Debugging and expanding Meta Casanova

Bachelor Thesis

Written by

LOUIS VAN DER BURG

0806963

Creating 010
Rotterdam

June 16, 2016

Abstract

This document describes the new programming language Meta Casanova and its development.

The author was part of the research group which was developping Meta-Casanova.

This document is the thesis that was issued by the Rotterdam University of Applied Sciences, as a test of skill and knowledge gained after four years of education at their Computer Engineering department.

Contents

Part I

Preface

Chapter 1

Introduction

1.1 Company

The graduation assignment is carried out at Kenniscentrum Creating 010. The company is located in Rotterdam. *Kenniscentrum Creating 010 is a trans-disciplinary design-inclusive Research Center enabling citizens, students and creative industry making the future of Rotterdam* [1].

The assignment is carried out within a research group, who is building a new programming language. The new programming language is called *Casanova*.

1.2 The assignment

We will now describe the assignment. What the motive is and how it will be executed.

1.2.1 Motive

Video game development is difficult [2]. Video games are getting bigger and, with it, more complex. However the time between releases of the video games does not lengthen [2].

One of the major challenges video game developers face is a short time to market. When the developers are able to produce games faster, they have an advantage over their competitors.

This is where *Casanova* comes in. *Casanova* is a programming language to make the development of games easier [3].

Casanova uses higher order types to make the programming of complex constructs easier. These constructs are often used in game development. A few of these constructs will be explained in detail from chapter 5 onward.

Because of the higher order types the compiler for Casanova became complex. The compiler for Casanova worked, but was still in development. So when a bug was found in the compiler, fixing it became more and more time consuming. This frustrated the developers so much a new language was created [4].

Meta Casanova (from now on called *MC*) is this new language. MC serves as the language in which Casanova will be written as a library

At the beginning of the assignment MC was still in development. The syntax was nearly complete, but untested, and the standard library of MC was just beginning to take shape. The assignment was created to debug and expand MC and further refine the standard library of MC.

1.2.2 Research question

As stated above the main goal of the assignment is to debug and expand MC and refine the standard library of MC. This translates to the following research question:

How can the programming language Meta Casanova be improved for the user within the timeframe of the internship?

Here the user is a programmer using the language to create applications.

This research question is quite broad and several sub-questions have been created to define a clearer scope for the assignment.

1. What is a good programming language to the user?
2. What is MC and how does it work?
3. How can the current syntax be improved to serve the user?
4. How can the standard library be improved to serve the user?

After we have a clear image of what a good programming language is, we can look at MC. MC can then be checked and improved where necessary.

To further test MC from a users perspective a few test programs will be written in the language.

The definitions and scope of the improvements are defined by the preliminary research.

1.2.3 Correlation with other projects

During the internship the research group keeps developing Casanova. The research group consists of Francesco di Giacomo[1], Mohamed Abbadi[1], Agostino Cortesi[1], Giuseppe Maggiore[2] and Pieter Spronck[3].

Within the research group the research team developing MC consists of Douwe van Gijn, responsible for developing the back-end of the bootstrap MC compiler, Jarno Holstein, responsible for developing the front-end of the bootstrap MC compiler, and Louis van der Burg, responsible for developing the MC language. The research team are all students and are supervised by Giuseppe Maggiore.

The student will work closely with the research team and cooperate with the research group.

[1]Universita' Ca' Foscari, Venezia
[2]Hogeschool Rotterdam
[3]Tilburg University

Part II

Preliminary research

Chapter 2

A good programming language

For a programming language to be practically usable it needs to be tested on the following criteria [5, 6]:

1. Read- & Writability
2. Simplicity
3. Definiteness
4. Predictability
5. Expressiveness
6. Implementability
7. Efficiency
8. Custom Libraries
9. Time
10. Hackability
11. Succinctness
12. Redesign
13. External Factors

Let us touch on each of these briefly, before we connect them to MC.

2.1 Read- & Writability

The read- and writability of a programming language determines how good the connection between man and machine is. The programmer must be able to write programs he understands easily, even after months or years of not looking at them.

With easy to read and write programs also comes ease of debugging. We can quickly notice any faults in the source code.

Documentation makes up a basic part of read- & writability. To really be able to make a program readable, even after a long delay between writing, the language needs to be its own documentation. This can be done in a manner of ways:

- Keywords
- Abbreviations and concise notation
- Comments
- Layout or format of programs
- No overuse of notation

2.2 Simplicity

The simplicity of a language is decided by its features. These should be easy to learn and remember. To quantify this we will make use of these sub points:

- Structure
- Number of features
- Multiple ways of specification
- Multiple ways of expressing

The main concept through which simplicity is achieved is basing the language around a few simple well-defined concepts [5].

2.3 Definiteness

Definiteness clarifies for which purpose the language is designed and how it is to be used. When a language has a clear view and definition of itself, it will also be clear to the user for what purposes to use the language.

2.4 Predictability

Predictability makes using the language easier for the user. When one user writes the same code as another user, the result should be the same. This implies that if the user understands the basics of the language, he can than expand beyond them by combining these basics.

However, the rules of the language should not hinder the user by making the available combinations too complex.

2.5 Expressiveness

Expressiveness goes hand in hand with abstraction. The user should be able to write code in a way that matches the structure in his thoughts.

Machine language is hard to understand and use for a human, it has nearly no abstraction and the programmer has to keep track of the registers by hand. This can however be remedied by the language the user is using.

It is the link between what the user has in his mind and what the machine will execute. Therefore the language should be similar to the way the user thinks. This makes it easier for the user to express his thoughts in code.

2.6 Efficiency

Efficiency is something most users want out of their computer. Programmers like to write fast programs [6, 2]. The language should facilitate this. Even if the language is very abstract, which usually means it is less efficient [6], it should be able to generate fast code, or making visible to the user where the bottlenecks are.

2.7 Custom libraries

Custom libraries are a necessity for every programming language. Without them the language would be useless and the programmer would have to build every feature from the ground up. This makes programming in the language take much longer then necessary.

2.8 Time

Time is something everything needs to build momentum and a stable user base, even programming languages cannot escape this. For now we will not be able to test this, as MC is still in its development stages. For this reason we will leave this out.

2.9 Hackability

Hackability is the ability to bend the language to the will of the user or to form the language to the needs of the user. In other words how much you can do with the language and how malleable and versatile it is. This also depends on the ability of the programmer, namely to think outside the pre-defined box of the language. We will try to see how well MC supports this type of thinking.

2.10 Succinctness

The succinctness makes a language more abstract. When you have to say less to make the computer do precisely what you want, that is something very powerful. It also makes the programs shorter and clearer to read.

2.11 Redesign

Redesign enables the evolution of a program written in the language. When source code can be reformatted and restructured easily without a lot of refactoring, it is easier to go from a rough prototype to a fully featured program.

2.12 External factors and implementability

The external factors and implementability play a big role in the adaptation of a language. Without some use for the language, it might as well not exist. Having a platform for the language plays a major part in the way it will be adopted as a standard. This does not have to be a physical platform, like UNIX or Windows. It can be a virtual platform, like an already existing major library. The only problem with the latter is, that the new language has to compete with other languages who already implement the library.

Do keep in mind that most of these criteria can be subjective to the user of the language and are not 100% verifiable. We will try to discuss the most objective parts for sake of the verifiability.

Chapter 3

MC in detail

We will now look at MC. After we have a basic understanding of MC, we will look at the improvements made.

Because the latest version of MC is not documented, the following information comes from interviews with the research team [7]. This chapter also serves as the documentation for this version of MC.

3.1 What is MC

MC is a declarative functional language. It tries to be a completely pure language, which means no side-effects are allowed directly.

3.2 Goal

The goal of MC is to use higher order types with type safety, in a natural way of programming. These higher order types give the ability to resolve certain computations at compile time, which would normally be performed at runtime. This can substantially increase performance at runtime. The higher order types also add expressive power by being able to capture abstractions.

MC also supports the .NET library natively. Because .NET has mutables [8], MC can have side-effects through the use of .NET.

MC aims to be as flexible as possible, with the safety of a strong type system.

3.3 Basics

We will now go through the basics of MC. We will look at how the syntax works and use small code samples to explain them.

The improvements made to these basics will be given in part III. This will give us a better understanding of why and how MC evolved.

Inside the code blocks a line wrap is indicated by $\hookrightarrow$.

3.3.1 Terms, types & kinds

With MC you program on three different levels:

1. Terms
2. Types
3. Kinds

Terms are the values of variables, like `5`, `'c'` or `''Hello''`. Types are the types of variables, like `Integer`, `String` or `Boolean`. Kinds are to types, what types are to terms. You could say that kinds are the types of the types.

3.3.2 Func

First we need to declare a function before defining it. For this we use the keyword `Func`.

```
Func "computeNumber" -> Boolean -> Int -> Int -> Int
```

Here we see that the function `computeNumber` is a function which goes from a `Boolean` and two `Integers` to an `Integer`.

The parameters are separated by the ASCII arrow, `->`. The last parameter is the return type of the function and all parameters between the name and the return type are the arguments of the function.

A function is defined by one or multiple *rules*. A *rule* consists of a *conclusion* and can contain multiple *premises*. They are separated by the *bar*. The conclusion is placed underneath the bar and the premise(s) above.

Now we define a rule for `computeNumber`.

```
a == True
add b c -> res
--------------------------
computeNumber a b c -> res
```

The conclusion is on the fourth line, the bar on the third and the premise on the first and second. The conclusion has the name of the function and the input arguments on the left of the arrow and the output argument on the right. This syntax is similar to that of natural deduction [9].

Locally defined identifiers

The identifiers named in the conclusion of a function are locally defined. Within the rule of `computeNumber` the identifiers `a`, `b` and `c` are examples of these local identifiers.

3.3.3 Multiple rules

There can also be multiple rules which form the function definition:

```
a == True
add b c -> res
---------------
computeNumber a b c -> res

a == False
mul b c -> res
---------------
computeNumber a b c -> res
```

When this happens the rules are activated in order of definition. A rule fails when the arguments do not match. If the rule fails, the next rule is activated. If none of the rules match, the program crashes.

Let us go through this process with an example.

First we will try the following function call, from inside a premise:

```
computeNumber False 5 3 -> newValue
-----------------------------------
```

The first rule to be attempted is the one written first in the source code, so:

```
a == True
add b c -> res
--------------------------
computeNumber a b c -> res
```

This rule will fail, because argument `a` has the value `False` and not `True`. So this rule will not be executed.

The next rule to be attempted is:

```
a == False
mul b c -> res
-------------------------
computeNumber a b c -> res
```

Argument a is still False and b and c are both Integers, so the rule matches. The rule will be executed and the result will be put in newValue.

3.3.4 Constants

Constants can be created by using Func without any parameters. The return type still has to be declared.

```
Func "bar" -> Int
bar -> 5
```

Here the constant bar is set to the value 5.

We also see that not every rule has a *bar*. This is possible only when no premise is needed in the rule.

As we have seen Func uses types in its declaration and terms in its definition. The types enable the detection of wrong function calls during compile time.

Important to note is that functions declared by Func are runtime constants. MC also has the ability to declare and define functions that run at compile time.[1]

3.3.5 Generic type identifiers

To indicate generic types the ' notation is used.

This is enhanced with the use of *identifiers* after ', for example: 'a or 'b.

These give the generic type a name and can be used to express if one generic type is the same as another generic type.

3.3.6 Functions as parameters

Aside from variables, functions can also be given as parameters. This needs to be specified in the declaration of the function.

```
Func "isThisEven" -> ('a -> 'b) -> 'a -> 'd
```

[1]As will be explained in section 3.3.8 and shown in section 3.3.9.

Here we see that the first argument (`'a -> 'b`) is a function. This is clear because the entire argument goes from `'a` to `'b`, `'a` is the input and `'b` the output of the function (`'a -> 'b`).

3.3.7 Data

With `Data` we can declare a constructor and deconstructor for a new type. Because `Data` can work two ways, by constructing and deconstructing types, it effectively creates an alias for types.

Say that we want to create a union, a type which can be any of the predefined types within a set [10]. For this we need a way to construct the union type and a way to deconstruct the union back to its original type. The deconstructor is needed to get to know which of the constructors inside the union is actually used. Only then can we know which of the types inside the union is used.

As an example we will create a union of a String and a Float.

```
Data "Left"  -> String -> String | Float
Data "Right" -> Float  -> String | Float
```

Here we create two constructors whom both go to the same type, `String | Float`. `Left` and `Right` can be seen as an alias for the `String | Float` union.

Let us see how this is used. We need a function which takes a union as argument.

```
Func "isFloat" -> ( String | Float ) -> Boolean
```

The parentheses in the declaration can be left out, but are used to clarify that the union is one argument.

In the definition we want to check which type the union is. This can be done directly in the conclusion of a rule.

```
isFloat (Right x) -> True
```

The deconstructor of the argument given automatically checks if the types match. This can be done because they are aliases created by `Data`. Aliases are automatically resolved to their basic types by the compiler.

Conditionals using aliases can be put in the conclusion. Other conditionals have to be put in the premises, because they need to be computed.[2]

To make the definition of `IsFloat` complete we also need a rule that checks for `String`.

[2]As is shown in the example of section 3.3.2

```
isFloat (Left x) -> False
```

Now we will test the function `isFloat` with the help of constant `iAmFloat`.

```
Right 9.28 -> iAmFloat
```

`iAmFloat` has type `String | Float` and can now be passed to `isFloat`.

```
isFloat iAmFloat -> output
```

When `iAmFloat` is deconstructed to its original type it will match with the first rule of `isFloat`. The identifier `output` now contains the value `True`.

