University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Marian Dziubiak
Student no. 370784

Implementation of a lazy runtime environment on the .NET platform

Master's thesis
in COMPUTER SCIENCE

Supervisor:
dr Marcin Benke
Institute of Informatics

Warsaw, June 2020

Abstract

Functional programming languages may have very desirable properties. We take a look at compiling Haskell, a non-strict functional language, to work on the .NET platform. In order to achieve that goal, a framework runtime for evaluation of lazy computations is presented. The performance of the runtime is important in order for it to be practically usable. To simplify the translation of Haskell programs to use this runtime an experimental compiler has been created and some of its details are presented.

Keywords

functional programming, lazy evaluation, .NET CLR

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Computer Science

Subject classification

Software and its engineering
Software notations and tools
Compilers
Runtime environments

Tytuł pracy w języku polskim

Implementacja leniwego środowiska uruchomieniowego na platformie .NET

Contents

Introduction

Functional programming is reliving a peak in its popularity. One of the more interesting features of pure functional programming is lazy evaluation. This means delaying the evaluation of an expression until its value is actually needed. This allows the programmer to use infinite data structures and certain types of algorithms that wouldn't be easily implementable in a strict language. Currently the main programming language supporting lazy evaluation is Haskell.

Haskell is compiled to native code or to LLVM representation by the optimizing compiler GHC. Like many other programming languages Haskell is mostly tied to its compiler and the native platform (as opposed to a managed platform). Actually, Haskell did in fact have more compilers available, however, GHC is leading the language development and left competition far behind.

There exist a couple of managed platforms that provide a common ecosystem for multiple languages. The two most popular are JVM and .NET. There has been some work done for the JVM [Tul96; CLH01; Ste02] and there exist two Haskell implementations: Frege and Eta. In this work I will focus on the idea of bringing Haskell to the .NET platform.

In 2002 Microsoft has released the first version of .NET Framework with C# programming language. Since the initial release, the .NET ecosystem has grown mature with many languages targeting the platform. Some of them are functional programming languages – F#, SML.NET, Nemerle. In 2015 most of the platform has been open sourced and gained a lot of interest from the community. This lead to some interesting performance improvements.

Previous attempts to bring Haskell to the .NET CLR (*Common Language Runtime*):

- In 2001 Nigel Perry, Erik Meijer and Arjan van Yzendoorn implemented a non-strict scripting language called Mondrian [MPY01; PM04].
- In 2005 Monique Monteiro, Mauro Araújo, Rafael Borges and André Santos created Haskell.NET [Mon+05].
- In 2006 Oliver Hunt created a compiler from Core language to C#, leveraging OOVM (*Object Oriented Virtual Machine*) features to simplify interoperability with non-strict code from CLI languages [Hun06].

Those previous attempts did not result in widespread adoption of created solutions. Due to the passage of time and lack of interest, it is currently impossible to obtain the created compilers or their source code. In order to see practical results a new compiler had to be made and with it a new runtime focused on code execution efficiency.

What is the reason for bringing Haskell to a managed platform? While the performance may be lower, the generated code will be portable and users can leverage a variety of libraries, including the mature web framework ASP.NET. This also goes the other way – .NET developers could access some of the libraries in the Hackage package repository. Ever tried debugging Haskell code? After compiling to C# it's possible to setup breakpoints and analyze the state of the program (with some limitations).

This paper presents a lazy runtime implementation called Lazer as well as a description of an experimental compiler from Haskell to C#. You can find full source code for the runtime, compiler and benchmarks at GitHub: github.com/manio143/Lazer.

Chapter contents

- Chapter 1. *Laziness in functional languages* introduces the desired properties of functional languages like purity and laziness. Then it goes over a lazy program evaluation process.
- Chapter 2. *Lazy runtime* showcases an implementation of a lazy runtime: the representation of data, functions and computations as well as framework classes that the compiled code uses.
- Chapter 3. *Comparison to related work* compares this and other implementations by showing important differences and arguing for their pros and cons.
- Chapter 4. *Performance* looks at performance bottlenecks, presents benchmark results and compares those to Haskell compilers.
- Chapter 5. *Compiling Haskell to C#* presents some aspects of the translation mechanism from STG representation of a Haskell program into C#.
- Chapter 6. *Conclusions and future work* summarizes the thesis and looks into future improvements to the runtime and extending Haskell for CLI (*Common Language Infrastructure*) interoperability.

Chapter 1

Laziness in functional languages

Functional programming languages are characterized by first class functions and easy to model data types. Many modern languages have functional features, but much fewer have syntax and features that make them easy to use. In particular, languages from the ML family such as Haskell, Elm, OCaml and F# have become fairly popular in the community interested in functional programming.

1.1. Desired properties of functional languages

One of the great things about functional languages is that they bring the program closer to mathematical notations. Why is that a good thing? Mathematicians are able to provide proofs for correctness of their formulas. Therefore, an algorithm expressed in mathematical functions can be reasoned about in a similar manner. A great example of that is Coq – a formal proof management system. The proofs can be extracted into functional programs in OCaml, Haskell or Scheme [Let02].

1.1.1. Purity

All functions in mathematics are *pure*. However, functions in programs can have side effects, like modifying memory or communicating with external devices. This makes a lot of algorithms harder to model in terms of mathematical functions. We call a function *pure* if it has no side effects.

A pure function will always return the same result for the same set of arguments. This means expressions that apply a pure function to some arguments are referentially transparent [SS90] – can be substituted by the value they would evaluate to. Referential transparency isn't a property of impure functions that may rely on their code getting invoked multiple times, each time returning a different value.

Furthermore, when dealing with just pure functions, the compiler may easily apply an optimization called Common Subexpression Elimination (CSE) which computes the value of an expression once and uses it in multiple places.

```
  f a <*> f a <*> f a
----------------------
    let x = f a in
    x <*> x <*> x
```

1.1.2. Laziness

In mathematics it doesn't matter which part of the formula is computed first. In a language with only pure functions, such as Haskell, since there are no side effects that may have to happen in a particular order, the order of expression evaluation can be changed. In particular we can delay the evaluation until the moment when we actually need to know the computed value. It may turn out we don't actually need to perform the computation. But if we do evaluate the expression, its value should be saved so that subsequent evaluations of that same expression can use that value.

In most strict languages there is a minimal amount of laziness. It can be most often found in conditional expressions. For example: if the left argument of the boolean `&&` operator is false then the right argument is not evaluated. This is especially useful when eager evaluation would lead to an error. Higher level languages also include some sort of lazy type (e.g. `Lazy<T>` in C#) that allows delaying computations. However, it is more cumbersome to use and making it an explicit type makes it harder to use in contexts where laziness was not considered.

There are a couple reasons why using lazy evaluation can be beneficial. First of all it allows to think about certain problems in more mathematical ways – e.g. list of all prime numbers. Such a data structure cannot be computed in finite memory, but it's possible to operate on it as if it was. This is especially important when dealing with Big Data as large amounts of information cannot be kept fully in memory, but their subset can. This is leveraged by frameworks such as Spark and found to increase performance of computations [HBG19]. Lazy evaluation also allows to avoid performing expensive computations when the result is not going to be used. If we are passing a value to an unknown function we don't know if it will be used. In a strict language this computation would have to be performed either way. Finally, there exist certain data structures that leverage lazy evaluation to provide amortization of asymptotic complexity, e.g. queues [Oka96].

1.2. Representation of functional programs

A functional program can often be expressed in terms of nested expressions that form a tree. By rewriting the simple expressions representing a named value into a reference to the bound expression, we obtain an expression graph. The process of evaluation of that program is equivalent to the process of graph reduction of the expression graph.

An example of such a reduction is the β-reduction in lambda calculus. Given a function application `f` $\cdot$ `a` where `f` is a lambda abstraction λ`x.e`, performing a β-reduction yields a new expression `e'` that is equal to `e` with all occurrences of `x` replaced by `a`.

Expressions are reduced to weak head normal form (WHNF) which is either a primitve value (e.g. an integer), a data cell (structured data, e.g. a tuple), a function value or a non-saturated function application (not enough arguments). For more theoretical information on evaluating expression see [Pey87].

1.2.1. Practical approaches

Graph reduction is rather simple conceptually, but it turns out to be harder to implement efficiently. Over the years different approaches were created to compute lazy languages:

- combinator machines (e.g. SKIM) [Tur79; Cla+80]

  Any program in lambda calculus can be transformed into application of basic combinators S and K. The SKI Machine is a description of instructions and hardware specific to functional program reduction that has been compiled into combinatorial form.

  ```
  S = \f \g \x -> f x (g x)
  K = \x \y    -> x
  ```

- lazy SECD machine CASE [Lan64; DM90]

  In this approach lambda expressions are reduced by following operational semantics of an abstract machine. CASE extends SECD with closures which eliminate multiple copying parts of the environment multiple times onto the stack.

  Acronyms describe machine state:
  <Stack, Environment, Cotrol, Dump>
  <Code, Argument register, Stack, Environment>

- G-Machine [Kie85; Pey87]

  The Graph Reduction Machine was designed as a fairly efficient way to evaluate lazy functional programs on existing hardware. It was originally defined by Thomas Johnsson and Lennart Augustsson [Joh84].

- Categorical Abstract Machine [CCM85; MS86]

  CAM similarly to CASE is a successor of SECD. However, it is based on Categorical Combinators [Cur86].

- Three-Instruction-Machine (TIM) [FW87]

  TIM replaced abstract expression graph representation by actual machine code to be executed. The computations are compiled in supercombinator form and executed with just three instructions: Take and Push (stack manipulation), and Enter.

- Spineless Tagless G-Machine (STGM) [Pey92]

  An extension of the G-Machine that discarded spines (chains of thunk references) and tags (data structure information). Instead it treated everything equally as a closure whose code was executed on entry.

The Glasgow Haskell Compiler (GHC) uses the STGM approach with some additional optimizations accumulated over the years. The STG language extends simple lambda calculus with a few constructs. In particular, it defines a `let..in` expression for binding expressions to identifiers and delaying their evaluation, and a `case..of` expression for forcing expressions (evaluating them to WHNF) and pattern matching.

1.2.2. Closures and Thunks

Let's introduce the concept of a *closure*. Every lambda expression containing free variables is represented by a closure – an object in memory holding references or values of those free variables as well as a pointer to the code that evaluates this expression.

A closure can represent either a delayed computation or a function that would be applied to some arguments. Even data can be represented by a closure, with its code being a simple expression returning this data.

Section 1.1.2. *Laziness* required lazy computations to save the computed value and return it on all future evaluations. A closure that represents a computation that will be replaced by the computed value is called a *thunk*. Most `let` expressions create thunks.

After analyzing performance of lazy programs research shows that many algorithms are actually faster if they are evaluated eagerly. Therefore a process called strictness analysis is performed by the compiler to decide whether a value should be evaluated lazily or eagerly [SPV17]. Therefore, there are cases when a `let` expression may actually evaluate its subexpression instead of creating a thunk for it.

