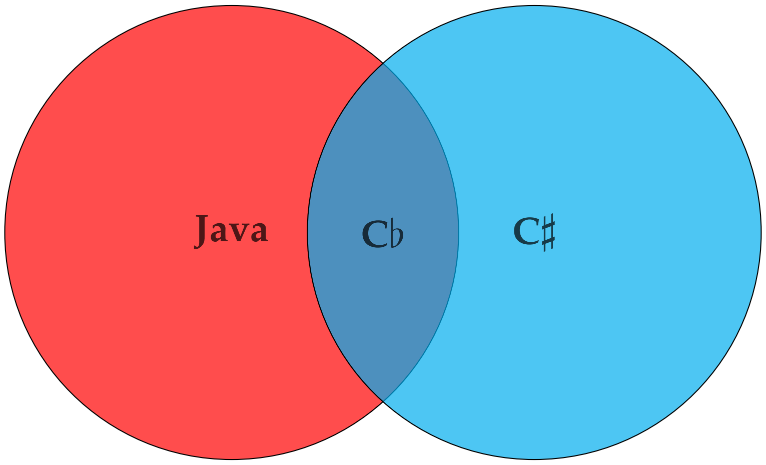


Figure 1.3: C♭ lives in the intersection of C♯ and Java

Previously we mentioned that the output of generated source code may not be of adequate quality. Therefore we need to create automated procedures which uses program transformations to refactor code. These will be discussed further in Chapter 6.

Although we are not writing compilers, we are still in the domain of compilation. A great book [1] once represented the complexities of writing compilers as a dragon. If compilers are indeed our dragons then we will need to be careful and use powerful, almost magical tools and techniques to handle such complexity. Throughout this thesis we will present language design and implementation, program transformation, semantic analysis and finally program testing. This is how we managed to cope with complexities that seem overwhelming (and fire breathing) at first sight!

1.5 Contributions

In this thesis, we make the following contributions:

- A study of the semantic *intersection* of Java and C♯ and the definition of C♭ based on this (Chapter 3).
- The design and implementation of a *serializer* that translates C♭ to Java and C♯ (Chapter 4).

- The definition of static semantic constraints for C♭ (Chapter 3), and an analysis tool to detect violations (Chapter 5).
- The design and implementation of a number of C♭ code transformations (Chapter 6).
- The design and implementation of a testing framework for C♭ and serialized code (Chapter 7).

1.6 Final Remarks

1.6.1 Raincode Labs

Raincode Labs is one of the largest independent compiler companies in the world. I had the privilege of working with them in the C♭ project. Our mission was to restart the C♭ project, and try to make it a viable product provided in the catalogue of Raincode Labs. Raincode Labs uses a number of languages, the relevant ones for this project are RCScript and Yafl[4] .

1.6.2 RCScript

RCScript is a procedural imperative language used with the purpose of building self contained scripts for the purpose of metaprogramming. It was the perfect tool for us as it provided excellent tree traversal functionality. Furthermore Yafl and RCScript works extremely well together, the Yafl parser generates the abstract syntax tree of the input C♭ program, then this ast is passed to the RCScript programs. In chapter Chapter 6 we will examine a program called the pipeline which will be the program which the ast is passed to.

1.6.3 The C♭ Project

C♭ is not a new project, in fact it was started six to seven years ago. However it was only in development for a short time with a small team of developers and it was abandoned quickly. When we revived the project we found a repository containing a C♭ grammar, a C♭ parser and a serializer to C♯. During the revival of the project we rewrote the serializer from scratch, redesigning the program. Furthermore the parser and grammar has been modified.

1.6.4 Parser

The parser was the one aspect of this project which I did not help implement. I gave a large amount of input on the design of the parser, however I was never the one implementing that input. Therefore there will be no discussion of the parser in this thesis, we will discuss the design of C♭ but not the implementation of the parser. The parser was written in Yafl.

1.6.5 Serializer

Earlier we explained the word "serializer". The serializer will refer to a program which translates a program from a source language to a target language. Such programs are often called compilers, transpilers, convertors, exporters and translators. However we will refer to them as serializers.

Figure 1.4: The Babylonian God Marduk conquering his complexities in ancient Mesopotamia.

CHAPTER TWO

RELATED WORK

In this chapter we will examine previous work which deal with source to source serialization and other concepts that are relevant to C♭.

2.1 GOOL: A Generic OO Language

A project which is particularly interesting to us is GOOL (Costabile [9]). It's a project where the author used a tool called SAGA, in order to show that there exists "a common core" contained between different object oriented languages. This common core refers to a set of features which OO languages share.

GOOL is interesting for several reasons, among them is the fact that they also use program transformation for serialization between languages. GOOL supports serialization to C♯, C++, Java, Objective-C, Python and LUA. The authors of GOOL expressed the choice of LUA as a means to provide proof that a language as different as LUA can contain the object oriented core they specified.

2.1.1 Goals of GOOL

The goal, as stated earlier is to show that there exists a common core for object oriented languages. They examine this common core by creating serializers targeting a set of object oriented languages.

This goal, greatly differ from our own, the core intentions are simply not the same. They state that this common core is not the intersection of these languages but a set of features [9, p. 14]. The authors never mention software migration as a motivation therefore we're lead to believe that the motivation is academic, in contrast with our motivation which is to create an industrial solution for software migration.

2.1.2 Methodologies of GOOL

Their methodologies was fairly straight forward. In order to provide a common core between object oriented programming languages, they had to define what that common core was. They considered representations for the common OO core via patterns found among OO languages [9, p .16]. These paterns include conditionals, loops, variables, and so on.

After finding the patterns, they designed a new generic OO language around these patterns and then proceeded to create a serializer for each target language.

They created GOOL by modifying and expanding a tool called SAGA [3] which is a domain specific language written in Haskell. SAGA (Story as an Acyclic Graph Assembly)is written For the purpose of Video game development, thus it had to be heavily modified. The advantage of SAGA was that it represented source code as algebraic data types and contained code generation capabilities to Java, C♯ and C++.

Their serializer design was similar to ours. They use an abstract syntax tree of the input program, traversed it and then use a dictionary in order to serialize the different constructs of the language. Similar to C♭, each target language used it's own dictionary in order to serialize every terminal and nonterminal.

2.1.3 GOOL Challenge: Memory management

Unlike C♭ where memory management is done via the garbage collector of the target language, GOOL had to provide their own object deletion semantics for memory management to the C++ and Objective-C serializers. The Objective-C serializer seem to be particularly challenging as they couldn't mimic the C++ serializer in it's creation due the difference in memory management semantics.

The solution was to create a “deletion priority” parameter in the config file specifying the target language. This deletion priority parameter would tailor the memory management to the ouput language.

funnily enough, the LUA serializer seem to have been fairly easy to implement due to previous work being done on implementing OO concept in LUA.

2.1.4 C♭ vs GOOL

Both GOOL and C♭ are shaped by their motivation. GOOL was motivated by trying to prove the existence of the “common core” between different object oriented languages. Where as we wanted to create a solution for software migration.

This difference is reflected in the entire thesis. The C♭ project was based on the intersection of Java and C♯. Although we designed C♭, we didn’t have as much freedom as one would imagine. The semantics in Java and C♯ was the foundation of the semantics of C♭. If the semantics in the target languages were equivalent for a construct, then we would simply use those semantics for that construct. Otherwise we would omit the construct or create the construct in some way. Further discussion of the C♭ design in chapter three.

GOOL on the other hand is presenting a set of constructs shared among the OOP languages. Thus they do not define ranges on datatypes or how the type system in the target language affect the semantics. For us reference types versus value type was an important issue, in GOOL that such matters are simply not the focus.

Yet for all the differences, the similarities stand. Both are languages which use program transformations in order to serialize code from one language to some other target language. Even though the purpose were different, GOOL and C♭ still contained a central idea in common which is the serializer, a “common core”, one could say.

2.2 The Realities of Language Conversions

Terekhov and Verhoef [27] discuss the difficulties of language to language conversions i.e, automatic source code migration via a serializer.

2.2.1 Pitfalls

A pitfall of software migration via serialization listed is the transformation of data types [27, p.114]. The authors examine problems with the size and range of datatypes in the context of serialization. That is an aspect which we did not have to consider due to the semantic similarities between Java and C♯. For instance we did expect floating points types to be an issue, however the fact that both target languages follow the IEEE 754 standard [13] simplified the definition of the float and double types [10, §4.2.3], [19, §floating-point-types]. On the other hand enumerated types are something which we wished to keep in C♭, however the semantics in the target language were different thus we did not add them.

Furthermore the authors argue that the semantic differences may simply be too large thus forcing the programmer to emulate the source language in the target language. The example they provide is the picture clause in a Cobol programmed as a Java class. They argue that such an approach can lead to further issues as one needs to create further constructs to manipulate the previously emulated construct. In the example with picture clause they use arithmetic operators as the constructs which needs to be implemented.

2.2.2 Advice

Terekhov and Verhoef provide advice on how to proceed with source to source conversion, including how one should make a listing of every construct in the source language and creating a planned strategy for the transformation of each construct to the target language.

During the development of C♭ we did that, however we had the privilege of being able to design the source language (C♭). Thus we could design the source language to be as semantically equivalent to the target languages as possible. We were extremely meticulous in the design of C♭ by examining multiple parts of the Java and C♯ language specifications over and over again to make certain that the semantics we specified for the C♭ construct would be correct for our purposes.

They also advice making the conversion as automatic as possible. This was one of our goals, yet we could not completely do that. We will discuss how a subset of the standard libraries was not contained in the intersection between Java and C♯ and thus required wrappers around them. That aside, we did succeed for the most part, especially at the serializer module. However I can not stress enough how massive of an advantage we had by being able to design C♭.

They provide further advice: Maintenance of the serializer, maintenance of the output program, serialization time and execution time of the serialized program and whether or not the tests of the original program is carried on. None of them are particularly relevant to the C♭ project, with the exception of the tests being serialized to the target language. However that decision has to be taken during the development of the front-end serializer which considers C♭ as the target language.

2.2.3 Conclusion

The purpose of the article is the discussion of the difficulties of serialization between languages. Their conclusion is predictably, that source to source serialization is difficult. They provide good points on why it is difficult and discuss the serialization of certain concepts within the language which may provide problems in the serialization process such as data types.

We had a massive advantage in the C♭ project. For one we are the ones designing the source language and we chose two semantically very similar languages as the target languages. Thus we had quite a large intersection and did not need to spend much time emulating constructs of a language. However it is still possible that their advice and warnings will be relevant to the construction of the serializer which targets C♭ as the target language.

2.3 Other Notable Work

In this section we will present other notable work relevant to C♭.

2.3.1 On source-to-source Compilers

On source-to-source compilers (Illushin and Namiot [14]) is another article which discusses principles of serializers. The more relevant points to C♭ are the requirements for the serializer: The input and output program should be semantically equivalent, The output program should be as correct as possible and the serialization process should be as automatic as possible. Furthermore they provide types of serialization: language to language serialization, serialization to another version, automatic parallelization and source code optimization. Only language to language serialization and source code optimization are relevant to the C♭ project.

2.3.2 Code Migration Through Transformation: An Experience Report

Kontogiannis et al. [16] report on their experiment of migrating a software system written in the PL/IX language to C++.

Similar to C♭, the authors represented the source code as an abstract syntax tree. Furthermore their serializer also inventories the constructs in the source language and then map them to some construct in the target language. Some constructs in the source language did not contain immediate mappings to ones in the target language. To remedy that issue, they created libraries and utilities for the execution of the serialized program [16, p .4]. These constructs include Array Classes and memory allocation mechanisms.

The serializer is similar to the C♭ serializer, in fact the serialization framework used in this project, GOOL and C♭ are very similar. It is natural to simply traverse an abstract syntax tree then, for every construct which contain a mapping to the target language, a serialization function is invoked. Furthermore they also match on the type of construct. For instance when a statement is found during the traversal of the ast they find the type of the statement by pattern matching. We do the same in our traverse procedure (read more in Chapter 4).