3.3.8 Runtime and compile time

So far we have seen the `->` arrow, which indicates a function is executed on runtime.

To indicate that a function is resolved or inlined at compile time, the big ascii arrow, `=>`, is used.

Types are resolved or inlined at compile time, except when they create data structures. These data structures keep existing on runtime. Kinds are also resolved or inlined at compile time.

We will see this starting from the next section onwards.

3.3.9 TypeAlias

Because the | (*pipe*) operator[3] is not built in the language we need to create it. To do this we need to manipulate types.

To make *pipe* work, we both need a constructor and deconstructor that works with kinds.

`TypeAlias` is the same as `Data` only on a higher level of abstraction. It constructs and deconstructs kinds.

With `TypeAlias` we can create a generic *pipe* with which we can create unions.

```
TypeAlias Type => "|" => Type => Type
```

Here we see how the generic *pipe* is declared. It takes two parameters of any type and returns a type.

The `Type` is a kind and indicates an unknown type is used.

Parameters may also be infix, as shown in the example above.

[3]Used in section 3.3.7

Infix parameters

There is a limitation to the use infix arguments, namely there may only be one. This is because of the parser[4] used for the bootstrap compiler of MC. The parser would become overly complex when adding multiple arguments on the left.

Because the benefits of having multiple arguments in front of the function name does not outweigh the time it consumes to expand the parser, the choice was made to only allow one infix argument.

3.3.10 TypeFunc

TypeFunc creates a function on type level. It can perform computations with both types and terms.

We will demonstrate the power of TypeFunc with a function that manipulates a type. The function will take a tuple and return the same tuple, but with switched types.

First we need to declare a tuple with TypeAlias.

```
TypeAlias Type => "*" => Type => Type
```

Now that we have a tuple we can use it in the function declaration. The tuple argument will have generic types so it can work with any tuple.

```
TypeFunc "switch" => Type => Type
```

Next we have the definition. Here we need to deconstruct the tuple to its original arguments. These arguments can then be used to create the return tuple.

```
a => (c * d)
(d * c) => res
---------------
switch a => res
```

Here we can see how the types inside the tuple are switched and returned as a new tuple.

[4]A parser analyses the source code and puts it in a structure easily used by the rest of the compiler.

Parameters

Important to note is that both `TypeAlias` and `TypeFunc` can use kinds, types and terms. Because they work on compile time only, they cannot replace `Data` and `Func`.

`Data` and `Func` exist for explicit runtime functions. The relations between the keywords can be seen in table 3.1.

Runtime	**Compile time**
`->`	`=>`
`Data`	`TypeAlias`
`Func`	`TypeFunc`

Table 3.1: Relations between the keywords and arrows in MC

3.3.11 Modules

`Module` is used to create a container at compile time. It is a collection of function declarations and definitions. Having a collection of functions is useful when they serve a combined or general purpose.

It does not fit within the levels defined in section 3.3.1. Modules can only be declared by `TypeFunc`.

Say we want to create a container for every type which does an addition of two identifiers of these types. The addition could be declared when we need them or we can put them in a `Module`.

When we create an addition container using a module, we do not have to type out the addition functions for every type. We can just call the module to create the functions for us.

A `Module` is declared with a `TypeFunc`.

```
TypeFunc "Add" => Type => Module
```

Here we declare Add to take a type and return a `Module`.

We then want to define the module Add with the + operator and the `identityAdd` constant.

```
Add 'a => Module {
  Func 'a -> "+" -> 'a -> 'a
  Func "identityAdd" -> 'a
}
```

This creates a basic container with generic addition functionality.

As shown above `Module` can contain `Func`, but can also contain `Data`, `TypeFunc`, `TypeAlias` and another `Module`.

The declarations within the Module do not have to be defined. This way we can define different behavior for every instance of the module.

It is possible to have a function declared and defined within a module.

When we want to define the above declared Add with an integer, we instantiate the module over Int. We then define the functions which are declared within Add to finish the module.

```
TypeFunc "IntAdd" => Module
IntAdd => Add Int {
  Int -> "+" -> Int -> Int
  identityAdd -> 0
}
```

Funcs are defined as they normally are, only wrapped inside a Module.

The caret

When we want to call a function within a module from outside the module, the ^ (*caret* operator is used.

```
Func "caretTest" -> Int
caretTest -> identityAdd^IntAdd
```

This creates a constant named caretTest with the same value as identityAdd from the module IntAdd. Important to note is that we cannot call a function outside the module from within the module. The *caret* notation is derived from LaTeX. In LaTeX the *caret* is used to indicate superscript [11]. This is used is MC to show that one element is part of another element like so: $\texttt{identityAdd}^{\texttt{IntAdd}}$.

Inherit

Modules can also inherit from other modules. This is done with inherit.

We will create a module which inherits from the Add module. The operator - will then be added as well.

```
TypeFunc "GroupAdd" => Type => Module
GroupAdd 'a => Module {
  inherit Add 'a
  Func 'a -> "-" -> 'a -> 'a
}
```

The module Add[5] is instantiated with 'a. The created Add module is then inherited into GroupAdd.

The function declarations and definitions of the inherited module are directly usable within the new module.

If a module takes another module as an argument, the module can also directly inherit.

[5]As used in section 3.3.11

```
TypeFunc "GroupAdd" => Add => Module
GroupAdd M => Module {
  inherit M
}
```

Now everything from module M is inherited into GroupAdd.

When inheriting a module which contains an inherit, these are also inherited in to the current module. This works recursively.

3.3.12 Priority

We can also give a priority together with a declaration. The priority indicates the order of function execution.

The priority is indicated by the #> (*hash arrow*) and is placed after the declaration. When we look at Data it will look like this:

```
Data "Left"  -> String -> String | Float  #> 7
Data "Right" -> Float  -> String | Float  #> 5
```

And with Func and TypeFunc it is used like this:

```
Func "bar" -> Int  #> 12
TypeFunc "foo" => Float => 'a => 'b  #> 9
```

The *hash arrow* is also used to indicate associativity. Everything is left associative by default, but if we want to make it right associative we can do that by placing an R after the *hash arrow*.

```
Func Int -> "\" -> Int -> Int  #> R
```

Priority and associativity can also be used in combination with each other.

```
Func Int -> "\" -> Int -> Int  #> 25 R
```

3.3.13 Import

import is used when importing other files in the current file.

If the file imported has any imports, those are ignored. Unlike inherit, import is not recursive.

The programmer can directly use the declarations and definitions used in the imported file. When the programmer wants to explicitly specify the use of a function from the imported file, the *caret* is used.

For example we can import the file *vector* and use the Vector2 type from it, as seen below:

```
import vector

Func "location" -> Vector2
location -> createVector2^vector 8.9 19.0
```

Importing of MC files always works with non capitalized import statements, even when the actual files does contains capital letters. This is done to differentiate between MC imports and .NET imports.

The order of elements is the same as with modules, only when using .NET imports there is a difference in syntax. When calling a .NET function the .NET syntax is used to differentiate between MC and .NET functions.

```
import System

Func "dotNetTest" -> String
dotNetTest -> DateTime.Now.ToString()
```

Here `dotNetTest` becomes a String which contains the current date and time.

.NET functions can be used on both run- and compile time

3.3.14 Builtin

Some things cannot be created with MC and need to be buildin in the compiler. An example of this is the boolean data type.

When using such a builtin literal the keyword `builtin` is used. We can now correctly implement the function `computeNumber`[6] using the `builtin` keyword:

```
a == True^builtin
add b c -> res
---------------
computeNumber a b c -> res

a == False^builtin
mul b c -> res
---------------
computeNumber a b c -> res
```

It can be seen as if `False` and `True` are imported from the language itself.

[6]As used in section 3.3.2

3.3.15 Lambdas

MC also has anonymous functions to implement lambdas [12].

The basic syntax is as one would expect of a lambda, it has arguments and returns what the function body of the lambda returns

```
(\ arguments -> functionBody )
```

In practice this would look like:

```
(\ a -> divide 10 a )
```

Lambdas do not need to be declared because they always are generic functions. The typechecker will check lambdas the same way as any other generic function. From the context of what the function body does together with the types of the arguments, it can deduce what the output type should be.

3.3.16 ArrowFunc

`ArrowFunc` always needs an argument on the left of the name and at least one function as argument on the right. A unique feature of `ArrowFunc` is the two syntax options it has when the function gets called.

The first syntax option is a regular function and the second converts the function to a lambda.

As with `Func` it needs a declaration and a definition.

```
ArrowFunc Int -> ":~>" -> (Int -> Int) -> Int
f a -> res
-------------
a :~> f -> res
```

Here `:~>` takes an integer and a function that takes an integer and returns an integer.

The declaration and definition are the same as the rest of MC.

When we call `:~>`, there is the regular way:

```
a :~> f -> res
```

And there is the second way:

```
{a :~> out
  f out} -> res
```

The brackets are used to indicate that everything inside them belongs together. When an `ArrowFunc` is called like this it gets converted to a lambda. The first argument is then used as the argument for that lambda.

```
(\ out -> f out ) a -> res
```

The result from this is then returned as the output in `res`.

This syntax exists for when the nesting of lambdas occur. Lambdas become unreadable when nested.

```
(\ b -> (\ c -> f c ) b ) a -> res
```

When using the `ArrowFunc` syntax it nests neatly.

```
{a :~> b
  {b :~> c
    f c}} -> res
```

This syntax stems from the use of the *bind* function of monads, which will be explained in further detail in section 5.4.

3.3.17 Partial application

Lastly MC supports partial application. This means that you do not have to give all the arguments to a function. When this is done a closure is created and given as output.

We will create a closure with the function `add`. `add` takes two `Integers` and returns an `Integer`.

In the definition we will say the two arguments will be added together.

```
Func "add" -> Int -> Int -> Int

a + b -> res
--------------
add a b -> res
```

Now we will give only one argument to `add` and create a new function `addThree`.

```
add 3 -> addThree
```

With this construct we have created the partially applied function `addThree`. When we look at what it contains we see that it has a closure with a partially implemented `add`.

```
3 + b -> res
--------------
add 3 b -> res
```

We can now give addThree its second argument to complete the closure. This can be done multiple times.

```
addThree 6 -> foo

addThree 7 -> bar
```

The identifier foo has a value of 9 and bar has a value of 10.

3.3.18 The main function

While MC has no specific main function it does specify which function acts the main function. The final function in the file compiled acts as the main function and is executed when the program is run.

Now that we know the basics of MC lets take a look at how the syntax has evolved and improved.

Part III

MC expanded

Chapter 4

Syntax evolution

Here we will look into the overall syntax changes of the language.

4.1 Generic type identifiers

Generic type identifiers are indicated by `'` together with a name, `'a`.[1] The `'` notation is derived from *ML language*.

In *ML language* the `'` notation is used to indicate a letter from the Greek alphabet [13]. `'a` stands for *alpha*, `'b` for *beta* and so on.

By using this notation the readability of the language is improved. It is now clear to the user when generic types are the same or different from each other.

4.2 Generic kind identifiers

Generic kind identifiers are currently unused, but it was thought to be necessary after the implementation of `TypeAlias`.

When working with kinds they are always generic, as they indicate an unknown type. That was why they were indicated by an *asteriks*, `*`.

When `TypeAlias` was created it became apparent that the type would be the same in some cases. This resulted in generic kind identifiers.

The notation devised was similar to the generic type identifiers. Instead of the *prime* the *hash* was used.

[1]As described in section 3.3.5

```
TypeFunc "unknownKinds" -> #a -> #b -> #a -> #c
```

Here we can see that the first and third arguments are of the same kind and the second argument and the return value are different kinds.

However this was an error in reasoning.

The fact that they are kinds means that they can be any type. The official notation used for kinds is either `Type` or `*`.

The syntax of kinds was changed to `Type`. This way the `*` was free to be used for other things, like the tuple in section 3.3.10.

This also improved the readability of the language. It is now clear when a kind is used.

Because of the use of `Type` it is also instantly clear what a kind is. This reduces the complexity of the language.

4.3 Modules

Modules have had two major changes, both will be described here.

4.3.1 Signature

Earlier in development `Module` was called `Signature`. It performed the same functionality as `Module`.

From the language creators perspective `Module` creates a specific signature at compile time. For the user it looks more like a container or class, when coming from object-oriented programming.

Because the user will be the one actually using the language, the name was changed to what it resembles to the user. It clarifies the syntax and makes it more predictable for the user.

4.3.2 Expanding

During development there were two ways of inheriting a module, via `inherit` and via expanding an existing module. Expanding a module is removed from MC, because of inconsistency issues.

A `Module` could be expanded via a function. The function takes a module as argument and expands this module by adding declarations and, optionally, definitions.

```
TypeFunc "expandModule" => Module => Module
expandModule M => M{
  Func "modulo" -> Float -> Float

  a % 10 -> res
  ---------------
  modulo a -> res
}
```

The module `M` is opened up again and `modulo` is added. Module `M` is then returned and contains the new function `modulo`.

This would have been syntactic sugar for creating a new module which inherits from `M`. In this new module `modulo` is then added.

```
TypeFunc "expandModule" => Module => Module
expandModule M => Module{
  inherit M

  Func "modulo" -> Float -> Float

  a % 10 -> res
  ---------------
  modulo a -> res
}
```

The expand-syntax makes it look like modules are mutable, which they are not. This makes it inconsistent with the rest of MC and the expand-syntax was removed.

By not using the expanding functionality the language becomes more coherent and in line with itself. This makes the language friendlier to the user, because there is no two ways of doing things. It is clear how inherit works and there is no confusion possible.