1.2.3. Example evaluation process

To make sure the reader can better understand the evaluation process a simple example is presented. The Haskell code below is written very close to its STG representation. It shows a `Maybe` data type with a monadic `bind` operation, a source computation `divide` and a modifying computation `add n` represented in terms of bind.

```
data Maybe a = Nothing | Just a
bind f m = case m of
            Just x -> f x
            Nothing -> Nothing
divide x y =
    case y of
        0 -> Nothing
        _ -> let v = x / y
             in Just v
add n = let f = \x -> let v = x + n
                      in Just v
        in bind f
main = let d = divide 4 2
       in add 5 d
```

The steps below represent evaluation of `main`:

Code	Action	Heap	Stack
`main`	create thunk `d{divide 4 2}`		return `main`
`main`	apply `add` to 5	`d{divide 4 2}`	return `main`
`add`	create function `f` capture `n = 5`	`d{divide 4 2}`	apply in `main` return `main`
`add`	return partial application `bind f`	`d{divide 4 2}` `f[n = 5]{`*`fun`*`}`	apply in `main` return `main`
`main`	apply `bind f` to `d`	`d{divide 4 2}` `f[n = 5]{`*`fun`*`}`	return `main`
`bind`	evaluate argument `m` → `d`	`d{divide 4 2}` `f[n = 5]{`*`fun`*`}`	return `main`
`d`	apply `divide` to 4 and 2	`d{`*`evaluating`*`}` `f[n = 5]{`*`fun`*`}`	case in `bind` return `main`

`divide`	evaluate argument `y` → `2`
	check if `2` is `0`
	create thunk `v{4 / 2}`
	return `Just v`
		`d{`*`evaluating`*`}`
		`f[n = 5]{`*`fun`*`}`
		`v{4 / 2}`
			update `d`
			case in `bind`
			return `main`

`bind`	check if the result is `Just`
	take out `x` → `v` from `d`
		`d{Just v}`
		`f[n = 5]{`*`fun`*`}`
		`v{4 / 2}`
			return `main`

`bind`	apply `f` to `x` → `v`	`f[n = 5]{`*`fun`*`}` `v{4 / 2}`	return `main`
`f`	create thunk `v'{v + 5}` return `Just v'`	`v{4 / 2}` `v'{v + 5}`	return `main`

If we now unpack `Just v'` and evaluate `v'` we will obtain the value `7`.

Chapter 2

Lazy runtime

This chapter presents an implementation of a lazy runtime environment on the .NET platform. The runtime is called *Lazer* inspired by Razor and Blazor, web page frontend scripting systems for the .NET platform. Lazer has been implemented in C#. It uses a C# 9.0 feature called *function delegates* that at the time of writing was not yet released.

The implementation is best explained by following the example code execution from chapter 1.2.3. *Example evaluation process*. First, let's introduce some key concepts regarding data representation and then move on to computation representation.

2.1. Types and data representation

Types can be divided into *lifted* and *unlifted*. Lifted types include $\perp$ (bottom) as a possible value. Actually, $\perp$ represents an infinite computation or a program error. Unlifted types on the other hand will always represent a proper value.

The bottom value is quite interesting in lazy languages. It is possible to apply a function to an expression that will clearly result in an error, but because it's not evaluated eagerly, program execution will halt only when the applied function tries to evaluate this expression.

Unlifted types such as `Int#`, `Double#` and `Char#` can be represented in C# by primitive types like `int`/`long`, `double` and `char`. Other unlifted types include tuples `(#,#)` and arrays `ByteArray#`, which can be represented by *structs* (.NET value types) and arrays (`byte[]`).

Lifted types will be represented by a pointer reference. For example an instance of a lifted type `Int` is an object representing its data constructor `I#` that has one field of type `Int#`. In practice this means data constructors represent .NET classes.

```
public sealed class IHash : Data {
    public int x0;
    public I(int x0) {
        this.x0 = x0;
    }
    public override int Tag => 1;
}
```

Haskell's type system cannot be represented in C# directly. One of the more advanced Haskell features are higher-kinded types (HKT). This allows type constructors to take polymorphic parameters that are also type constructors, i.e.:

```
newtype IdentityT f a = IdentityT { runIdentityT :: f a }
```

Here `f` is a type variable representing a type constructor and is later applied to the type variable `a`. The .NET runtime does not allow for generic type parameters to be type constructors. In other words .NET's generic type parameters are equivalent to Haskell's type variables of kind `*`. So types like `IdentityT` are not expressible in C# without type erasure or *implementation leakage*. The user of `IdentityT` would have to know how it was defined and pass it the applied type `f a`. This can be problematic – especially when type arguments are used as an abstraction layer. Therefore this runtime ignores types, by means of type erasure.

2.1.1. Data superclass

Data constructors implemented in C#, like `IHash` defined above, will share some common properties. Thus, they will extend the `Data` class. As mentioned in 1.2.2. *Closures and Thunks*, data constructors are also closures, but very simple ones, having a simple `return this;` in their evaluation method.

```
public abstract partial class Data : Closure
{
    public override Closure Eval() => this;
}
```

The `Data` class doesn't override the `Tag` property, thus forcing its subclasses to do so.

2.1.2. Data constructor definition

All data constructors fit a simple template. Fields in the constructor are all public, ordered and named accordingly (`x0`, `x1`, `x2`, ...). The constructor's class has a *constructor* that takes as many arguments as there are fields. The class is sealed (cannot be extended) so that efficient type checks can be performed.

```
public sealed class [Name] : Data {
    public [Type0] x0;
    ...
    public [TypeN] xN;

    public [Name]([Type0] x0, ..., [TypeN] xN) {
        this.x0 = x0;
        ...
        this.xN = xN;
    }

    public override int Tag => [Tag]
}
```

The `[Type]` is either an unlifted value type (e.g. `int`, `double`) or a pointer reference of type `Closure`. Why `Closure` and not `Data`? Because, data constructors can be applied to unevaluated computations.

If a type has more than one constructor, each one has to override the `Tag` property with an integer value. This value is the constructor's index in the type definition of the Haskell program and goes from 1 to n. Tag can be used for `case` expressions explained in 2.2.4. *Performing conditional computations*.

Let's take a look at the representation of the Haskell's `Maybe` type constructors `Nothing` and `Just`, which have been defined in 1.2.3. *Example evaluation process*.

```
public sealed class Nothing : Data {
    public Nothing() { }
    public override int Tag => 1;
}
public sealed class Just : Data {
    public Closure x0;
    public Just(Closure x0) {
        this.x0 = x0;
    }
    public override int Tag => 2;
}
```

2.2. Computation representation

In C# the unit of computation is a method. Given a Haskell expression we can create a method which when executed will return the value obtained by evaluating that expression.

C# is an Object Oriented language and it provides two kinds of methods: instance methods and static methods. An instance method is defined in a class and accessed through a method table on an object of that class. This allows for subclasses to override instance methods of their parent. Static methods are not bound to any objects and are accessed directly by their qualified name.

When we look at an expression as the steps to evaluate it, we don't have any state which would justify implementing expressions as objects. However, closures do have state – the captured free variables, and as such will be represented by objects.

For any expression we may construct a static method that describes the expression's evaluation. This method will be called from an object representing the closure. It may also be called directly from another static method as a performance optimization.

The Lazer runtime uses generic classes to provide closure implementation (see 2.3.5. *Universal classes*). However, it is also possible to generate a class per closure which will call the static method in a specialized overloaded method. This requires a different design of the runtime (see 3. *Comparison to related work*).

2.2.1. Computation as a static method

The Haskell program consists of bindings – named expressions. A binding is represented in memory as a closure with a pointer to the computation. A computation is represented by a static method that follows a simple template:

```
public static [RetType] [Name]_Entry(
    [FreeType0] [FreeName0], ..., [FreeTypeK] [FreeNameK],
    [ArgsType0] [ArgsName0], ..., [ArgsTypeN] [ArgsNameN]) {
        [Expression]
    }
```

The `[RetType]` is the return type of the method, equivalent to the type of `[Expression]` (for all lifted types it's `Closure`). The `[Name]` is the bound identifier of the computation.

The first k arguments are the free variables of the computation (see 2.3.5. *Updatable class family* for a practical optimization). The following n arguments are formal parameters, if the computation represents a function.

2.2.2. Computing a data cell

Let's look at the `divide` function from 1.2.3. *Example evaluation process*. It evaluates its second argument and performs a value check on it. Then it delays the actual division (a costly operation) and returns a new data cell. This can be represented in C# by the following methods:

```
public static Closure divide_Entry(Closure x, Closure y) {
    var yI = y.Eval() as IHash;
    var yVal = yI.x0;
    switch (yVal) {
        case 0:
            return new Nothing();
        default:
            var v = new Updatable<Closure, int>(&divide_v_Entry, x, yVal);
            return new Just(v);
    }
}
public static Closure divide_v_Entry(Closure x, int yVal)   {
    var xI = x.Eval() as IHash;
    var xVal = xI.x0;
    var d = xVal / yVal;
    return new IHash(d);
}
```

Creating a new heap allocated object — a data cell – is easy in C#. In STG all constructor applications are saturated so we don't have to deal with an insufficient number of constructor arguments.

Here we can also see that `v` is not a data value, but a delayed, updatable computation. The class `Updatable` is a generic thunk that takes a pointer to a static function that performs the computation and captures the free variables (here `x` and `yVal`). Its implementation is described in 2.3.5. *Updatable class family*.

When `Eval` method is called on `v`, the `divide_v_Entry` method will be invoked with captured values `x` and `yVal`.

2.2.3. Computing a function

Let's look at the `add n` function from 1.2.3. *Example evaluation process*. It creates a new function `f` that captures argument `n` and partially applies function `bind` to the created function. This can be represented in C# by the following methods:

```
public static Function bind = new Fun2<Closure,Closure,Closure>(&bind_Entry);
...
public static Closure add_Entry(Closure n) {
    var f = new Fun1<Closure, Closure, Closure>(&add_f_Entry, n);
    return bind.Apply<Closure,Closure>(f);
```

```
}
public static Closure add_f_Entry(Closure n, Closure x) {
    var v = new Updatable<Closure, Closure>(&Int_add, x, n);
    return new Just(v);
}
```

The `FunN` class takes a reference to the computation method. The `N` in its name represents the function's arity – number of arguments it can be applied to. It can also capture free variables, as is the case here when we pass `n` to the constructor. The `Apply` method checks the number of provided arguments and if saturated performs the computation. Otherwise it returns a partial application object (PAP).

2.2.4. Performing conditional computations

Let's look at the `bind f m` function from 1.2.3. *Example evaluation process*. It performs pattern matching on its second argument. An alternative computation is performed based on the satisfied condition. This can be represented in C# by the following method:

```
public static Closure bind_Entry(Closure f, Closure m) {
    m = m.Eval();
    switch (m) {
        default:
            throw new ImpossibleException();
        case Nothing m_Nothing:
            return new Nothing();
        case Just m_Just:
            var x = m_Just.x0;
            return f.Apply<Closure,Closure>(x);
    }
}
```

We represent STG `case` expressions in C# with `switch` statements. An example of this has already been shown in 2.2.2. *Computing a data cell* when checking if an argument had value `0`. While switching on primitive values is simple, it's a little more complex with data constructors.

