Design & Implementation of Separate Compilation for Encore

Jonas Olander
Abstract

Design & Implementation of Separate Compilation for Encore

Jonas Olander

Separate compilation can bring several benefits to a module system such as allowing modules to be pre-compiled and then linked with different programs on the same system, which can result in shorter compilation times for repeated compilations of a program. Encore is a programming language that currently does not support separate compilation of modules. This project presents a design and implementation of separate compilation support for the Encore module system which leverages the fact that Encore compiles to C. The problems and challenges that were faced and their solutions are presented alongside some of the design choices that were made. The design and implementation were then evaluated with a focus on compiler performance and correctness. The final result of the project is an implementation of separate compilation that shows promising results when it comes to decreasing the compilation speed of Encore programs, but it requires some optimizations and future work, especially when it comes to improving usability.
Contents

1 Introduction ........................................... 4
   1.1 Purpose ........................................ 5
   1.2 Goal of The Project ............................. 5

2 Background and Related Work .......................... 5
   2.1 Modules in Encore ............................... 6
   2.2 Compiling Encore ............................... 8
      2.2.1 The Encore Compiler ....................... 10
   2.3 C Representation ................................ 10
      2.3.1 Naming of C Identifiers .................. 12
      2.3.2 Traits .................................... 12
   2.4 Separate Compilation in C ....................... 13
   2.5 Module Systems in Other Languages ............. 13
      2.5.1 The Haskell Module System ............... 15
      2.5.2 Go Packages ................................ 17
      2.5.3 Rust Crates and Modules ................. 19

3 Design & Implementation of Separate Compilation for Encore ....... 20
   3.1 Compiling and Linking a Module ................. 21
   3.2 Implementation Challenges and Design Choices .... 23
      3.2.1 Naming of C Identifiers .................. 23
      3.2.2 Trait Polymorphism ....................... 24
      3.2.3 Detecting Compiled Modules ............... 24
      3.2.4 Handling Imports .......................... 26
      3.2.5 Storing Information About Exported Identifiers 26
      3.2.6 Circular Dependencies .................... 27
      3.2.7 Resolving Module Dependencies ............ 27
      3.2.8 Resolving Name Conflicts Between Libraries 29
   3.3 Changes Made to the Compiler .................... 29
      3.3.1 Structural Changes ....................... 30
      3.3.2 Changes to Code Generation ............... 30

4 Evaluation ........................................... 32
   4.1 Performance ..................................... 32
      4.1.1 Compilation Speed ........................ 32
      4.1.2 Runtime Performance ....................... 34
   4.2 Testing and Validation ......................... 36

5 Conclusion ........................................... 37
   5.1 Future Work .................................... 38

Bibliography ........................................... 40
1 Introduction

Modular programming is defined as a style of programming in which a program is separated into a collection of components (modules), each of a manageable size, with a clear purpose and a well-defined interface [1]. Often this takes the form of splitting the program into multiple different source files that each contain definitions for some related type of functionality, usually with the possibility to hide parts of the implementation (information hiding) [1]. The source files are then combined during the compilation process to form a complete program.

Splitting the program into modules contained in separate files brings some benefits over the opposite where the smallest part of a program is the whole. It makes the code easier to read and to reason about since each source file will contain less code. It also makes the code easier to maintain since a change to a module need not affect the rest of the program and it promotes code reuse since other programs could potentially use the module as well.

Many modern programming languages implement support for modular programming in some way, usually in the form of a module system that allows the programmer to structure the code into reusable components, often contained in different files. A concept which is related to modular programming and module systems is modular linking. With modular linking a module is compiled and statically type checked [2] separately and then linked [3] together with other modules to form a complete program [4].

Some of the benefits that modular linking brings to a module system are:

- The programmer only have to recompile the parts of a program that have changed since the last compilation which may result in shorter compilation times for large programs, especially over time.

- Modules can be pre-compiled and linked with different programs on the same system.

- It decouples modules from the programs that use them, making them simpler to test and making it easier to change parts of a program without affecting the rest.

- It makes it easier to distribute code across multiple systems.

Modern programming languages such as Go [5], Haskell [6] and Rust [7] support modular linking as a part of their module systems. Section 2.5 contains descriptions of their respective module systems.

Encore is an object-oriented and actor-based programming language that uses C [8] as an intermediate compilation target [9]. Encore supports modular programming with its module system. However, the module system does not support modular linking in its current form. This thesis presents a design and implementation of modular linking support for the Encore module system that leverages the fact that Encore compiles down to C, a language that supports separate compilation.
1.1 Purpose

The purpose of this thesis is to design and implement modular linking support for the module system in Encore by enabling separate compilation and linking of modules.

1.2 Goal of The Project

The primary goal of the project is to improve programmer efficiency by reducing the compilation time for repeated compilations of large Encore programs. Laying a foundation for future improvements to the module system is also considered important. Special consideration is taken to ensure that the implementation is backward compatible with the current version of Encore. The implementation focuses on enabling separate compilation for modules but does not include any solution for making sure that compiled modules are kept up to date or to recompile modules when one of its dependencies have changed.

2 Background and Related Work

Encore is a programming language in development at Uppsala University in Sweden [9]. It is a research language with a focus on making parallel programming easier for the programmer. The type system eliminates the possibility of Data races, and all parallelism is handled behind the scenes, letting the programmer focus on what the program should do instead of managing threads and synchronizing data access. Encore is a statically typed, compiled, object-oriented and garbage collected language with support for functional programming techniques and concepts such as pattern matching and lambda functions.

Similarly to other object-oriented languages such as Java and C++, Encore use classes as templates for creating objects. Each Encore program defines a Main class with at least one method main, serving as the entry point for the program. Listing 1 shows a bare-bones definition of the class Main.

```en
active class Main
  def main() : unit
  ...
end
end
```

Listing 1: Encore Main Class

Encore does not support inheritance between classes, meaning that there is no class hierarchy. Instead, traits are used to provide default method definitions and to specify fields and methods to be defined by all classes that include the trait. Traits enable trait polymorphism: a method or function can specify that it accepts any object whose class include a particular trait and this object is then assumed to have access to the fields and methods defined in it. A class can include any number of traits, and the order of inclusion does not matter.
Listing 2 shows an example of how traits are defined and used in Encore. The *Student* class includes the *BasicMath* trait, inheriting the *addition* method and implementing its own version of the *multiplication* method. In the *main* method a new instance of the *Student* class is created, and the trait methods are then used to perform addition and multiplication.

Listing 3 shows an example of trait polymorphism in Encore. The *ClassRoom* class defines a method *doMult* which takes an argument called calculator of type *BasicMath*. Any class that include the *BasicMath* trait can take on the role of the *BasicMath* type.

```scala
read trait BasicMath
  require
def multiplication(a : int, b : int) : int
    a * b
end

def addition (a : int, b : int) : int
  a + b
end
end

read class Student : BasicMath
  def multiplication(a : int, b : int) : int
    a * b
end
end

active class Main
  def main() : unit
    val student = new Student
    val sum = student.addition(5, 5)
    val prod = student.multiplication(5, 5)
    print(sum) -- 10
    print(prod) -- 25
end
end
```

Listing 2: Encore Trait definition and Usage

```scala
active class ClassRoom
  def doMult(calculator : BasicMath, x : int, y : int) : int
    calculator.multiplication(x, y)
end
end
```

Listing 3: Encore Trait Polymorphism

### 2.1 Modules in Encore

A module in Encore corresponds to a single Encore source file in the file system. Modules provide a way to group related functionality in a separate file that can be
imported by other modules and programs. This promotes code reuse and makes building large programs more manageable.