4.4 .NET libraries

When importing and calling .NET libraries the current syntax is using the original .NET syntax:

```
Func "dotNetTest" -> String
dotNetTest -> System.DateTime.Now.ToString()
```

This was not always used. The syntax for .NET imports went through several stages.

The first implementation was the MC inherit order with .NET elements, while using the `^`:

```
dotNetTest -> ToString()^Now^DateTime^System
```

This seemed strange considering the method calls .NET has. The order of the elements was then switched.

```
dotNetTest -> System^DateTime^Now^ToString()
```

This syntax seemed out of sync with the rest of the language. That is why the `^` was changed back to the `.`.

That is how we arrived at the current syntax used. The current syntax shows the user explicitly when a .NET call is used.

Using these different syntaxes for .NET and MC creates a clear distinction between the two.

4.5 Builtin

The keyword `builtin` was first called `primitives`.

For developers this was a good choice, as they indicate the primitives that needed to be built in the compiler.

For the user however this seemed confusing. The user sees the `primitives` as types that are built into the language.

The name was changed to `builtin`, to better reflect the way how the user sees this functionality.

4.6 Conditionals

Two sorts of conditionals exist in MC and here we will describe why they are this way.

4.6.1 Inside the conclusion

During development we tried implementing conditionals inside the conclusion of a rule. This could make the code more compact.

We will take the example of `Func` from section 3.3.2, and implement this. The code is then compacted to this:

```
add b c -> res
------------------------------------
computeNumber True^buitin b c -> res
```

This is just as easy to read as the original.

It does make the conclusion less conclusive as there is now something happening inside the conclusion. The conclusion is meant to show the input and output of the rule, it is not meant to tell what the function does.

That is where the premises are for.

The compiler also needed to be modified heavily. It now needed to do computations inside the conclusion.

To keep the language and the compiler modular, a comprise was done. Types created with `Data` and `TypeAlias` can be used as conditionals inside the conclusion. This is possible because they act as aliases and need no computation to be checked.

Conditionals requiring computation, a comparison of values, will be done in the premises.

This is also syntactically predictable. Especially because the `Data` and `TypeAlias` used in conditionals, are aliases. They could be replaced with that from which they are constructed.

4.6.2 Equals

When comparing values the == operator is used. This is done to keep the single equals sign free to be used by the programmer.

The single = could be mistaken for an alias creation or value assignment, when first learning MC. With the double == there is no such confusion.

Chapter 5

Standard library

We will now look at the standard library and explain the evolution it went through during the project.

When we have an idea of how it should work, the choices made during the development will be clearer.

5.1 Prelude

Prelude is the first item in the standard library and contains a few definitions making basic programming easier. *Prelude* contains the definitions which are not large enough to grant there own item within the standard library.

The .NET *System* namespace is imported and the type `Unit` is created.

```
import System

Data "unit" -> Unit
```

The `Unit` is used as an empty or null value.

Next we have the declaration and definition of a generic tuple and union. Because they are generic `TypeAlias` is needed for the type manipulation.

```
TypeAlias Type => "*" => Type => Type
Data 'a -> "," -> 'b -> 'a * 'b    #> 5

TypeAlias Type => "|" => Type => Type
Data "Left" -> 'a -> 'a | 'b       #> 5
Data "Right" -> 'b -> 'a | 'b      #> 5
```

A standard if-else construct is then implemented as follows:

```
TypeAlias "Then" => Type
Data "then" -> Then
TypeAlias "Else" => Type
Data "else" -> Else

Func "if" -> Boolean^System -> Then -> 'a -> Else -> 'a -> 'a
if True^builtin then f else g -> f
if False^builtin then f else g -> g
```

The `then` and `else` `Data`s are syntactic sugar and could be left out. We have left them in to enhance the clarity of the if-else construct.

Now that we have the if-else construct we can do basic boolean math. It also makes the use of multiple rules an option and not a necessity, which in turn creates more freedom for the programmer.

Here we see how *System* of .NET is used. The boolean of .NET can be imported as shown, however the values `True` and `False` cannot be imported. They are built in .NET itself. This is why the boolean literals are built into MC as well and need to be called using `builtin`.

5.1.1 Match

Next we have the `match` function. `match` is used to check the constructor of the *pipe* operator and makes computations with the *pipe* easier.

It takes a variable, matches it on either `Left` or `Right` and executes the function specified.

```
TypeAlias "With" => Type
Data "with" -> With

Func "match" -> ('a | 'b) -> With -> ('a -> 'c) -> ('b -> 'c) -> 'c
```

There is some syntactic sugar created to make it clearer how `match` works. The first argument is the union which needs to get matched. The arguments `('a -> 'c)` and `('b -> 'c)` are the functions to be executed on each branch of the match.

In this case the two functions both take an argument from the union, the `'a` and the `'b`, and both return `'c`.

Next we have the definition of `match`. `match` needs a definition for both cases of the match, namely the `Right` and the `Left`.

```
match (Left x) with f g -> f x
match (Right y) with f g -> g y
```

When it matches the `Left` it executes function `f` with `x` as its argument. And when it matches the `Right` it executes function g with `y` as its argument.

5.1.2 List

We have a generic list implementation, which very useful considering MC has no built in arrays. For this we need to create a list that can work with types.

The list is declared with a union. The union is necessary because we need an end to the list.

```
TypeAlias "List" => Unit | (Type * (List Type))
```

The list is created with a tuple. An end to the list is created with a union of the list tuple and `Unit`.

We also need the basic operators to create lists. This is done with `::`, which takes a type and a list of that same type. And with `empty` an empty list is created.

```
Data 'a -> "::" -> 'b -> List (Right ('a * 'b))
Data "empty" -> List (Left unit)
```

With `::` we can specify the head and tail of a list.

But we would also like to concat lists together. Concatenation is done with the `@` operator.

```
Func List 'a -> "@" -> List 'a -> List 'a  #> 200
empty @ l -> l
(x :: xs) @ l -> x :: (xs @ l)
```

And when we want to apply a function to the entire list we call `map`. `map` executes a function and creates a new list, which it then returns.

```
Func "map" -> List 'a -> ('a -> 'b) -> List 'b
map empty f -> empty
map (x :: xs) f -> (f x) :: (map xs)
```

When we want to find one or more elements in the list we can call `filter`. It checks the entire list using a predicate function and returns the matching elements.

```
Func "filter" -> List 'a -> ('a -> Boolean^System) -> List 'a
filter empty p -> empty

(if p x then
  (x :: (filter xs p))
  else
  (filter xs p)) -> res
-------------------------
filter (x :: xs) p -> res
```

The programmer can specify the predicate function `p`.

5.1.3 Evolution of prelude

We will now go over the evolution of the separate parts of *prelude*.

Boolean

Boolean currently does not exist anymore as a separate part of the standard library. Here we will see why this is the case.

Boolean was created to implement boolean logic. After many iterations it was discarded due to the evolution of the language.

When first implementing Boolean it was part of *Prelude*.

```
Data "True" -> Boolean^System
Data "False" -> Boolean^System
```

This might seem correct at first glance, but `True` and `False` have no meaning here. There is no way to check whether a boolean value is true or false. That is why a new notation had to be created for the boolean literals, which ended up into a module with the literals as `Funcs`.

```
import System

TypeFunc "Boolean" => Module
Boolean => Module {
  Func "True" -> Boolean^System
  Func "False" -> Boolean^System
}
```

The `Boolean` module could then be implemented in *prelude*.

```
TypeFunc "boolean" => Boolean
boolean => Boolean {
   True -> True^builtin
   False -> False^builtin
}
```

The boolean literals could now be called using `True^boolean`. This way was noticeably the same as what happened inside the implementation of `boolean`. The choice was then made to call the boolean literals directly with `True^builtin`, which made the boolean module obsolete.

This choice brings us to the current state of the boolean literals.

Match

Match started as a separate part of the standard library containing the `match` module. We will now look at how it was first created and how it has evolved into *prelude*.

The first iteration of the `match` module started with the import of *prelude* and the declaration of the `match` module.

```
import prelude

TypeFunc "match" => Type => Module
```

So far there are no problems yet with us a module for `match`.

Next we see the first part of implementation of the `match` module: Some syntactic sugar is created and the actual function is defined which will do the matching.

```
match ('a | 'b) => Module {
  Data "with" -> With

  TypeFunc "Left" => Type
  Left => 'a

  TypeFunc "Right" => Type
  Right => 'b

  Func "doMatch" -> 'c -> With -> (Left -> 'd) -> (Right -> 'd) -> 'd
```

Now we see why the module is needed. It was thought that the `match` module would get a separate instance for every match that was executed.

When separate functions, like these, are needed to check the validity of another function it is better to group them together inside a module.

The definitions of `doMatch` have not changed compared to what they are currently.

```
  doMatch (Left x) with f g -> f x
  doMatch (Right y) with f g -> g y
}
```

It became clear that `Left` and `Right` were not necessary, because they were simply passing type of the union.

With this removal the use of a module became obsolete, because we do not need to group a single function. The `doMatch` function was then renamed `match` and placed inside *prelude* directly.

List

The first list was put together with the list monad. This seemed the logical choice at the time, because the list monad needed a list implementation to use inside the monad.

Monads will be explained in section 5.4 and the list monad in section 5.6.2.

When coming back to the list monad it became apparent that `List` was not specific to the list monad. It could be used without the monad as a regular list implementation.

The choice was then made to move it to *prelude*.

5.2 Number

Number is created to give the user a generic interface to create numbers.

Because of the import system of MC, MC can directly import the integer and float types with all their functions from .NET. *Number* is therefor used for self defined number types.

Number is build up from different modules to give the programmer the freedom to choose what their custom numbers can do. The modules are arranged according to the mathematical groupings to built up numerical operators [14].

It starts with the `MonoidAdd` module. It declares the `+` operator for the custom number.

```
TypeFunc "MonoidAdd" => Type => Module
MonoidAdd 'a => Module {
  Func 'a -> "+" -> 'a -> 'a #> 60
  Func "identityAdd" -> 'a
}
```

`MonoidAdd` needs the identity as a base number from which the operations will work.

Next we have the `GroupAdd` module. `GroupAdd` inherits everything from the module `MonoidAdd` and adds the `-` operator.

```
TypeFunc "GroupAdd" => Type => Module
GroupAdd 'a => Module {
  inherit MonoidAdd 'a
  Func 'a -> "-" -> 'a -> 'a #> 60
}
```

We do the same for multiplication and dividing.

```
TypeFunc "MonoidMul" => Type => Module
MonoidMul 'a => Module {
  Func 'a -> "*" -> 'a -> 'a #> 70
  Func "identityMul" -> 'a
}

TypeFunc "GroupMul" => Type => Module
GroupMul 'a => Module {
  inherit MonoidMul 'a
  Func 'a -> "/" -> 'a -> 'a #> 70
}
```

Now we have declared the basic number operations of addition, subtraction, multiplication and dividing.

We can combine them into a basic number like so:

```
TypeFunc "Number" => Type => Module
Number 'a => Module {
  inherit GroupAdd 'a
  inherit GroupMul 'a
}
```

Or create a Vector without the dividing operator, because vectors cannot be divided [15].

```
TypeFunc "Vector" => Type => Module
Vector 'a => Module {
  inherit GroupAdd 'a
  inherit MonoidMul 'a
}
```

Of course all the operators still have to be defined, but that is up to the programmer. This gives the programmer the freedom to choose what the operators do.

5.2.1 Evolution of number

During development *number* was used to create integers and floats. This was before the .NET import system was in place. Currently it is only needed for user defined number types.

The first iteration of *number* was just one module with all the operators declared. The first iteration works for the standard number types like `Integer` and `Float`, but not `Vectors`, because they cannot be divided.

For this reason the current *number* design is similar to the mathematical way. When implementing *number* we only define what we need, which reduces the overhead generated.

5.3 Record

With *Record* we can create a list of key/value pairs which can be searched and manipulated. The special thing about this record is that it does all the searching on compile time. This makes the runtime code much faster.

Record is declared as a module and contains `TypeFuncs` to make it work at compile time.

```
TypeFunc "Record" => Module
Record => Module{
  TypeFunc "Label" => String
  TypeFunc "Field" => Type
  TypeFunc "Rest" => Record
```

Here we see the declarations which are the basis of a list of key/value pairs. `Label` acts as the key, `Field` as the value paired with the key and `Rest` contains the rest of the record.

When there is just one pair in the record we can get that record out easily, as we already have it. But when there are multiple record entries in a record we need a function to search through the record to find the sub record we want to have. For this the `get` function was created.

```
  TypeFunc "get" => String => Record => Record

  (if (l == Label^rs) then
    rs
  else
    get l Rest^rs) => res
  ----------------------
  get l rs => res
```

The `get` function takes a labels and a record. It checks the label of the record and if it is the same it returns the current record and if not the it recursively calls itself with the rest of the record.

To manipulate these records there the `set` function was created. The `set` function takes a label, a field and a record. The field is the new value that needs to be paired with the label.

```
  TypeFunc "set" => String => Type => Record => Record
  (if (l == Label^rs) then
    RecordEntry l f rs
  else
    (set l f Rest^rs -> rs'
     RecordEntry Label^rs Field^rs rs')) => res
  -------------------------
  set l f rs => res
```

```
}
```

Because MC does not allow direct manipulation of values, the set function returns a new record instead of changing the value of the specified record. When the record does not match the current record `set` is called recursively, and the result is used as the new `Rest` for the current record.

With this the declaration of the record is complete. Now we can look at how the definition of a record works.