To get the arguments of a data constructor, the object has to be cast from the general type `Closure` to the specific type of the constructor. The .NET Runtime is type safe in that it performs type checks on casts to validate data consistency. The most efficient cast available is an `isinst` CIL instruction that applied on a sealed type results in four ASM instructions:

1. get pointer to object's MethodTable,
2. compare it to the sealed type's MethodTable pointer
3. jump forward one instruction if pointers are equal
4. otherwise assign `null` to the variable

There are two ways to perform constructor choice in a switch statement: by tag or by type. Doing a switch on type performs the type cast and choice simultaneously but it is linear to the number of alternatives. Doing a switch by tag still requires the type cast, but for more

than 2 constructors (3 branches including the `default`) it is implemented efficiently as a jump table. Tests in 4.2.1. *Switching on Tag vs Type* show that switching on tag becomes beneficial with switches of 5 or more alternatives.

Enumerator types are types whose every data constructor has no arguments. Switching on those is always performed by tag, because no casts are needed.

In the `default` branch of the example an exception is thrown that marks an impossible branch. The reason why it had to be included is because C# cannot tell that no other values are possible. That's actually not due to type erasure – any switch statement that is the last statement in a function has to have a default case.

2.3. Runtime implementation

The lazy runtime is a set of abstractions and tools that allow for efficient execution of lazy code. This section presents the details of what has been implemented, how it works, and why it is efficient.

2.3.1. Abstract Closure class

All heap allocated objects: computations, lifted values, and functions share a common superclass — the `Closure`. Closure provides three distinct interfaces for each of the aforementioned object types:

1. `Eval()` method – computations are invoked and expressions evaluated to WHNF, resulting in the returned value. For data and functions there is nothing to compute as they are already in WHNF so they can just return themselves.

2. `Tag` property – allows to switch on data constructors without a cast beforehand. Functions and computations throw an exception if used.

3. `Apply(...)` methods – used to apply functions to their arguments. Data values cannot be applied and throw an exception. Computations are first evaluated and then the result is applied.

```
public abstract class Closure {
    public abstract Closure Eval();
    public abstract     int Tag { get; }
    public abstract       R Apply<A0,R>(A0 a0);
    public abstract       R Apply<A0,A1,R>(A0 a0, A1 a1);
    ...
}
```

Every method in C# has a fixed number of parameters (no varargs). This means the runtime establishes maximal arity of functions (with the number of `Apply` methods). Thanks to currying any function of greater arity can be rewritten to return a function of smaller arity [Cur36]. Furthermore, `Apply` methods are generic to achieve the most flexibility and succinctness. This in turn means type parameters need to be provided at every call site.

2.3.2. Abstract Computation and Data classes

From the previous section follow the definitions of abstract `Computation` and `Data` classes:

```
public abstract class Computation : Closure
{
    public override int Tag
        => throw new NotSupportedException(
                        "Accessing Tag on a computation.");

    public override R Apply<A0, R>(A0 a0)
        => this.Eval().Apply<A0, R>(a0);

    public override R Apply<A0, A1, R>(A0 a0, A1 a1)
        => this.Eval().Apply<A0, A1, R>(a0, a1);
    ...
}

public abstract class Data : Closure
{
    public override Closure Eval() => this;
    public override int Tag => 1;

    public override R Apply<A0, R>(A0 a0)
        => throw new NotSupportedException(
                        $"Cannot apply a value ({GetType()})");

    public override R Apply<A0, A1, R>(A0 a0, A1 a1)
        => throw new NotSupportedException(
                        $"Cannot apply a value ({GetType()})");
    ...
}
```

2.3.3. Abstract Function class

While previous abstractions were pretty straightforward, functions are a bit more complicated. As mentioned before, C# methods have to be invoked with all of the parameters they require. This means functions have to provide an additional check of arity. There are three possible situations:

1. function is applied to the exact number of arguments – invoke the computation method;
2. function is applied to too many arguments – apply the function to the exact number of arguments and then apply the result to the rest;
3. function is applied to too few arguments – create a partial application object.

```
public abstract partial class Function : Closure
{
    public override Closure Eval() => this;
```

```
    public override int Tag
        => throw new NotSupportedException(
                        "Accessing Tag on a function.");

    public int Arity;
    ...
}
```

The above part of the `Function` class implements `Eval` and `Tag` inherited from `Closure` and introduces a new member: the `Arity` field. Below is one of the overloaded `Apply` methods that shows two of the situations outlined above:

```
public override R Apply<A0, A1, R>(A0 a0, A1 a1)
{
    Closure h;
    switch (this.Arity)
    {
        case 1:
            // too few arguments
            h = this.Apply<A0, Closure>(a0);
            return h.Apply<A1, R>(a1);
        default:
            // too many arguments
            var pap = new PAP<A0, A1>(this, a0, a1);
            return Unsafe.As<R>(ref pap);
    }
}
```

The case when `Arity` is 2, when we provided the exact number of arguments, is handled by an overload in the subclass for the given arity. This way when `Apply` is called with the right number of arguments, the proper code gets executed right away, without the need of another indirection.

The partial application object holds a reference to the applied function and captured arguments. When applied it applies the original function to all available arguments.

```
public abstract class PAP : Closure
{
    public Function f;
    protected PAP(Function f) => this.f = f;

    public override Closure Eval() => this;

    public override int Tag
        => throw new NotSupportedException(
                        "Accessing Tag on a PAP.");
}

public sealed class PAP<B0> : PAP
{
    public B0 x0;
```

```
    public PAP(Function f, B0 x0) : base(f)
        => this.x0 = x0;

    public override R Apply<A0, R>(A0 a0)
        => f.Apply<B0, A0, R>(x0, a0);

    public override R Apply<A0, A1, R>(A0 a0, A1 a1)
        => f.Apply<B0, A0, A1, R>(x0, a0, a1);

    public override R Apply<A0, A1, A2, R>(A0 a0, A1 a1, A2 a2)
        => throw new NotSupportedException(
                        "Application exceeds runtime argument limit.");
}
```

With this implementation PAPs never form a chain of objects, but rather return a new "flat" PAP when applied to too few arguments.

2.3.4. Abstract Thunk class

In 1.2.2. *Closures and Thunks* a *thunk* has been defined as a closure that updates itself after it is evaluated. The `Thunk` class is going to extend `Computation` class and provide this update mechanism. Upon invoking the `Eval` method the first time, the computation is performed and the result saved. The thunk now becomes an indirection and on future evaluations the saved result is returned.

This can be implemented in two ways. One is to perform a comparison of the result pointer with `null` to see if the computation has been performed. The other one involves modifying the method that is called to perform evaluation. Performance tests show that on the .NET platform the first approach is more efficient due to a more expensive constructor call for every thunk in the second approach (see 4.2.2. *Dynamic Thunk indirection call* for more details).

```
public abstract class Thunk : Computation
{
    public Closure ind;

    protected abstract Closure Compute();
    protected virtual void Cleanup() { }

    public override Closure Eval()
    {
        if (ind != null)
            return ind.Eval();

        ind = Blackhole.Instance;
        ind = Compute();
        Cleanup();
        return ind;
    }
}
```

The `Compute` method is implemented in subclasses and performs the actual computation. It's result is saved in the `ind` field. But during the computation `ind` is assigned a special value called the *black hole* that allows detecting invalid self-references (computation loops). If the computation of a thunk tries to evaluate itself again it will end up evaluating the black hole, which stops the program. Black holes can be further used to implement synchronization in multithreaded contexts [HMP05]. However, the currently used implementation is extremely simple:

```
public sealed class Blackhole : Computation
{
    public override Closure Eval() =>
        throw new System.Exception("BLACKHOLE");
    public static Blackhole Instance = new Blackhole();
}
```

For explanation on the `Cleanup` method, see 2.3.5. *Updatable class family*.

2.3.5. Universal classes

In order to limit the amount of types, rather than having a type per closure, a different approach has been chosen. Universal types take a function pointer and make indirect calls to it. They also have generic type parameters to allow creating type safe and efficient instances with customized fields.

Updatable class family

Section 2.2.2. *Computing a data cell* presented usage of an Updatable class. It is a subclass of `Thunk` that takes a function pointer and free variables. When evaluated, it performs an indirect call to the function pointer with those free variables as arguments.

In order to simplify these classes, rather than having an `Updatable` class per number of free variables, only two classes are created: one without free variables and one with a single generic field. However, this field can be a structure with multiple subfields, thus allowing for more free variables. This field is passed to the method by reference, rather than by value, which saves some time as there's no need to copy the structure onto the stack.

```
public unsafe class Updatable : Thunk
{
    private delegate*<Closure> f;

    public Updatable(delegate*<Closure> f)
        => this.f = f;

    protected override Closure Compute() => f();
}

public unsafe class Updatable<F> : Thunk
{
    private delegate*<in F, Closure> f;
    public F free;
```

```
    public Updatable(delegate*<in F, Closure> f, F free)
    {
        this.f = f;
        this.free = free;
    }

    protected override Closure Compute() => f(in free);

    protected override void Cleanup()
        => this.free = default;
}
```

The `Cleanup` method is responsible for freeing any references to other heap allocated objects, so that they can be collected by the Garbage Collector (GC). Cleanup is invoked after the computation has finished and those free variables are no longer needed. The JIT can use SIMD instructions to clear the whole structure in one swoop.

Fun class family

Similarly to `Updatable` classes, the `FunN` classes take a function pointer and free variables. Below is an example of such a class:

```
public unsafe class Fun1<F, T0, R> : Function
{
    private delegate*<in F, T0, R> funPtr;
    public F free;

    public Fun1(delegate*<in F, T0, R> f, F free)
    {
        this.Arity = 1;
        this.funPtr = f;
        this.free = free;
    }

    public override R ApplyImpl<A0, R>(A0 a0)
    {
        var fun = (delegate*<in F,A0,R>)funPtr;
        return fun(in free,a0);
    }
}
```

The function delegate is strongly typed and therefore requires the `FunN` to have generic parameters that match the methods signature. Moreover, compared to *normal* delegates it can only point at a static method, but provides superior performance (see 4.2.3. *Function pointers vs Delegates*).