Every module starts with a module declaration at the top of the file. A module declaration consists of the keyword `module` followed by the module name, ending with an optional list of exported identifiers. Module names in Encore start with a capital letter and must match the name of the source file that the module is declared in, not considering the file extension. For example, the file `String.enc` would contain a declaration of the module `String`.

By default, modules export everything defined in it, but it is possible to control what the module exports by providing a list of identifiers in the declaration. If a list is provided then only the identifiers included will be visible to other modules and programs that imports the module. Listing 4 shows the two different ways of declaring a module. In Version 2 only the two identifiers `exportedIdentifier` and `otherExportedIdentifier` would be usable by other modules and programs whereas in Version 1 everything is exported.

```
-- Version 1
module String
...

-- Version 2
module String (exportedIdentifier, otherExportedIdentifier)
...
```

Listing 4: Encore Module Declarations

Importing a module gives access to it in another file. The import declaration is used to import a module. Encore provides several options regarding imports, but at its core, an import declaration consists of the `import` keyword followed by an import path. Import paths are a sequence of names separated by dots, ending with the name of the module to import. The dots correspond to directory separators, and the whole path translates to a relative path used to locate the module. For example, the import path `A.B.C` translates to the relative path `A/B/C.enc` where the file `C.enc` would contain the requested module `C`. The Encore compiler has a set of standard paths to use as the base when searching for imported modules. The directory containing the file targeted for compilation, and the path to the base directory of the standard library are the two paths that are used by default. A compiler flag can be used to add additional base paths. There is no requirement for having unique module names in Encore, but their import paths must be unique. Listing 5 shows an example of two different ways the `Math` module can be imported depending on its location relative to a base path.

```
import A.B.C.
```

Listing 5: Importing a Module

Two modules are imported implicitly in every Encore program, the `String` module, and the `Std` module. The `String` module provides the `String` class which is used to represent strings in Encore, and the `Std` module currently provides some standardized Traits such as `Eq` and `Id`. Importing an Encore module brings its exported identifiers into the local scope. Name conflicts can occur if two imported modules exports identifiers with the same name. A qualified import can be used to resolve such name conflicts. By adding the `qualified` keyword before the import path in an
import declaration, the names of that module will be qualified with the full import path. To avoid having to type the full import path to access identifiers imported using a qualified import it is possible to give an alias to the module. Using the keyword as followed by the new name effectively renames the module locally, and the new name will act as the qualifier instead of the import path. Listing 6 shows an example where the modules C and D both export a class named ExportedClass. The module C is then imported using a qualified import and D is imported using a qualified import and an alias. To resolve the name conflict only one of the imports would have to be a qualified import.

Encore also supports controlling what to bring into the local scope by including a list of identifiers to filter against in the import declaration. It is also possible to exclude specific identifiers by using the hiding keyword. Listing 7 shows an example of this.

2.2 Compiling Encore

The process of compiling an Encore program can be divided into three major steps. During the first step, the Encore compiler processes the Encore code and then emits a set of source files in the C programming language [8]. After translating the Encore code, a C compiler framework such as Clang [10] or GCC [11] is used to compile the generated C-code, creating a set of object files [3]. The last step consists of using a linker to link the object files representing the Encore program with the Encore Runtime (EncoreRT) which is a collection of libraries written in C [12]. The final result is an executable file. Figure 1 shows a diagram which gives an overview of the Encore compilation process.

![Encore Compilation Process](image.jpg)
module C

read class ExportedClass
...
end

module D

read class ExportedClass
...
end

--File: main. enc
--ExportedClass from C is now referred to as A.B.C.
ExportedClass
import qualified A.B.C
--ExportedClass from D is now referred to as NewName.
ExportedClass
import qualified A.B.D as NewName
...

Listing 6: Qualified Imports in Encore

--Only the identifier named ExportedIdentifier will be imported.
import Math (ExportedIdentifier)

--Everything except the identifier named ExportedIdentifier will be imported.
import Math hiding (ExportedIdentifier)
...

Listing 7: Controlled Imports in Encore
2.2.1 The Encore Compiler

The compiler for the Encore programming language is called encorec. It is a command line tool written in Haskell that takes an Encore source file as its input and outputs C-code. The compiler handles the full compilation process, so there is no need for a user to execute a C compiler manually for the generated code.

Internally the compiler must go through a series of steps before generating any C-code. The first thing that happens is that it parses the targeted Encore source file. During parsing, it looks for syntax errors while building an in-memory representation of the code in the form of an Abstract Syntax Tree (AST)[2].

If the compiler does not detect any errors during parsing, then it proceeds to the next step which involves importing modules. The compiler looks for modules recursively while parsing each module in the same way as in the previous step resulting in a collection of AST’s.

In the next few steps, the compiler performs a series of operations on the AST’s such as type checking and optimization [2]. During these steps the AST’s are checked to make sure that they each represent a correct Encore program, and this includes making sure that imported modules are used correctly.

Before generating any C-code, the compiler combines the AST’s into one big AST that represents a complete and correct Encore program.

2.3 C Representation

The output of the Encore compiler is a set of C source files. Every Encore class is represented by its own C file using the naming scheme Class Name.encore.c. Since each Encore program has a Main class, there will always be a file called Main.encore.c. There is also another class file that the compiler generates for every program, and that is the file representing the Encore String class, defined in the String module. The String module is imported into every Encore program implicitly, and it enables the use of strings and provides methods to operate on them. In addition to the class files, two other files will always be generated for every program, header.h and shared.c.

The shared.c file contains the code which could potentially be used anywhere in the program such as global functions. It also contains the main function serving as the entry point to the C version of the program.

The header.h file is a C header file [5]. It contains declarations and type definitions for most of the generated C identifiers in the program, and it is included in all of the other generated files using the #include preprocessor directive[5]. The header file is what ties the rest of the program together by providing a common interface to use when accessing something defined in another file. The header file also includes header files belonging to the Encore runtime libraries, and this means that the rest of the generated C files also has access to their internals due to the mechanics of the #include directive.
Figure 2 shows the files generated for a bare-bones Encore program. The arrows in the figure represent the use of the \#include directive. Figure 3 shows an in-depth overview of the full compilation process for the same program.
2.3.1 Naming of C Identifiers

The generated C identifiers are named using a combination of predefined values and the path to the file where its Encore counterpart was defined. Some examples of what the predefined values can include are information about what type of Encore identifier it represents and the name of the Encore identifier.

The path is the same that was used to locate the file during the Encore compilation process. For the file targeted for compilation, this is simply the path given to the compiler. Usually, the compiler is invoked in the same folder as the targeted file with just the name of the file given to the compiler as the argument. For modules, it is the same path that was used to locate the module during the compilation process. Depending on the base path used, this can be either a relative path or an absolute path.