First we need to create a record entry. This is done with the `TypeFunc` `RecordEntry`.

```
TypeFunc "RecordEntry" => String => * => Record => Record
RecordEntry label field rest => Record{
  Field => field
  Label => label
  Rest => rest
}
```

We also need to create an empty record entry so we can end the record. This is done by the `TypeFunc` `EmptyRecord`.

```
TypeFunc "EmptyRecord" => RecordEntry
EmptyRecord => RecordEntry {
  Field => Unit
  Label => Unit
  Rest => Unit
}
```

The programmer has to use `EmptyRecord` as the `rest` argument of the first record entry. This creates an end to the record.

5.3.1 Updatable record

The current record can only be instantiated and not updated. To expand the record to be updatable we need to add the update function.

First we declare and define a `TypeFunc` which creates a new record from the original with inherit.

```
TypeFunc "updatableRecord" => Record => Record
updatableRecord r => Module {
   inherit r
```

The new module now contains everything of record `r`.

Then we declare and define the update function.

```
  TypeFunc "update" => Record => Type => Record
  r == Empty
  --------------------
  update r dt => Empty
}
```

Only the definition of `update` which checks for an empty record can be defined. It is the only situation in which we can be certain what the return value will be, namely `Empty`.

If the record is not `Empty` we cannot know what the programmer wants to do with `update`. That is why we leave that definition of `update` up to the programmer.

5.3.2 Evolution of record

Several parts of `Record` have changed. We will go over them here.

Set

The original version of `set` checked the field-argument to be the same type of `Field` inside the record. For this `Fields` was needed, which contained the type of `Field`.

However the checking of `Fields` needed to happen in the declaration of the `set`. This required the declaration to have premises and this is not possible, because the declaration cannot be variable.

The field-argument also did not need to be checked, because `Field` of a record can be anything.[1]

Set return

The return value of `Set` was first only the record that was changed and not all the previous records in the record structure. This was fixed by returning the recursive call to `set` as the new value of the current record.

Now the entire record structure is returned and not just the record that is changed.

[1]As described in section 5.3.

Cons

During the development the idea occurred that every `Record` had to have its own type signature. Because the field can be anything, it was thought to impact the type of the `Record`.

This would be a problem when trying to declare the use of a specific `Record`.

This resulted in the creation of `Cons`. `Cons` would be the type signature of that particular `Record`.

It would consist of the types of the `Label`, `Field` and `Rest`.

```
TypeFunc "Cons" => Type
Cons => (Label,Field,Rest)
```

It became apparent that every record is of type `Record` and that the type of `Field` had no impact on the type of the record. `Cons` was then removed.

This also reduced the complexity of the records for the user. Now the user can use the records without having to keep in mind which type belongs to which record.

5.4 Monad

Now we come to the module `Monad`. This module takes a mathematical concept and makes it into a practical programming construct.

Monads are the main reason MC has type manipulation.

Monads enable purity of code with the power of mutables [16]. The structure of monads also enables a customizable flow of control [17].

First we are going to look at what monads exactly are within programming. Then we will see how they are implemented within MC.

5.4.1 Monads

Monads are a container-like generic interface [18]. They contain at least two functions, *return* and *bind*.

These two functions are the basic way we can use monads [16].

The return

The return function takes a value and wraps it inside a monad [16].

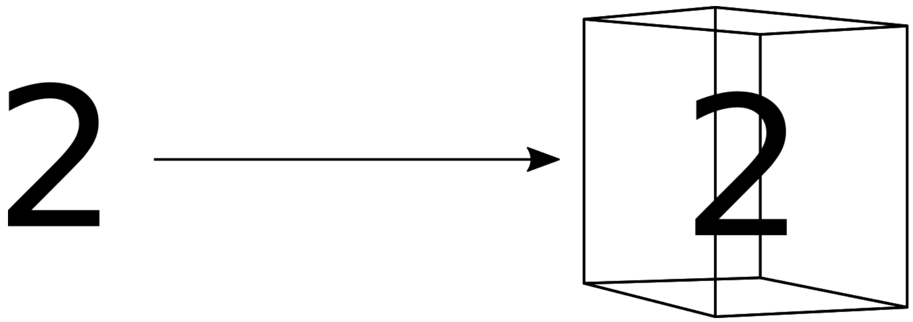

Figure 5.1: The return

The box in figure 5.1 visualizes the monad and 2 is the value that is being put inside the monad. With the return function any value can be put inside a monad. When it is inside the monad it cannot be seen by the outside world.

This is where the bind comes in.

The bind

Having a monad is good, but what if you want to manipulate the value of the monad? The `>>=` (pronounced *bind*) function supplies this functionality [16].

When we want to apply the function `(+3)` to the monad containing 2, we call bind. Bind unpacks the monad, as visualised in figure 5.2.

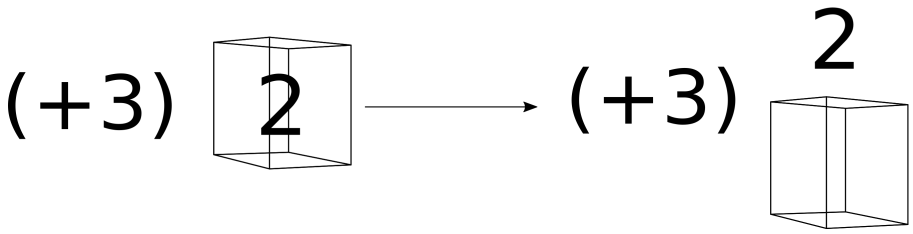

Figure 5.2: A function and a monad

It then applies the function to the value. Finally the new value gets packed inside the monad by the function, as visualised in figure 5.3.

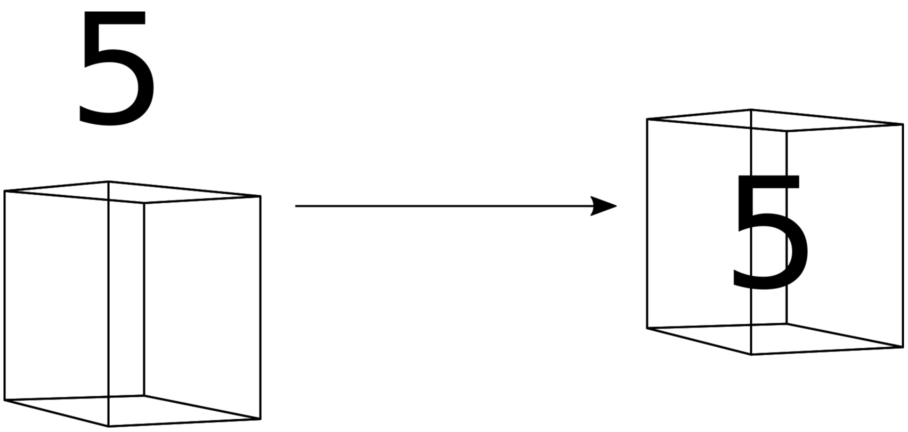

Figure 5.3: The return

Using these two basic functions, we can always work with monads and the values they contain.

To use the monads further we will need a few more functions. These functions are generic for all the monads and will be given an implementation.

More than a wrapper

But just a monad offers very little besides being a wrapper for values. That is why there have been created different sorts of monads. This means a lot of different functionalities can be mapped to monads [19].

We will discuss two of these for now. More will be explained when looking at the implemented monads in MC in section 5.6.

The maybe monad offers the ability of the value to be either a value or nothing [20]. For example we could utilize the pipe operator for this:

```
Data "Maybe" -> 'a -> 'a | Unit

Func "test" -> Maybe -> Boolean^System
(match a with
  (\ x -> True^builtin)
  (\ unit -> False^builtin)) -> res
-----------------------------------
test a -> res
```

Here we use `Unit` as the nothing and 'a as the value. In the example the term a is checked for a being a value, the `Left`, or being nothing, the `Right`.

This is a very basic example of how the maybe works. It acts like a pipe and can contain either a value, `Left`, or nothing, `Right`.

The state monad gives monads the ability to behave like mutables. It takes a *state* and returns a new state and a return value [20].

Say we want a random number generator. The state monad will simulate the *state* of the number being generated. The state monad is given a *state* in the form of a number and from this it will compute the new state of the number and the random number generated.

In MC the functionality might look like this:

```
Func "state" -> 's -> ('a,'s)
state number -> (randomGeneratedNumber,newNumber)
```

The implementations of these monads are far from complete, these examples simply show the functionality of these monads.

We will now look at how the monad is really implemented within MC.

5.4.2 Implementation

Because monads need to be able to manipulate type information, it is a good example of what MC is useful for.

With this monad we will be able to create actual monads, this can be seen as the base module for monads. First we take a look at the declaration:

```
TypeFunc "Monad" => (Type => Type) => Module
```

Monads are implemented as modules. This way we can put the return and bind functions inside a monad.

The first argument is a function on type level. It represents the monad it uses to wrap the result in.

When looking at the implementation we see that a module is created with the return and the bind:

```
Monad 'M => Module {
  ArrowFunc 'M 'a -> ">>=" -> ('a -> 'M 'b) -> 'M 'b   #> 10
  Func "return" -> 'a -> 'M 'a
```

The bind is declared with `ArrowFunc` and is called `>>=`. It takes a monad as its first argument and a function as its second argument.

The function creates the new monad which is then returned.

The return takes a value, `'a`, and wraps it inside the monad, `'M`.

Both bind and return are only declared and not defined, because they are different for every monad created. When creating a new monad bind and return will need to be defined.

Now we will add functionality to make the monads more practical for the user when programming.

Returnfrom

Sometimes we want to directly pass the value inside the monad instead of wrapping it. This is done with `returnFrom` [20].

```
Func "returnFrom" -> 'a -> 'a
returnFrom a -> a
```

As we can see the identifier `a` is directly passed on.

Lift

Lift is used when a regular function, it takes a value and returns a value, needs to be executed over a monad [21]. Lift unwraps the monad, executes the function with the value of the monad and then wraps the new value inside the monad.

An important difference with the bind, is that the function given to the lift does not pack the value in a monad. The function passed to the bind does pack the value inside the monad.

```
  Func "lift" -> ('a -> 'b ) -> 'M 'a -> 'M 'b
  {a >>= a'
     return f a'} -> res
  ----------------------
  lift f a -> res
```

Here we can see clearly that the lift repacks the value with `return`.

The `Lift` can also be declared with functions which take two arguments. It is then called `Lift2`.

`Lift2` has to bind twice, once for each monad.

```
  Func "lift2" -> ('a -> 'b -> 'c) -> 'M 'a -> 'M 'b -> 'M 'c
  {a >>= a'
     b >>= b'
     return f a' b'} -> res
  --------------------------
  lift2 f a b -> res
```

This can be expanded to functions which take any number of arguments.

With these functions we have the complete interface of a monad. Now we can use the module Monad to implement them.

Combining monads

Monads can also be combined [22]. Since they have the same generic interface they can be combined into another generic interface. These new monads can utilise the functionality of the all combined monads.

In this way the functionality of a monad can be extended [22]. Combining monads can be seen as putting one monad inside another monad.

We will take the state and maybe monad and combine these into a parser monad. A parser monad can parse through a piece of text and scan for certain elements [17].

The state monad will be used for the scanning functionality and the maybe monad will be used to determine of the element has been found. The state monad returns the maybe monad as its result. When the maybe monad has a value as result the parser monad stops.

Like normal monads, combined monads have to be built manually. This is a very error prone process when combining more than two monads or when using more complex monads.

What we actually want is to write the monads once and combine them automatically. This is where monad transformers come in.

5.4.3 Monad transformers

Monad transformers are a way to automatically combine monads [17]. Instead of simply having a monad which takes the arguments it needs, it also takes another monad as one of its arguments. The monad given as argument is then used to process the other arguments given to the monad transformer.

MC has an implementation of the basic monad transformer with which all monad transformers and monads can be created.

For monad transformers to work a generic function is added which contains the signature of the monad wrapped inside the transformer.

```
TypeFunc "MCons" -> Type
MCons -> 'M
```

The identifier `'M` is the monad given to the monad transformer. `MCons` can be called when this monad is needed directly.

LiftM

There are of course functions that work specifically with the wrapped monad in a monad transformer. For this `liftM` is created.

For this we have to use the bind and return of the wrapped monad.

```
  Func "liftM" -> (MCons^'M 'a -> MCons^M' 'b) -> 'M (MCons^'M 'a) ->
    ↪ 'M (Mcons^'M 'b)
  {M >>=^'M a
    f a -> b
    return^'M b} -> res
  ---------------------
  liftM f M -> res
}
```

It works the same as the other lift functions, with the specification of using the bind and return of the wrapped monad, `'M`, instead of the bind and return of the current monad.

5.4.4 Evolution of monad

Monad was first implemented with just `MCons`, `>>=`, `return` and `returnFrom` functions. This enabled basic use of the monad.

`lift`, `lift2` and `liftM` were then added to extend the usecases of monads.

5.5 Tryable monad

Apart from the regular monadic module there is also the tryable monadic module. The tryable monad has as extra functionality that a function can be attempted.

This means that we can have functions that fail, without breaking the program.