SingleEntry class family

Thanks to the demand analysis [SPV17] the compiler can tell that some thunks are evaluated just once. In that case it's not necessary for them to perform updates. For example, list

concatenation `(++)` only evaluates its second argument once, when returning it if the first argument is `[]`, therefore you can pass a one time computation to it.

```
(++) x y = case x of
            [] -> y    -- return y.Eval();
            (x:xs) -> let u = xs ++ y in x : u
```

A thunk-like one time computation is called a *single entry thunk*. A `SingleEntry` takes a function pointer and free variables. When evaluated it makes an indirect call to the function pointer with free variables as arguments. Upon returning the result the `SingleEntry` object is free to be collected by the GC. Below is an example of such a class:

```
public unsafe class SingleEntry<F> : Computation
{
    private delegate*<in F, Closure> f;
    public F free;

    public SingleEntry(delegate*<in F, Closure> f, F free)
    {
        this.f = f;
        this.free = free;
    }

    public override Closure Eval() => f(in free);
}
```

Number of universal classes

How many predefined classes is enough? It is a hard question, because Haskell programs have no limitation on the number of free variables in an expression. Same goes for number of function arguments. An approximation based on a few sample programs can be made. In particular, compiling the Prelude, that is Haskell's standard library, should give a good view of most Haskell programs.

In my efforts of compiling Prelude, described below, I came across functions with up to 8 free variables and up to 7 arguments to apply to. For thunks it was 11 free variables. With the structured free variables it is possible to just have

- 2 updatable classes,
- 2 single entry classes,
- $2n$ function classes, where n is the number of `Apply` methods.

2.4. Standard library

One of the goals of this project is to allow compiling Haskell code to run on Lazer, the lazy .NET runtime. Majority of existing Haskell code uses its Prelude library. It includes primitive operations (e.g. manipulating byte arrays), arithmetic operations, popular type classes, IO operations, system interaction, and data manipulation (lists, character strings, optional `Maybe`, etc).

With the experimental compiler described in 5. *Compiling Haskell to C#* it was possible to compile a very small part of Prelude that allowed running some basic Haskell programs. Compiled modules include

- `GHC.Prim` - primitive operations like throwing exceptions, manipulating arrays, and sequencing evaluation. Many operations described in Haskell's `GHC.Prim` are accessed through other means, native to the .NET platform.
- `GHC.Types` - basic lifted types: `Int`, `Word`, `Char`, `Float`, `Double`, `Bool`, `Ordering`, `List`.
- `GHC.Tuple` - tuple types.
- `GHC.CString` - conversion between .NET's `string` and Haskell's `[Char]`.
- `GHC.Classes` - `Eq` and `Ord` type classes with instances for basic types.
- `GHC.Maybe` - the optional type with `Eq` and `Ord` instances.
- `GHC.Char` - creating characters from integers.
- `GHC.Num` - the `Num` type class with instances for `Int`, `Word` and `Integer`.
- `GHC.Enum` - the `Bounded` and `Enum` type classes with some instances.
- `GHC.List` - list operations.
- `GHC.Base` - most common operations and type classes.
- `GHC.Integer.Type` - arbitrary precision integer arithmetics.

Compiling the base package turned out to be much harder then initially expected. There is complex integration between the GHC compiler and the base library as well as holes in their documentation. The biggest issue is cyclic dependencies between the modules (solved in GHC with hs-boot files), because the experimental compiler currently processes only one file at a time. Therefore, some parts of the modules above had to be removed. Nonetheless, the achieved results are good enough to be usable in benchmarks (see 4.3. *Benchmarks*).

It should be noted that the .NET runtime comes with its own rich standard library. Further investigation needs to be done to see when it can be utilized.

Chapter 3

Comparison to related work

The development of the lazy runtime was influenced by a number of existing solutions. However, most papers on the topic did not provide publicly available source code material, or such source code is no longer available because of time passage. This makes it a bit harder to reason about practical efficiency of those solutions.

First, we'll look at the work done for the Java Virtual Machine (JVM), especially the very robust GHC port called Eta. Then we'll compare the lazy runtime to other .NET based works.

3.1. JVM based solutions

In 2007 Brian Alliet wrote a paper [All07] in which he pointed out previous work done on the subject of translating Haskell to Java and suggested possible improvements. It mentions the works of Mark Tullsen [Tul96], Kwanghoon Choi et al. [CLH01] and Don Stewart [Ste02] and their key differences. It also points out that the use of *eval/apply* application method may be better suiting for the Object Oriented Virtual Machine (OOVM) then the previously popular *push/enter* method [MP04].

One of the biggest challenges on the JVM is the lack of tail calls. For most functional recursive programs this is a huge obstacle. The solution to this problem usually comes down to implementing an iterative interpreter that substitutes recursion. Potentially, a modified version of the JVM supporting tail calls could be used, but this lowers the *portability* of the compiled program as it now has to be shipped with its runtime. The .NET platform supports tail calls out of the box (at least in 64-bit environments).

Moreover, there is no efficient mechanism similar to CLR's function pointers in the JVM. Instead more emphasis is placed on passing around objects that implement a particular interface. Generics on the JVM are also *weaker* in that they don't support primitive types.

There is, however, a somewhat successful implementation of Haskell (particularly GHC Haskell) on the JVM – the Eta platform developed by Typelead. Unfortunately, there seem to be no papers related to its development, but it is open source and available on GitHub.

3.1.1. Eta

Eta is presented as a variant of Haskell. It is based on GHC 7.10.3. The project is composed of a GHC fork with modified backend and additional tooling for building and managing packages.

One of the key features of Eta is its Foreign Function Interface (FFI) that allows to export Haskell functions as Java methods as well as to import and use Java objects. The latter is

done by providing a special `Class` type class that can be evaluated in the `IO` monad.

The Eta runtime is much more complex than this lazy runtime. It provides advanced features for IO, managing threads, mutable variables, selector thunks, Software Transactional Memory, and more. But it is still limited by the JVM and the aforementioned constraints. This leads to more work involved in porting Haskell libraries as special trampolines have to be used to mark recursive functions.

Let's compare the way a `map` function would be compiled on Eta and this lazy runtime:

```
// Eta_map.java
public static class Map extends Function2 {
  public static Map INSTANCE = new Map();

  public static Closure call(StgContext ctx, Closure f, Closure l) {
     Closure l_ = l.evaluate(ctx);
     if (l_ instanceof Nil) {
        return Types.NilInstance();
     } else {
        Cons cons = (Cons)l_;
        MapThunk tail = new MapThunk(f, cons.x2);
        Ap2Upd head = new Ap2Upd(f, cons.x1);
        return new Cons(head, tail);
     }
  }

  public final Closure apply2(StgContext ctx, Closure f, Closure l) {
     if (ctx.trampoline) {
        Stg.apply2Tail(ctx, this, f, l);
     }
     return call(ctx, f, l);
  }
  public final Closure enter(StgContext ctx) {
     if (ctx.trampoline) {
        Stg.enterTail(ctx, this);
     }
     return call(ctx, ctx.R1, ctx.R2);
  }
}
```

The `map` function is represented by a class deriving from `Function2` – a function of arity 2. The computation is described by the static method `call` that evaluates its second argument and checks which data constructor it is. If it is `Cons` then it allocates a new thunk that applies `map` on the function and tail, another thunk that applies the function on the head, and returns a new data cell.

Then two instance methods are shown: `apply2` and `enter`. Eta supports both *eval/apply* and *push/enter* function application methods. Both of those functions check if the computation is executed on a trampoline in which case they have to perform a 'bounce' (unroll the stack) and then call the static method `call`.

```
// Lazer_map.cs
```

```
public static Function map =
    new Fun2<Closure,Closure,Closure>(&map_Entry);

public static Closure map_Entry(Closure f, Closure l)
{
    var l_ = l.Eval();
    switch (l_)
    {
        default: { throw new ImpossibleException(); }
        case Nil l_Nil: { return nil; }
        case Cons l_Cons:
            {
                var h = l_Cons.x0;
                var t = l_Cons.x1;
                var tail = new Updatable<(Closure,Closure)>(
                    &map_tail_Entry, (f, t));
                var head = new Updatable<(Closure,Closure)>(
                    &map_head_Entry, (f, h));
                return new Cons(head, tail);
            }
    }
}
public static Closure map_tail_Entry(in (Closure f, Closure t) free)
{
    return map_Entry(free.f, free.t);
}
public static Closure map_head_Entry(in (Closure f, Closure h) free)
{
    return free.f.Apply<Closure,Closure>(free.h);
}
```

The main difference is that there is no special class for map, but rather a universal class `Fun2` is instantiated with the pointer to the static computation method. The same goes for the thunks created in the `Cons` branch. Thanks to tail calls no trampolines are involved, which increases performance.

The *eval/apply* model has been implemented just a bit differently in Eta which doesn't use generics. Similarly to Lazer there are abstract subclasses `FunctionN` of the base class `Function` that implement all apply methods except `applyN` that is implemented by the actual function implementation. Those already implemented methods take care of creating PAPs and further result applications.

3.2. Mondrian

Mondrian was a lazy functional scripting language for the .NET platform. It was developed during early access to the first version of the runtime. It was the first approach to bring non-strict semantics to .NET.

Two papers have been discovered that describe Mondrian. The first one [MPY01] introduces the language and provides its implementation details. The second paper [PM04] discusses more generally the implementation of Mondrian and other lazy languages for both

.NET and JVM (OOVMs in general). Mondrian is a separate language from Haskell and has been designed for interoperability with other .NET languages.

Let's take an overview look on its implementation. There is no common base class (except for `Object`). Data types are POCO (plain old CLR object) and functions implement the following interface:

```
interface Code {
    public Object ENTER();
}
```

This means there is one class per function. From the interface we can also see that it uses *push/enter* function application model. The argument stack is just a stack of `Objects` which leads to inefficient boxing of primitive values. A trampoline is used to apply functions with exception handling as means of creating PAP objects. This is probably due to a lack of tail calls in the early versions of the .NET runtime.

Mondrian is no longer available in any shape or form (even the projects web domain points to a different site). However, the description of its implementation can show us where optimizations can be done:

- calling an interface method is costly, calling an overloaded method from a base class is cheaper;
- using *push/enter* with boxing can be substituted by seperate stacks for primitive values (see below) or by using *eval/apply* method.

3.3. Haskell.NET

In 2005 appeared the first project to bring Haskell to the .NET platform. Monique Monteiro et al. published a paper [Mon+05] describing the implementation. This work also uses predefined universal classes for thunks and functions and was a primary motivator to do the same for this lazy runtime.

Haskell.NET also uses universal objects for data constructors. Therefore all switching is done on their tags. This is less efficient for small number of alternatives as shown in 4.2.1. *Switching on Tag vs Type*.

For function application *push/enter* method is used with multiple stacks – one for object references and one for each primitve type. This removes costly boxing. However, when evaluating a thunk the current arguments have to be hidden. The state of the stacks has to be saved and set as empty. This simulates the fact that in GHC arguments and update frames share the same stack and the thunk computation shouldn't have access to arguments not meant for it.

The paper mentions a modified version of GHC with a new backend for .NET compilation, which unfortunately is no longer available. Similarly to the compiler described in chapter 5. *Compiling Haskell to C#*, the modified GHC generated code from STG representation. This way it utilizes a lot of GHC's optimizations and obtains explicit closure definitions that are easy to translate.