After determining the path, the compiler converts it into a valid C identifier name by replacing directory separators and dashes by underscores and by removing the extension. Listing 8 shows an example of a class named Calculator located in the file Utils/Math.enc and a type definition of a C struct in the header file. The struct _enc_class_Utils_Math_Calculator_t is the C equivalent object type of the Encore Calculator class, and the compiler determines its name using a combination of predefined values and the path used to locate the Math module.

```
--File: Utils/Math.enc
module Math
read class Calculator
...
end

//File: header.h
typedef struct _enc_class_Utils_Math_Calculator_t ...
```

Listing 8: Naming of a C Identifier

2.3.2 Traits

Since Encore supports trait polymorphism, it is not always possible to know at compile time which implementation of a trait method that should be used. For each class that includes a trait, new implementations of the trait methods will be created, and if the class of an object is not known then there is no way to determine which implementation to use beforehand. A technique called dynamic dispatch can be used to solve this problem.

With dynamic dispatch, the implementation of a method call is determined at runtime. There are different ways to implement this type of behavior. For example, in C++ a mechanic known as a virtual table (vtable) is used.

In Encore, a lookup function is used to implement dynamic dispatch for polymorphic trait method calls. Each C representation of an Encore class defines a static function called the trait_method_selector which takes an integer as its input and returns a
pointer to a function representing a trait method implementation for that class. All
trait methods are given unique IDs through the use of a C enum type in the header
file. These IDs are then used in the lookup function to create a mapping between
an id and the corresponding implementation. When performing a polymorphic trait
method call, the lookup function of that object can be used to retrieve a pointer to
the correct method implementation.

Listing 9 shows the lookup function and the trait id definitions for the Student class
and the BasicMath trait defined in listing 2.

2.4 Separate Compilation in C

In the C programming language, a program consists of one or more translation
units that are processed and compiled separately [15]. The C standards define a
translation unit as a source file that has passed through the C-preprocessor. After
preprocessing, the contents of every file included using the #include directive are
now considered as being part of the file itself as if it the contents of the included file
had been copied and pasted.

Compiling a translation unit results in the creation of an object file. This object file
is then linked with other object files at a later stage to form a complete program. It
is also possible to combine several object files into a library and then use the library
during linking instead of each separate file.

To use an entity defined in another translation unit a matching declaration must be
provided. Declarations are typically placed in header files to avoid errors [16] but also
as a means of providing a public interface for a translation unit. Listing 10 shows an
example of this where the file addition.c contains a definition of a function named
performAddition and the header file addition.h provides a matching declaration.
The file main.c then includes the header file and the performAddition function can
now be used, given that the two translation units are linked together.

During linking, the linker processes each translation unit while noting declarations
without a matching definition. The linker then tries to resolve missing definitions
in one translation unit by checking if another provides it. If the linker does not find
a matching definition for some declaration, it will report an error. Exactly how the
linker resolves the missing definitions depend on the implementation. Some linkers
care about the order in which the translation units are provided as arguments.
For example, some linkers can only resolve missing definitions in a left to right
order.

2.5 Module Systems in Other Languages

There are many different ways to implement a module system for a programming
language. The programming languages Haskell, Go and Rust each have their own
take on what a module is and how they should interact with each other. Their
respective module systems are described in this section with the purpose of providing
static void* trait_method_selector(int id)
{
    switch (id)
    {
    case ENC_MSG_main_BasicMath_multiplication:
    {
        return enc_method_main_Student_multiplication;
        break;
    }
    case ENC_MSG_main_BasicMath_addition:
    {
        return enc_method_main_Student_addition;
        break;
    }
    .
    .
    .
    default:
    {
        printf("error, got invalid id: \%d", id);
    }
    }
    return NULL;
}

Listing 9: Dynamic Dispatch for Trait Methods in Encore
Listing 10: C Declarations and Usage in Other Files

a solid background and motivation for some of the design decisions made in the project and for the ideas on future work and optimizations seen in section 5.1.

2.5.1 The Haskell Module System

The implementation of modules in Haskell varies between different Haskell compilers but for this thesis only the Glasgow Haskell Compiler (GHC) [17] will be considered. When using GHC every Haskell source file will start with a module declaration consisting of the keyword `module` followed by the module name and an optional list of exported identifiers, ending with the keyword `where` which indicates that the module body is starting. If a module definition is left out then a declaration of a module named Main will be inferred, acting as the root of the Haskell program. All modules except the Main module must be defined in a file with the same name, not including the file extension.

Module names must start with a capital letter. Dots in a module name correspond to directory separators, and this means that a module named A.B.C would correspond to the file A/B/C.hs on a Unix system, relative to some base directory. Listing 11 shows a definition of the Strings module with a list of exported identifiers enclosed in parenthesis.

Listing 11: Haskell Module Definition

The default behavior of a module is to export all identifiers defined in it. To control what should be accessible from other modules a list of exported identifiers can be added to the module declaration. To gain access to the contents of another module the `import` statement is used. Writing `import` followed by the module name grants
access to the exported identifiers of that module. All import statements must be placed at the top of the source file beneath the module declaration.

Module namespaces in Haskell are flat, meaning that when a module is imported, its identifiers are brought into the local scope and they can be used as if they were defined there. Conflicts can occur if two imported modules export identifiers with the same name. The problem can be solved by using a qualified import for one or both of the modules. Adding the keyword qualified before the module name in an import statement will prefix the names of the exported identifiers of that module with the module name. Listing 12 shows an example of this where the imported modules A and B both export an identifier called exportedIdentifier. The conflict is in this case solved by qualifying the import of module B.

```haskell
module A (
    exportedIdentifier
    ...
) where
...

module B (
    exportedIdentifier
    ...
) where
...

module Main
import A
import qualified B
-- The identifier exportedIdentifier from module B is now
-- called B.exportedIdentifier.
...
```

Listing 12: Haskell Qualified Imports

GHC supports separate compilation of source files containing a module declaration [17]. Compiling a module generates an object file and an accompanying interface file. The object file can then later be linked with other object files to form a complete program. The interface file contains information needed by the compiler when the module is imported by another module and can be seen as a compiler readable version of the object file. It includes data type information, function definitions, function type signatures and other things needed by the compiler to determine the contents and correct use of the module. If optimizations are enabled, then additional information will be added to the interface.

To help keep track of when recompilation of a module is needed, GHC assigns a sort of version number in the form of an MD5 hash to the interface file and all the signatures contained in it. A module interface file also stores a list containing the version numbers of the module’s dependencies. To determine if a module should be recompiled GHC first checks if the source file has been modified by comparing its modification date to that of the object file. When the modification date of the source file is more recent, then the module will be recompiled. If the module source file has
not changed, then GHC compares the version numbers of the module’s dependencies stored in the interface to the current version numbers of those modules. If there is a mismatch, then the module will be recompiled. If nothing has changed then the current object file and interface file is left unchanged.

GHC does not directly support compiling mutually recursive modules. If two modules import each other, then the compiler will try to locate a non-existing interface file and fail. To resolve this an interface file can be hand written for either module, this file, also called a boot file only have to contain the minimum amount of information needed by the other module in the mutually recursive relationship.