2.3.3 Others

Stratego/XT (Visser [28]) is a framework built for the purpose of development of transformation software. Stratego itself is a language which focuses on program transformation and XT is a set of transformation tools.

Another classic work within source-to-source serialization is the work of Boyle [6], where numerical software written in pure functional Lisp is translated to high-performance Fortran. A newer approach for Lisp-like languages is that of Hasu and Flatt [11], where the macro expansion facility is used to perform the serialization.

In the PIPS project [15], the goal is detecting the maximum level of parallelism in sequential program. The source language is Fortran77 and the target language is Fortran90 or Fortran77. PIPS is motivated by source program parallelization. There are a few other source-to-source serializers which are motivated by parallelization by the use of program transformation such as POLARIS (Blume et al. [5]).

2.4 Conclusion

Program transformation is a massive field, with a lot of work being done. Furthermore, the work being done does not necessarily have to contain legacy languages or legacy code; there is an ever growing list of languages that compile to JavaScript [2].

Having said that, to the best of my ability I did not manage to find anyone who uses an intermediate representation language in order migrate programs from one language to another. The closest we got to that was GOOL where they used a domain specific language in order to serialize programs to other OOP languages.

On the other hand there is a common pattern among the serializers. They are built by having a procedure traverse the abstract syntax tree, once the traversal finds a construct which is to be serialized, that construct is then passed to a serialization function.

CHAPTER
THREE

INTERSECTION OF JAVA AND C♯

C♭ is designed to be easily mappable to java and C♯ . Therefore it is extremely important to design C♭ in such a manner that we can capture the intersection of java and C♭ . Fortunately, C♯ and Java shares many similarities, however there are subtle differences which we will get into which made the design more difficult than anticipated. We will also discuss the elements of the language that were not part of the intersection and the problems caused by said elements.

These sections are for the discussion and contextualiziation of some of the topics. The specifications are designed to be as compact and specific as possible. A distinction of the actual specifications and discussion of the specifications is designed not to confuse the reader.

3.1 The C♭ language specifications

In this section we will define the language specifications for C♭ . This is one of the more challenging parts of the C♭ project, we had to be precise and ensure the correct semantics. The semantics had to be defined in terms of Java and C♯ semantics, ideally we want the semantics of Java and C♯ to be equivalent, such that C♭ contains the same semantics to the target language and which would make the serialization trivial. As this wasn't always the case we had to get somewhat creative with our solutions.

3.1.1 Types

All types in C♭ can be categorized as either Value types (referred to as primitives in Java) or reference types.

Value Types

Value Types, also known as primitive types are the simplest types within C♭. The interesting aspect of value types are that they can not be of null value and that they are passed by value, that is a copy of the value is passed.

Type	Default Value	(Range) From	(Range) To
`BYTE`	0	-128	127
`SHORT`	0	-32768	32767
`INT`	0	-2147483648	2147483647
`LONG`	0	-9223372036854775808	9223372036854775807
`CHAR`		0	65535
`FLOAT`	0	-128	127
`DOUBLE`	0	-128	127
`BOOLEAN`	`FALSE`	`FALSE`	`TRUE`

Default values are the same as in C♯ [19, §default-values] and Java [10, §4.12.5].

`FLOAT` and `DOUBLE` are subject to the IEEE754 standard [13] in both C♯[19, §floating-point-types] and Java [10, §4.2.3], with the same notation for literals.

`CHAR` is considered an integral type. Furthermore In Java character can only represent UTF-16 code units [10, §3.10.4] . Where as in C♯ character literals represent unsigned 16-bit[19, §CsTypes].

The C♯ type `sbyte` [19, §integral-types] corresponds to Java's `byte` [10, §4.2.1] type which is the type C♭'s `BYTE` is based upon, therefore in the serialization from C♭ to C♯ the type `BYTE` is serialized to `sbyte`

Serialization to Java The default value for char in Java is the null character:

`\u0000`

[10, §4.12.5].

Serialization to C♯ The default value for char in C♯ is:

```
\x0000
```

[19, §default-values]

When referring to value types not containing the null value we should make it very clear that it is possible for C♯ to define nullable value types [19, §nullable-types], however Java does not contain nullable types primitives/-value types.

Reference Types

The following are all reference types in C♭, C♯ [19, §reference-types] and Java [10, §4.3]

- Class types
- Array types
- Parameterized types

Enumerations (`enum`) are a type that we wished for in C♭. However implementing enumerations contradict our philosophy of C♭ being the intersection of Java and C♯. Enumerated types in Java are class types, like other class types they are passed by reference and inherit from object. C♯ on the other hand uses enumerated types as value types and thus differ in semantics from Java's types.

Another type we considered for C♭ are interfaces (`interface`). Interfaces are contained both in Java and in C♯. In the real world we expect the C♭ code we are given to be machine generated and we did not include a transformation into interface types within our plans. Furthermore the line between class and interface is starting to blur with default methods in interface and abstract methods.

The String type

String are class types, the string objects are immutable objects and string literals are instances of a string class. Cf. Java [10, §4.3.3], C♯ [19, §the-string-type].

The Object type

The class hierarchy in C♭ contains a root class called Object. All classes inherit from the Object type, with a single rooted class hierarchy as in Java [10, §4.3.2], C♯ [19, §system.object(v=vs.110).aspx].

3.1.2 Generics

A type is said to be *generic* if it contains one or more type parameters in the type declaration. A type parameter is an identifier included in the type declaration. C♭ supports generic types in a similar way to Java [10, §8.1.2] and C♯ [19, §type-parameters]. However, C♭ does not allow for intersection types [10, §4.9] or any other form of constraint on type parameters [19, §type-parameter-constraints].

Type variables are not permitted to be value types, thus are only allowed to be reference types, as in Java [10, §4.4], and unlike C♯, which allows value types as arguments [19, §type-arguments].

Generic Types Semantic Errors

- Type parameters are not permitted to be declared in a static members of a generic class or any nested class of the generic class.

- A generic class is not permitted to be a direct or indirect child class of the SYS#Exception class.

Generics was one of those areas we just knew would be a difficult construct. Java uses *type erasure* [7], where generic types can also be used as 'raw types' with all parameters replaced by the upper bound on the parameter (typically Object). Both languages allow for restrictions on type parameters. Due to type erasure Java does not permit the usage of value types as type parameters; fortunately auto-boxing (automatic conversion between value and referene types) makes this reasonably convenient in practice. That constraint does not apply to C♯, which also allows value type parameters. Our conclusion is something as simple as possible: The simplest form of generics supported by both languages, the declaration of one or more type variables where every type variable have to be a reference type.

3.1.3 Inheritance and Object Oriented Semantics

Inheritance in C♭ works in the following manner: All members are inherited from the super/parent type, private members are inherited but not accessible, cf. Java [10, §8.4.8], C♯ [19, §inheritance].

3.1.4 Exceptions

C♭ contains the class Exception. Exception is an abstract class which is the parent class of all other exception classes. The exception class is not meant to be instansiated or inherited from, instead use the subclasses CheckedException and UncheckedException; cf. Java [24], C♯ [20]

This is an example of a construct not in the intersection, but is of such great importance that we are forced to add it to C♭. If we did omit checked exceptions, then a large subset of the Java standard library would have to be discarded, as it uses checked exceptions.

3.1.5 Project Management

C♭ uses modules for namespace management. Multiple classes shares the same namespace if they are declared in the same module. Each C♭ file is required to contain exactly one module. Furthermore C♭ requires the directory structure to reflect the module structure in a project, similarly to Java package organization [10, §7.1].

Given a module M and a submodule S, the fully qualified name of S is M@S.

C♭ Semantic Errors

- A C♭ file is permitted to contain exactly one module.
- A module may not contain two members of the same name.
- A C♭ file must contain exactly one top level class.
- A C♭ file must be have the same name as the name of the top level class declared followed by the ".cflat" extension [10, §7.6].
- It is not permitted to import an inaccessible type.

Importing Modules

Importing modules is done by using the EXTERNAL keyword. The grammar is described below.

```
ExternalModuleClause ::= 'EXTERNAL' 'MODULES'
{CFlatModuleName}    + BY ',' ';'
```

3.1.6 Classes

C♭ only contains classes as types. The class modifiers are the following:

C♭ Keyword	Semantics
ABSTRACT	Non-Instansiable
STATIC	All members must be static

The grammar for class declaration is as follows:

```
CFlatClass ::={CFlatComment} * ['PUBLIC'] ['STICKY']
['STATIC'] ['ABSTRACT']'CLASS' CFlatClassName ';'
['INHERITS' CFlatClassName  ';']
{CFlatClassElement}  *      'ENDCLASS' ';'
```

A class is said to be a top level class if it is not declared as a member class of some enclosing class declaration.

The alternative, nested classes are classes declared as class members of some class.

Top level C♭ classes must be public. Furthermore C♭ classes can only contain one of the optional modifiers ABSTRACT, FINAL or STATIC. Only abstract classes are permitted to declare abstract methods, cf. C♯ [19, §abstract-classes], Java [10, §8.1.1.1].

Classes Semantic Errors

- Attempting to instantiate an object from an abstract class.
- Declaring a class with more than one of the following modifiers: ABSTRACT, FINAL and STATIC.

- Attempt at instantiating a static class, attempting to derive from a static class or trying to declare non static members in a static class.
- Attempting to declare top level classes private.
- Attempting to declare an abstract method in a non abstract class
- A static class declaring non-static members.

In C♯ assemblies often define the scope of access modifiers. Java on the other hand uses packages to define the scope of access modifiers. The only access modifiers in the intersection between Java access and C♯ are `public` and `private`, thus they are the only access modifiers that should be in C♭.

Thus the current grammar of C♭ allows for the usage of either no access modifier or public. This is a bug in the parser.

Nested classes

A class is said to be nested if it is declared as a class member of some other class. C♭ supports nested classes [19, §nested-types] [10, §8.1.3]. A C♭ class declaration contains exactly one access modifier, furthermore it can also use the optional modifiers: `STATIC`, `FINAL` and `ABSTRACT`. Non-static nested classes may not declare static members with the exception of constant variables [10, §4.12.4]. A nested class has access to all types that are accessible to its containing class regardless of access modifiers. However the opposite is not true, the outer class does not have access to the inner class's private members.

The `THIS` keyword in a nested class refers to the object for which the method was invoked, as in Java [10, §15.8.3], C♯ [19, §this-access].

Nested Classes Semantic Errors

- If declared static a nested class is not permitted to contain any non-static members.
- If the nested class is not declared static then it is not permitted to declare static members, unless the members are constants.
- Local variables used but not declared in an inner class must either be `FINAL` or be effectively final [10, §4.12.4].

- Local variables used but not declared in an inner class must be definetly assigned before the body of the inner class [10, §16].
- Blank FINAL fields of enclosing types are not permitted to be assigned within an inner class [10, §4.12.4].

3.1.7 Field Variables

C♭ field declarations must contain exactly one access modifier, at most one static declaration and at most one immutability declaration.

Construct	Keyword	Keyword	Semantics	Java	C♯
Access modifier	PRIVATE	PUBLIC	Encapsulation	[10, §8.3.1]	[19, §fields]
Static modifier	STATIC	[none]	Static variables	[10, §8.3.1.1]	[19, §static-and-instance-fields]
Immutability modifier	FINAL	[none]	Immutable variables	[10, §8.3.1.2]	[19, §readonly-fields]

The grammar is as below:

```
CFlatAttribute ::={'PRIVATE'|'PROTECTED'|'PUBLIC'}
['STICKY']    ['STATIC'] {'CONST'|'VAR'}
Ident  [':='Expression] ':' CFlatType    ';'
```

Again, there is an error in the parser. The only access modifiers that are allowed in C♭ should be PUBLIC and PRIVATE.

3.1.8 Constructors

C♭ contains constructors. Constructors are used to instantiate instances (objects) of a non-static class; [10, §8.8], [19, §instance-constructors].

The constructor name must be the same as the class name, have no return type and is not inherited. Furthermore instance constructors has exactly one access modifier, can be overloaded and is invoked by using the NEW operator.

Constructor related semantic errors The following produces C♭ errors.