However, it compiled Haskell straight to CIL rather than to C#. This results in more flexibility in the compiler and removes the need to use a secondary compiler. The downside is the amount of work that has to be put into creating this compiler and its optimizations. Additionally, using C# for Lazer cheaply allowed for error debugging using existing tools.

3.4. Oliver Hunt's work

In 2006 Oliver Hunt published his Master's paper on *„The Provision of Non-Strictness, Higher Kinded Types and Higher Ranked Types on an Object Oriented Virtual Machine"* [Hun06]. In this paper he provides a lot of theoretical details on lazy functional programming and comparison with imperative programming.

Hunt puts more emphasis on strongly typed data constructors in order to simplify interoperability. This way a `Cons<A>` has a head element of type `A` rather than `Closure`. Therefore, the external user of the type can easily know what was it supposed to return and can switch on its variants. However, it may introduce additional monomorphisms, i.e. a `Nil<A>` is no longer generic for all `A` and cannot be shared.

Hunt describes potential support of Higher Kinded Types (HKT) by the means of generic type arguments with partial type erasure. Rather than resorting to *implementation leakage* (see 2.1. *Types and data representation*) in the compiled code, he uses casts to convert from an abstracted (erased) type to a representable type (see chapter VII of his paper). This of course is somewhat costly.

Higher Ranked Types are for example functions with a type parameter that has a universally quantified variable, i.e:

```
f :: (forall s. [s] -> [s]) -> ([a],[b]) -> ([a],[b])
f g (x,y) = (g x, g y)
```

This means that generic type instantiation (here `s`) has to be done on the function application level rather than on the function's type. Otherwise `g` would only support one type - either `a` or `b`. Hunt supports this and describes a solution that utilizes generic methods as well as generic types (see chapter VIII of his paper).

While stronger typing has benefits in terms of readability and interoperability of the compiled code, it adds complexity to the compiler as well as increases the number of classes the runtime library has to provide. In Hunt's approach there has to be a `Thunk` class created for every data type (e.g. `ListThunk<A> : List<A>`), because the specific type is referenced rather than a generic common superclass.

When it comes to function application Hunt removes the need for either *eval/apply* or *push/enter* by performing lambda lifting and explicit partial application at compile time. Functions are implemented with classes extending a set of common base classes for given arity.

Additionally Hunt presents that type classes may be implemented as interfaces with instances implementing those interfaces. The Lazer runtime uses the same unified approach as GHC where instance dictionaries are compiled the same way as data constructors and then for each entry in the dictionary a selector function is created.

Hunt's paper also provides a description for an experimental compiler. It uses GHC to compile Haskell to the Core intermediate language which is then processed to produce C#. Unfortunately, again, access to this compiler wasn't possible.

3.5. Comparison summary

How is this runtime different from the previous solutions for the .NET platform?

1. It focuses on performance. Although, a little bit of performance may be sacrificed for the sake of code readability.

2. It uses *eval/apply* function application strategy which fits very well an OOVM. Additionally, function application is heavily generic, which allows to use performance benefits of passing and returning primitive types the same way as closures, putting the runtime behaviour implementation details on the .NET JIT.

3. It uses function pointers explicitly in C#. While it's hard to say if Monteiro et al. used function pointers, an argument can be made that they did, because they emitted CIL directly and that is the most efficient way to do it.

4. It uses type erasure, but not universal data containers. The interoperability is still limited, but properly named data constructors make it easier to see what's happening during debugging.

However, the main motivation for this work was not to be different, but to provide a complete working runtime to the reader. The classes defined in 2.3. *Runtime implementation* are the exact classes the runtime uses. Therefore, even if some time passes and the runtime's source code disappears, it can be recreated from this paper.

Chapter 4

Performance

There are many considerations to be made when creating an efficient runtime. Some performance issues are easy to identify and others are extremely vague and dependent on very specific details of how the CPU interacts with our code and data.

This chapter presents some of the considered performance aspects. Next, performance tests are presented that support the choices made regarding switching, thunk indirections, and function pointers. Finally, Lazer runtime and compiler are compared to the GHC by running a few benchmarks.

4.1. Performance considerations

While chapter 3. *Comparison to related work* mentioned a lot of similar works, a lot of Lazer development happened before they've been discovered. During those early phases of development the runtime was very slow and with each iteration of work performance gains were discovered.

One of the first mistakes was trying to explicitly check whether the closure is already evaluated or if it should be entered. Extensive branching may lead to an increase in branch prediction fails which slows things down. However, here the culprit was an inefficient type cast. Removing the checks resulted in about 10% performance increase.

Why was the type cast inefficient? As mentioned in 2.2.4. *Performing conditional computations* there is a fast type cast in .NET, but it only works for sealed types. However, the sealed data constructors had a base class that the cast was made to. And the base class obviously could not be sealed.

4.1.1. Garbage Collection

In .NET there is a global GC that stops all threads and performs the collection. The stoppage may occur at any time when the program runs out of free allocation space. It is a generational tracing GC, which means that short living objects are cheaper to collect. Still, the less objects are allocated, the less time is spent in GC and the program is faster.

The way the GC works is that it traverses the stack looking for pointers. References between objects form a graph with the stack being a root (and static objects). If there is no path from the root to an object, then this object can be collected. A side effect of this implementation is that non-tail recursive functions that grow the stack make the traversal a bit more expensive.

When dealing with lots of short lived objects it may be a good idea to implement some kind of pooling mechanism to decrease the number of allocations and the frequency of the GC process. The singleton pattern can be utilized for argumentless data constructors and local functions without free variables.

4.1.2. Data structures

During the development of the runtime a question was posed whether heap based continuations would alleviate the problems of limited stack space. The idea has been turned down due to a performance slowdown. However, with that implementation came an issue of choosing the correct data structure for a continuation stack. An array based stack proved to be faster then linked list of references. But the most efficient approach was to form a linked chain from the continuation objects themselves, by giving each a reference to the next continuation. This way much fewer allocation were made, while keeping the O(1) cost of push and pop operations.

4.2. Performance Tests

It often happens that we perceive one way as faster then the other, but in reality things are different. The best way to handle that is to run performance tests, because those usually don't lie. Below are four tests I felt necessary to present to support claims I make in this paper, regarding the efficiency of the runtime.

4.2.1. Switching on Tag vs Type

As explained in 2.2.4. *Performing conditional computations* there are two ways to switch on expressions, by type casting or by switching on the constructor's tag. This test is meant to compare performance of both. The following Haskell code has been implemented in C#:

```
data O = O0 Int# | O1 Int# | ... | O9 Int#

extractO :: O -> Int
extractO o =
    case o of
        O0 o0 -> I# o0
        O1 o1 -> I# o1
        ...
        O9 o9 -> I# o9

suma !a []     = a
suma !a (x:xs) = suma xs (a+x)

take 0 _      = []
take _ []     = []
take n (x:xs) = x : take (n-1) xs

loop :: O -> [O]
loop o = o : loop o

test o = suma 0 (take 500000 $ map extractO $ loop o)
```

The test was ran on constant sequences of `O0`, `O4` and `O9` to see if there is a noticeable difference in run time between linearly increasing time spent on switching on types versus indirect jump on the tag in constant time.

		MIN	AVG	MAX	σ
O0	Switch Type	164ms	171ms	189ms	6.8
	Switch Tag	167ms	172ms	186ms	5.3
O4	Switch Type	168ms	173ms	200ms	7.2
	Switch Tag	165ms	173ms	195ms	8.5
O9	Switch Type	170ms	177ms	201ms	7.3
	Switch Tag	169ms	174ms	185ms	5.4

The results of both types of switches are very close. Switching on types is initially a little faster, however, it does get slower with more alternatives. The reason is that indirect jumps are more expensive than comparisons, but the number of comparisons increases, while the cost of an indirect jump and a cast is constant. Switches with 5 or more alternatives should choose them by tag.

4.2.2. Dynamic Thunk indirection call

In 2.3.4. *Abstract Thunk class* a mechanism has been shown were on each call to `Eval` a comparison is made to check if the indirection pointer is null. This may seem inefficient and could potentially be implemented by calling a "dynamic" method. However, .NET objects reference their types' method tables and cannot modify them. This forces the use of an indirect call.

```
public unsafe abstract class Thunk : Computation {
    public Closure ind;
    private delegate*<Thunk,Closure> eval = &PerformCompuation;

    public override Closure Eval()
        => this.eval(this);

    private static Closure PerformComputation(Thunk t) {
        t.ind = Blackhole.Instance;
        t.ind = Compute();
        eval = &ReturnIndirection;
        t.Cleanup();
        return t.ind;
    }

    private static Closure ReturnIndirection(Thunk t)
        => t.ind.Eval();
}
```

The following tests were performed on a slightly simplified implementation, keeping the idea intact:

		MIN	AVG	MAX
Create a million thunks	Comparison	0.3ms	0.3ms	0.5ms
	Dynamic method	7.0ms	7.5ms	10.2ms
Call `Eval` a million times	Comparison	1.0ms	1.1ms	1.5ms
	Dynamic method	3.5ms	3.6ms	4.0ms

It turned out that the cost of an indirect call is much higher then the cost of comparison, not to mention the additional cost of assigning the dynamic method in the constructor.

4.2.3. Function pointers vs Delegates

In .NET the legitimate way of passing functions around are method delegates. The standard library provides universal classes such as `Action` and `Func` that are created from a function pointer and possibly an additional reference to an object in case of instance methods.

A small test example has been created to measure the time it takes to create a million delegates or pointers and then measure the time it takes to invoke a single one a million times.

		MIN	AVG	MAX
Create	Delegate	22ms	23.4ms	35ms
	Pointer	8.8ms	9.4ms	12.7ms
Invoke	Delegate	4.6ms	4.8ms	5.6ms
	Pointer	4ms	4.3ms	5.4ms

The results show that invocation takes approximately the same time (there's one additional memory read for delegates per invocation), but there's nearly a 3x difference in creation time. This proves pointers to be a superior solution when efficiency is required.

4.2.4. Function application

Unknown function application is slower than known function application that can be performed by a direct call. To support this a simple test has been created that performs addition in a loop (100 thousand times).

	MIN	AVG	MAX
Direct call inlined	0.09ms	0.09ms	0.12ms
Direct call no-inline	0.27ms	0.28ms	0.37ms
Static application	0.30ms	0.31ms	0.40ms
Generic application	0.93ms	0.91ms	1.19ms

The inlining is done by the JIT when a function is small. The actual application process is very close to direct calls. However, the Lazer runtime uses overloaded generic methods for application and those have an additional invocation cost, because the .NET runtime needs to check if the method with that particular type signature has been emitted by the JIT.

While this is a fairly big performance hit, in practice unknown function application happens less often then known function application. Moreover, the cost of Garbage Collection exceeds the generic application costs by a an order of magnitude.

4.3. Benchmarks

While the Lazer runtime is efficient in theory, practical benchmarks needed to be ran in order to compare its performance with GHC. If the speed difference is too large than it's not so clear if other benefits of bringing Haskell to .NET outweigh the performance loss.