2.5.2 Go Packages

The Go module concept is called a package. All Go source files start with a package declaration consisting of the keyword *package* and the name of the package [5]. Package names do not need to be unique across all packages used in a program, except the package called main which serves as the starting point of an executable program.

Instead of identifying packages with a unique name as seen in 2.5.1 a package is identified by a unique string called the import path. Each import path can be used to identify a folder in the file system, and all Go source files located in the folder are assumed to be part of the package.

Import declarations are used to gain access to the contents of a package from another. When importing a package, the *import* keyword is used, followed by the import path or a list of import paths. Import declarations must be put at the top of the file below the package declaration. Listing[13] shows a declaration of the *strings* package followed by an import declaration listing three packages to import. Note that by convention the package name is the last part of the import path.

```go
package strings

import (
    "fmt"
    "encoding/json"
    "github.com/somelib/lib"
)
```

Listing 13: Go Package Declaration and Imports

To avoid conflicts between identifiers defined in different packages each package name defines a distinct namespace which is used to qualify the identifiers it encloses. If two packages with the same name are imported, then one can be renamed. Listing[14] shows an example of a Go source file where two packages with the same name are imported. In this case the second *lib* package is given the name *libNrTwo* in order to solve the name conflict.

It is possible to control which identifiers should be visible to other packages by defining a public interface for the package. By default, all identifiers whose names start with a capital letter will be exported from the package and accessible from the
outside. Listing 15 shows a package where two functions are defined, one which will be exported and one that is only accessible from within the package itself.

```
package strings

func Exported() {
    ...
}

func notExported() {
    ...
}
```

Listing 15: Encapsulation in Go packages

One of the design goals of the Go language is that both large and small programs should compile quickly [18]. Go’s implementation of modular linking and handling of packages reflects this. Go supports modular linking by the notion of installing a package. When installing a package, its source files are compiled and combined into an object file. This object file is read by the compiler when the package is imported and later linked with other object files. If one of its source files or one of the packages it depends on changes, then a package will be recompiled.

Information about what the package exports and what packages it depends on are recorded at the top of the object file. In addition to its own export information, the information for all packages that it depends on is also included. By including the export information of packages that it uses the number of files that the compiler have to read and process is kept to a minimum, and this helps to keep the compilation time down for large programs.

Other design choices regarding packages that help with keeping the compilation time down include that unused imports are considered a compile-time error and that circular imports are prohibited. By treating unused imports as an error, the compiler ensures that only code in use gets processed. Not allowing circular imports helps with keeping the complexity down and reduces the number of source files that the compiler have to deal with at a time.
2.5.3 Rust Crates and Modules

The module system in Rust is implemented using two main concepts, crates and modules [19]. Crates are individual units of compilation similar to the Go packages described in section 2.5.2 and can be compiled into either an executable or a library.

Every crate consists of an implicit root module contained in a Rust source file. The root module contains the Rust code for the crate but can also define a set of sub-modules which together with the root forms a tree. Figure 4 shows an example of the module tree for a crate called strings. In this example, the root module defines a set of sub-modules in the form of the translation module, the format module, and the utils module. The translation module also defines two sub-modules of its own, the english module and the french module. The examples found in this section are modified versions of the examples found in the Rust documentation [19].

![Figure 4: Rust Module Tree](image)

When defining a module, the `mod` keyword is used followed by the module name and the contents of the module enclosed by square braces. The code for the module is placed within the brackets, and this is also where additional sub-modules are defined. Referring to a sub-module is done using double colons(::), for example, the code `strings::translation::english` would be used to access the english module in the previous example.

Rust also supports defining modules in separate files to avoid having to put all of the code in a crate in a single source file. By replacing the square braces with a semicolon in the module definition, Rust expects to find the contents of the module in a Rust source file with the same name as the module or in a file called `mod.rs` located in a sub-folder named after the module.

By default, identifiers defined in a crate are private, meaning that they cannot be used outside of the crate. Rust supports defining a public interface for a crate with the `pub` keyword. Writing `pub` before the definition of a module or any other Rust identifier makes it publicly available outside of the crate. Each module defines its own public interface, and making a module public does not automatically make the identifiers and sub-modules contained in it public as well.

Compiling a library crate makes it available to be imported by other crates. To gain access to the public interface of a crate in another file the `extern crate` dec-
laration is used followed by the name of the crate. When referring to an identifier from the imported crate, the full path is used as the qualifier. For example, `strings::translation::english::someIdent` refers to the identifier with the name `someIdent` defined in the `english` module from the previous examples. Avoiding such long names are possible, there is support for importing modules into the local scope with the `use` keyword. Importing a module with `use` makes it possible to refer to its public identifiers with a shorter name. Listing 16 shows a Rust source file where the `strings` crate is imported followed by an import of the `english` module.

```
extern crate strings

use strings::translation::english
// Identifiers from the english module can now be
// referred to with english::identifier

// Identifiers from the french module needs
// to be fully qualified using strings::translation::french::
// identifier
```

Listing 16: Importing a Rust Crate and Modules

Rust crates can be compiled separately and then linked together when building an executable crate. Building an executable create can either be done manually by calling the Rust compiler for each crate and then calling the linker with the compiled libraries as arguments or by using a build tool called Cargo that comes with the Rust installation. Dependencies on other crates can be specified in a configuration file and Cargo will then take care of compiling and linking each crate after downloading it if needed.

### 3 Design & Implementation of Separate Compilation for Encore

The design of separate compilation support for the Encore compiler is based around the fact that Encore compiles down to C. The C compiler frameworks support separate compilation, and this is in fact already utilized in the Encore compilation process. The Encore runtime libraries are written in C, compiled when a user installs Encore on the system, and then linked with the compiled C-code of an Encore program. A similar approach could work when implementing separate compilation for modules. Modules can be pre-compiled and then later linked with the programs that import them. When compiling a program that imports a compiled module there would be no need to generate and compile any C code for that module, this could improve the compilation speed of all programs using it.

Figure 5 shows an overview of an updated Encore compilation process with support for separate compilation of modules.
### 3.1 Compiling and Linking a Module

Compiling a module is not that different from compiling a regular Encore program. Internally in the compiler, both are treated the same way, and the only real difference is that a module does not contain a `Main` class. The C representation of a module would, therefore, be very similar to the current representation of an Encore program.

The main difference lies in what happens after the Encore compiler has generated the C-code for the module. When compiling a regular program, the C source files would be compiled into object files and passed to the linker to create an executable. When compiling a module, there is no need to create an executable file. Instead, a tool such as the Unix archiver utility (ar) can be used to combine the object files into an archive, also known as a static library. A static library is simply a collection of object files contained in a single file. When another program imports the module, the linker can be passed the library as an argument instead of the individual object files.

The names of static libraries must start with the prefix "lib". The proposed naming scheme for the static libraries of compiled modules is, therefore, the prefix "libenc" followed by the module name.

The generated header file will serve as the public interface of the compiled module. Instead of having a single header file for an entire Encore program, each compiled module can provide its own. The `#include` directive can then be used in the header file of a program that imports a compiled module to get access to the necessary declarations and type definitions provided by the module header file. If a compiled module imports other compiled modules, their header files will be included in the header file for the compiled module. The generated header file for a compiled module will use the same naming scheme as for the static library.