- Constructors are not permitted to contain a return type in the signature.

- Constructors are not permitted to contain the following keywords: STATIC, ABSTRACT or OVERRIDE.
- A class is not permitted to contain two constructors with override equivalent signatures [10, §8.4.2].

Static constructors (e.g, `static { ... }`) are both in Java and C♯ however they are not within C♭. It was simply not prioritized during development.

3.1.9 Methods

C♭ contains member methods. More on the semantics below.

Method Signature

Method declaration in C♭ is defined by the grammar below.

```
CFlatMethod ::=
    { 'PRIVATE' | 'PROTECTED' | 'PUBLIC' }
    [ 'STICKY' ] [ 'STATIC' ] [ 'ABSTRACT' ]
    'METHOD' [ 'OVERRIDE' ] Ident
    [ '(' { CFlatParam } * by ',' ')' ]
    [ 'THROWS' { CFlatClassName } + by ',' ]
    [ ':' SuperRef '(' { FunctionArg } * by ',' ')']
    [ ':' CFlatType ] ';'
    [ StatementList ]
'ENDMETHOD' ';'
```

Figure 3.1: C♭ method grammar

Again there is a bug in the access modifiers: The only access modifiers that should be available are PUBLIC and PRIVATE.

Semantically method declaration in C♭ mimic that of Java and C♯ . Below is more information on the modifiers.

Keyword	Semantics	ref
`STICKY`	Marked as a sticky	C♭ construct
`STATIC`	class variable	[10, §8.4.3.2], [19, §static-and-instance-methods]
`ABSTRACT`	non-instansiable	[10, §8.4.3.1], [19, §abstract-methods]
`OVERRIDE`	Overrides a method in parent class	[10, §9.6.4.4], [19, §override-methods]

The semantics for C♭ modifiers are all defined by the Java and C♯ ones. However the `STICKY` modifier is a unique C♭ modifier we use for marking methods which are not to be deleted. For instance it is used in the Dead Method removal procedures in the transformation section.

Method Overriding C♭ supports method overload [10, §8.4.9] [19, §basic-signatures-and-overloading]. For a method m1 to be overridable by a method m2 all of the following must be true:

- m1 must be accessible to subclasses, therefore it can not have the access modifier private.
- m1 can not contain the modifiers `FINAL` or `STATIC`.
- m1 and m2 must have the same signature, the same access modifiers and finally the same return type.
- m2 must be in a super class of m1.

Method related semantic errors The following produces C♭ errors [19, §methods] [10, §8.4.3]:

- Method declarations can not contain the `ABSTRACT` modifier and one of the following modifiers: `PRIVATE`, `STATIC` or `FINAL`.
- Methods declared as `ABSTRACT` are only permitted in `ABSTRACT` classes.
- Given an abstract class C with an abstract method m, all classes that inherit C must implement m.
- It is not permitted for a static method to use the name of a type parameter of surrounding declaration in the signature or body.
- Static methods are not permitted to reference `THIS` keyword or `SUPER` keyword.

- Abstract methods are not permitted to include the static modifier.
- If a statement in the method body throws an exception, then that exception must be specified at the method declaration.
- It is not permitted for a static class to be overriden by an instance class.
- private methods can not be overridable, override other methods or be abstract.
- A method with the return type void is not permitted to contain a return statement which contains an expression.
- A method with a return type that is not void must contain a return statement which contain an expression at every execution path.
- a method with a return type that is not void must contain a return statement which contains an expression.

Method Overload

C♭ allows for the overloading of constructors and methods. This is the same as in Java [10, §8.4.9] and C♯ [19, §basic-signatures-and-overloading]: Using the same method name, provided their signature is unique within a class. The actual argument types and the number of arguments are used to determine the method to be invoked.

Checked Exceptions In Method Signature

The throws clause in the method signature can only contain names of C♭ CheckedExceptions classes. throws an exception which directly or indirectly belong to a CheckedException class [10, §11.1.1].

A C♭ method containing a throws clause in the method signature will not use that information in any way during the serialization to C♯ as C♯ does not contain the concept of checked exceptions.

3.1.10 Statements

C♭ contains the statements defined in this section.

Break

C♭ contains the break statement which contains the same semantics as the `break` statement in Java [10, §14.15] and C♯ [19, §the-break-statement].

C♭ does not include the optional label specified in the Java grammar.

The grammar for a break statement is the following:

Break Statement Grammar:

```
BreakStatement ::= 'BREAK' ';'
```

Semantic Errors

- Break statements are not permitted unless contained in the body of a switch, while, do while, foreach or for statement.

Continue

C♭ contains the continue statement which contains the same semantics as in Java [10, §14.16] and C♯ [19, §the-continue-statement]

Continue Statement Grammar:

```
ContinueStatement ::= 'CONTINUE' ';'
```

Semantic Errors

- Continue statements are not permitted unless contained in the body of a while, do while, foreach or for statement.

Return

C♭ contains the return statement with the same semantics as in Java [10, §14.17] and C♯ [19, §the-return-statement].

Return Statement Grammar:

```
ReturnStatement ::= 'RETURN' [Expression] ';'
```

Semantic Errors

- A return statement without an accompanying expression must be contained in either a method with the return value `VOID` or a constructor.
- A return statement with an accompanying expression must be contained in a method which return any value not `VOID`.
- Return statements are not permitted to be contained in a finally block.

If Statement

C♭ contains if statements, within the if statements one can declare else-if statements and else statements. The semantics are the same as in Java [10, §14.9] and C♯ [19, §the-if-statement].

If Statement Grammar:

```
IfStatement ::= 'IF' Expression 'THEN'
  [StatementList]
  {ElsifClause}*
  ['ELSE'
  [StatementList]]
  'ENDIF' ';'
```

elseif clause grammar:

```
ElsifClause ::=
  'ELSIF'     Expression 'THEN'
    [StatementList]
```

Semantic Errors

- The expression contained in the if must be of type `BOOLEAN` or the boxed reference type for `BOOLEAN`.

For loop

For loops are included in C♭ they contain the same semantics as in Java [10, §14.14.1] and C♯ [19, §the-for-statement] counting-style for-loops.

For Loop Grammar:

```
ForStatement ::=
  'FOR' SimpleExpression ':=' Expression 'TO'
  Expression
  ['BY' Expression] 'DO'
    [StatementList]
  'ENDFOR' ';'
```

Semantic Errors

- The condition expression must be of type BOOLEAN or the boxed type of BOOLEAN.

Foreach Loop

C♭ also contains the foreach loop which also contains the same semantics as in Java [10, §14.14.2] and C♯ [19, §the-foreach-statement].

Foreach Statement Grammar:

```
ForeachStatement ::=
    'FOREACH' CFlatParam 'IN' Expression
    'DO' [StatementList]
    'ENDFOREACH'     ';'
```

Semantic Errors

- The type of the collection must be an array type or an Iterable type.

While

C♭ contains the while statement with the same semantics as in Java [10, §14.12] and C♯ [19, §the-while-statement].

While Statement Grammar:

```
WhileStatement ::=
  'WHILE' Expression 'DO'
  StatementList
  'ENDWHILE' ';'
```

Semantic Errors

- The expression must evaluate to a `BOOLEAN`, or the boxed type of `BOOLEAN`.

Do While

C♭ contains the do while statement, the semantics are the same as in Java [10, §14.13] and C♯ [19, §the-do-statement].

Do While Grammar:

```
DoWhileStatement ::=
  'DO'
    [StatementList]
  'WHILE' Expression ';'
```

Semantic Errors

- The expression must evaluate to a `BOOLEAN`, or the boxed type of `BOOLEAN`.

Throw Statement

C♭ contains the throw statement. Throw statements contain the same semantic in Java [10, §14.18] and C♯ [19, §the-throw-statement].

Throw Statement Grammar:

```
ThrowStatement ::=
  'THROW' Expression ';'
```

Semantic Errors

- The exception expression must have type Exception or a subclass of Exception.

Try Statement

C♭ contains the try, catch and finally statements. The semantics are the same as in Java [10, §14.20] and C♯ [19, §the-try-statement].

Try Statement Grammar:

```
TryStatement ::=
  'TRY'
    StatementList
    {CatchBlock}*
    [FinallyBlock]
  'ENDTRY'';'
```

Semantic Errors

- The exception expression must be Exception or a subclass of Exception.
- A finally clause in a try statement is not permitted to contain a return statement.

3.1.11 Expressions

Expressions uses the infix notation for binary operators.

The grammar for binary expressions:

```
BinaryExpression ::= Expression '<'.. '=' Expression

BinaryExpression ::= Expression {'-'|'+'} Expression

BinaryExpression ::=
          Expression {'MOD'|'*'|'/'|'&'} Expression

BinaryExpression ::= Expression 'AND' Expression

BinaryExpression ::= Expression 'OR' Expression
```

Other expression types include CastExpression, IsCondition, NewExpression, SimpleExpression, TestExpression, TypeOfExpression and UnaryExpression. For details please see the grammar.

There are other expression types, however those were expression types we did not work with. For instance there are database expressions. When C♭ is used as a solution for a client, it is expected that the database expressions will come in handy. We however wanted to focus on the core of the language thus we did not do much with them.
These constructs are expected to be useful at some point.

IS operator

C♭ contains the `IS` operator. Given an expression e and a type T, the `IS` operator is used to check if the runtime type of *e* is compatible with T, that is if the value of *e* can be cast to T [10, §15.20.2], [19, §the-is-operator].

CHAPTER
FOUR

SERIALIZER

In this chapter we will look into the implementation of the C♭ Serializer and programs around the serializer. We will also look at the history of C♭ project. Finally, as mentioned earlier, the parser is not written by me, thus I won't go into the details of the parser.

4.1 History of C♭

C♭ was a project which was started 6 years before I arrived at Raincode. What we found when the project was resumed was a parser, a grammar and the beginnings of a serializer to C♯.

However, there were clear signs of the project being very much a work in progress. There was no Java serializer and the grammar had some constructs which are not in the intersection. For instance, the parameters of a function could use the keywords `out` [19, §method-parameters] and `ref` [19, §method-parameters]{, which can only be supported by C♯.

One advantage for me personally when I arrived at Raincode and found a project which was already started was that I did not know the internal language RCScript. Thus finding a program which I could analyze to learn the syntax and semantics of RCScript helped ease me into the project.

4.2 Restarting the C♭ project

We started out by writing tests for C♭ in order to familiarize ourselves with the work that had previously been done.

The main structure of the grammar was something we kept, although we heavily modified it. An example of how we modified the grammar, was with Modules. This is what a C♭ module and class looked like before we redefined it.

```
        MODULES EnclosingModule;

        PUBLIC CLASS MyClass#EnclosingModule;
        ENDCLASS;
```

Module translates to `package` in Java [10, §7] and `namespace` in C♯ [19, §namespace-declarations]. The grammar for module declaration was:

```
        'MODULES' {Ident}
```

This grammar poses a problem as one would often like to nest modules as is the case in Java [10, §7.4] and C♯ [19, §namespace-declarations]. To fix this problem we created a seperator character '@' within the module name. For example

```
        MODULES java@lang;
```

would denote the 'java.lang' library. The actual transformation, which in this case is to replace the '@' character with a '.' character is handled by the Serializer.

4.3 Serializer

The serializer is a mixture of multiple scripts executed in a particular order. These scripts traverse a given parse tree abstract syntax tree using mutually recursive procedures that define a depth first search.

Serialization is initiated when a construct which is to be serialized is reached. For instance consider this if statement.

```
IF TRUE THEN
  BREAK;
ENDIF;
```

This if statement is represented as nonterminal with the following production:

```
IfStatement ::=
  'IF' Expression 'THEN'
      [StatementList]
    {ElsifClause}*
    ['ELSE'
      [StatementList]
    ]
  'ENDIF' ';'

If statement contains the following properties:

Condition: Expression
ElseStatementList: StatementList
ElsifClauses: LIST of ElsifClause
StatementList: StatementList
```

Figure 4.1: The if statement nonterminal as defined in the parser

The traversal is done via pattern matching on the type of nonterminal. For instance, the if statement nonterminal above first matches on being a statement, then matches on the type of statement: IfStatement. Once the type of the construct is determined we start the serialization of the subtree of the nonterminal. In this case the condition expression (TRUE literal) and the block (BREAK statement). Once the subtree has been traversed and serialized we backtrack to the if statement nonterminal and serialize it. We will return to this if statement at a later point, let us first examining the abstract syntax tree traversal in details.