It uses the regular monadic module to create a new interface. It also takes a monad as an argument which it then inherits.

```
TypeFunc "TryableMonad" => (Type => Type) => Module
TryableMonad 'M => Monad(MCons^'M) {
  inherit 'M
```

It uses the type signature of 'M to create a new monad.
Here we see how MCons is used to get the signature of the original monad.
Then we declare the try function. It takes a monad and two functions.

```
  Func "try" => MCons^'M 'a => ('a -> MCons^'M 'b) => ('e -> MCons^'M
    ↪ 'b) => MCons^'M 'b
```

The value inside the monad is checked and then one of the functions is executed. This will be done in the definition.
First the value inside the monad is put into a match.

```
  {pm >>=^'M x
    (match x with
```

If x contains an error the error function processes it. And if it contains a value the value is passed to the function.

```
      (\e -> err e)
      (\y -> k y)) -> z
```

The result of match is then wrapped in monad 'M and returned.

```
    return^'M z} -> res
  ---------------------
  try pm k err -> res
}
```

With try we can now check if the monad contains a value or an error and process them.
The try function was put to the test by manually typechecking it. The resulting proof can be found in appendix **??**.
We also want to be able to get the original monad out of the tryable monad. This is done with getMonad.

```
  TypeFunc "getMonad" => Type
  getMonad => MCons^'M
}
```

Now we can create a Monad which has two options when calling bind. This can be used for error handling or for creating complex dataflows within monads.

5.5.1 Evolution of tryable monad

The TryableMonad started out as the either monad. either used the available options to create error handling.

It occured then that we could also create more complex dataflows, when creating a generic tryable monad. This was then implemented as the `TryableMonad`.

5.6 Implemented monads

We will now look at the monads that are implemented in MC.

5.6.1 Id

If monad transformers always take another monad, there has to be at least one monad to end the chain. That is where the *id* monad comes in [17].

It does not take a monad as part of its arguments and simply returns all values as they are.

First we need a type signature to tell the monad interface which monad we are creating.

```
TypeAlias "Id" => Type => Type
Id 'a => 'a
```

Here we can see what the id monad should do, simply pass the type on as it was given.

Now we will declare and define the id monad. The type signature is created with `Id` and bind and return are defined.

```
TypeFunc "id" => Monad
id => Monad(Id) {
  bind x k -> k x
  return x -> x
}
```

`Id` also takes an `'a`, but this is never used as the id monad does not need an extra type to say which type it really uses. While the `'a` is not used, it necessary to have them in the type signature. Otherwise `Id` cannot be used to instantiate `Monad`, as evident from section 5.4.2.

Other monad transformers which do actually use the `'a` also do not instantiate them immediately. The programmer can decide for which type the monads will be created, therefor it is left open. This effectively creates an interface for the defined monad to be instantiated with a type later specified.

`return` simply returns the exact same value and `bind` executes the function `k` with `x` as its argument. It is the monad which *passes through* the values.

When using the id monad transformer with another transformer, it becomes that monad.

As an example we pass the id monad to the maybe monad transformer. When the return of the created maybe monad is called, it first calls the return of the inner monad. In this case is the id monad.

The return of the id monad returns the value directly as a result. This result is then passed back to the maybe monad, where it gets wrapped inside the maybe monad and returned.

In this way the id monad will always end the cycle.

5.6.2 List

The list monad uses a list to store the values [22, 20].

The list monad transformer uses List from *prelude* to create the transformer.

First we have to declare and define the signature.

```
TypeAlias "ListT" => (Type => Type) => Type => Type
ListT 'M 'a => 'M(List 'a)
```

Using ListT together with the signature of 'M we create the list monad.

```
TypeFunc "list" => Monad => Monad
list 'M => Monad(ListT MCons^'M) {
```

ListT also takes a 'a, which is the type for the list created. As stated in section 5.6.1 the programmer has to decide which type the list monad uses.

The return simply puts the value in a one-element list wrapped inside a monad.

```
  return x -> return^'M(x :: empty)
```

bind looks a bit more complicated. It uses the ArrowFunc syntax together with a match.

```
  {lm >>= l
    (match l with
```

The match checks whether the list is empty.

```
      (\empty -> return^'M empty)
```

When a list is present the first element of the list gets unpacked by the bind of 'M, which gets named y. The function k gets executed with y as its argument and the result gets repacked by the return.

```
      (\(x :: xs) ->
        {x >>=^'M y
          return^'M k y} -> z
```

Now the rest of the list, `xs`, gets passed to the bind of `list` so the entire list gets passed to `k`. The results of processing `x` and `xs` are then concatenated into a single list.

```
        xs >>= k -> zs
        (z @ zs)))} -> res
  -------------------------------
  lm >>= k -> res
}
```

All the nested functions are closed with the proper brackets and the resulting list is returned as the result of the bind.

If the use of the different syntaxes of the `ArrowFunc` seems confusing, take an extra look at section 3.3.16.

Evolution of list

The list that is used within the list monad, was first implemented within the list monad. But the list could be used outside of the monad as well.

It was then moved to *prelude*.

5.6.3 Either

The `either` monad is an implementation of the `TryableMonad` module.

It creates a monad which can be either a value or a list of error messages [21]. The type signature is as follows:

```
TypeAlias "EitherT" => (Type => Type) => Type => Type => Type
EitherT 'M 'e 'a => 'M('a | 'e)
```

It uses the pipe to be able to be either failed or a value. In the definition of `either`, `'e` will be specified as a list.

The `TryableMonad` is then called with the signature of the monadic argument.

```
TypeFunc "either" => Monad => Type => TryableMonad
either 'M 'e => TryableMonad(EitherT MCons^'M (List 'e)) {
```

The `'a` is again not used, like the id monad in section 5.6.1.

Next we have is the `fail` function. It takes an error message and concatenates it with the existing error messages.

```
Func "fail" -> 'e -> MCons^'M
fail e -> return^'M(Right (Right^'M :: e))
```

The `fail` is needed when we call the `try`. It will provide the error function to the `try`.

Because `either` is a `TryableMonad` it can contain a value or an error message. When calling the bind of `either` this needs to be checked.

That is why the bind calls `try`.

```
pm >>= k -> try pm k fail
```

The monad and the function is passed on to `try` together with the function `fail`.

The `return` passes the value to `'M`.

```
return x -> return^'M(Left x)
```

`Left` is used because we know `x` is a value.

Evolution of either

`either` was first implemented without the use of `TryableMonad`. It had the same functionality. All the functions of the `TryableMonad` were implemented directly within `either`.

When the idea occurred of using a generic tryable monad, these were ported to the `TryableMonad`. `either` was then implemented by using the `TryableMonad`.

5.6.4 Result

The result monad is a complete implementation of the either monad transformer from section 5.6.3.

The `either` monad still needs an error type and a type for the values it takes. The result monad specifies the error type, leaving the type of the value for the programmer to specify.

Because all the functions in `either` are defined, result can directly call `either` with an error type. No further function declarations or definitions are needed.

In the declaration we specify it returns a `TryableMonad`, which `either` actually is. In the definition we specify the error type to be `String`.

```
TypeFunc "result" => Monad => TryableMonad
result 'M => either MCons^'M (List String)
```

This creates an `either` with `String` as error type.

Evolution of result

The original idea of the `result` monad was a `maybe` monad with error handling in the form of a `string`.

The `result` monad was first implemented in much the same way as the `either` monad, but with explicit use of `String` as the error handler. When this was noticed, the `result` monad was implemented by using `either`.

5.6.5 State

We have had a basic explanation of the state monad in section 5.4 where we have seen that it is different from most monads. The state monad does not acts like a lambda in monadic form.

Now we will see how the state monad transformer is implemented. The type signature of the state monad transformer shows that it is a function.

```
TypeAlias "StateT" => (Type => Type) => Type => Type => Type
StateT 'M 's 'a => ('s -> 'M('a * 's))
```

It takes a state and returns a monad containing a tuple of the resulting value and the new state.

We use `StateT` in the call to `Monad` so it knows what the type signature will be.

```
TypeFunc "state" => Monad => Type => Monad
state 'M 's => Monad(StateT MCons^'M 's) {
```

When looking at return we see something new. A lambda is created to match the type signature of the state monad.

```
  return x -> (\ s -> return^'M(x,s))
```

`x` is set as the result of the state.

We see this happening also in the bind. A lambda is created and in the function body of the lambda the computation happens.

```
  (\ s ->
    {sm s >>=^'M x
      x -> (x',s')
      k x' s'}) -> res
  ---------------------
  sm >>= k -> res
```

Bind of `'M` is called and the result is then deconstructed to get to the tuple which it contains. We know `x` is a tuple because it is the result from state monad `sm`. That `x` is a tuple, is evident by the type signature of the state monad.

The state monad also contains two extra functions. They make it possible to get and set the state.

To get the state we call `getState`.

```
Func "getState" -> MCons 's
getState -> (\ s -> return^'M(s,s))
```

It sets the state as the result in the return lambda. By defining return this way, you will get the state back when it is entered into the lambda.

To set the state we call `setState`.

```
  Func "setState" -> 's -> MCons Unit
  setState s -> (\ unit -> return^'M(unit,s))
}
```

It sets the result to unit and the state to the input state.

5.7 Other monad implementations

When comparing the monad transformers of MC with monad implementations of other languages, we quickly come to the conclusion that MC is very compact.

When looking at a monad implementation in Python[2], we see that the code for most monads is much longer. And these are the direct implementations, not using monad transformers.

There are only few attempts made when it comes to monad transformers. The main reason for this is the need for higher order types.

Haskell also has an implementation of monad transformers [23], but these are far longer than the MC implementation.[3]

This shows what MC is capable of compared to most other languages. With the use of higher order types the high abstractions used for monad transformers can be implemented with very little code.

[2]Appendix B

[3]See appendix C for the source code of the state monad transformer.

5.8 Advantages

Now that we know what monads are and that MC has a clear implementation of them, we can look at the practical advantages.

We will focus on the video game industry, but the concepts are useful in other industries as well.

When programming with monads and monad transformers we can create a monad which does one function and combine it with other monads. When combining monads more complex functionalities are created, but the way they are used does not change [17, 20].

When the way of using monads does not change there is no need to comprehend how a specific monad does things. We only need to know how to use it.

Monads then become very powerful black boxes [17, 16, 20].

For example we can create a networking monad which takes care of all the network communication. We also create a graphical monad which takes care of filling the screen. And as a final touch we create an input monad which takes care of all the user input.

When we combine these monads for video game development the only thing left to create is the game itself.

The video game developer can now focus on the video game itself, saving time on networking, user input and graphics. When combining monads with the typesafety of MC, the video game developer has a significant advantage over his competitors.

Not only is the video game developer able to develop faster, the video game is also less prone to bugs.

Chapter 6

Test programs

There have been two test programs written to test the language. The first is a physics simulation to test the updatable records.

Because the final goal of the language is to make creating games easier, the second test program is a game.

Neither of the programs are implemented with the monads, because of time constraints.

Both of the programs do not have their graphical parts implemented, because it would remove the focus from MC The graphical parts of the programs would utilise the XNA framework and the focus would then be on how XNA is implemented within MC. We want to test how MC works and not how it handles a specific library. For this reason the graphical parts of the programs are left out.

6.1 Bouncing ball

Bouncing ball is the physics simulation used as a preliminary test of updatable records. It will show how the updatable `RecordEntry` are used in practice.

This program bounces a ball up and down.

The ball will have two properties, velocity and position, which will implemented as records. Both of these will then be put inside the updatable `RecordEntry` of the ball.

The `update` function will then add implement how the ball bounces.

First we import prelude, record and the XNA framework. The Microsoft XNA library will be used for the `Vector2` it has and the Thread library will be used to call the `Sleep` function.

```
import Microsoft^Xna^Framework
import System^Threading^Thread
import record
import prelude
```

The recordentry of the ball will be build bottom-up. This is done because every recordentry needs to have a `rest` as an argument.

`Velocity` is created first, because it needs to be updated less. It keeps deep searches of the recordentry to a minimum.

```
Record "Velocity" Vector2(0.0f, 98.1f) Empty
```

Then `Position` will be created with `Velocity` as its `rest` argument.

```
Record "Position" Vector2(100.0f, 0.0f) Velocity
```

Lastly we create the updatable recordentry `Ball`. This is done in one line by creating a recordentry entry and passing it directly as argument to `updatableRecord`. `Position` is used as `rest` argument to complete the record.

```
UpdatableRecord (Record "Ball" unit Position) => Ball {
```

We then need to define the `update` function.

The `Position` and `Velocity` RecordEntry are put in identifiers, so we can use them directly.

```
  get^e "Position" Rest^e => position
  get^e "Velocity" Rest^e => velocity
```

Then the height of the ball is checked and the according calculations executed.

```
  (if (Y^Field^position <= 500.0f) then
    ((set^e "Position"
           (Field^position Vector2.+ (Field^velocity Vector2.* dt))
           Rest^position)
     (set^e "Velocity"
            (Field^velocity Vector2.+ (Vector2(0.0f, 98.1f) Vector2.*
    ↪ dt))
            Rest^velocity))
  else
    (set^e "Position"
           Vector2(X^Field^position, 500.0f) Vector2.- Field^velocity)
    ↪ ) -> res
  --------------------------------------------
  update e dt -> res
}
```

The final step is to create the `run` function to start the program. It will call the update function of `Ball`, sleep for a set amount of time and then call itself.

```
Func "run" -> RecordEntry -> String

update^ball ball time -> ball'
Sleep(1000)
run ball' (time +^Int 1000) -> res
---------------
run ball time -> res
```

This creates an infinite loop, so the ball will keep on bouncing. A return value will never be set as it will keep calling itself until the program stops.

6.2 Tron

Tron is the game and utilises user input.

Tron [24] is played with two or more players. Each controlling a bike which leaves a trail. The bikes can move freely within the playing field, but cannot stop, slow down or speed up. The trails the bikes leave behind become walls.

A bike may not hit a wall or trail, when it does it is destroyed. The player who remains driving is the winner.

The game tron has been chosen because of the minimalistic graphics.

Tron has been split up in two three parts, bike, playfield and main. The main file calls all the needed parts and starts the game. The playfield contains all the bikes.