Conditional inclusion (include guards) are used in the generated header files. For modules, the naming scheme for the constant used in the include guard will be the predefined value `ENCORE_LIB_`, followed by the module name in capital letters, ending with `.H`. For example, the `String` module would use the constant `ENCORE_LIB_STRING_H` in the include guard.

To avoid cluttering the file system and to make it clear which files belong to a
compiled module, the static library and the header file will be saved to a folder named using the module name and the suffix ", lib".

Figure 6 shows the compilation process for a module. In this example, the String Module is compiled.

Figure 7 shows the full compilation process for a program importing the compiled String module.
3.2 Implementation Challenges and Design Choices

Leveraging the support for separate compilation that is already in place for the C programming language goes a long way but there were still some challenges to overcome and design decisions to make. Some of the challenges come from incompatibilities between previous Encore implementation choices and the C way of doing separate compilation, while others are connected to separate compilation and modules in general. There are also some design choices that are more related to style and preference than being a technical difficulty.

3.2.1 Naming of C Identifiers

With the current naming scheme, the path used to locate a file plays a central role in the naming of C identifiers. This path can be either relative or absolute depending on the base path used. For identifiers located in the file targeted for compilation, this is usually the name of the file. This naming scheme works well when the program and all imported modules are compiled together as one unit, but when using separately compiled modules, there may be a mismatch between the actual names of the identifiers in the compiled module and the names assigned to them when the module is imported by another program or module.

When compiling a module, the names of its C identifiers will include the path passed to the compiler, usually the name of the file containing the module. Later, when another module or program imports the module, those identifiers are assumed to have a name based on the path used to locate the imported module. In this case, unless the imported module and the program or module importing it resides in the same folder, there will be a mismatch. There needs to be some global source of truth for how the C identifiers in an Encore program or module are named. For example, the names given to the identifiers in a module must be the same regardless if the module is compiled or imported. The names must also be unique across all modules used together in a program.

Two different solutions were considered during the project. The first solution use the module name to prefix the names of identifiers. Using the module name guarantees that the identifiers have consistent names and that they are unique for each module. However, this assumes that no module used in the same program has the same name. Enforcing unique names for the modules used in a program breaks backward compatibility, and for this reason, the solution would not work.

The second solution, which is the one that was ultimately implemented in the compiler, uses the absolute path to each file instead of the path used to locate it during the compilation process. This solution ensures that consistent and unique names get assigned to C identifiers across modules, and it does not break backward compatibility. Two modules with the same name can exist in the same program since the absolute path to their respective files will be different. One downside of the chosen solution is that a module source file and the folder containing the library and header file can not be moved to another place in the file system and then used without recompiling the module.
3.2.2 Trait Polymorphism

The current method for handling trait polymorphism in the generated C-code will not work out of the box with the suggested design for separate compilation. The reason for this is the mechanics of the enum type in C. The enum containing the trait method IDs starts counting from a dummy value, currently 1024, and then assigns a unique integer value to each ID by simply counting upwards. This approach works well when there is only one header file used in a program because then a single enum will hold all IDs.

If the header files for each compiled module contains an enum which assigns IDs to the traits defined in the module, then the IDs will not be unique any longer. If a class uses traits defined in two different modules, then the lookup function will not be able to tell which implementation is the correct one and it will just return the first match.

The implemented solution is similar to the previous one, but it does not rely on enums to assign unique IDs to trait methods. The trait method IDs are instead declared as uninitialized global variables in the header file for the module where the trait was originally defined. Uninitialized global variables in C are stored in the Symbol table section of the object files; they are then later assigned to the static data portion of a program’s address space and given a unique place in memory [20]. Since each global variable has a unique address, it can be used to identify a trait method id.

The lookup function used to find the correct method implementation have been changed to receive a memory address(pointer) as an argument instead of an integer value. The lookup function then compares the pointer to the address of each global variable representing a trait method ID. The new solution avoids the problem that the previous one had where trait methods could be assigned the same ID.

Listing [17] shows a comparison between the previous way trait method IDs were declared and the new way. Placeholder names are used for the IDs.

Listing [18] shows the new implementation of the lookup function. Placeholder names are used for the IDs and methods.

3.2.3 Detecting Compiled Modules

The decision of when to use a compiled module could be delegated to either the compiler or the programmer. A new statement similar to the `extern` keyword found in Rust can be added, letting the programmer explicitly declare that a compiled version of the module is required. The compiler could then report an error if it is unable to locate a compiled module. The compiler could also make the decision implicitly, for example, if a compiled version of the module exists, then it will be used. To make the transition to using compiled modules as seamless as possible it was decided that the compiler should make the decision implicitly. When determining if a compiled version of the module exists, the compiler looks for the folder where the library and header file would have been placed during the module compilation.
--Previous Solution
enum
{
   _TRAIT_METHOD_ID_1,
   _TRAIT_METHOD_ID_2,
   .
   .
   .
};

--New solution
int _TRAIT_METHOD_ID_1;
int _TRAIT_METHOD_ID_2;
...

Listing 17: Declaration of trait method IDs

static void * trait_method_selector(void* id)
{
   if ((id == (&(_TRAIT_METHOD_ID_1))))
   {
      return _enc__method__METHOD_1;
   }
   else if ((id == (&(_TRAIT_METHOD_ID_2))))
   {
      return _enc__method__METHOD_2;
   }
   else
   {
      printf("error, got invalid id: \%s", id);
   }
   return NULL;
}

Listing 18: New Implementation of the Lookup Function
process. The folder will be located in the same file system path as the module source file if it exists.

All static libraries located in the folder are passed to the linker as arguments when building an executable file, and the module header file is included in the header file for the module or program targeted for compilation. The reason why all libraries in the folder are passed to the linker is that the name of the library is not entirely deterministic due to the fact that it can be renamed manually as described in 3.2.8. The compiler still reads the full Encore source file for a compiled module to get the information it needs about its exported identifiers. No check is performed by the compiler to ensure that the contents of the source file and the compiled code match, it is left up to the programmer to make sure that compiled modules are kept up to date. When building C programs, the build tool `make` is often used to ensure that source files are recompiled if they are changed. The same tool can be used when building Encore programs.

### 3.2.4 Handling Imports

With support for separate compilation, a module can either be compiled or not. When compiling a regular program, compiled modules will be left out of the code generation step, while regular modules will be combined with the main program before the compiler generates any C-code.

A decision had to be made regarding how to handle the two different cases when compiling a module. If a module imports another compiled module then it can be handled the same way as for a regular program, but it is not as clear what to do for the other case. If the compiler combines the modules as it would when compiling a regular program, then there could be issues with duplicate names for identifiers if any of those modules were to be imported together with the compiled module in another program.

To avoid adding too much complexity to the implementation at this stage, it was decided that a compiled module would not be allowed to import regular modules. The compiler will report an error, letting the programmer know which module needs to be compiled.