4.3.1 Traversal

Traversal is done by a depth first search; i.e., a bottom-up traversal that starts at the leaf nodes. The target language is given as a system argument to a program called the *pipeline*, for now you can think of the pipeline as the program which runs the serializer. We will revisit the pipeline in the chapters on semantic analysis (Chapter 5) and transformation (Chapter 6).

The main procedure in the traversal module is Traverse, which takes an abstract syntax tree node corresponding to a nonterminal and matches on the type of the nonterminal to further pass it to the correct procedure. Nested constructs are handled by recursive calls to Traverse. This traversal dispatch mechanism is somewhat similar to the visitor pattern [25].

```
PROCEDURE Traverse(NT)
BEGIN
  IF NT IS Statement THEN
    RESULT := TraverseStatement(NT);
  ELSIF NT IS Expression THEN
    RESULT := TraverseExpression(NT);
  ELSIF NT IS CFlatClass THEN
    RESULT := TraverseCFlatClass(NT);
  ...
  ELSIF NT IS Ident THEN
    RESULT := TraverseIdent(NT);
  ELSE
    RESULT := Utils.Fail(NT);
  END;
END;
```

Figure 4.2: The central traversal procedure. The '...' represents omitted source code from this example.

Procedures such as TraverseStatement and TraverseExpression further matches the given item. This way we ensure that any nonterminal can be directed to the correct procedure. Once an item is reached that requires serialization of some information then that item is passed to a serializer function.

Let us reconsider the if statement example. The traversal of the if statement terminal is presented below. The traversal of the ELSEIF and

ELSE nonterminals are extremely similar, thus for the sake of clarity I removed them.

```
PROCEDURE TraverseIfStatement(NT)
  BEGIN

  -- RecursiveFlat returns the flattened list.
  ifBody := List.
          RecursiveFlat(Traverse(Nt.StatementList));

  ifCond := Traverse(Stmt.Condition);

  -- in the same manner
  -- traverse elsif subnodes and else subnodes.
  ...
  -- Each of these parameters aside from NT
  -- are string representations which has already
  -- run their corresponding serializer function.
  RESULT := targetSerializer.
      SerializeIfStatement(NT, ifBody, ifCond,
      ElsifBody, ElseBody);
END;
```

Internally we used multiple names to refer to the serializer. We mainly used serializer but also Exporter and processor. Thus it is named Processor in the source code. For clarity in this thesis I will rename it to Serializer.

It is important to fetch all the necessary information before the serialization is initiated, which is why we are using depth first search. As the program above demonstrates the traversal is run before the final line where the serialization of this particular construct starts.

4.4 Serialization

Serialization is the process of transforming or translating source code from a source language to a target language. This is simply done by traversal of the abstract syntax tree, fetching the necessary data and encoding it in the syntax of the target language. Keeping in mind that source code is a representation of some data in a tree can help. In the terminology of Zaytsev and Bagge [29], translation of a program P to the target language is done by $unparse_{target}(parse_{C\flat}(P))$.

Recall the earlier if statement as an example:

```
IF TRUE THEN
  BREAK;
ENDIF;
```

One could think of serialization as the opposite of parsing; *unparsing*. To encode this if statement in the correct syntax of the target language we simply unparse the abstract syntax tree to the target language. We traverse every subnode in the subtree starting at the `if` statement, finding a `true` literal and a `break` statement. Furthermore, we fetch them and pass them along to the serializer procedure which encodes them in an `if` statement in the target language.

```
PROCEDURE serializeIfStatement(self, ifBody, ifCond)
  BEGIN
  "if( " + ifCond + ") {" +
    ifBody +
    "}"
END;
```

Figure 4.3: Again, we are omitting the else and else if for the sake of clarity.

The end result is the following string:

```
  if (true) {
    break;
  }
```

To store the syntax of the target language we use procedures which contain the string literals. For instance to get the name of a predefined type in C♯ we simply pass along the C♭ type to the following procedure.

```
-- Returns string representation of predefined types
PROCEDURE GetPredefinedType(code);
BEGIN
  IF code = IntegerKeyword THEN RESULT := "int";
  ELSIF code = FloatKeyword THEN RESULT := "float";
  ELSIF code = DoubleKeyword THEN RESULT := "double";
  ELSIF code = StringKeyword THEN RESULT := "string";
  ELSIF code = BooleanKeyword THEN RESULT := "bool";
  ELSIF code = DateKeyword THEN RESULT := "DateTime";
  ELSIF code = ByteKeyword THEN RESULT := "byte";
  ELSIF code = ShortKeyword THEN RESULT := "short";
  ELSIF code = LongKeyword THEN RESULT := "long";
  ELSIF code = CharKeyword THEN RESULT := "char";
  ELSE
      RESULT := UnknownType;
  END;
END;
```

Figure 4.4: Procedure for C♯ type keywords

The serialization process is fairly straightforward. Although time consuming and a procedure which require a large amount of testing, the programming itself was not too difficult. For more example of C♭ programs and the results from serializing them to Java and C♯, see Appendix A.

CHAPTER
FIVE

C♭ SEMANTIC ANALYSIS

In this chapter we will discuss a set of semantic analysis procedures we implemented for C♭. We will go into details of how they were implemented and show source code as a tool for illustration.

5.1 Semantic Analysis Introduction

Semantic analysis is a process within compilation which enforces restrictions on the input program. These restrictions are defined in the language specifications. For instance type checking in Java and C♯ are great examples of semantic analysis. One usually differentiates between syntax errors, i.e., errors produced at the parsing stage due to an input file not conforming to the concrete syntax, and semantic errors which are errors that although syntactically correct contradicts the language specification. such as type errors and name binding errors (referring to names that haven't been defined).

Defined in the language specifications of C♭ are semantic errors. However our semantic analysis is not complete. Due to time limitations we only implemented checks for a small subset of the semantic errors we defined in the language specification. The thinking behind the semantic analysis was that we want to catch problems as early as possible. Furthermore we created independent programs such that we can add new stages to it in the future.Clarify last sentence.

Before listing the semantic analysis components, it is appropriate to

Semantic Error. *Static classes are only permitted to contain static members*

mention how we signal a semantic error. Errors are signalled by printing an error message with the appropriate source code location, then terminating execution of the pipeline with a −1 exit code. The pipeline is the enclosing program of the semantic analysis, program transformations and the serializer, more on the pipeline at chapter Chapter 6. Providing a specific exit code signalling the failure of semantic analysis is useful for testing purposes as it can be detected by scripts. More on testing in Chapter 7.

You will see a RCScript in this chapter quite frequently. The syntax of the language is quite understandable, I will add comments in the source code to make the more unusual syntax for operators more clear.

5.1.1 Class Analysis

Implementing that using RCScript is quite simple. A C♭ class is a node in the abstract syntax tree, one can easily extract all the subnodes and iterate over them. If subnode v has a property 'static', that is if v can be static then we simply check the static value of v. If v is not static then we fail the static analysis.

```
PROCEDURE StaticClassMembersCheck(inputClass)
BEGIN
    FOR node IN inputClass.DirectSubNodes DO
        IF node CAN static THEN
            IF NOT node!static THEN
                fail(inputClass);
            ENDIF
        ENDIF
    ENDFOR
ENDPROCEDURE
```

In Java there is no such thing as static top level classes [10, §8.1.1]. Thus this constraint is only at the C♭ level code. This is one of those times were C♭ contains a construct which is not in the intersection of the languages.

However this is one of the simpler cases as the implementation of the check was almost trivial.

5.1.2 Method Analysis

Semantic Error. *Static methods are not permitted to contain the keywords 'THIS' or 'SUPER'*

Again, the checker implementation for those semantics are almost trivial. Simply filter the subtree of the method such that only the 'THIS' and 'SUPER' references remain. If the list representation of the subtree that is retrieved is not empty, then fail.

```
PROCEDURE StaticMethodAnalysis(method)
BEGIN
   IF method!Static <> VOID THEN

      -- | is the filter operator.
      -- <> is 'not equal' operator.
       subTree := method.SubNodes |
      ((X IS ThisRef) OR (X IS SuperRef));

       IF subTree <> VOID THEN
     FAIL
       ENDIF;
   ENDIF;
ENDPROCEDURE;
```

5.1.3 Constructor Analysis

Semantic Error. *Constructors are not permitted to contain a return type in the signature.*

To enforce these semantics, we simply extract all constructors and iterate over them. If the 'ReturnedType' variable has any value besides 'VOID' then fail the analysis.

5.1.4 Statement Analysis

There are various other statements which are easier to implement located in the AnalyzeStatement module.

Try Statement

C♭ does not permit a finally clause to contain a return statement.

Semantic Error. *A finally clause in a try statement is not permitted to contain a return statement.*

The implementation is trivial. If a try statement contains a finally block, simply look at the subtree with the finally block as the root and filter away all subnodes that are not return. If the subtree still contains any children then throw a semantic error.

```
PROCEDURE retInFinally(try)
VAR errorMessage, subTree;
BEGIN
    errorMessage := "Finally clause is not"
    & "permitted to contain a return statement ";

    -- CAN operator decides whether the try node
    -- contains a non VOID Finally attribute.
    IF try CAN Finally THEN

        -- The | is the filter operator.
        subTree := try.Finally!SubNodes
          | X IS ReturnStatement;

        -- There exists a return
        -- subnodes in the finally clause..
        IF subTree <> VOID THEN
          AnalysisUtils.
          VerificationFail(errorMessage, try.Finally);
        END;
    END;
END;
```

Continue and Break statement

Continue and break statements are both statements which are meant to be contained in loops. Thus we created one procedure for both that takes a nonterminal in the abstract syntax tree and checks whether they are contained in a loop.

Semantic Error. *A break statement must be contained in a loop.*

And...

Semantic Error. *A continue statement must be contained in a loop.*

To enforce this is almost trivial we simply used RCScript which contains excellent tree traversal functionality to simply check the ancestor nodes of a nonterminal is a loop statement, if not throw a semantic error.

The exact line where the check is made looks like this.

```
RESULT := #(nt.Ancestors | X IS LoopStatement) > 0;
```

We produce a list of ancestor nodes which are loop statements and check if the size is larger than 0, that is non empty. The result of that check is assigned to the result variable.

Return Statement Analysis

In this section, we will examine several checks for return statements.

Semantic Error. *A Method with a return type that is not void must contain a return statement with a following expression at every execution path.*

Return statements have semantics which can be difficult to enforce. Calculating the execution path of a program and ensuring that if a method's return type is set to anything besides 'VOID' then every execution path contains a return statement. Another, simpler way of putting it is:

"Will the last statement run by this method be a return statement for all valid input?"

One needs to consider nested ifs and other statements in this analysis as they can diverge the computation into a path which does not contain a return statement.

What we chose to implement here is an analysis that checks the execution path of conditionals. If there exist a conditional statement with multiple paths, those are if statement followed by else statement and/or else if statements. If the analysis finds such a branching at a conditional statement it will check that every such path contains a return statement.

Two other checks for return statement that we implemented are closely tied. The first is the following.