Splitting the code will keep all the parts clean and specific to their function.

The bikes and playfield will be records. Some of the records will be made updatable records when necessary.

We start by looking at the bike.

6.2.1 Bike

First we import a few things. The XNA framework is needed for the `vector2`. The `System.Windows.Input` will be needed for the controls of the bike.

Lastly we will import recordentry and prelude.

```
import Framework^Xna^Microsoft
import System^Windows^Input
import record
import prelude
```

The bike needs several things to work in the game.

A bike needs a record which keeps track of its current position. The position needs to be updated so we use an updatable record. The position record is created seperately and included in the bike later.

This is done to make the definition of the bike smaller.

The `RecordEntry` is created in the premise and then passed to `UpdatableRecord`.

```
TypeFunc "PositionRecord" => String => Vector2 => Record => Record
RecordEntry label pos rest => r
--------------------------
PositionRecord label pos rest => UpdatableRecord r => Module{
```

The created record `r` will not be explicitly inherited as that happens with the `UpdatableRecord` function. We only need to define the update function.

The update function takes a record and a tuple. The tuple consists of delta time and speed. This is then used to calculate the new position.

A new record containing the new position is then created and returned as the result.

```
  dts -> (dt,speed)
  Field^p + (dt * speed) -> newPos
  set^p label^p newPos Rest^p -> res
  -----------------------------------
  update p dts -> res
}
```

Note that the `+` and `*` operators are inferenced to be the `Vector2` operators.

Now that we have an updatable position we also need an updatable trail. This trail changes when the bike moves.

As with `PositionRecord` an we first create a record which is then passed to `UpdatableRecord`.

```
TypeFunc "TrailRecord" => String => Vector2 => Record => Record
RecordEntry label field rest => r
----------------------------
TrailRecord label field rest => UpdatableRecord r => Module{
```

The update function only needs to create a new trail record with the original record as its `Rest`. This way the trail is a record of points.

```
  RecordEntry Label^t pos t => res
  ----------------------------------
  update t pos => res
}
```

Next we need the controls for each bike. As the bike is always moving forward we only need to steer left or right.

The two buttons used for this are placed in a record.

```
TypeFunc "Keys" => Key => Key => Record
RecordEntry "Left" left Empty
RecordEntry "Right" right Left
------------------------------
Keys left right => res
```

Then we come to the bike itself. It will use the `Keys`, `TrailRecord` and `PositionRecord`. And with those it also uses records to store the colour, speed and a boolean to check if it has not crashed.

The bike will also be given a unique identifier, its name.

In the definition we will see how the bike is built with multiple records.

```
TypeFunc "Bike" => String => Int => Int => Keys => Vector2 =>
    ↪ TrailRecordEntry => Record => Record
RecordEntry "Colour" color Empty
RecordEntry "IsAlive" True^builtin Colour
RecordEntry "Controls" keys IsAlive
RecordEntry "Speed" speed Controls
PositionRecord "Position" position Speed
RecordEntry "Trail" trail Position => field
RecordEntry label field rest => e
---------------------------------------------------------------------
Bike label color keys speed position trail rest => UpdatableRecord e{
```

The record entries are ordered according to how often they are needed, this speeds up compile time.

`color` is an *ARGB* value and `speed` is the amount of pixels per second.

The update function takes a bike and the delta time as argument. It first checks if the bike is still active.

```
  (if Field^IsAlive then
```

Next the trail, position are put in variables.

```
    get^b "Trail" Field^b => trail
    get^trail "Position" Rest^trail => position
    get^position "Speed" Rest^position => speed
```

Then the trail and position get updated.

```
    update^TrailRecord Field^trail Field^position => newTrailRecord
    update^PositionRecord position (dt,Field^speed) -> newPos
```

A new trail record is created and lastly a new bike record is created and returned.

```
    set^b label^trail newTrailRecord newPos => newTrail
    set^b Label^b newTrail Rest^b
```

The bike is returned if it is dead.

```
  else
    b) -> res
  -------------------------------------
  update b dt -> res
}
```

We now have a complete bike that can be updated.

6.2.2 Playfield

The playfield contains a record with the bikes, the filed size and a winner. When the winner is set it will be returned to the main function and the game will end.

First we import the XNA framework, System, record, prelude and bike.

```
import Framework^Xna^Microsoft
import record
import prelude
import bike
```

For the bikes a separate record will be created. This enables a dynamic number of bikes without interfering with the playfield.

```
TypeFunc "BikeRecord" => String => Record => Record => Record
BikeRecordEntry label bikes rest => RecordEntry label bikes rest{
```

When defining the update function we need a few extra functions to make it all work. These functions are necessary because they do recursive operations. This would be impossible without a separate function.

The first function is `checkTrails`. It goes through the trail of a bike and check for any collisions. It takes a position and trail and returns a boolean.

If any collisions are detected the boolean returns false, otherwise it returns true.

First we check for any collisions with the current point in the trail.

```
Func "checkTrails" -> Vector2 -> Record -> Boolean
(if ((X^Field^pos == X^Field^trail) ||
     (Y^Field^position == Y^fieldsize)) then
  False^builtin
else
  True^builtin) -> newIsAlive
```

If there are no collisions and there are more points in the trail, `checkTrails` is recursively called.

```
(if (newIsALive && (Rest^trail != Unit)) then
  checkTrails pos Rest^trail
else
  False^builtin) -> res
-----------------------------
checkTrails pos trail -> res
```

Next we have `checkCollisions`, which calls `checkTrials` for the trail of every bike. It takes a bike and a record of bikes. The position of the current bike and the trail of the first bike in the record, will be passed to `checkTrails`.

```
Func "checkCollisions" -> Record -> Record -> Record
get^bs "Position" Field^bs -> pos
get^Rest "Trail" Field^rst -> trail
checktTrails pos trail -> newIsAlive
(if newIsAlive && (Rest^trail != Unit))hen
  checkCollisions bs Rest^rst
else
  set^bs "IsAlive" False^builtin) -> newBs
------------------------------------------
checkCollisions bs rst -> newBs
```

Then we have the function which will update the `IsAlive` status of the current bike. First it checks if the position of the bike is within the playfield.

```
Func "updateIsAlive" -> Record -> Vector2 -> Record
get^bs "Position" Field^bs -> position
(if ((X^Field^position <= 0.0) ||
     (Y^Field^position <= 0.0) ||
     (X^Field^position >= X^fieldsize) ||
     (Y^Field^position >= Y^fieldsize))
        (set^bs "IsAlive" False^builtin Rest^bs)
      else
```

If the bike is within the playing field, any collisions with its own trail and the other bikes is tested.

```
          ((checkTrails bs (get^bs "Trail" Field^bs)
           (checkCollisions bs Rest^bs))) -> newBs
  updateIsAlive Rest^newBs fieldsize -> newBikeRecords
  ----------------------------------------------------
  updateIsAlive bs fieldsize -> updatedBikes
```

We then come to `updateBikes`. This goes through all the bikes and calls their update function. It then returns the updated bikes.

First we have to check if the bike record is `Empty`.

```
  Func "updateBikes" -> Record -> Float -> Record
  (if (b == Empty) then
    Empty
  else
    (update^b b dt -> newBike
     updateBikes Rest^b dt -> newRest
     RecordEntry Label^newBike Field^newBike newRest)) -> res
  ------------------------------------------------------
  updateBikes b dt -> res
```

In the update function we want to get the actual bikes.

```
  Field^bs -> bikeRecords
```

We also need `fieldsize` and `dt`. These are given to the function as a tuple, because `update` can only take two parameters and we need three.

The `update` function first calls `updateBikes`. And when the bikes are updated they are passed to `updateIsAlive`. The new bike record is then returned.

```
  dtfs -> (dt,fs)
  updateBikes bikeRecords dt -> newBikeRecords
  updateIsAlive newBikeRecords fs -> aliveBikeRecords
  set^bs label^bs aliveBikeRecords bs -> res
  ----------------------------------------
  update bs dtfs -> res
}
```

When defining the actual playfield we create a `BikeRecordEntry` and a record containing the winner. Both these records are passed to `Rest` of the playfield. The size of the playfield is put in the `Field` of the playfield record. The playfield record is also made updatable.

```
TypeFunc "Playfield" => String => Record => Vector2 => Record
RecordEntry "Winner" unit Empty
```

```
BikeRecordEntry "Bikes" bikes Winner -> rest
--------------------------------------------
Playfield label bikes size => updatableRecord (RecordEntry label size
    ↪ rest) {
```

Winner is emty until there is only one player left in the game.

Next we create the checkDeaths function. This function is needed as an intermediate function to update. checkDeaths calls itself recursively to check the state of all the bikes in the game.

It then keeps a counter of the amount of bikes that are still alive. When the counter is higher than one unit is returned.

```
  Func "checkDeaths" -> bike -> amountAlive -> 'a
  (if ((amountAlive > 1) || (amountAlive == 1)) then
    unit
  else
    ((if ((get^bike "IsAlive" Field^bike) == True^builtin) then
      amountAlive + 1
    else
      amountAlive) -> newAmountAlive
      (if (newAmountAlive == 1) then
        bike
      else
        checkDeaths Rest^bike newAmountAlive))) -> res
  ------------------------------------------------
  checkDeaths bike amountAlive -> res
```

Now that we can check that there is a winner we can define update of Playfield. First update updates Bikes.

```
  get^p "Bikes" Rest^p -> bikes
  update^bikes bikes (dt,Field^p) -> newBikes
```

Then checkDeaths is called and the winner is set. The new playfield is then updated with the updated bikes and this playfield is returned as result.

```
  checkDeaths Field^newBikes 0 -> winner
  set^p "Winner" winner Rest^p -> updatedPlayfield
  set^updatedPlayfield "Bikes" newBikes Rest^updatedPlayfield ->
    ↪ newPlayfield
  --------------------------------------------
  update p dt -> newPlayField
}
```

This makes the playfield complete. All that is left to do is create the main function which starts the game.

6.2.3 Main

The main combines the bikes and the playfield to start the actual game.

First `bike`, `playfield` and the XNA framework is imported.

```
import playfield
import bike
import xna stuff
```

Then we create a `play` function which runs the actual game until there is a winner. When there is no winner it increments the delta time and updates the playfield.

```
Func "play" -> Playfield -> Record
get^p "Winner" Rest^p -> winner
(if (winner == unit)
  (update^p dt -> p'
   play p' (dt+1))
else
  Winner) -> realWinner
-------------------------------
play p dt -> realWinner
```

When a winner has emerged, this winning bike is returned.

Then we declare and define `start` which creates all the bikes and the playfield and calls `play` to start the game.

```
Func "start" -> 'a
Keys^bike System.Windows.Forms.Keys.P System.Windows.Forms.Keys.L ->
    ↪ controlRecordOne
TrailRecord "trailBikeOne" Vector2(500.0,250.0) Empty -> trailBikeOne
Bike "playerOne" -16776961 5 controlRecordOne Vector2(500.0,250.0)
    ↪ trailBikeOne Empty-> bikeOne
Keys^bike System.Windows.Forms.Keys.Q System.Windows.Forms.Keys.A ->
    ↪ controlRecordTwo
TrailRecord "trailBikeTwo" Vector2(0.0,250.0) Empty -> trailBikeTwo
Bike "playerTwo" -16711681 5 controlRecordTwo Vector2(0.0,250.0)
    ↪ trailBikeTwo Empty -> bikeTwo
RecordEntry "gamebikes" BikeOne BikeTwo -> bikeRecord
Vector2(500.0,500.0) -> fieldsize
playfield "gamefield" bikeRecord fieldsize  -> gamefield
play gamefield 0 -> res
---------------------
start -> res
```

When a winner is present the game stops.

6.2.4 Discoveries

During the programming of the game several things became clear about MC.

Record

The `set` function of `Record` was not returning the original record, just the updatable record.[1]

Good language

The points described in chapter 2 also had there conclusions found during the creation of the test programs.

The read- and writability of the MC syntax takes some getting used to, as with most languages, but overall it is very compact and readable.

The simplicity of MC is a mixed bag. While the syntax is simple, the concepts used by the language can become anything but simple. These concepts will have the user do some studying to understand these concepts.

The goal and idea behind of MC is quite clear.[2] And because MC has a clear idea of what it wants to do, the user can easily use the language as it was intended.

By being a declarative language MC has de advantage of making itself predictable. The declarations always state what a function does and the definition then tells how this is achieved. Even when using monads MC keeps its predictability, because of how the monads work.[3]

The higher order types handle abstractions very well, as is evident by the short implementation of monad transformers.[4] The high abstractions MC is capable of increase the expressiveness of MC.

Because all these high abstractions are computed at compile time the efficiency does not suffer. The efficiency can even be increased when moving computations normally done at runtime to compile time.

The custom libraries have made quite a leap during its development and the standard library has a stable base. This stable base increases easy expansion in the future, which will help MC to thrive.

The hackability of MC has two sides. On the one hand it forces the user to be very secure by using declarations and definitions. On the other hand the

[1]See section 5.3.2
[2]As described in sections 3.1 and 3.2.
[3]As is shown in section 5.4.1.
[4]As is shown in section 5.6.

user can implement the definitions in numerous ways. Especially the latter enables the user to bend MC to be the language they need.

The higher order types and the high abstractions they enable allow MC to be very succinct. The user does need to have both declarations and definitions which makes MC more work, but they do not outweigh the power given to the user with the higher order types.

The part MC might be troublesome for user is redesigning existing code. Especially when editing what functions do, the user needs to change both the declaration and definition of a function. This double editing is not something programmers are fond of and might ruin the flow of programming is an evolutionary way. However when the user thinks about the code before it is written, MC works like a charm.