### 3.2.5 Storing Information About Exported Identifiers

There are several ways to store information about the exported identifiers of a compiled module, for example, the compiler could generate an interface file like in Haskell, or it could write the information to the object file like in Go. The compiler could also parse the full source file of a compiled module. Using the object file to store export information is not a feasible solution since then support for parsing an object file would have to be added to the compiler, and storing export information in the object file might corrupt it since the object file format used in C is not designed for it. That leaves the choice of either using an interface file or parsing the source file. The main benefit of using an interface file is that if only information about the exported identifiers is stored there, then the compiler will have less code to parse
compared to if it were to parse the full source file. It would also be possible to distribute only the static library, header, and the interface file together, completely hiding the implementation details from the user. However, using an interface file adds an extra layer of complexity to the implementation, and due to time constraints it was not possible to include it in the design. Therefore, the chosen solution was to simply parse the full module source file.

### 3.2.6 Circular Dependencies

A circular dependency is when two or more modules import each other, either directly or indirectly. Dependencies between modules in a program can be seen as a graph (dependency graph) where the modules make up nodes and an import statement an edge. The Acyclic Dependencies Principle (ADP) states that “The dependency structure between packages must be a directed acyclic graph” [21]. What it means is that there should be no way to follow a path through the graph in such a way that a node is visited twice.

If two modules depend on each other, either directly or indirectly, they become tightly coupled. It will be impossible to use one without the other, and that includes any other modules that can be reached by either one of them. Allowing circles in the dependency graph can have unforeseen consequences, a module can go from having a single dependency to suddenly depending on all modules in the graph. A detailed description of this problem and a description of the problems that circular dependencies could introduce in general can be found in the article Granularity by Downing [21].

Encore currently supports circular dependencies between modules, and in the interest of backward compatibility, the design in this thesis continue to support it for regular modules. However, when it comes to compiled modules, it will not be allowed. Similarly to the Go implementation, it will be considered a compile-time error. The solution can be seen in 3.2.7, an algorithm was implemented with the capability to detect cycles in the dependency graph.

Listing 19 shows an Encore program using three modules. Together, the modules form a circular dependency which can be seen in the dependency graph shown in figure 8. These modules are because of this tightly coupled with each other, none of them can be used without the other two.

### 3.2.7 Resolving Module Dependencies

Some linkers can only resolve missing definitions in a left to right order. Depending on the order of the static libraries provided as arguments to the linker, it might not be able to resolve all definitions. Because of this, dependencies between modules must either be resolved internally in the compiler to determine the correct order, or another solution must be used to avoid link-time errors.

To solve this problem and the problem seen in 3.2.6, an algorithm capable of resolving dependencies between modules has been implemented in the compiler to determine the correct order in which to pass the static libraries as arguments to the
-- File: One.enc
module One
import Two
...

-- File: Two.enc
module Two
import Three
...

-- File: Three.enc
module Three
import One

-- File: main.enc
import One

active class Main
...
end

Listing 19: Encore Program with Circular Dependency

Figure 8: Dependency Graph with Circular Dependency

linker. The problem could also have been solved by passing the arguments to the linker twice. The main benefit of using the algorithm over the other solution is that it can also be used to detect circular dependencies.
The algorithm uses the fact that the set of modules and imports in a program make up a graph (dependency graph). A modified version of the Depth-first search algorithm [22] is used to walk the dependency graph while using some data structures to keep track of resolved and unresolved dependencies. A module (dependency) is considered resolved first when all of its imports (dependencies) have been resolved. Also, a circular dependency occurs if an unresolved module is seen twice.

Figure 9 contains a step by step description of the algorithm. Two sets [22] are used to keep track of resolved and unresolved dependencies while walking the graph. A stack [22] is used to store modules in the correct linking order.

For each module
1. Add module to the unresolved set.
2. For each module import:
   • If the imported module is not in the resolved set then:
     – If the imported module is in the unresolved set and it is a compiled module. Report an error, a circular dependency involving a compiled module exists.
     – If the imported module is in the unresolved set and it is a regular module. Skip to the next import. Circular dependencies between regular modules are allowed.
     – If none of the above, resolve dependencies recursively for the imported module.
3. Add module to the resolved set.
4. Remove module from the unresolved set.
5. If the module is a compiled module, push it onto the link order stack.

Figure 9: Dependency Resolving Algorithm

3.2.8 Resolving Name Conflicts Between Libraries

The names of static libraries must be unique to avoid conflicts during linking, with the proposed naming scheme, conflicts will occur if a program imports two compiled modules with the same name. The same applies to the include guards used in the generated header files for compiled modules, except that in this case, the contents of one of the header files will be excluded which may or may not cause an error.

To resolve the name conflicts, one of the libraries will have to be renamed. The Encore compiler already supports changing the name of the output file when compiling a regular program by using the output (-o) compiler flag. Using the same flag when compiling a module will change the name of the resulting static library, and also the constant used in the header include guard.

3.3 Changes Made to the Compiler

Most of the changes made to the compiler are either structural changes needed to support compilation of modules or related to the C-code generation and the way that the compiler handles imports of compiled modules.
3.3.1 Structural Changes

The largest structural change is that the compiler now supports compiling modules as well as importing and linking other compiled modules. The actual process of compiling a module is very similar to that of a regular program, and internally it is mostly the last step of the compiler that differs. The step where the compiler constructs the final output using a C compiler framework. Figure 10 shows how the compiler processes the Encore source file Math.enc. After generating the C-code, the compiler will either create an executable file or a static library and an accompanying header file depending on if the Math.enc file contains a module declaration or not.

A compiler flag (-cl) can now be used to target a module for compilation instead of a regular Encore program. When using the -cl flag, only source files containing a module declaration can be passed as the input to the compiler. The restriction placed on a source file that it must contain a Main class does not apply when compiling a module. The output from the compiler will be a folder containing a static library and a header file, instead of an executable file.

The compiler now makes a distinction between regular and compiled modules. When importing a module, the compiler first checks if a compiled version of the module exists. If it does, then the compiled version will be used, meaning that the compiler will not generate any C-code for it. The compiler tags the AST representing that module with a precompiled flag. This flag is mostly used during code generation to separate regular and compiled modules, but it is also used to make sure that a compiled module only imports other compiled modules.

Another important change, albeit a small one, is that all file system paths used in the compiler are now absolute paths. The base paths that the compiler use to locate modules are now all transformed into absolute paths. The compiler also determines the absolute path to the file targeted for compilation. Only working with absolute paths are good for consistency, it also ensures that the import paths used to locate modules are absolute which is important since the import path is used to name the C identifiers for imported modules. When compiling a module, the compiler uses the absolute path to the targeted file to name C identifiers, and by making all paths absolute, it is guaranteed to be equal to the path used when importing the module, which ensures a consistent naming scheme.

3.3.2 Changes to Code Generation

Before the compiler generates any C-code, it now separates the regular modules from the compiled modules. The algorithm found in §3.2.7 is used to resolve dependencies between modules, detecting circular dependencies and determining the linking order of the compiled modules. The compiler then combines the regular modules with the main program while keeping the compiled modules separate. When compiling a module, there will be no imported regular modules, so the compiler does not combine any modules in that case.