4.3.1. Running tests

The following benchmarks are performed on three platforms: the Lazer runtime described in this paper, GHC interactive (via runhaskell), GHC compiled. Lazer programs were created by using the compiler described in 5. *Compiling Haskell to C#*. The GHC interactive runs are mostly the slowest ones, thus they form a nice upper bound – if Lazer was slower then it is clearly inefficient.

In order to test the computation multiple times within a single process instance, to circumvent runtime initialization costs, the computations had to be wrapped in a non-constant function. Otherwise, the *optimizing* compiler GHC replaced the function call with a thunk and the test would only run once.

4.3.2. Nofib: exp3_8

This test is from the nofib suite and is meant to test the efficiency of arithmetic operations (addition and multiplication) on natural numbers in unary form. Below are the Haskell source code and test results:

```
infix 8 ^^^

data Nat = Z | S Nat deriving (Eq,Ord)
instance Num Nat where
    Z   + y   = y
    S x + y   = S (x + y)
    x   * Z   = Z
    x   * S y = x * y + x
    fromInteger x = if x < 1 then Z else S (fromInteger (x-1))

int :: Nat -> Int
int Z     = 0
int (S x) = 1 + int x

x ^^^ Z   = S Z
x ^^^ S y = x * (x ^^^ y)
```

		MIN	AVG	MAX
`int (3 ^^^ 8)`	GHC (compiled)	0.160s	0.163s	0.168s
	GHC (interpreted)	1.87s	1.89s	1.91s
	Lazer (JIT)	1.09s	1.14s	1.30s
`int (3 ^^^ 9)`	GHC (compiled)	1.96s	1.98s	2.0s
	GHC (interpreted)	17.3s	17.6s	17.9s
	Lazer (JIT)	15.3s	15.6s	16.9s

We can see that Lazer is about 7-8x slower than compiled GHC. It's highly likely to be due to a large number of allocations and lengthy GC process. GHC's GC is optimized for a lazy functional language, whereas .NET's GC is more general purpose.

4.3.3. Nofib: digits of e

This test is from the nofib suite and is meant to test the efficiency of arithmetic operations on arbitrary precision integers. The GHC `Integer` implementation uses GNU GMP library.

The .NET implementation uses `System.Numerics.BigInteger`. Below are the Haskell source code and test results (function executed 100 times):

```
type ContFrac = [Integer]

eContFrac :: ContFrac
eContFrac = 2:aux 2 where aux n = 1:n:1:aux (n+2)

ratTrans :: (Integer,Integer,Integer,Integer) -> ContFrac -> ContFrac
ratTrans (a,b,c,d) xs |
  ((signum c == signum d) || (abs c < abs d)) && -- No pole in range
  (c+d)*q <= a+b && (c+d)*q + (c+d) > a+b       -- Next digit is determined
     = q:ratTrans (c,d,a-q*c,b-q*d) xs
  where q = b `div` d
ratTrans (a,b,c,d) (x:xs) = ratTrans (b,a+x*b,d,c+x*d) xs

takeDigits :: Int -> ContFrac -> [Integer]
takeDigits 0 _ = []
takeDigits n (x:xs) = x:takeDigits (n-1) (ratTrans (10,0,0,1) xs)

e :: Int -> [Integer]
e n = takeDigits n eContFrac
```

		MIN	AVG	MAX
`e 50 · 100`	GHC (compiled)	0.36s	0.35s	0.38s
	GHC (interpreted)	1.90s	1.91s	1.96s
	Lazer (JIT)	0.45s	0.48s	0.56s
`e 150 · 100`	GHC (compiled)	3.48s	3.50s	3.55s
	GHC (interpreted)	17.9s	18.0s	18.2s
	Lazer (JIT)	5.25s	5.35s	5.47s

Here Lazer is around 1.5x slower than compiled GHC. There is a difference in the underlying representation of arbitrary precision integers and their operations implementations, which is definitely a factor.

4.3.4. Nofib: primes

This test is from the nofib suite and is meant to test the efficiency of list operations (`map`, `filter`, `iterate`). Below are the Haskell source code and test results (function executed 100 times).

```
isdivs :: Int  -> Int -> Bool
isdivs n x = mod x n /= 0

the_filter :: [Int] -> [Int]
the_filter (n:ns) = filter (isdivs n) ns

prime :: Int -> Int
prime n = map head (iterate the_filter [2..n*n]) !! n
```

		MIN	AVG	MAX
`primes 500 · 100`	GHC (compiled)	0.77s	0.78s	0.82s
	GHC (interpreted)	3.95s	4.00s	4.44s
	Lazer (JIT)	1.85s	1.90s	2.07s
`primes 1500 · 100`	GHC (compiled)	4.0s	4.0s	4.1s
	GHC (interpreted)	37.9s	38.1s	38.4s
	Lazer (JIT)	29.2s	29.9s	31.4s

Here Lazer is 2.5∼7x slower than compiled GHC. Unknown function application and the garbage collections are what slows down code the most. Here we can see that the longer chains of lists allocate more long living objects, thus populating older GC generations which take more time to collect.

4.3.5. Sums

In this test a few different approaches were applied to obtain the sum of 100 thousand natural numbers. Below is the Haskell source code and test results:

```
sum [] = 0::Int
sum (x:xs) = x + sum' xs

suma [] !acc = acc::Int
suma (x:xs) !acc = suma xs (x+acc)

sumfold = foldl (+) (0::Int)
foldl f x0 h = go x0 h
    where
        go x0 [] = x0
        go x0 (x:xs) = go (f x0 x) xs

inf = 1 : map (+1) inf
```

		MIN	AVG	MAX
`sum (take 100000 inf)`	GHC (compiled)	2.27ms	3.36ms	7.54ms
	GHC (interpreted)	94.6ms	106ms	144ms
	Lazer (JIT)	2.30ms	2.55ms	5.18ms
`suma (take 100000 inf) 0`	GHC (compiled)	1.06ms	1.18ms	2.08ms
	GHC (interpreted)	75.2ms	76.9ms	84.0ms
	Lazer (JIT)	1.23ms	1.35ms	2.8ms
`sumfold (take 100000 inf)`	GHC (compiled)	1.08ms	1.23ms	2.44ms
	GHC (interpreted)	59.5ms	71.1ms	100.8ms
	Lazer (JIT)	1.42ms	1.63ms	3.80ms
`sum [1..100000]`	GHC (compiled)	2.06ms	2.97ms	6.47ms
	GHC (interpreted)	40.7ms	52.5ms	105ms
	Lazer (JIT)	1.65ms	1.79ms	3.80ms
`suma [1..100000] 0`	GHC (compiled)	0.84ms	0.95ms	1.28ms
	GHC (interpreted)	31.4ms	32.6ms	37.2ms
	Lazer (JIT)	0.80ms	0.90ms	2.00ms
`sumfold [1..100000]`	GHC (compiled)	0.60ms	0.76ms	2.65ms
	GHC (interpreted)	15.0ms	27.3ms	63.8ms
	Lazer (JIT)	0.82ms	0.89ms	2.69ms

The addition operation in those tests is executed eagerly. Thanks to strictness analysis special worker functions are created, which resamble imperative loops. Lazer is mostly within the 1-1.5x slower range.

4.3.6. Tight arithmetic loops

Both GHC and Lazer show a speed up when performing operations eagerly for tight arithmetic loops. Below are two loops in Haskell and test results:

```
fiba :: Int -> Int -> Int -> Int
fiba !a !b n = if n <= 0 then a else fiba b (a+b) (n-1)

fibt = fiba 0 1

sumFromTo :: Int -> Int -> Int
sumFromTo from to = go from to 0
    where go !f !t !n | f > t     = n
                      | otherwise = go (f+1) t (n+f)
```

		MIN	AVG	MAX
`fibt 800000`	GHC (compiled)	0.85ms	0.90ms	1.01ms
	GHC (interpreted)	645ms	659ms	709ms
	Lazer (JIT)	0.68ms	0.78ms	1.06ms
`sumFromTo 1 100000`	GHC (compiled)	0.061ms	0.063ms	0.097ms
	GHC (interpreted)	81ms	83ms	92ms
	Lazer (JIT)	0.057ms	0.066ms	0.25ms

In those tight loops Lazer performs just as well as GHC or even a little faster.

4.3.7. Uknown function application

This test measures the cost of calling `Apply` on an unknown function.

```
loopApp :: (Int -> Int) -> Int -> Int -> Int
loopApp f w 0 = f 0 + w
loopApp f w n = loopApp f (f n + w) (n-1)

loopCall :: Int-> Int-> Int
loopCall w 0 = add1 0 + w
loopCall w n = loopCall (w + add1 n) (n-1)

{-# NOINLINE add1 #-}
add1 :: Int -> Int
add1 x = x+1
```

		MIN	AVG	MAX
`loopApp add1 0 100000`	GHC (compiled)	0.88ms	0.94ms	1.19ms
	GHC (interpreted)	79.4ms	95.6ms	129ms
	Lazer (JIT)	0.91ms	0.94ms	1.32ms
`loopCall 0 100000`	GHC (compiled)	0.460ms	0.511ms	0.823ms
	GHC (interpreted)	71.4ms	94.3ms	125ms
	Lazer (JIT)	0.27ms	0.28ms	0.35ms

For compiled GHC unknown function application is at most 2x as slow as known function application. For Lazer its 3-4x slower. See 4.2.4. *Function application* for details.

Why is Lazer faster than GHC when using direct calls? This probably comes from the fact that GHC checks stack depth on function entry, whereas .NET uses *Segmentation fault* system interrupt to handle stack overflows. This is because GHC can use potentially infinite stack space and has to allocate more space when it runs out.

4.3.8. Summary

The benchmarks clearly show that Lazer runtime is performing best when it's doing strict evaluation. Unknown functions application and large amounts of object allocations are the major causes of slowdown. Some of the slowdowns can be fixed by making a better compiler, others are unfortunately bound to the runtime limitations.

Overall, the performance is decent and the runtime can be used for example with graphical user interfaces that don't require extremely low latency. The processing parts that do require high performance have to be implemented eagerly or reference external .NET libraries.

Chapter 5

Compiling Haskell to C#

The primary objective of this paper was to create a lazy runtime. However, translating Haskell code to C# by hand is a rather tedious task. Thus, an experimental compiler has been developed to aid this process. The compilation isn't fully automated and the produced code may require a few fixes here and there.

This chapter presents the steps taken to create the compiler as well as what the compiler does and what could be better. The compiler has been named `stg2cs` after the fact that it performs translation of Haskell in STG form into C#.

5.1. GHC compiler library

The Glasgow Haskell Compiler (GHC) can be used as a library by referencing the `ghc` package. This means that it is not necessary to fork the whole compiler. Instead only the important parts can be referenced.

Currently `stg2cs` can only compile one file at a time. First, the GHC loads and compiles the Haskell source code, producing `.hi` and `.o` files. Next, custom code can access the compilation pipeline and repeat the compilation process having therefore access to every step of the compilation. After generating and optimizing STG representation of closures the converter to C# is called.