By implementing a direct link to the .NET libraries MC has an enormous amount of functionality added at very little cost. This does mean MC has to compete with C# and F#, both of which are strong languages using the .NET library. One advantage MC has over both C# and F# is the higher order types and the high abstractions it can implement. Adding to that the ability to compute anything you want at compile time instead of runtime, brings in major advantages over C# and F#.

Part IV

Results

Chapter 7

Conclusion

To summarize the main research question was: *How can the programming language Meta Casanova be improved for the user within the timeframe of the internship?*

To answer the main question we first had to answer the subquestions.

What is a good programming language to the user? We have seen that a good programming language to the user needs to have the qualities described in chapter 2.

What is MC and how does it work? MC is a declarative functional language and works as is described in chapter 3.

How can the current syntax be improved to serve the user? The syntax has been improved in a manner of ways as is described in chapters 4 through 6.

How can the standard library be improved to serve the user? The standard library has been improved by implementing basic programming constructs, a generic number interface, updatable records and monads, as is described in chapter 5.

From the subquestions we have seen that MC can be improved for the user by focussing the syntax on the user and by expanding the standard library with programming constructs that are useful to the user.

The improvements made to MC during this internship will add to the development of the MC and Casanova programming languages. With Casanova video games become easier to develop. The video game developers will be able to develop their video games faster with Casanova. And by being able to develop faster they will gain an advantage over their competitors in the video game market.

7.1 Other conclusions

Complex datastructures can have increased performance at runtime when doing most of the computation at compile time.

Higher order types enable clear and compact writing of complex programming constructs.

Monads enable programmers to have a safe and clear way of building complex programs with more ease.

Chapter 8

Recommendations

These recommendations are for the people continuing with the project.

8.1 Further development

When developing MC further I would recommend expanding the standard library even more. Especially when the focus of the language is put more on monads.

8.1.1 Monads

Because Monads are a very strong example of where MC thrives, it is a logical next step to move most functionality to monads.

By doing this MC can be used better for quick and safe development of applications, for example video games.

The first step in moving to monads would be the IO monad [25]. The IO monad would be the only interface with the outside world and unpure functions, making MC a completely pure language. Being pure MC becomes more strict, but also less prone to bugs and errors [16].

The implementation of the coroutine monad would come next. With coroutines comes the possibility of multithreading [26]. Multithreading gives an even bigger performance boost to MC.

With the IO- and coroutine monads MC has an effective base for complex programming and the other monads can be implemented as needed by the users.

8.1.2 The compiler

I would also recommend to keep syntax changes to a minimum. This will give the compiler developers a chance to finish the compiler.

An other option would be to define the language before hand and build the compiler after the language has a finished specification.

Chapter 9

Evaluation

Here I will show that I have the competences which are associated with computer science according to the Rotterdam University of applied science.

9.1 Administering

I have learned to use type theory and applied it to MC in accordance to the specifications given to me by the research group. I have learned to use MC and program the standard library of MC with it.

I have made clear goals before starting the actual assignment with minimal requirements.

9.2 Analysing

I have researched what a good programming language is to the user and what MC is and does. I have combined these researches to improve the syntax of MC. I have also combined the research about good programming languages with the standard library of MC to refine and expand the standard library of MC.

9.3 Advising

During the assignment I have put forth improvements for MC to the research group. I had to explain and justify these improvements to the entire research group.

During the second half of the internship members of the research group also came to me with questions concerning MC, because I have become the central resource for MC. This thesis also serves as the documentation of MC and contains a *recommendations* chapter[1] with recommendations for future improvements.

9.4 Designing

I have designed several syntax changes to improve MC while keeping the core of MC the same. I have also refined and expanded the standard library to show what MC is capable of.

[1]See chapter 8.

Chapter 10

Bibliography

[1] C. 010, "Creating 010 - kenniscentrum creating 010." `http://creating010.com/en/`. [Online; Accessed: 18-03-2016].

[2] J. Blow, "Game development: Harder than you think," *Queue*, vol. 1, no. 10, p. 28, 2004.

[3] G. Maggiore, A. Spanò, R. Orsini, G. Costantini, M. Bugliesi, and M. Abbadi, "Designing casanova: a language for games," in *Advances in Computer Games*, pp. 320–332, Springer, 2011.

[4] F. D. Giacomo, P. Spronck, and G. Maggiore, "Building game scripting dsl's with the metacasanova metacompiler," 2015.

[5] U. P. Khedker, "What makes a good programming language," tech. rep., Technical Report TR-97-upk-1, Department of Computer Science University of Pune, 1997.

[6] P. Graham, *Hackers & painters: big ideas from the computer age*. " O'Reilly Media, Inc.", 2004.

[7] F. D. Giacomo and G. Maggiore. Personal communication, 2016.

[8] Microsoft, "Overview of the .net framework." `https://msdn.microsoft.com/en-us/library/a4t23ktk(v=vs.100).aspx`, 2016. [Online; Accessed: 18-02-2016].

[9] D. Prawitz, "Natural deduction: A proof-theoretical study," 1965.

[10] A. Igarashi and H. Nagira, "Union types for object-oriented programming," in *Proceedings of the 2006 ACM symposium on Applied computing*, pp. 1435–1441, ACM, 2006.

[11] N. H. Beebe, "Latex editing support," *Te verschijnen in TUGboat en MAPS*, vol. 92, 1992.

[12] H. P. Barendregt, *The lambda calculus*, vol. 3. North-Holland Amsterdam, 1984.

[13] R. Harper, D. MacQueen, and R. Milner, *Standard ml.* University of Edinburgh. Department of Computer Science. Laboratory for Foundations of Computer Science, 1986.

[14] N. Bourbaki, "Commutative algebra. chapters 1–7, elements of mathematics," 1998.

[15] M. Hazewinkel, *Encyclopaedia of Mathematics: Volume 6: Subject Index—Author Index.* Springer Science & Business Media, 2013.

[16] P. Wadler, "Monads for functional programming," in *Advanced Functional Programming*, pp. 24–52, Springer, 1995.

[17] S. Liang, P. Hudak, and M. Jones, "Monad transformers and modular interpreters," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 333–343, ACM, 1995.

[18] P. Wadler, "Comprehending monads," *Mathematical structures in computer science*, vol. 2, no. 04, pp. 461–493, 1992.

[19] E. Moggi, "Notions of computation and monads," *Information and computation*, vol. 93, no. 1, pp. 55–92, 1991.

[20] W. Swierstra, "Data types à la carte," *Journal of functional programming*, vol. 18, no. 04, pp. 423–436, 2008.

[21] M. Jaskelioff, "Monatron: An extensible monad transformer library," in *Implementation and Application of Functional Languages*, pp. 233–248, Springer, 2008.

[22] D. J. King and P. Wadler, "Combining monads," in *Functional Programming, Glasgow 1992*, pp. 134–143, Springer, 1993.

[23] A. Gill and R. Paterson, “Transformers: Concrete functor and monad transformers.” `http://hackage.haskell.org/package/transformers`, 2016. [Online; Accessed: 02-06-2016].

[24] “Tron,” 1982.

[25] P. Hancock and A. Setzer, “The io monad in dependent type theory,” in *Electronic proceedings of the workshop on dependent types in programming, Göteborg*, pp. 27–28, 1999.

[26] C. T. Haynes, D. P. Friedman, and M. Wand, “Continuations and coroutines,” in *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pp. 293–298, ACM, 1984.

[27] J. DeLaat, “Pymonad source.” `https://bitbucket.org/jason_delaat/pymonad/src/cbecd6796cd1488237d2a0f057cefd2a50df753a/pymonad/State.py?at=master&fileviewer=file-view-default`, 2016. [Online; Accessed: 01-06-2016].

Part V

Appendices

Appendix A

Type proof of try

See the next two page.

$$
\begin{aligned}
&Mcons_{\alpha} && : M\alpha \\
&Mcons^{st}_{\alpha} && : (\sigma \to \alpha\sigma) \\
&Mcons^{res}_{\alpha} && : (\alpha|\epsilon) \\
&Mcons^{prs}_{\alpha} && : (\sigma \to (\sigma \to (\alpha|\epsilon) \times \sigma) \times \sigma)
\end{aligned}
$$

$$try^{prs}\ k\ err\ pm = lift^{st}(lift^{st}(try^{res}k(\lambda e_1.try^{res}(>>=)^{id}(\lambda e_2.fail^{prs}(e_1+e_2)))err))pm : (\alpha \to M\beta) \to (\epsilon \to M\beta) \to Mcons^{prs}_{\alpha} \to Mcons^{prs}_{\beta}$$

$$\dfrac{\dfrac{\dfrac{try^{res} : (\alpha \to M\beta) \to (\epsilon \to M\beta) \to Mcons^{res}_{\alpha} \to Mcons^{res}_{\beta} \qquad (>>=)^{id} : (\alpha \to M\beta)}{try^{res}(>>=)^{id} : (\epsilon \to M\beta) \to Mcons^{res}_{\alpha} \to Mcons^{res}_{\beta}} \qquad \dfrac{\dfrac{fail^{prs} : \epsilon \to Mcons^{prs}_{\alpha} \qquad \dfrac{e_1 : \epsilon \qquad e_2 : \epsilon}{e_1 + e_2 : \epsilon}}{fail^{prs}(e_1+e_2) : Mcons^{prs}_{\alpha}}}{\lambda e_2.fail^{prs}(e_1+e_2) : \epsilon \to Mcons^{prs}_{\alpha}}}{try^{res}(>>=)^{id}\ (\lambda e_2.fail^{prs}(e_1+e_2)) : Mcons^{res}_{\alpha} \to Mcons^{res}_{\beta}}}{\lambda e_1.try^{res}(>>=)^{id}\ (\lambda e_2.fail^{prs}(e_1+e_2)) : \epsilon \to Mcons^{res}_{\alpha} \to Mcons^{res}_{\beta}}$$

$$\dfrac{\dfrac{try^{res} : (\alpha \to M\beta) \to (\epsilon \to M\beta) \to Mcons^{res}_{\alpha} \to Mcons^{res}_{\beta} \qquad k : (\alpha \to M\beta)}{try^{res}k : {\color{red}(\epsilon \to M\beta)} \to Mcons^{res}_{\alpha} \to Mcons^{res}_{\beta}} \qquad \lambda e_1.try^{res}(>>=)^{id}\ (\lambda e_2.fail^{prs}(e_1+e_2)) : {\color{red}\epsilon \to Mcons^{res}_{\alpha} \to Mcons^{res}_{\beta}}}{\textbf{Contradiction:}\ \text{application of } try^{res}\ k \text{ expects } (\epsilon \to M\beta) \text{ but got } (\epsilon \to Mcons^{res}_{\alpha} \to Mcons^{res}_{\beta})}$$

$$try^{prs}\ k\ err\ pm = lift^{st}(lift^{st\_ctxt}(try^{res}k(\lambda e_1.try^{res}return^{id}(\lambda e_2.fail^{prs}(e_1+e_2))err)))pm : (\alpha \to Mcons^{prs}_{\beta}) \to (\epsilon \to Mcons^{prs}_{\beta}) \to Mcons^{prs}_{\alpha} \to Mcons^{prs}_{\beta}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{try^{res} : (\alpha \to Mcons^{res}_{\beta}) \to (\epsilon \to Mcons^{res}_{\beta}) \to Mcons^{res}_{\alpha} \to Mcons^{res}_{\beta} \qquad return^{id} : (\alpha \to Mcons^{res}_{\beta})}{try^{res}return^{id} : (\epsilon \to Mcons^{res}_{\beta}) \to Mcons^{res}_{\alpha} \to Mcons^{res}_{\beta}} \qquad \dfrac{\dfrac{fail^{prs} : \epsilon \to Mcons^{prs}_{\alpha} \qquad \dfrac{e_1 : \epsilon \qquad e_2 : \epsilon}{e_1 + e_2 : \epsilon}}{fail^{prs}(e_1+e_2) : Mcons^{prs}_{\alpha}}}{\lambda e_2.fail^{prs}(e_1+e_2) : \epsilon \to Mcons^{prs}_{\alpha}}}{try^{res}return^{id}\ (\lambda e_2.fail^{prs}(e_1+e_2)) : Mcons^{res}_{\alpha} \to Mcons^{res}_{\beta}} \qquad err : Mcons^{res}_{\alpha}}{try^{res}return^{id}\ (\lambda e_2.fail^{prs}(e_1+e_2))err : Mcons^{res}_{\beta}}}{\lambda e_1.try^{res}return^{id}\ (\lambda e_2.fail^{prs}(e_1+e_2))err : \epsilon \to Mcons^{res}_{\beta}}$$

$$\dfrac{\dfrac{try^{res} : (\alpha \to Mcons^{res}_{\beta}) \to (\epsilon \to Mcons^{res}_{\beta}) \to Mcons^{res}_{\alpha} \to Mcons^{res}_{\beta} \qquad k : (\alpha \to Mcons^{prs}_{\beta})}{try^{res}k : (\epsilon \to Mcons^{res}_{\beta}) \to Mcons^{res}_{\alpha} \to Mcons^{res}_{\beta}} \qquad \lambda e_1.try^{res}return^{id}\ (\lambda e_2.fail^{prs}(e_1+e_2))err : \epsilon \to Mcons^{res}_{\beta}}{try^{res}k\ (\lambda e_1.try^{res}return^{id}\ (\lambda e_2.fail^{prs}(e_1+e_2))err) : Mcons^{res}_{\alpha} \to Mcons^{res}_{\beta}}$$

$$\dfrac{lift^{st} : (\alpha \to \beta) \to Mcons^{st}_{\alpha} \to Mcons^{st\_ctxt}_{\beta} \qquad \dfrac{lift^{st\_ctxt} : (\alpha \to \beta) \to Mcons^{st\_ctxt}_{\alpha} \to Mcons^{res}_{\beta} \qquad try^{res}k\ (\lambda e_1.try^{res}return^{id}\ (\lambda e_2.fail^{prs}(e_1+e_2))\ err) : Mcons^{res}_{\beta}}{lift^{st\_ctxt}(try^{res}k\ (\lambda e_1.try^{res}return^{id}\ (\lambda e_2.fail^{prs}(e_1+e_2))\ err)\ pm) : Mcons^{res}_{\beta}}}{lift^{st}(lift^{st\_ctxt}(try^{res}k\ (\lambda e_1.try^{res}return^{id}\ (\lambda e_2.fail^{prs}(e_1+e_2))\ err)))\ pm : Mcons^{res}_{\beta}}$$

Appendix B

Monads in Python

More can be found in the *PyMonad* library [27].