The C-code is generated in the same way for both modules and programs except
that no main function is included when compiling a module and that the naming scheme for the header files differs. For modules, the naming scheme defined in \[3.1\] is used, while for programs, the fixed name "encMain.h" is used. In the last step of the code generation, the compiler will create a static library when compiling a module, and an executable when compiling a regular program. When creating the executable file, the static libraries representing the imported compiled modules are passed to the linker as arguments.
4 Evaluation

The main goal of the project has been to decrease the compilation time for large Encore programs. Because of this, the evaluation focuses heavily on the different performance aspects of the implementation. Performance is not everything though, making sure that the implementation is correct and that it does not break backward compatibility is equally important. Therefore, a significant amount of the time spent on the evaluation of the project was dedicated to writing tests and to try and validate the implementation by other means. Usability aspects were not considered during the evaluation, but some conclusions regarding the subject are made in 5.

4.1 Performance

The implementation is evaluated in two different performance categories, compilation speed, and runtime performance. The runtime performance of a program should not be affected by using separate compilation, but testing for it serves as a validation against performance altering bugs. Also, since the implementation of trait method calls was changed, a separate test was created to measure the differences between the two different implementations.

The profiling capabilities of GHC were used to test the compilation speed of the compiler and the Unix time utility was used to measure the running time of Encore programs. Each test was performed ten times and then the result was averaged out. All tests were performed on a 13-inch Macbook Pro 2016 with a 2.9 GHz Intel Core i5 processor (two cores, two virtual threads) and 16 GB of 2133 MHz LPDDR3 Ram.

4.1.1 Compilation Speed

When measuring the performance of the Encore compilation process, there are two main things to consider, the performance of the Encore compiler itself, and the performance of the C compiler used to compile the generated C-code.

The speed of the Encore compiler will be affected by the amount of Encore code it needs to process, and the complexity of that code. Time spent performing file I/O also plays a part since the compiler writes C source files to disk before calling the C compiler.

The speed of the C compiler is affected by the size and complexity of the translation unit it is processing. By adding up the compilation time for all translation units in the C representation of an Encore program, we get the total time spent compiling C code. The combined size of all translation units in a C program will be referred to as the translation size of the program.

Something worth noting is that a big Encore program does not necessarily have a larger translation size than a small Encore program. The reason for this is that every class in Encore are represented by a separate C source file. Each source file then includes the generated header file and transitively also all header files representing
the Encore runtime. The C compiler then compiles the source files separately, and this means that each translation unit will contain not only the code for that class but also everything contained in the generated header file and the header files for the runtime. The runtime header files currently consist of roughly two thousand lines of C code. The effect of this is that an Encore program with two empty classes can have a larger translation size than a program with one class containing a lot of Encore code.

Figure 11 shows an example of the phenomenon. Each class added is an empty Encore class without any methods or fields. When reaching 32 classes, the translation size of the program has reached a minimum of roughly 64 thousand lines of C-code. The compilation time has reached 2 seconds even though the program only consist of about 100 lines of Encore code.

![Figure 11: Number of Classes vs Compilation Time](image)

The effect that the separate compilation of modules have on the compilation time of an Encore program lies on a spectrum.

The worst case occurs when all compiled modules have to be recompiled. For this case, the total compilation time is expected to be higher than if separate compilation was not used. The overhead comes from executing the Encore compiler for each module instead of just once. If the number of imports connecting the modules are high, then the total overhead will be larger because this would mean that some modules are processed by the compiler more than once.

The time spent generating and compiling C-code can be ignored for any compiled module that does not require recompilation. For cases other than the worst case, the overhead of recompiling some modules have to be weighed against the time saved by the other compiled modules. As a rule, compiled modules with large translation sizes and a small amount of Encore code that requires processing will provide a greater relative gain than the opposite. Also, modules with more Encore code that requires processing will cause a higher overhead during recompilation. The Encore code of imported modules is included in the code that requires processing when compiling a module.
Figure 12 shows a comparison between the worst case, best case and the case where no separate compilation is used, here called the regular case. The same program is used as in Figure 11 but the classes are now placed in a module. In the best case, the module containing the classes is pre-compiled. The difference between the worst case and the regular case can be attributed to the overhead of executing the Encore compiler for the module. This is an extreme example which is not likely to occur in practice.

Figure 12: Compilation Time Comparison, Number of Classes vs Compilation Time

Figure 13 provides a more realistic example. Here, a comparison for the program found in Listing 20 can be seen. The main file of the program does not do anything interesting in itself but a number of standard library modules are imported. The overhead seen in the worst case is quite large in this example. This is attributed both to the fact that using more modules involves executing the Encore compiler more times and that some of the standard library modules contain a large amount of Encore code. Standard library modules do in general not require recompilation unless a new version has been released, so for this set of imported modules, the best case can be seen as the general case.

4.1.2 Runtime Performance

The runtime performance tests focused on two main things. Evaluating the new implementation of polymorphic trait method calls and evaluating the runtime performance of typical Encore programs, with and without using separate compilation.

A test was created specifically to compare how the new implementation of trait method calls perform in comparison to the previous implementation. The test uses a basic Encore program which performs a polymorphic trait method call a large number of times. The called method adds two numbers and returns the result. Each test was executed ten times, and the results were then averaged. Table 1
--File: Main.enc
--In addition, several other modules are imported transitorily
--by the standard library modules.
import HashMap
import ArrayList
import Random
import Task
import Data.Either
import Set.OrderedSet

active class Main
  def main() : unit
    ()
end
end

Listing 20: Balanced Encore Program

Figure 13: Compilation Time Comparison, Balanced Encore Program

shows the results. The differences between the two implementations are negligible and should be attributed to natural fluctuation.

Table 1: Runtime Performance - Polymorphic Trait Method Calls

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Calls</th>
<th>Avg Time(s)</th>
<th>Standard Deviation(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Previous</td>
<td>100 000</td>
<td>0.0188</td>
<td>0.003</td>
</tr>
<tr>
<td>New</td>
<td>100 000</td>
<td>0.0179</td>
<td>0.004</td>
</tr>
<tr>
<td>Previous</td>
<td>1 000 000</td>
<td>0.1097</td>
<td>0.0033</td>
</tr>
<tr>
<td>New</td>
<td>1 000 000</td>
<td>0.1109</td>
<td>0.00038</td>
</tr>
<tr>
<td>Previous</td>
<td>10 000 000</td>
<td>1.0493</td>
<td>0.029</td>
</tr>
<tr>
<td>New</td>
<td>10 000 000</td>
<td>1.0423</td>
<td>0.0245</td>
</tr>
</tbody>
</table>
Some of the Encore implementations of the Savina [23] benchmarks were used to test the runtime performance of typical Encore programs, with and without separate compilation. The Savina benchmarks use common patterns found in Actor based languages which make them suitable to use for this comparison. Which Savina benchmarks to use was decided mostly at random with some consideration towards diversity. The Encore code was first modularized. All benchmarks were then executed ten times using compiled modules, and ten times with regular modules. The results were then averaged.

Table 2 shows the result of running the Savina benchmarks. The differences in execution time between using regular modules and compiled modules are too small to attribute it to anything but natural fluctuation.