5.1.1. Simplified example

The following code shows how to get declared types and bindings (functions and thunks) from the GHC. The imports have been removed for brevity.

```
compile :: String -> IO ([StgTopBinding], [TyCon])
compile moduleName = runGhc (Just libdir) $ do
    env <- getSession

    -- setup flags (i.e. optimizations to perform)
    dflags_ <- getSessionDynFlags
    let dflags = customModify dflags_
    setSessionDynFlags dflags

    target <- guessTarget moduleName Nothing
    addTarget target
    load LoadAllTargets  -- here GHC does the full compilation
```

```
depanal [] True
modSum <- getModSummary $ mkModuleName moduleName

-- redo the compilation steps
pmod <- parseModule modSum      -- ModuleSummary
tmod <- typecheckModule pmod    -- TypecheckedSource
dmod <- desugarModule tmod      -- DesugaredModule
let coreMod = coreModule dmod   -- CoreModule

-- extract information for further processing
let mod   = ms_mod modSum
let loc   = ms_location modSum
let core  = mg_binds coreMod
let tcs   = filter isDataTyCon (mg_tcs coreMod)

-- optimize Core and prepare for further processing
let env' = env {hsc_dflags = dflags'}
guts' <- liftIO $ core2core env' coreMod
let core' = mg_binds guts'
(prep, _) <- liftIO $ corePrepPgm env' mod loc core' tcs

-- convert Core to STG and run optimizations
let (stg,_) = coreToStg dflags' mod prep
stg_binds2 <- liftIO $ stg2stg dflags' stg

return (stg_binds2, mg_tcs coreMod)
```

5.1.2. Working with GHC source code

While it may not feel like `stg2cs` is complex, the hard part of creating it was definitely a lack of examples on how to interact with the GHC. Some information could be obtained on the wiki of the GHC repository, but mostly I had to figure out in which module there might be a function that was needed, read its source code to make sure it is indeed what I want and how to use it, read any comments surrounding it, and finally try to use it in my code.

Hackage is a very good platform for browsing source code, because it gives you a search box, if you can guess the functions name, and when viewing source there are links to declarations of the functions used. Although, I believe it doesn't support C preprocessor declarations, which break the links to declared functions in the module that uses CPP.

5.2. Conversion from STG to C#

Following the steps outlined in the previous section we were able to obtain closure bindings in STG form as well as declared types that need to be translated. In the following subsections we'll dive into how this information is used to produce equivalent C# code.

5.2.1. Module representation

While GHC performs further compilation to `C--` and thus uses the C way of exposing labeled elements, like code and data, a different approach had to be used for C#. A module is

represented by a static class with fields for top level closures and constants, and methods for their computations. If the module name is dot-separated then the last part is the name of the class whereas the initial part forms the namespace.

When a static class is referenced in executed code for the first time, the .NET runtime stops current execution to first make sure the static fields are initialized. The initialization is performed by a static constructor, which in C# is declared like a normal constructor, but with just the static modifier and no arguments.

5.2.2. Processing types

There are two kinds of types that have actual representation: data types and type classes (their dictionaries). We extract data constructors from the types and extract the crucial information:

1. data constructor name,
2. data constructor integer tag,
3. data constructor represantable arguments.

The constructor name is processed by the `safeConName` function (see 5.2.3. *Safe names*). The tag is just a number, so it's directly copied. The arguments are types, which are translated according to the rules in 5.2.2. *Translating types*. There is one type that has no actual representation and it is the state token. While it may appear in the data constructor definition it is not used when applying the constructor to arguments and therefore it is removed from the C# representation. Data constructor types are emitted as inner classes in the module class, defined as in 2.1. *Types and data representation*.

For type classes a dictionary data constructor is processed which contains fields for any other dictionaries it depends on (superclasses) and all the declared instance functions and constants. We also need to emit selector functions that extract the requested function from the dictionary. For example, given a `Functor` type class:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

We create a dictionary and a selector function:

```
public sealed class CColFunctor : Data
{
    public Closure x0;
    public CColFunctor(Closure x0)
    {
        this.x0 = x0;
    }
    public override int Tag => 1;
}

public static Function fmap = new Fun1<Closure,Closure>(&fmap_Entry);
public static Closure fmap_Entry(Closure a0)
{
    var dict = a0 as GHC.Base.CColFunctor;
```

```
    var dictItem = dict.x0;
    return dictItem.Eval();
}
```

Translating types

Most Haskell types will be lifted and therefore uniformly represented by the `Closure` type. There's more to do when representing unlifted types and there's also a special case for levity polymorphism.

Basic types are translated directly according to the following table:

`Int#`	→	`long`
`Int64#`	→	`long`
`Word#`	→	`ulong`
`Char#`	→	`char`
`Float#`	→	`float`
`Double#`	→	`double`
`ByteArray#`	→	`byte[]`
`MutableByteArray#`	→	`byte[]`

Then there are unlifted tuple types. In C# 7 a new syntax for tuples was added as well as a family of generic types `System.ValueTuple`. GHC tuple representation is a bit complicated, because returing a tuple means returning multiple values in different registers. In C# we leave the runtime representation up to the JIT. So the `RuntimeRep` arguments of a tuple type are dropped. Then if the state token is part of a tuple it is also dropped. Finally if a tuple has only one element then we drop the notion of it being a tuple and just treat it as just the argument type.

There are some cases when expressions actually reference the state token or a void token, e.g. in imperative monadic compuations. Rather than removing those, they are represented by empty C# structs which are eliminated on the JIT level.

Finally, there's levity polymorphism [EP17]. Essentially, this notion describes functions that can return either a lifted type or an unlifted type, depending on the caller context. In GHC it is much more complex than it is in C#, because here we can just use generics. So a levity polymorphic function will be represented by a generic method with one generic type parameter. Whenever an expression has a type that is neither lifted or unlifted it is just generic.

```
public static GenericR doll_Entry<GenericR>(Closure f, Closure a)
{
    return f.Apply<Closure, GenericR>(a);
}
```

5.2.3. Processing bindings

Now that we have processed the types it's time to process bindings – all thunks, closures and constants of the module.

Sorting declarations

First we need to sort the declarations into delayed closures, like thunks and functions, and initial closures, i.e. constructed constants. All top level closures may only have free variables

that are also top level closures. Thunks and functions can be called *delayed*, because they will only need to access static fields after the static class initialization has been completed. On the other hand, top level constants that are applications of data constructors need to access static fields during the initialization, e.g. a constant list `[1]` will reference the constant number `1`.

The order of initialization for delayed closures doesn't matter, because it only relies on loading a pointer to the computation and creating a closure object. For initial closures a recursive `let` expression is generated that handles any dependencies between them.

Gathering callables

As noted in 4.2.4. *Function application*, function application is slower than direct calls. Therefore an optimization is introduced. Whenever we declare a function that has no free variables (other than top level) it is marked as *callable* and we may call its method directly. All top level functions are callable.

Translating expressions

The core of Haskell programs are expressions that can be translated into C# statements fairly easily. There are some quirks mentioned below that need to be kept in mind. Every expression is placed inside a static method that can be used to invoke that computation.

Literals Literals are translated one to one, except for some character escape sequences that have symbolic representation in Haskell, but need hexadecimal representation in C#.

```
1##   =>   1.0
```

Application Application has a function and arguments. But it may also have no arguments. In that case it's the value of the function (or other type of variable). Here we also check the environment to see if the function is callable and if so we emit a method call rather than a slow application.

```
    x  =>  x
  f a  =>  f.Apply<T,R>(a)
cal a  =>  cal_Entry(a)
```

Constructor application Data constructor application is trivial, as it's always saturated. However, in order not to create many identical instances of argumentless constructors, their static wrapper is used.

```
(:) x y  =>  new Cons(x, y)
     []  =>  nil_DataCon
```

Operation application Operators are translated to equivalent operations in .NET. Some don't have direct equivalents (e.g. `IntQuotRemOp` is translated to two instructions instead of one). GHC defines a lot of operations and currently the compiler supports only the most frequently used onces.

```
                  +# x y  =>  x + y
uncheckedIShiftRL# x s    =>  (long)((ulong)x >> (int)s)
```

What's worth noting is that operators are defined in a Haskell usable way in the `GHC.Prim` module. However, the data structure describing them uses different names, e.g. the shift above is `ISrlOp`.

Alternative selection Case expressions assign the evaluated value of the expression to an alias and then do alternative selection. There's additional logic to deal with switching on type vs on tag (see 2.2.4. *Performing conditional computations*).

```
case f x of (as pf)      var pf = f.Apply<Closure,Closure>(x);
    [] -> e1         =>  switch(pf) {
    (:) h t -> e2         default: throw new ImpossibleException();
                           case Nil pf_Nil: return <e1>;
                           case Cons pf_Cons:
                               var h = pf_Cons.x0;
                               var t = pf_Cons.x1;
                               return <e1>;
```

While in theory case expressions should only be done on simple expressions, i.e. applications, there are cases when a more complex expression is given. To handle such a case the compiler generates a method that computes the subexpression (if it's small enough it may be inlined by the JIT).

Let bindings Non-recursive binding are treated like recursive bindings of just one element. First there's analysis done on the free variables of each binding to determine recursive assignments. Then each right handside is emitted - either a data constructor application or a closure (i.e. thunk, function). Free variables of closures are passed via tuples. Every recursive free variable is replaced by `null` when instantiating closure objects and after initialization's completed they're assigned back. The closure's computation expression is placed in a new static method.

```
let f =                  var f_Free = ((Closure)null, n);
    \[f n] r [x] ->      var f = new Fun1<(Closure,long)>(&f_Entry, f_Free);
        ...          =>  f.free.x0 = f;
in ...                   ...
```

Safe names Haskell is rather loose with what can be a name, whereas C# isn't. This meant that all symbols in operators had to be converted to words, apostrophes removed, etc. For `C--` name generation GHC uses z-encoding that escapes special characters (e.g. `#` becomes `zh`) and this can be borrowed. Another issue was conflicting names. GHC assigns each variable a unique value that is appended to its name if the variable is only used locally. Finally, a module name is added.

This results in `safeVarName` and `safeConName` functions in the compiler. They work great 90% of the time. Unfortunately, GHC is riddled with weird undocumented quirks that make this a nightmare for the other 10% of cases.

For instance, local worker functions may be used from other modules as an optimization, but their not set to external in single module compilation, so I cannot check that and prevent appending a unique value, which breaks callers code. Or sometimes GHC just adds a digit at the end of an imported name. This is one of the reasons why the compiler's output may have to be manually fixed. I suspect most of the issues would disappear if all of the source files were processed in a single GHC session.

5.2.4. Simplifying generated expressions

To simplify readability of the generated code a few simplification rules are applied:

1. If a case expression has just one branch, skip the switch;
2. If a case expression has just one data constructor branch and a default impossible branch then use a cast and skip the switch;
3. If the result of an application of method call is being evaluated, you can skip the evaluation (computations have to return evaluated results).