Semantic Error. *Methods with the return type void must not contain return statements which contains an expression.*

and the opposite

Semantic Error. *Methods with any return type not void must contain return statements which contains an expression.*

These two analysis are extremely simple to implement, trivial really. One simply iterates over every return statement and checks that it contains an expression if it is supposed to or check that it does not contain an expression. Both of them are tested in this procedure. Aside from a few comments which explains the less obvious operators and line breaks, this is the actual source code of the procedure in the repository that makes the analysis.

```
PROCEDURE RetMethodType(Method)
VAR Message, MethodHasType, RetHasType, RetList;
BEGIN
    Message := "The type of a return statement"
    & "must correspond to the methods returned type";

    -- <> is the inequality operator.
    MethodHasType := Method.ReturnedType <> VOID;

    -- | Filter operator.
    RetList := Method.SubNodes | X IS ReturnStatement;

    IF RetList = VOID AND MethodHasType THEN
      Fail(Method, Message);
    END;

    IF RetList <> VOID THEN
        FOR ret IN RetList DO
            RetHasType := ret.Value <> VOID;

            IF MethodHasType AND (NOT RetHasType) THEN
              Fail(ret, Message);
            END;

            IF (NOT MethodHasType) AND RetHasType THEN
              Fail(ret, Message);
            END;
        END;
    END;
END;
```

CHAPTER

SIX

C♭ TRANSFORMATION

In this chapter we will discuss program transformations which takes C♭ source code and transforms it into equivalent C♭ programs while improving the code quality and making it more readable for the programmer.

6.1 C♭ to C♭ Transformations

During the development of C♭ we envisioned a pipeline in which the input would be C♭ source code and the output would be equivalent C♭ source code. At each stage of this pipeline the code would be modified while preserving the semantics, the purpose is to output code of higher quality. The pipeline is illustrated in Figure 6.1.

The naming convention of the stages may not make sense now, however we will explain each and everyone of them when we will examine the implementation.

We included semantic analysis in the pipeline even though it does not perform any transformation on the input program. The semantic analysis stage is the only such case where a transformation is not performed. The inclusion of the semantic analysis is due to how naturally it fit the first stage of the pipeline.

The pipeline is divided into two parts: The language independent section and the language specific section.

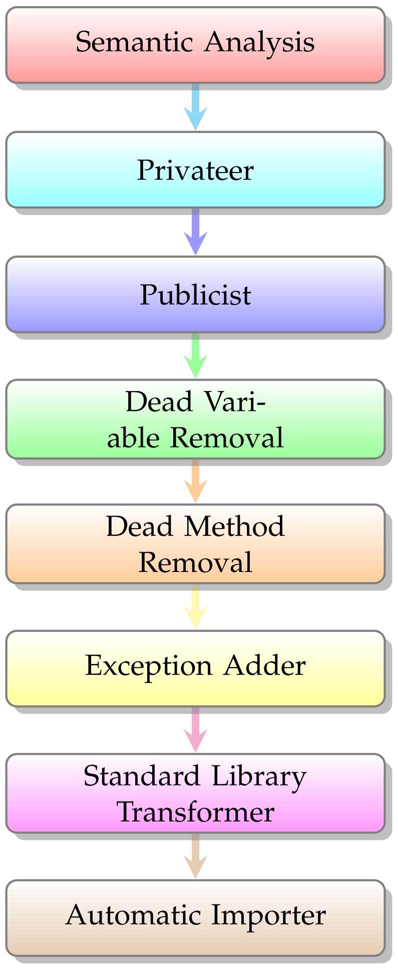


Figure 6.1: The C♭ pipeline

6.1.1 Language-Independent Section

The language-independent section operates generally on any of the serialization languages by altering the underlying abstract syntax tree in order to alter the program. For instance changing the accessibility modifier of a method nonterminal for the purpose of data hiding.

The language-independent section is composed of the following procedures.

1. Semantic analysis
2. Privateer
3. Publicist
4. Dead variable removal
5. Dead method remover
6. Exception Adder

6.1.2 Language Specific Section

The language specific section alters the nodes in the abstract syntax tree to correspond to some data in the target language The reason these stages of the pipeline is language dependent is that the transformation is in a way an extension of the serializer. I will get into further detail in their specific section.

1. Standard Library
2. Automatic importer

One could view the language specific part as the part of the pipeline which runs transformations which does not capture the intersection. For instance the Standard Library module is dependent on the target language and thus is not in the intersection.

6.2 Dead Method Removal

We created a module which detects dead methods and marks them as such. Dead Methods are methods which are never called during the execution of the program. To detect these we first need an entry point, in both Java [10, §12.1.4] and C♯ [19, §basic-application-startup] that is the main method. Valid signature for the main method is shown below.

```
// Java main method.
public static void main(String [] args) {..}

// Java main method.
public static void main(String... args)

// Cs Main Method.
static void Main(string[] args) {...}

// Cs Main Method.
static int Main(string[] args) {...}

// Cs Main Method.
static int Main() {...}

// Cs Main Method.
static void Main() {...}
```

In order to proceed we must somehow mark the main method such that it should never be removed. To do that we have the 'STICKY' modifier in the grammar.

The 'STICKY' keyword is used to mark a method as sticky, sticky is a C♭ modifier which can be included in class, method and attribute non-terminals. As it is a C♭ modifier, 'STICKY' does not contain any inherent semantics during execution, however as of now we use it to keep track of classes and members which should not be removed.

We can illustrate the running of the dead method removal better by representing the methods as vertices in a directed graph where the edges represent the calls, i.e a method A has a directed edge that points towards a method B if A has a call to B in A's method body.

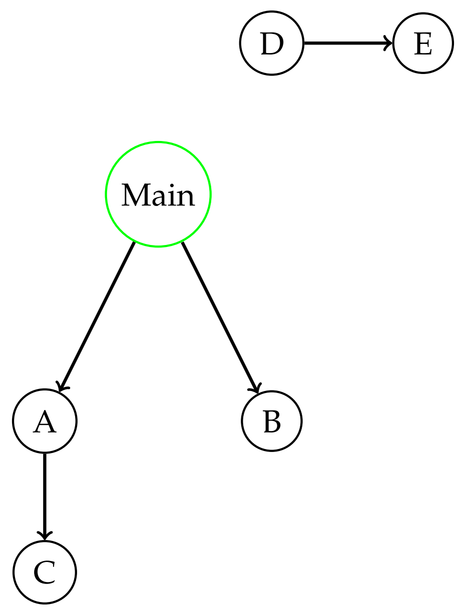


Figure 6.2: Example of determining what methods should be deleted.

Suppose you have the following program. There are multiple methods, method Main calls methods A and B, A calls method C and D calls method E. The method Main represents the main method (entry point) in this example. The methods D and E are not connected to the rest of the graph, thus will never be called during the execution of the program.

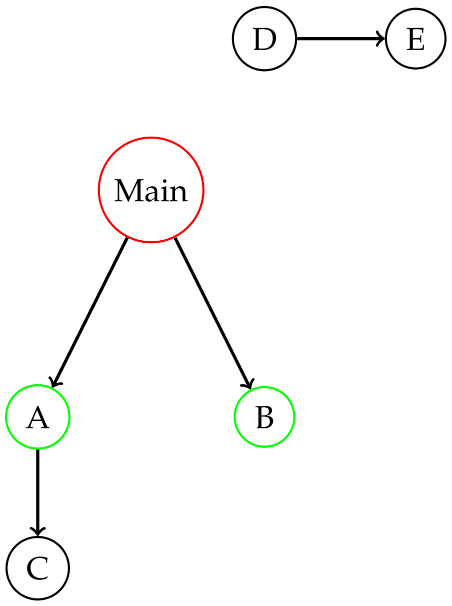
Furthermore we use color to illustrate the marking of a vertex. A green vertex means that a method is marked as sticky, a blue vertex means it is set to the ToMark state, a red vertex means that it has the Marked state and a black vertex means that it has no Mark state. A vertex without a mark state after execution of the Mark procedure (see Listing 6.1) the dead Method removal will be considered as a dead method and will have the string "DEAD_" added as a prefix.

We have three procedures that must be mentioned. The first is Mark which takes a list of C♭ file root nodes representing a program and iterate over every method contained in that program. If a method is sticky and not marked OR a method is set to ToMark then it is sent to the MarkMethod procedure. This is run until no further changes can occur i.e until no further methods can be sent to the MarkMethod procedure.

The MarkMethod procedure takes a method m as input, set's it to marked and finds every method m2 called by m and sets m2 as toMarked.

Finally Sweep runs through every method and if they are not sticky or marked then their name gains the prefix “DEAD_”. Simplified pseudocode of these procedures are on the next page.

First we call Mark then Sweep in that order. Let us demonstrate with an example, Recall the example earlier (Figure 6.2). When we run Mark we will at the first iteration set Main as marked and then set A and B as ToMark.



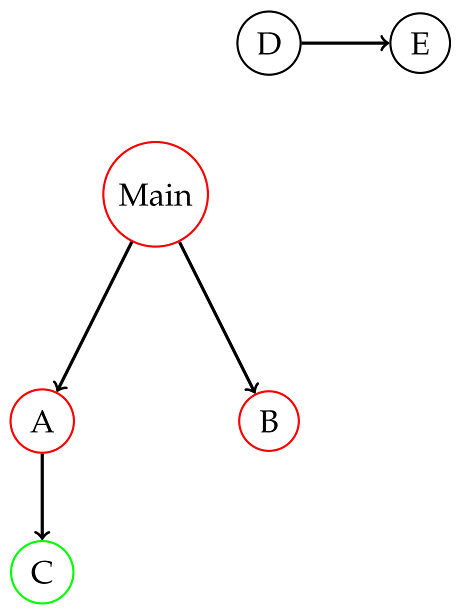
In the next iteration A and B will be set to Marked and C will be set to ToMark.

Listing 6.1: Pseudocode of the mark program

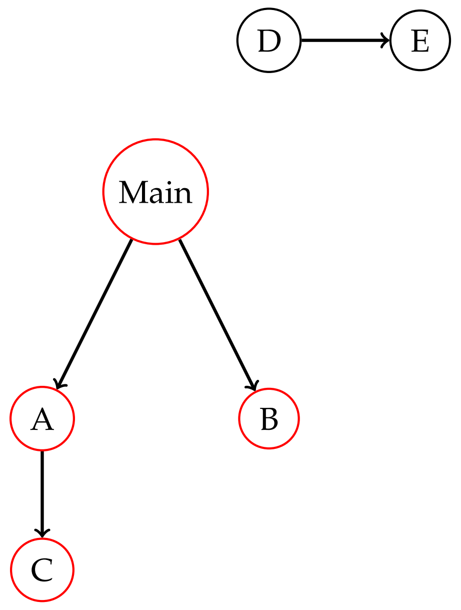
```
PROCEDURE Mark(Program)
  While loop DO
    loop := FALSE;
    FOR method in Program DO
      IF method is Sticky AND method.Mark = VOID THEN
        loop := TRUE;
        MarkMethod(method);
      END;
      IF method.Mark = ToMark THEN
        loop := TRUE;
        MarkMethod(method);
      END;
    END;
  END;
END;

PROCEDURE MarkMethod(m);
BEGIN
  m.Mark := Marked;
  FOR method m2 called by m DO
    IF m2.Mark = VOID AND m2 <> VOID THEN
      m2.Mark = ToMark;
    END;
  END;
END;

PROCEDURE sweep(program);
BEGIN
  FOR method in program DO
    IF NOT (method.Sticky) OR method.Mark = VOID THEN
    method.Id := ‘‘DEAD_’’+Method.Id;
    END;
  END;
END;
```



Finally we Mark C. As C does not call any other method the procedure will not be able to send any further method to the MarkMethod procedure and thus terminate, yielding this graph:



Methods D and E are never called thus never marked. Therefore they are marked as the dead methods "DEAD_D" and "DEAD_E" by adding the "DEAD_" prefix to the method name.

That concludes the dead method removal algorithm.

6.3 Dead Variable Removal

In this section we discuss how to delete dead attributes, that is attributes which are never used or referred to. To implement the dead variable removal we iterate over each attribute and find the list of each member that refers to said attribute. This is trivially easy in RCscript with the given YAFL parser. Simply using the attribute RefBy provided a list of members that refers to this specific attribute.

If the list is empty then it is not referred to by any member and thus should be marked as dead. One marks the attributes as dead by giving it the prefix "DEAD". Finally one adds the comment "//Dead variable detected".

6.4 Checked Exceptions Finder

Java contains the construct of checked and unchecked exceptions. Unchecked exceptions are all the exception classes that inherit from the ERROR class, the RuntimeException class or any of their subclasses. The checked Exceptions are all other Exception classes [10, §8.1.1] .