<table>
<thead>
<tr>
<th>Table 2: Runtime Performance - Regular Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmark</td>
</tr>
<tr>
<td>1.PingPong</td>
</tr>
<tr>
<td>1.PingPong</td>
</tr>
<tr>
<td>3.Counting</td>
</tr>
<tr>
<td>3.Counting</td>
</tr>
<tr>
<td>8.Big</td>
</tr>
<tr>
<td>8.Big</td>
</tr>
<tr>
<td>21.ParallelQuickSort</td>
</tr>
<tr>
<td>21.ParallelQuickSort</td>
</tr>
</tbody>
</table>

### 4.2 Testing and Validation

Validating that the implementation is correct has been challenging and time-consuming overall. Some changes have been easier to validate and test than others. For example, validating that the new implementation of polymorphic trait method calls is correct is relatively easy. It is a change to a specific feature of the language, something that can be tested in isolation. Encore has an extensive suite of unit tests that is used to test the different features of the language, and there are already tests in place to make sure that the trait implementation works correctly. Encore also uses a benchmarking test suite known as the Savina benchmarks and this together with a benchmark written for the new trait implementation have been used to ensure that the implementation does not affect the runtime performance of Encore programs. These test suites have been essential tools when it comes to validating the backward compatibility of the implementation.

The hardest part has been to validate that Encore programs using compiled modules are correct, mainly that the compiler generates C-code correctly for programs using compiled modules. The C-code for a single compiled module can be assumed to be correct since the compiler does not treat it differently from a regular program. It is the code generated when using an identifier located in a compiled module that is hard to validate fully, essentially the interplay between compiled modules and the modules and programs that import them. Throughout the project, generated C source files for different types of programs have been inspected visually but to know
for sure tests have to be written. However, it is not feasible to write tests for all of
the possible cases and combinations where an Encore feature is used together with
one or more compiled modules.

The existing test suite includes tests for the module system which cover key features
of the language and other concepts related to importing and exporting identifiers
from modules. These tests have been duplicated and altered to use compiled modules
instead. Also, based on recommendations from members of the Encore research
group, additional tests have been written that covers the use of traits, local functions
and closures originating in compiled modules. The result of creating these tests is
that even if the implementation of separate compilation is not proved to be correct
in all possible use cases, it has an equivalent and in some cases, a better test coverage
than regular modules.

5 Conclusion

The implementation shows promising gains in compilation speed for Encore pro-
grams that use separately compiled modules. For some programs that only use
compiled modules, tests show a 40% decrease or more in total compilation time.
However, the cost of recompiling a module is high. The overhead of executing the
Encore compiler for each module adds up. In the worst case where all modules
used in a program require recompilation, the total compilation time can far exceed
that of the same program if it had not used compiled modules. These facts lead to
the conclusion that the implementation in its current form only provides a reliable
reduction in total compilation time when used together with modules that do not
change often.

Standard library modules are a good example where it would be very beneficial to
use separate compilation. Also, the C representation of Encore class is expensive to
compile. If an Encore program can be structured so that classes that rarely change
are put together in compiled modules, then a larger reduction in compilation time
is to be expected.

There is no measurable impact on runtime performance when using separately com-
piled modules. Some tests even show a slight increase in speed, but the difference
is so small that it should be disregarded. The implementation is fully backward
compatible with current Encore programs and should not have a measurable im-
 pact on the runtime performance, or compilation speed of code written before the
implementation of separate compilation support.

Usability was not a focus in the project and this is reflected in the implementation.
The compiler does not make an effort to detect if the compiled version of a module
and the module source file are out of sync. This can lead to confusing errors during
linking unless the programmer use a build tool such as make to ensure that modules
are recompiled when the source file changes. It is also not possible to move the static
library and source file to another location in the file system without recompiling due
to the way the C identifiers are named. This makes it impossible to distribute
pre-compiled modules to other systems because a module must be compiled on the
system that it will be used on.
Due to time constraints, the implementation has not been tested and validated for correctness as rigorously as it could have been. The test coverage matches that of the pre-existing module system, but more tests should have been written to increase the confidence in the correctness of the implementation further.

5.1 Future Work

The implementation of separate compilation presented in this thesis could be improved further. The overhead caused by executing the Encore compiler when re-compiling a module should be addressed.

An interface file could be generated by the compiler when compiling a module, similar to how the Haskell module system works [17]. The interface file would then only contain the exported identifiers of the module. Only the type signatures of methods and functions are needed for the compiler to determine that the module is used correctly. Therefore, there would be no need to include the method and function bodies in the interface. With the current implementation, the full source file of a module is parsed and processed by the compiler when importing a module. When using an interface file instead, there would be much less Encore code for the compiler to process for each import of a compiled module, which would result in shorter compilation times. The improvements would affect both compilations of regular programs and recompilations of modules.

Another change that could improve the performance of the Encore compiler is that the interface file of a compiled module could include the export information of all of the modules that it can reach in the dependency graph. By doing this, the compiler would not have to parse as many files, and this should translate to improvements in compilation time. The object files of Go packages store export information from other packages it depends on, and this is considered to be one of the main contributing factors to the speed of the Go compiler [5].

Finally, instead of creating one C source file for each class, all classes belonging to a module could be put in one file. Putting all classes in one file would significantly reduce the translation size of a module that uses more than one class. A smaller translation size equals less time spent compiling C code, resulting in a reduced total compilation time.
List of Figures

1. Encore Compilation Process ............................................. 8
2. Encore - C Program Structure ........................................... 11
3. Encore - Full Compilation Process ................................... 11
4. Rust Module Tree ....................................................... 19
5. Encore Compilation Process with Separate Compilation ............ 21
6. Encore Module Compilation Process ................................... 22
7. Compilation Process when Importing a Compiled Module ............ 22
8. Dependency Graph with Circular Dependency .......................... 28
9. Dependency Resolving Algorithm ....................................... 29
10. Encore Module & Program Internal Compilation Process ............ 31
11. Number of Classes vs Compilation Time ............................... 33
12. Compilation Time Comparison, Number of Classes vs Compilation Time ......................................................... 34
13. Compilation Time Comparison, Balanced Encore Program .......... 35

Listings

1. Encore Main Class ....................................................... 5
2. Encore Trait definition and Usage ..................................... 6
3. Encore Trait Polymorphism ............................................. 6
4. Encore Module Declarations ............................................ 7
5. Basic Encore Import Declarations ..................................... 8
6. Qualified Imports in Encore ............................................ 9
7. Controlled Imports in Encore ......................................... 9
8. Naming of a C Identifier ............................................... 12
10. C Declarations and Usage in Other Files ........................... 15
11. Haskell Module Definition ............................................ 15
12. Haskell Qualified Imports ............................................. 16
13. Go Package Declaration and Imports ................................ 17
14. Go Duplicate Package Name .......................................... 18
15. Encapsulation in Go packages ...................................... 18
16. Importing a Rust Crate and Modules ................................. 20
17. Declaration of trait method IDs ...................................... 25
18. New Implementation of the Lookup Function ......................... 25
19. Encore Program with Circular Dependency ......................... 28
20. Balanced Encore Program ............................................ 35

List of Tables

1. Runtime Performance - Polymorphic Trait Method Calls .......... 35
2. Runtime Performance - Regular Programs ........................... 36
References


[12] Brandauer S. Encore Glimpse: Encore intro slides;.


[23] Shams Imam VS. Savina - An Actor Benchmark Suite;.