The first two are meant to lower the indentation level (which grows pretty quickly) and the last saves a little bit of time.

5.2.5. Emitting the code

The processing of STG bindings and types is done in a state monad and results in creating a data structure that represents particular C# declarations, expressions and statements. These structures are instances of the GHC `Outputable` class where the pretty printer is used to actually produce C# code in text form. The text output is then printed to the user.

5.3. Debugging Haskell code

One of the interesting ideas was to enable debugging of Haskell code after it has been compiled to C#. The .NET platform has very good debugging tools and this could be leveraged to allow for debugging Haskell code. However, it is not easy to properly support it. GHC performs a lot of code transformations to increase performance, which means the generated code looks very differently than the source code. Luckily there is an option to generate code mapping in the DWARF format (GHC `-g` flag).

Unfortunately the C# compiler doesn't provide enough flexibility for external source information. Below is an example of what could be done, however, the current implementation of the Haskell to C# compiler doesn't emit source mapping information.

5.3.1. Code mapping

The Roslyn compiler provides few preprocessor directives for use in C# source code. One of them is the `#line` directive which allows to annotate generated source code with information about its origin. This may be used to allow setting breakpoints in Haskell source code, while debugging the .NET program.

```
1:  map f l =
2:      case l of
3:          [] -> []
4:          (x:xs) ->
5:              let nx  =
6:                       f x
7:                  nxs =
8:                       map f xs
9:              in nx : nxs
```

```
  public static Closure map_Entry(Closure f, Closure l)
  {
#line 2 "map.hs"
      var l_ = l.Eval();
      switch (l_)
      {
          default: { throw new ImpossibleException(); }
#line 3 "map.hs"
          case Nil l_Nil: { return nil; }
          case Cons l_Cons:
          {
#line 4 "map.hs"
              var x = l_Cons.x0;
              var xs = l_Cons.x1;
#line 5 "map.hs"
              var nx = new Updatable<(Closure,Closure)>(
                  &map_head_Entry, (f, x));
#line 7 "map.hs"
              var nxs = new Updatable<(Closure,Closure)>(
                  &map_tail_Entry, (f, xs));
#line 9 "map.hs"
              return new Cons(nx, nxs);
          }
      }
  }
  public static Closure map_tail_Entry(in (Closure f, Closure xs) free)
  {
#line 8 "map.hs"
      return map_Entry(free.f, free.xs);
  }
  public static Closure map_head_Entry(in (Closure f, Closure x) free)
  {
#line 6 "map.hs"
      return free.f.Apply<Closure,Closure>(free.x);
  }
```

Unfortunately, this preprocessor directive is limited to just the line number. This means that column information will be taken from the C# file rather than the Haskell file and wrong part of the line may be highlighted in the IDE. If there are multiple expressions on the same line, multiple usages of the directive will be ignored and only the last one may be used during debugging.

It's worth noting that the .NET debugger uses unoptimized code by default which doesn't use tail calls and it's easy to overflow the stack. For proper debugging one should enable *Optimize* flag in the C# project and disable *Just My Code* debugger option.

Chapter 6

Conclusions and future work

Compiling lazy functional languages isn't a trivial task and thanks to the research done on that topic in the last 40 years new solutions can be created more easily. Lazer, the .NET lazy runtime introduced in this work builds upon the work of people who previously tried to achieve the same goal – bring Haskell to .NET.

The runtime in the current form is working decently and is small enough to be easily explained, provided the reader has prior knowledge of the C# language. The experimental compiler was very useful in attempting to compile the Haskell's `Prelude` library and `nofib` benchmarks. It showed that the way code is compiled to work within the runtime is also very important and the compiler itself has to take care of many optimizations.

Many scientific papers provide theoretical information on the solution, but lack practical examples of performance tests. Often we might perceive something as slow or fast, but without actual benchmarks we may be incorrect. I was able to test my runtime (with the compiler) and show when it works as fast as GHC and when it doesn't, providing a reference for future implementations whether they are an improvement or not.

When choosing this topic I hoped to arrive at a great spot that merges the purity of a lazy functional language and the versatility of a mature programming framework such as the .NET platform. Unfortunately, a lot of performance issues linked to increased GC pressure and generic virtual methods may make it very hard for programmers to adopt Haskell on a the .NET CLR. It would be interesting to see the overall performance of a lazy .NET runtime in a more real world scenario.

There is still a lot of potential for further improvements, especially on the compiler side. Let's take a look at the areas that could be further worked on.

6.1. Areas of improvement

6.1.1. GHC integration

The `stg2cs` compiler is a proof of concept which shows that compiling Haskell to C# is generally possible. In order to properly compile programs that have many files and to make better use of GHC utilities, more work has to be put into forking the GHC, adding a .NET backend and emitting .NET assemblies directly, which will also allow to provide proper debug symbols. However, this means some code optimizations done by the Roslyn compiler may have to be reimplemented.

6.1.2. Multithreading

Currently, the runtime is only working in a single thread context. In order to allow multithreading Lazer's standard library would need to provide functions for managing threads or paralelizing computations. At the runtime level there's one thing that is required and that is thunk evaluation synchronization. Adapting work in [HMP05] may be non-trivial, but .NET platform has a rich multithreading toolset, which should make things easier.

Initial test with a simple implementation based on locks showed a slight performance degradation in a single thread context.

6.1.3. Foreign imports

During the porting of the `GHC.Integer.Type` module from the GHC's base package, foreign imports were used as stubs for `BigInteger` operations. However, current system validates what is imported and declines declarations that don't conform to C function names. This meant that imported functions had to be defined in the same class as the module.

In order to support proper imports of static methods from the .NET world, modifications would have to be done in the GHC itself, adding new foreign import types with different validation rules. While it is currently not possible to declare a foreign import with a lifted type (or to declare a new unlifted type), additional support for non-constructible types could be added.

```
data BigInteger#

foreign import dotnet unsafe "System.Numerics.BigInteger.Compare"
    compareBigInt# :: BigInteger# -> BigInteger# -> Int#
```

6.1.4. Compiler optimizations

There are a couple of things that could be improved in the `stg2cs` compiler, but there wasn't enough time to get them done.

- Partially applied functions that are later applied in the same computation with all arguments can be modified into direct calls.
- Let bindings that are shared among multiple branches as a final expression (`let-no-escape`) can skip allocations (there's quite a lot of work here).
- Local recursive functions should recurse by direct calls rather than application.

6.1.5. Providing libraries

Bringing Haskell to a new platform is an opportunity to provide new libraries. Compiling the `Prelude` is mandatory, as it is defined in the Haskell standard. It may even be possible to compile Haskell libraries that use native libraries with .NET's Platform Invoke. But whenever the solution is provided in the .NET standard libraries it should be favoured.

Bibliography

[All07] B. Alliet. *Efficient Translation of Haskell to Java Master's Thesis Proposal*. 2007.

[CLH01] K. Choi, H. Lim, and T. Han. "Compiling Lazy Functional Programs Based on the Spineless Tagless G-machine for the Java Virtual Machine". In: *FLOPS* (2001).

[Cla+80] T.J.W. Clarke, P.J.S. Gladstone, C.D. MacLean, and A.C. Norman. "SKIM – The S, K, I Reduction Machine". In: *Proceedings 1980 LISP conference* (1980).

[CCM85] G. Cousineau, P.-L. Curien, and M. Mauny. *The categorical abstract machine*. 1985.

[Cur86] P.-L. Curien. *Categorical Combinators*. 1986.

[Cur36] H. Curry. "Functionality in combinatory logic". In: *Proceedings of the National Academy of Sciences (U.S.A.)* XX (1936).

[DM90] A. Davie and D. McNally. *CASE – A Lazy Version of an SECD Machine with a Flat Environment*. 1990.

[EP17] R.A. Eisenberg and S. Peyton Jones. *Levity Polymorphism*. 2017.

[FW87] J. Fairbairn and S. Wray. "Tim: a simple, lazy abstract machine to execute supercombinators". In: *Proceedings of 1987 Functional Programming Languages and Computer Architecture Conference* (1987).

[HMP05] T. Harris, S. Marlow, and S. Peyton Jones. *Haskell on a Shared-Memory Multiprocessor*. 2005.

[HBG19] V. Hayot-Sasson, S.T. Brown, and T. Glatard. "Performance Evaluation of Big Data Processing Strategies for Neuroimaging". In: *CCGRID* (2019).

[Hun06] O. Hunt. *The Provision of Non-Strictness, Higher Kinded Types and Higher Ranked Types on an Object Oriented Virtual Machine*. 2006.

[Joh84] T. Johnsson. "Efficient Compilation of Lazy Evaluation". In: *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction* (1984).

[Kie85] R.B. Kieburtz. "The G-machine: a fast, graph-reduction evaluator". In: *Technical Report CS/E-85-002, Dept. of Computer Science, Oregon Graduate Center* (1985).

[Lan64] P.J. Landin. "The Mechanical Evaluation of Expressions". In: *Computer Journal* 6.4 (1964).

[Let02] P. Letouzey. "A New Extraction for Coq". In: *TYPES* (2002).

[MP04] S. Marlow and S. Peyton Jones. "Making a fast Curry: Push/Enter vs. Eval/Apply for Higher-order Languages". In: *ICFP'04* (2004).

[MS86] M. Mauny and A. Suárez. “Implementing functional languages in the categorical abstract machine”. In: *Proceedings 1986 ACM Conference on Lisp and Functional Programming* (1986).

[MPY01] E. Meijer, N. Perry, and A. van Yzendoorn. “Scripting .NET using Mondrian”. In: *ECOOP* (2001).

[Mon+05] M. Monteiro, M. Araújo, R. Borges, and A. Santos. “Compiling Non-strict Functional Languages for the .NET Platform”. In: *Journal of Universal Computer Science* 11.7 (2005).

[Oka96] C. Okasaki. “The Role of Lazy Evaluation in Amortized Data Structures”. In: *ICFP’96* (1996).

[PM04] N. Perry and E. Meijer. “Implementing functional languages on object-oriented virtual machines”. In: *IEE Proceedings on Software* 151(1):1-9 (2004).

[Pey87] S. Peyton Jones. *The Implementation of Functional Programming Languages*. 1987.

[Pey92] S. Peyton Jones. *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine*. 1992.

[SPV17] I. Sergey, S. Peyton Jones, and D. Vytiniotis. *Theory and Practice of Demand Analysis in Haskell*. 2017.

[SS90] H. Søndergaard and P. Sestoft. “Referential Transparency, Definiteness and Unfoldability”. In: *Acta Informatica* 27 (1990).

[Ste02] D. Stewart. *Multi-paradigm just-in-time compilation*. 2002.

[Tul96] M. Tullsen. “Compiling Haskell to Java”. In: *YALEU/DCS/RR-1204* (1996).

[Tur79] D.A. Turner. “A new implementation technique for applicative languages”. In: *Software – Practice and Experience* 9:31-49 (1979).