The semantics checked exceptions is that if a method m throws a checked exception e then m has add e to the thrown type in the method signature [10, §11.2.3]. All methods that calls m must also add the classes m throws to their method signature.

```
static void m() throws IOException {
   throw new FileNotFoundException();
}
```

The checked exception finder is a stage in the pipeline which automatically detects checked exceptions and adds them to the method signature.

We chose to implement checked exceptions in C♭. We added a list of C♭ class names to the C♭ method nonterminal. This class name list represents every class which is thrown by the method. This attribute is called "ThrownType". A member method m should add checked exception class c to it's ThrownType if m throws c directly, or if m indirectly calls a method that throws c.

The procedure which finds checked exceptions and throws them works in two stages. The first stage is to find the methods which throws a class which inherit from the C♭ class "CheckedException". The other stage is to find all methods which calls a method which contains a non empty "ThrownType" list.

This stage of the pipeline is not in the intersection of Java and C♯ as it is not contained in C♯. However we still added it to the language independent section of the pipeline. The reason for that is that all we're doing in this stage is adding variables to an attribute in the method nonterminal. What we do with that attribute is determined at the execution serialization. Thus this section will have no negative impact on the serialization to C♯.

6.4.1 Methods That Directly Throws Exception

First we find all the methods which throw an exception within the method body. Let us call the program which does that for P1 such that we can refer to it in the future. To detect methods which contains throw statements in the method body we iterate over every method in the program. For every method, we iterate over every statement in the method body. If there is a throw statement then there is the potential for that throw statement to throw a checked exception class. Thus we examine the class name given at the throw statement, let us call the class name found c. If c is "CheckedException" or "UncheckedException" then we know whether to add c to the ThrownType list of the method. Otherwise we recursively climb up the inheritance tree of c arriving at class named c′. There are three cases which terminate this recursive procedure:

- c′ is "CheckedException" Terminate recursion and add c′ it to the m's ThrownType list.
- c′ is "UnecheckedException" Terminate recursion.
- c′ does not inherit from any class, aside from implicitly inheriting from the Object class, terminate recursion.

Thus we managed to enumerate all Checked Exceptions thrown within a Method's body into the method signature. However, there is still the case where a method calls other methods which throw a Checked Exception. We need another script to handle such a scenario.

6.4.2 Methods That Indirectly Throw An Exception

A method throws a checked exception if it contains a throw statement which throw a checked exception class or if it calls a method that or indirectly throws a checked exception class. A method m throws a checked exception indirectly if there is a sequence of calls between m and some other method which throws a checked exception. The previous program, P1 solves the case where a method throws a checked exception class in it's body by appending that class name to the throws variable. However the case where a call an lead to a thrown checked exception has still not been solved. We will solve that case now.

We use P1 as a base case. Suppose we have a program which throw a fair number of checked exceptions. After running P1 we expect to have a some methods which has non empty ThrownType property. Let us call that set of methods for s. Every method that calls a method m in s should add all of m's exceptions to it's method signature. Furthermore the list containing the exceptions being thrown should not contain duplicates, though that is trivial to accomplish.

This program uses a while loop which does not terminate until no further modifications can be applied. It iterates over every method m1 in the program and iterates over every method call made by m1. Suppose m1 calls a method m2, for m1 to have the thrown classes of m2 added to it's method signature all of the following conditions must be fullfilled:

1. The call to m2 must not be enclosed by a try statement.
2. m2 must throw classes not in the method signature of m1.
3. The call to m2 must contain the same number of parameters as the one defined in the declaration of m2.

If all of these conditions are satisfied then thrown classes of m2 are added to m1 and any duplicate classes are deleted.

6.5 Data Hiding

Data hiding is the concept where access to certain parts of the implementation is restricted. Not to be confused with encapsulation, the idea of bundling data and methods. data hiding is used extensively in the industry, and not only by object oriented languages. The functional language Haskell contain data hiding by the use of the exports keyword in a module definition [17]. We considered data hiding to be a good practice, thus adding a stage in the pipeline for that purpose. That way we can achieve automatic data hiding.

We use the two sections of the pipeline with the oddest names, the Privateer and the Publicist. The Privateer hides all methods and field variables. The publicist sets the access modifier of field variables and methods to public if that field variable or method is referred to by some entity outside of the enclosing class. Thus the order on which these stages of the pipeline run is of crucial importance. If they are run in the incorrect order, which is the Publicist before the Privateer then all fields and methods would be hidden.

6.5.1 Privateer

The privateer is trivial. We iterate over every field variable and method in a class. If that field variable or method can be set to private then we set it to private. That's it.

6.5.2 Publicist

At this point in the execution of the pipeline every member is set to the private access modifier. The publicist iterates over every member in a program and checks if is it ever referred to by any member which is not enclosed by the same class, then set it to public.

That concludes the pipeline stages on data hiding.

6.6 Standard Library Transformations

We defined a standard library for C♭, however we had no implementation of such a standard library. One of the advantages behind C♭ is the ability to reuse tools which are provided in the target language. That way we saved a lot of time not having to reinvent the wheel. Due to the similar design of Java and C♭ there is an intersection even in the standard library. By that we mean that part of the standard library in Java and C♯ have the same semantics or extremely similar semantics. Thus we used that to our advantage by defining a standard library which we can map to some some part of the standard library in Java or C♯. This was not always possible as there are times where Java and C♯ simply does not have an equivalence in the libraries. For instance one language, usually C♯ is using some feature which is not in the intersection, such as indexer types in C♯ [19, §indexers]. When such a case occurred we created something we call the run time libraries. However before that, let us define the standard library.

6.6.1 Standard library

C♭ contains the following types in the standard library:

Type	Description
Object	The primordial class.
STRING	String type.
Dict	Dictionary type
Input	Console input type.
FileIO	Simpler file reading and writing.
Filehandler	A type which provides easy to use methods for File IO.
FileWriter	A type for file writing.
FileReader	A type for file reading.
Exception	The root class of Checked exceptions and Unchecked exceptions.
CheckedException	The root type for Checked exceptions.
UncheckedException	The root type for Unchecked exceptions.
ERR	A type responsible for printing to the error stream.
OUT	A type responsible for printing to the out stream.

Object Class

The object type contains the following accessible members.

```
-- Compares equality between caller and obj.
PUBLIC METHOD equals(Object obj) : BOOLEAN
```

```

-- Returns a string representation of the caller object.
PUBLIC METHOD toString() : STRING

-- Returns the runtime class of the caller object.
PUBLIC METHOD FINAL getClass() : Class

-- Default hash function.
PUBLIC METHOD hashCode() : INT
```

String Class

The String class contains the following accessible members.

```
-- Compares 'this' object with the
--string str lexicographically.
PUBLIC METHOD compareTo(STRING str) : INT

-- Concatenates str to the end of 'this' string.
PUBLIC METHOD concat(STRING str) : STRING

-- Returns TRUE if 'this' string contains str.
PUBLIC METHOD contains(STRING str) : BOOLEAN

-- Returns TRUE if 'this' string ends with str.
PUBLIC METHOD endsWith(STRING str) : BOOLEAN

-- Returns the length of 'this' string.
PUBLIC METHOD length() : INT

-- Removes leading and trailing whitespace.
PUBLIC METHOD trim() : STRING
```

Dict

C♭ contains the SYS#Dict class which is a dictionary type used for mapping a key to a value [18, 22].

The Dict type contains the following members:

```
-- Returns TRUE if the map contains 'key'.
-- Otherwise returns FALSE.
PUBLIC METHOD containsKey(Object key) : BOOLEAN

-- TRUE if the map contains 'value'.
-- otherwise returns FALSE,
PUBLIC METHOD containsValue(Object value) : BOOLEAN

-- Returns the value mapped to the key
-- null if no such value exist.
PUBLIC METHOD get(Object key) : Object

-- Associates the specified value with the
-- Specified key.
-- The value returned is undefined.
PUBLIC METHOD put(Object key, Object value) : Object

-- Removes the mapping for the
-- specified key from this map, if present.
PUBLIC METHOD remove(Object Key) : Object
```

The put method returns an undefined value due to the semantics of C♯ and Java being different in the implementation of their dictionary type. Java returns the previous value associated with a key or null if no such value exist. C♯ does not return any value.

In general dictionary gave us some trouble implementing due to the semantic differences between the Java and C♯. In fact the dictionary type was the one which prompted us to create the runtime libraries, which is a library we will discuss later in this chapter.

Input

C♭ contains the input class which is meant for reading input from console and contains one method:

```
// Reads console input.
PUBLIC METHOD readLine()
```

FileIO

C♭ contains the FileIO type which provides easier ways to read and write from file.

```
-- Writes to file.
PUBLIC METHOD write(path : STRING, message : STRING)

-- Appends to file.
PUBLIC METHOD append(path : STRING, message : STRING)

-- Reads from file.
PUBLIC METHOD read(path : STRING) : STRING
```

FileHandler

C♭ contains the FileHandler class which is meant for manipulation of files.

```
-- Creates a new FileHandler instance.
PUBLIC METHOD FileHandler(path : STRING)

-- Creates a file.
PUBLIC METHOD create()

-- Determines whether a file exists or not.
PUBLIC METHOD exists() : BOOLEAN

-- Deletes a file.
PUBLIC METHOD delete()
```

FileWriter

C♭ contains the FileWriter type for writing to file. The FileWriter type contains the following accessible members.

```
-- Creates a new instance of FileWriter.
PUBLIC METHOD FileWriter(path : STRING)

-- Closes the stream.
PUBLIC METHOD close()

-- Flushes the stream.
PUBLIC METHOD FLUSH()

-- Appends the str to the end of the file.
PUBLIC METHOD append(str : STRING)

-- Overwrites File with str.
PUBLIC METHOD write(str : STRING)
```

FileReader

C♭ contains the FileReader type, which is used for reading files. The fileReader class contains the following accessible methods.

```
-- Creates a new FileReader instance.
PUBLIC METHOD FileReader(path : STRING)

-- Reads the file.
PUBLIC METHOD read() : STRING

-- Closes the file stream.
PUBLIC METHOD close()
```

ERR

C♭ contains the ERR type which is used to print to the error stream.

```
-- write the message to the error stream.
PUBLIC STATIC METHOD write(message : STRING)

-- Write the message on a new line on the error stream.
PUBLIC STATIC METHOD writeLn(message : STRING)
```

OUT

C♭ contains the OUT type which is used to print to the output stream.

```
-- write the message to the output stream.
PUBLIC STATIC METHOD write(message : STRING)

-- Write the message as a new line on
-- the output stream.
PUBLIC STATIC METHOD writeLn(message : STRING)
```

Exception

Exception class contain the following methods:

```
-- Returns the instance
-- that caused the current exception
PUBLIC getCause() : Exception

-- Returns a string description
-- of the current exception instance
PUBLIC getMessage() : STRING

-- Returns a stacktrace.
PUBLIC getStackTrace() : STRING
```

Furthermore CheckedExceptions and UncheckedExceptions both inherit these methods.

6.6.2 Standard Libraries Transformer

What we want to achieve is an automatic transformation which looks at every class name nonterminal and at every method invocation nonterminal and potentially transform certain properties in these nonterminals.

Let us consider an example: suppose you create a program containing an exception class E. You make E inherit from CheckedExceptions. After you complete the implementation, you serialize your program to Java. At this point you do not wish for E to inherit the C♭ class CheckedException but the actual checked exception class defined in Java: the class Exception. This is one of the scenarios which the standard library transformers are meant to solve. We need to transform certain class names and method invocations into the corresponding ones in the target language.

This stage of the pipeline can be viewed as an extension of the serializer. The serializer is not a compiler, it does not execute a program. It's a program which transforms a string s to some other string s′. To execute s′ one must use a C♯ or Java compiler. The same applies for this stage of the pipeline. We are transforming references to C♭ standard library to references in the target language. Although there are exceptions, we had to create wrappers for part of the C♭ standard library. These special cases will be discussed further in the section of "Runtime Libraries".

The implementation of the standard library transformer is straight forward although we need to consider a few cases. In our previous example we needed to change the class which E inherits from namely, CheckedException. The transformation done on CheckedException transforms the class name to Exception and the module to java.lang. To retrieve the corresponding name in the target language we created a map to every type in the target language.

```
CheckedException => {
  CName => "Exception",
  Module => "java.lang",
  getCause => "getCause",
  getMessage => "getMessage",
  getStackTrace => 'getStackTrace().toString'
},
```

Figure 6.3: CheckedException map to Java in, source code written in RC-script

CName corresponds to the class name in the target language. Furthermore we also change the module to the package/namespace in the target language which contains the type we wish to transform. That is because it is required by the automatic importer. We will explore that in details in section 6.7 .

The algorithm runs as this, we consider the cases of references to the standard library which we may transform. The cases are the following:

1. ClassVarMember

2. ObjectVarMember

3. CFlatClassName

Those are the nonterminal names for: class names, object (instance) members and class (static) members. We iterate over these nonterminals and simply check if they are contained in the standard library, if they are contained in the standard library then we transform the terminals corresponding to the new identifiers fetched from the map.

```
PROCEDURE ChangeClassReferenceToStdLib(CName)
BEGIN
  -- Check whether the class is in the std library.
  IF ClassMap![CName.ClassId.Image] <> VOID THEN
     ChangeModuleName(CName);
     ChangeClassName(CName);
  END;
END;
```

Figure 6.4: The procedure which changes class name references

Thus we transform the following:

```
PUBLIC CLASS SysUser#MyClass; INHERITS SYS#Exception;
ENDCLASS;
```

To the following:

```
public class MyClass extends Exception
{

} // End of class MyClass
```

6.6.3 Runtime Libraries

As mentioned earlier, certain parts of the standard library are not easy to map due to a difference in the semantics. The libraries might be extremely similar over both languages yet one library is using some feature which is not in the intersection of Java and C♯ . One example of a type that uses some feature not in the intersection is the Dictionary type of C♯ . Dictionary types in C♯ uses indexers which is a construct that is not in the Java programming language and thus not equivalent to Java's Dictionary type.

To remedy such issues we created the runtime libraries. The runtime libraries are wrappers which use the already implemented libraries in the target language in order to force the semantics to be equivalent. Going back to the dictionary type, this is how a value is retrieved in Java.

```
// dict is a Hashmap instance.
dict.get(k); // Where k is the key.
```

As mentioned earlier, the C♯ implementation of dictionaries uses indexers.

```
// dict is a Dictionary instance.
dict[k]; // Where k is the key.
```

Therefore we created these wrappers. These (wrapper) libraries would be bundled with every C♭ program in their own module. For the dictionary example we implemented the class Dictionary in the runtime library. It would contain an instance of Dictionary and then manipulate the operations associated with dictionary in order make the semantics correspond to Java's Hashmap type.

```
// C# dictionary type as a field variable
private Dictionary<Key, Value> Map;
```

Most methods were trivial to implement.

```
  // Returns true if the map contains the input key.
  public bool ContainsKey(Key key) {
    return Map.ContainsKey(key);
  }
```

Even some semantically different methods were trivial to implement. For instance this is the get method which would wrap the indexer property.

```
  // Returns the value associated with the key.
  public Value Get(Key key) {
    return Map[key];
  }
```

This way we solve the inequality of semantics, however there are downsides to such an implementation. For instance there is an overhead in performance. Overall the runtime libraries, although used very sparingly fulfilled their purpose.

6.7 Automatic Import

Java and C♯ uses Packages and Namespaces in order to create naming scope for project management. The usage of external libraries or other packages/namespaces requires importing them. In C♭ the equivalent to namespaces and packages are modules, which also requires import. Thus we decided to create a program which would automatically import modules which are not already imported.

Given a class c we want to import the modules of all classes used in c. Let us call the list of modules already imported in c for M. To expand M we do the following:

1. traverse c and fetch the list of CFlatClassName nonterminal in the subtree of c.
2. The module name is stored within the CFlatClassName nonterminal, thus one can easily generate a CFlatModuleName list from a CFlatClassName list.
3. Transform the list of CFlatClassName to a list of CFlatModuleName, let us call it M′.
4. Create M″ by concatenating M and M′.
5. Remove duplicates of M″.
6. Attach M″ as the ExternalModuleClause of c.

Knowledge of the C♭ grammar are needed in order to fully understand the implementation.

A CFlatClassName is the following:

```
CFlatClassName ::= CFlatModuleName '#' Ident [Generics]
```

A CFlatModuleName nonterminal is :

```
CFlatModuleName ::= { Ident } + BY '@'
```

An External Module clause is:

```
ExternalModuleClause ::=
  'EXTERNAL' 'MODULES' { CFlatModuleName } + BY ',' ';'
```

Finally an Ident is just an identifier, a string without the quotation marks.

With this we have examined all of the program transformations in the pipeline. What I particularly like about the pipeline is the ability to expand and come up with more stages. For example, an automatic transformer that takes procedural source code and transforms it to object oriented source code.

CHAPTER
SEVEN

VALIDATION

C♭ is an industry grade solution for software migration. If C♭ is chosen as the tool for a project migration, then it needs to work as intended. Thus we created a vast amount of tests, the only issue is that RCscript did not contain a unit testing framework. That lead us to be quite inventive in our testing. We created our own unit testing framework consisting of Powershell scripts as well as batch script, Java programs and C♯ programs.

7.1 Serializer Tests

As mentioned earlier, the C♭ project was a dormant project in the Raincode repositories for years before it was restarted. I didn't really know the design or syntax of C♭ or how to program in RCScript. Therefor I was told to write C♭ programs as tests to gain some intuition to the syntax of C♭ as well as the design. These tests are serializer tests and they remain till this day, although modified. The recipe for testing the serializer is the following.

Write a C♭ file f_1, serialize it Java and C♯ respectively producing j_1 and s_1, where j_1 is the Java serialization and s_1 is the C♯ serialization of f_1. Inventory all three copies of the file, i.e f_1, j_1 and s_1. These files will test an aspect of the language e.g expressions, statements or other constructs. Running the tests will run the serializer on f_1 producing j_2 and s_2, where j_2 is the java serialization of f_1 and s_2 is the C♯ serialization of f_1. We compare the files j_1 and j_2 as the Java test and we compare s_1 to s_2 as the C♯ test.

The purpose of this test is to find modifications in the serializer. If j_1 and j_2 are not equal then there has been some modification on the serializer. When we pushed changes to the serializer, we expected some of the serializer tests to fail. Furthermore when we needed new functionality in C♭ we would create new serializer tests after implementing the feature in the parser.

7.2 Semantic Analysis Tests

The semantic analysis procedures needed to be tested to ensure correctness. When working with with semantic analysis we used methodologies from test driven development. We wrote tests prior to the development of of the semantic analysis.

We would start by writing a pair of test files, one which would succeed and one which would fail. To mark a failure we printed out some error message and changed the exit code to -1. The procedure in Listing 7.1 signals a failure in the semantic analysis, it takes a message, that is a string containing some message, and a nonterminal node where the error occured.

This worked in conjunction with a powershell script. The powershell script would validate a file by taking the file, an exit code and a language parameter. If the file would run through the semantic analysis and return the expected exit code then the script would print out a success message written in green, otherwise it would print out a red failure message and write the expected exit code and the actual exit code.

The powershell script was a unit testing framework we created from scratch. This testing framework uses the so called functional testing [12] process where a single procedure is not the object being tested, but rather a certain functionality of the program as a whole.

Using exit codes helped determine programmatically whether or not the semantic analysis we were implementing was working as intended or not. Furthermore we could make it easier to visualize whether or not our tests were succeeding or not. Using the colours to output the test results made it very easy for the programmer to evaluate the test results.

Listing 7.1: VerificationFail procedure

```
-- Used to write verification errors.
-- Exit code -1 is used for errors
-- concerning the validation of the input.
PROCEDURE VerificationFail(message, nt)
VAR error;
BEGIN
   SYS.SetExitCode(-1);

   IF nt!LineNr <> VOID THEN

      error := ("Validation error L:" &
      nt.LineNr &
      " : " & nt.ColNr &
      " " & message);

      OUT.WriteLn(error);
   ELSE
      OUT.WriteLn(message);
   END;

   -- Exit the execution of the program
   -- at the first occurence of a semantic error.
   SYS.Exit;
END;
```

7.3 Target Language Unit Tests

Finally we wanted to be able to test the serialized source code in the target language using the semantic analysis implemented in the compilers of these languages. Thus we created Unit test frameworks in the target language using the reflections [21, 23] APIs in the target languages. This way we could catch errors not caught in the semantic analysis we implemented, we could test the semantics of the runtime libraries and finally we could catch errors we didn't expect.

More importantly, we could write tests in C♭, serialize the tests to Java or C♯ and then use this implementation of the testing framework to invoke the tests. This unit testing framework is compiled as libraries in the target language and bundled with every program serialized by C♭. Thus we can write C♭ tests that invoke this testing framework, serialize said tests and run them, yielding a testing framework in C♭.

We created a framework for each of the two target languages. These framework consists of three classes. The Assert class, the AssertException class and the main class. In the case of Java we needed an additional class which is the class which loads the input classes.

The Assert class contains a single static method assertEquals which takes an "expected" and an "actual" instances of the SYS#Object class, a string with the testname and returns a boolean value.

Recall SYS#Object serializes to the Object type in both Java and C♯ i.e the root class which all objects inherit from.

If the expected and actual values comparison fails that is, they are not equal then an AssertException is thrown and the unit test fails, otherwise the test succeeds.

The AssertException class provides three constructors which which are used to throw exceptions, an empty one, one which takes a string message and one which takes a string message and an exception. Another important fact to note is that the AssertException class inherits from the SYS#UncheckedException class which makes it an unchecked exception.

Finally the Main class simply loads a file using the reflections libaries and foreach class fetches all the method. Furthermore it filters out the methods such that only methods which are public, static, has return type boolean, does not contain any parameters and whose name starts with “test” remain.

```java
methods = methods.stream().filter(
    x -> Modifier.isPublic(x.getModifiers())
    && Modifier.isStatic(x.getModifiers())
    && (x.getReturnType() == boolean.class)
    && (x.getParameterCount() == 0)
    && x.getName().startsWith("test")
).collect(Collectors.toList());
```

Figure 7.1: The Java source code which filters out all but the test methods

Then for each method, that method is invoked. If the test fails then the name of the failed method is printed and an AssertException is thrown. Otherwise the test succeeds, printing a success message.

This testing framework was used in the testing of the runtime standard libraries. It worked as intented, providing a testing framework in the target language. We have a testing framework in C♭.

CHAPTER

EIGHT

CONCLUSION

Our time slaying dragons are coming to an end — we have explored a variety of topics within this thesis.

We have presented the design of C♭, a language designed to be within the intersection of Java and C♯, and how we handled constructs in C♭ which were not in the intersection. Furthermore we have presented the implementation of the serializer, a set of semantic analysis procedures, a set of program transformations for the purpose of improving the code quality and the testing framework for C♭.

During the introduction we asked a few research question. Throughout this thesis we have explored the C♭ project, thus we can finally give proper answers.

The question: Can we use program transformation to improve the quality of the serializer output?

Research Answer 1. *Yes, we have examined a set of program transformations that mark dead variables and methods, that hides data and procedures that helps the programmer by automatically detect not imported modules and undeclared checked exceptions. This set of program transformations is by no means final, the pipeline is built on such a way that we can expand on it in the future.*

The question: Is it feasible to design and implement a language entirely contained in the intersection of Java and C♯?

Research Answer 2. *No, we could not design a language entirely contained in the intersection of Java and C♯ for our purposes. There were constructs outside the intersection which we wished to include in C♭ such as checked exceptions which is contained in Java and not in C♯. Furthermore a set of constructs had semantic differences, such as generics. In Java, Generics require the type parameter to be a reference type where as type parameters in C♯ can be both reference and value types. Thus we defined C♭ generics to work in the same manner as "the strictest" target language, Java. However, we did end up with a language* mostly *in the intersection of Java and C♯.*

We will now ask questions which may be viable work for the future of C♭.

Future Work. *The creation of a serializer to C♭.*

This is the most obvious part of any future work. C♭ is a programming language which is designed to be used in software migration. Thus we may have to alter it for the sake of the source language and source program. Depending on the language we may have to add new constructs to C♭. To be tested properly we need real world usage.

Future Work. *The creation of a set of program transformations that transform a procedural program to an OO program.*

We can not guarantee that the generated C♭ code will be OO, which is a paradigm that both Java and C♯ are designed around. Having an automatic transformation which bundles data and procedures together and then proceed to transform them to a class could change the idiomaticity of the C♭ source code before the serialization to Java and C♯.
We may also try to use C♭ for language research:

Future Work. *C♭ provides us with an intriguing opportunity for exploring the semantics of Java and C♯. If we create a sufficiently wide range of tests for the C♭ language constructs, we should be able to see if the constructs in the intersection are actually semantically equivalent, or if there are corner cases where there are differences. For example, whether there are small differences in loop scoping.*

Finally, this concludes the thesis.

GLOSSARY

abstract syntax tree abstract tree representation of source code. 6, 10, 14, 15, 36–40, 44, 47, 53

algebraic data types Type system for composite data types often used in functional programming languages, includes the feature of sum types. 10

Checked exceptions All the exception classes that inherit from the Checked Exception class. 63

Data hiding The practice of hiding specific implementation details i.e, hiding variables or methods or even classes. 62

depth first search A graph traversal algorithm which traverses the full depth of the branch before backtracking and traversing alternate branches. 38, 39

domain specific language a language which specializes in solving problems within a particular domain. 10, 15

entry point A block of source code which is executed during the execution of a program. 55

intermediate representation language A language which is designed for the use of a compiler or interpreter, the content represents source code. 3

legacy code Source code that uses technologies that are no loger supported. 15

legacy languages Some language that are no longer supported. 15

metaprogramming Programs which consider other programs as data. 2

mutually recursive procedures A collection of procedures that call each other, Used a lot in parsing. Particularly in a recursive descent parser. 36

nonterminal Lexical elements of a language: Nonterminals are elements which can modified by the use of production rules. 10, 37, 47, 53, 59, 68, 70

semantic analysis A stage during compilation which checks for errors defined in the language specifications. 43

semantics the meaning of programs and programming phrases, i.e what happens at execution. 35

serializer A program that transforms code from a language to a target language.. 2, 3, 7, 11, 14, 35, 38, 53

source language The language we serialize from that is, in a source to source program transformation it is the language in which the input program is written in. 4, 7, 12, 39, 82

syntax The rules which specify the structure of the language. 35

target language The language which we serialize to, in C♭ it is either Java or C♯ . 3, 7, 10, 12, 13, 39, 53

terminal Lexical elements of a language: Terminals are the basic elements, literals which can not be modified by production rules. 10

type checking A process which analyzes the programs for type errors, those are errors concerning incorrect use of a type such as a conversion error. 43

Unchecked exceptions All the exception classes that inherit from the Unchecked Exception class. 63

Value Types Some datatype which contains a value within the memory space rather than holding a reference. 18

APPENDIX

A SERIALIZATION EXAMPLES

In this appendix we will present a few C♭ programs and their serialization to Java and C♯.

A.1

The following C♭ program:

```
MODULES Serializer;

CLASS Serializer#ClassMembers;
  PUBLIC METHOD ClassMemberAndArray;
    VAR a := a#b!c : INTEGER;
    VAR b := a#b!c[d] : BOOLEAN;
    VAR c := Car.wheel : WHE#EL;
    VAR d := Car.Wheel[1] : WHE#EL;
    VAR e := CA#R!c?WHEEL : INTEGER;
    VAR g := Car#Cars!SportCar[n] : ARRAY OF Car#Cars;
    VAR f : ARRAY OF INTEGER;
  ENDMETHOD;
ENDCLASS;
```

serializes to the following Java program:

```
package Serializer;

class ClassMembers
```

```
{

    public void ClassMemberAndArray()
    {
        int a = b.c;
        boolean b = b.c[d];
        EL c = Car.wheel;
        EL d = Car.Wheel[1];
        int e = R.c.WHEEL;
        Cars[] g = Cars.SportCar[n];
        int[] f;
    }

}  // End of class ClassMembers
```

and the following C♯ program:

```
using System;

namespace Serializer
{

class ClassMembers
{

  public void ClassMemberAndArray()
  {
      int a = b.c;
      bool b = b.c[d];
      EL c = Car.wheel;
      EL d = Car.Wheel[1];
      int e = R.c.WHEEL;
      Cars[] g = Cars.SportCar[n];
      int[] f;
  }

}  // End of class ClassMembers

} // End of namespace Serializer
```

A.2

The following C♭ program:

```
MODULES Serializer;

CLASS Serializer#Classes;

    PRIVATE VAR a := 5 : INTEGER;

    PUBLIC CONST b : STRING;

    CLASS Serializer#SomeOtherClass;

        PRIVATE CONST a := 5 : INTEGER;

        PUBLIC VAR b : STRING;

        CLASS AnotherModule#SomethirdClass;

            PRIVATE VAR a := 5 : INTEGER;

            PUBLIC VAR b : STRING;

        ENDCLASS;

        PUBLIC METHOD run;
            RETURN TRUE;
        ENDMETHOD;

    ENDCLASS;

ENDCLASS;
```

serializes to this java program:

```
package Serializer;

class Classes
{

    private int a = 5;

    public String b;

    class SomeOtherClass
    {

        private int a = 5;

        public String b;

        class SomethirdClass
        {

            private int a = 5;

            public String b;

        }  // End of class SomethirdClass

        public void run()
        {
            return true;
        }

    }  // End of class SomeOtherClass

}  // End of class Classes
```

and this C♯ program:

```
  using System;

namespace Serializer
{

class Classes
{

    private int a = 5;

    public readonly string b;

    class SomeOtherClass
    {

        private readonly int a = 5;

        public string b;

        class SomethirdClass
        {

            private int a = 5;

            public string b;

        }  // End of class SomethirdClass

        public void run()
        {
            return true;
        }

    }  // End of class SomeOtherClass

}  // End of class Classes

} // End of namespace Serializer
```

A.3

The following C♭ program:

```
MODULES Serializer;

CLASS Serializer#IndexedExpr;

    PUBLIC METHOD IndexedExpr;
        VAR a := A[3] : STRING;
        VAR b := A["test"] : INTEGER;
        VAR d := A[B[C[D[E]]]] : BOOLEAN;
    ENDMETHOD;

ENDCLASS;
```

serializes to the following Java program:

```java
package Serializer;

class IndexedExpr
{

    public IndexedExpr()
    {
        String a = A[3];
        int b = A["test"];
        boolean d = A[B[C[D[E]]]];
    }

}  // End of class IndexedExpr
```

and serializes to the following C♯ program:

```csharp
using System;

namespace Serializer
{

class IndexedExpr
{

    public IndexedExpr()
    {
        string a = A[3];
        int b = A["test"];
        bool d = A[B[C[D[E]]]];
    }

}  // End of class IndexedExpr

} // End of namespace Serializer
```

For more, C♭ files and implementation please see the within the cflat folder provided with this thesis.

BIBLIOGRAPHY

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers Principles Techniques & Tools Second Edition*. Pearson Education Inc, 2006.

[2] J. Ashkenas et al. List of languages that compile to JS. `https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js`, 2018.

[3] L. Beyak. SAGA: A Story Scripting Tool for Video Game Development. Master's thesis, Department of Computing and Software, McMaster University, April 2011.

[4] D. Blasband. The YAFL programming language. *JOOP*, 8(7):42–49, 1995.

[5] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: Improving the effectiveness of parallelizing compilers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 141–154, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. ISBN 978-3-540-49134-7. doi: 10.1007/BFb0025876.

[6] J. M. Boyle. Lisp to Fortran—Program Transformation Applied. In P. Pepper, editor, *Program Transformation and Programming Environments*, pages 291–298, Berlin, Heidelberg, 1984. Springer Berlin Heidelberg. ISBN 978-3-642-46490-4.

[7] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the Future Safe for the Past: Adding Genericity to the Java

Programming Language. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 183–200, New York, NY, USA, 1998. ACM. ISBN 1-58113-005-8. doi: 10.1145/286936.286957.

[8] Computerworld. Java 8 Language specification. `https://www.computerworld.com/article/2502430/data-center/cobol-brain-drain--survey-results.html`, 2012.

[9] J. Costabile. GOOL: A Generic OO Language. Master's thesis, Department of Computing and Software, McMaster University, April 2012.

[10] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, May 2014. ISBN 9780133900767.

[11] T. Hasu and M. Flatt. Source-to-source compilation via submodules. In *Proceedings of the 9th European Lisp Symposium on European Lisp Symposium*, ELS2016, pages 7:56–7:63. European Lisp Scientific Activities Association, 2016. ISBN 978-2-9557474-0-7.

[12] W. E. Howden. Functional program testing. *IEEE Transactions on Software Engineering*, SE-6(2):162–169, March 1980. ISSN 0098-5589. doi: 10.1109/TSE.1980.230467.

[13] IEEE. *IEEE Standard for Floating-Point Arithmetic (IEEE Std 754-2008)*. IEEE Computer Society, Aug. 2008.

[14] E. Illushin and D. Namiot. On source-to-source compilers. *International Jounal of Open Information Technologies*, 4(6):48–51, 2016. ISSN 2307-8162.

[15] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical Interprocedural Parallelization: An Overview of the PIPS Project. In *Proceedings of the 5th International Conference on Supercomputing*, ICS '91, pages 244–251, New York, NY, USA, 1991. ACM. ISBN 0-89791-434-1. doi: 10.1145/109025.109086. URL `http://doi.acm.org/10.1145/109025.109086`.

[16] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. Müller, and J. Mylopoulos. Code Migration Through Transformations: An Experience Report. In *Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '98, pages 13–. IBM Press, 1998. URL http://dl.acm.org/citation.cfm?id=783160.783173.

[17] S. Marlow. Haskell 2010 Language Report. https://www.haskell.org/onlinereport/haskell2010/haskellch5.html, 2010.

[18] Microsoft. C# Class Libraries. https://msdn.microsoft.com/en-us/library/xfhwa508(v=vs.110).aspx, 2017.

[19] Microsoft. C# 6.0 draft language specification, 2017. URL https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/.

[20] Microsoft. C# Exception Class documentation. https://msdn.microsoft.com/en-us/library/system.exception(v=vs.110).aspx, 2017.

[21] Microsoft. C# Reflection API Documentation. https://msdn.microsoft.com/en-us/library/system.reflection(v=vs.110).aspx, 2017.

[22] Oracle. Java SE 8 Collections API, Map Interface. https://docs.oracle.com/javase/8/docs/api/java/util/Map.html, 2014.

[23] Oracle. Java SE 8 Reflection API Documentation. https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/package-summary.html, 2014.

[24] Oracle. Java 8 Throwable class documentation. https://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html, 2017.

[25] J. Palsberg and C. B. Jay. The Essence of the Visitor Pattern. In *22nd Intl. Computer Software and Applications Conf. (COMPSAC '98)*, pages 9–15. IEEE Computer Society, 1998.

[26] Stack Overflow. Developer Survey 2017. https://insights.stackoverflow.com/survey/2017#most-popular-technologies, 2017.

[27] A. A. Terekhov and C. Verhoef. The realities of language conversions. *IEEE Software*, 17(6):111–124, Nov 2000. ISSN 0740-7459. doi: 10.1109/52.895180.

[28] E. Visser. Program Transformation with Stratego/XT. In *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*, pages 216–238. Springer, 2004. ISBN 978-3-540-25935-0. doi: 10.1007/978-3-540-25935-0_13.

[29] V. Zaytsev and A. H. Bagge. Parsing in a broad sense. In J. Dingel and W. Schulte, editors, *Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS'14)*, volume 8767 of *LNCS*, pages 50–67. Springer, 2014.