B.1 State

```
# --------------------------------------------------------
# (c) Copyright 2014 by Jason DeLaat.
# Licensed under BSD 3-clause licence.
# --------------------------------------------------------

from pymonad.Monad import *

class State(Monad):
        """ Represents a calculation which produces a stateful side-
    ↪ effect.  """

        def fmap(self, function):
                """
                Applies 'function' to the result contained within the
    ↪ monad and passes the state
                along unchanged.

                """
                @State
                def newState(state):
                        result, st = self(state)
                        return (function(result), state)
                return newState

        def amap(self, functorValue):
```

```
        """
        Applies the function contained within the monad to the
↪  result of 'functorValue'
        and passes along the state unchanged.

        """
        @State
        def newState(state):
            function = self.getResult(state)
            value = functorValue.getResult(state)
            return (function(value), state)
        return newState

    def bind(self, function):
        """
        Chains together a series of stateful computations. '
↪ function' accepts a single value
        and produces a new 'State' value which may or may not
↪ alter the state when it is
        executed.

        """
        @State
        def newState(state):
            result, st = self(state)
            return function(result)(st)
        return newState

    @classmethod
    def unit(cls, value):
        """
        Produces a new stateful calculation which produces '
↪ value' and leaves the passed in
        state untouched.

        """
        return State(lambda state: (value, state))

    def getResult(self, state):
        """ Returns only the result of a stateful calculation,
↪  discarding the state. """
        return self.value(state)[0]

    def getState(self, state):
        """ Returns only the final state of a stateful
↪ calculation, discarding the result.  """
        return self.value(state)[1]

    def __call__(self, state):
        """
```

```
            Executes the stateful calculation contained within the
↪  monad with an initial 'state'.
            Returns the result and the final state as a 2-tuple.

            """
            return self.value(state)

    def __eq__(self, other):
            """
            Always raises a TypeError.
            The State monad contains functions which can not be
↪ directly compared for equality,
            so attempting to compare instances of State with
↪ anything will always fail.

            """
            raise TypeError("State: Can't compare functions for
↪ equality.")
```

Appendix C

Monad transformers in haskell

More can be found in the *transformers* package [23].

C.1 State

```
{-# LANGUAGE CPP #-}
#if __GLASGOW_HASKELL__ >= 702
{-# LANGUAGE Safe #-}
#endif
#if __GLASGOW_HASKELL__ >= 710
{-# LANGUAGE AutoDeriveTypeable #-}
#endif
-----------------------------------------------------------------------
-- |
-- Module      :  Control.Monad.Trans.State.Lazy
-- Copyright   :  (c) Andy Gill 2001,
--                (c) Oregon Graduate Institute of Science and
    ↪ Technology, 2001
-- License     :  BSD-style (see the file LICENSE)
--
-- Maintainer  :  R.Paterson@city.ac.uk
-- Stability   :  experimental
-- Portability :  portable
--
-- Lazy state monads, passing an updatable state through a computation
    ↪ .
-- See below for examples.
--
-- Some computations may not require the full power of state
    ↪ transformers:
```

```
--
-- * For a read-only state, see "Control.Monad.Trans.Reader".
--
-- * To accumulate a value without using it on the way, see
--   "Control.Monad.Trans.Writer".
--
-- In this version, sequencing of computations is lazy, so that for
-- example the following produces a usable result:
--
-- > evalState (sequence $ repeat $ do { n <- get; put (n*2); return n
    ↪  }) 1
--
-- For a strict version with the same interface, see
-- "Control.Monad.Trans.State.Strict".
-----------------------------------------------------------------------

module Control.Monad.Trans.State.Lazy (
    -- * The State monad
    State,
    state,
    runState,
    evalState,
    execState,
    mapState,
    withState,
    -- * The StateT monad transformer
    StateT(..),
    evalStateT,
    execStateT,
    mapStateT,
    withStateT,
    -- * State operations
    get,
    put,
    modify,
    modify',
    gets,
    -- * Lifting other operations
    liftCallCC,
    liftCallCC',
    liftCatch,
    liftListen,
    liftPass,
    -- * Examples
    -- ** State monads
    -- $examples

    -- ** Counting
    -- $counting
    -- ** Labelling trees
    -- $labelling
  ) where

import Control.Monad.IO.Class
import Control.Monad.Signatures
import Control.Monad.Trans.Class
import Data.Functor.Identity

import Control.Applicative
import Control.Monad
#if MIN_VERSION_base(4,9,0)
import qualified Control.Monad.Fail as Fail
#endif
import Control.Monad.Fix

-- ---------------------------------------------------------------------------
-- | A state monad parameterized by the type @s@ of the state to carry
    ↪ .
--
-- The 'return' function leaves the state unchanged, while @>>=@ uses
-- the final state of the first computation as the initial state of
-- the second.
type State s = StateT s Identity

-- | Construct a state monad computation from a function.
-- (The inverse of 'runState'.)
state :: (Monad m)
      => (s -> (a, s))  -- ^pure state transformer
      -> StateT s m a   -- ^equivalent state-passing computation
state f = StateT (return . f)
{-# INLINE state #-}

-- | Unwrap a state monad computation as a function.
-- (The inverse of 'state'.)
runState :: State s a   -- ^state-passing computation to execute
         -> s           -- ^initial state
         -> (a, s)      -- ^return value and final state
runState m = runIdentity . runStateT m
{-# INLINE runState #-}

-- | Evaluate a state computation with the given initial state
-- and return the final value, discarding the final state.
--
-- * @'evalState' m s = 'fst' ('runState' m s)@
evalState :: State s a  -- ^state-passing computation to execute
          -> s          -- ^initial value
          -> a          -- ^return value of the state computation
evalState m s = fst (runState m s)
{-# INLINE evalState #-}
```

```
-- | Evaluate a state computation with the given initial state
-- and return the final state, discarding the final value.
--
-- * @'execState' m s = 'snd' ('runState' m s)@
execState :: State s a  -- ^state-passing computation to execute
          -> s          -- ^initial value
          -> s          -- ^final state
execState m s = snd (runState m s)
{-# INLINE execState #-}

-- | Map both the return value and final state of a computation using
-- the given function.
--
-- * @'runState' ('mapState' f m) = f . 'runState' m@
mapState :: ((a, s) -> (b, s)) -> State s a -> State s b
mapState f = mapStateT (Identity . f . runIdentity)
{-# INLINE mapState #-}

-- | @'withState' f m@ executes action @m@ on a state modified by
-- applying @f@.
--
-- * @'withState' f m = 'modify' f >> m@
withState :: (s -> s) -> State s a -> State s a
withState = withStateT
{-# INLINE withState #-}

-- ---------------------------------------
-- | A state transformer monad parameterized by:
--
--   * @s@ - The state.
--
--   * @m@ - The inner monad.
--
-- The 'return' function leaves the state unchanged, while @>>=@ uses
-- the final state of the first computation as the initial state of
-- the second.
newtype StateT s m a = StateT { runStateT :: s -> m (a,s) }

-- | Evaluate a state computation with the given initial state
-- and return the final value, discarding the final state.
--
-- * @'evalStateT' m s = 'liftM' 'fst' ('runStateT' m s)@
evalStateT :: (Monad m) => StateT s m a -> s -> m a
evalStateT m s = do
    ~(a, _) <- runStateT m s
    return a
{-# INLINE evalStateT #-}

-- | Evaluate a state computation with the given initial state
```

```haskell
-- and return the final state, discarding the final value.
--
-- * @'execStateT' m s = 'liftM' 'snd' ('runStateT' m s)@
execStateT :: (Monad m) => StateT s m a -> s -> m s
execStateT m s = do
    ~(_, s') <- runStateT m s
    return s'
{-# INLINE execStateT #-}

-- | Map both the return value and final state of a computation using
-- the given function.
--
-- * @'runStateT' ('mapStateT' f m) = f . 'runStateT' m@
mapStateT :: (m (a, s) -> n (b, s)) -> StateT s m a -> StateT s n b
mapStateT f m = StateT $ f . runStateT m
{-# INLINE mapStateT #-}

-- | @'withStateT' f m@ executes action @m@ on a state modified by
-- applying @f@.
--
-- * @'withStateT' f m = 'modify' f >> m@
withStateT :: (s -> s) -> StateT s m a -> StateT s m a
withStateT f m = StateT $ runStateT m . f
{-# INLINE withStateT #-}

instance (Functor m) => Functor (StateT s m) where
    fmap f m = StateT $ \ s ->
        fmap (\ ~(a, s') -> (f a, s')) $ runStateT m s
    {-# INLINE fmap #-}

instance (Functor m, Monad m) => Applicative (StateT s m) where
    pure a = StateT $ \ s -> return (a, s)
    {-# INLINE pure #-}
    StateT mf <*> StateT mx = StateT $ \ s -> do
        ~(f, s') <- mf s
        ~(x, s'') <- mx s'
        return (f x, s'')
    {-# INLINE (<*>) #-}

instance (Functor m, MonadPlus m) => Alternative (StateT s m) where
    empty = StateT $ \ _ -> mzero
    {-# INLINE empty #-}
    StateT m <|> StateT n = StateT $ \ s -> m s `mplus` n s
    {-# INLINE (<|>) #-}

instance (Monad m) => Monad (StateT s m) where
#if !(MIN_VERSION_base(4,8,0))
    return a = StateT $ \ s -> return (a, s)
    {-# INLINE return #-}
#endif
```

```
    m >>= k  = StateT $ \ s -> do
        ~(a, s') <- runStateT m s
        runStateT (k a) s'
    {-# INLINE (>>=) #-}
    fail str = StateT $ \ _ -> fail str
    {-# INLINE fail #-}

#if MIN_VERSION_base(4,9,0)
instance (Fail.MonadFail m) => Fail.MonadFail (StateT s m) where
    fail str = StateT $ \ _ -> Fail.fail str
    {-# INLINE fail #-}
#endif

instance (MonadPlus m) => MonadPlus (StateT s m) where
    mzero       = StateT $ \ _ -> mzero
    {-# INLINE mzero #-}
    StateT m `mplus` StateT n = StateT $ \ s -> m s `mplus` n s
    {-# INLINE mplus #-}

instance (MonadFix m) => MonadFix (StateT s m) where
    mfix f = StateT $ \ s -> mfix $ \ ~(a, _) -> runStateT (f a) s
    {-# INLINE mfix #-}

instance MonadTrans (StateT s) where
    lift m = StateT $ \ s -> do
        a <- m
        return (a, s)
    {-# INLINE lift #-}

instance (MonadIO m) => MonadIO (StateT s m) where
    liftIO = lift . liftIO
    {-# INLINE liftIO #-}

-- | Fetch the current value of the state within the monad.
get :: (Monad m) => StateT s m s
get = state $ \ s -> (s, s)
{-# INLINE get #-}

-- | @'put' s@ sets the state within the monad to @s@.
put :: (Monad m) => s -> StateT s m ()
put s = state $ \ _ -> ((), s)
{-# INLINE put #-}

-- | @'modify' f@ is an action that updates the state to the result of
-- applying @f@ to the current state.
--
-- * @'modify' f = 'get' >>= ('put' . f)@
modify :: (Monad m) => (s -> s) -> StateT s m ()
modify f = state $ \ s -> ((), f s)
{-# INLINE modify #-}
```

```
-- | A variant of 'modify' in which the computation is strict in the
-- new state.
--
-- * @'modify'' f = 'get' >>= (('$!') 'put' . f)@
modify' :: (Monad m) => (s -> s) -> StateT s m ()
modify' f = do
    s <- get
    put $! f s
{-# INLINE modify' #-}

-- | Get a specific component of the state, using a projection
    ↪ function
-- supplied.
--
-- * @'gets' f = 'liftM' f 'get'@
gets :: (Monad m) => (s -> a) -> StateT s m a
gets f = state $ \ s -> (f s, s)
{-# INLINE gets #-}

-- | Uniform lifting of a @callCC@ operation to the new monad.
-- This version rolls back to the original state on entering the
-- continuation.
liftCallCC :: CallCC m (a,s) (b,s) -> CallCC (StateT s m) a b
liftCallCC callCC f = StateT $ \ s ->
    callCC $ \ c ->
    runStateT (f (\ a -> StateT $ \ _ -> c (a, s))) s
{-# INLINE liftCallCC #-}

-- | In-situ lifting of a @callCC@ operation to the new monad.
-- This version uses the current state on entering the continuation.
-- It does not satisfy the uniformity property (see "Control.Monad.
    ↪ Signatures").
liftCallCC' :: CallCC m (a,s) (b,s) -> CallCC (StateT s m) a b
liftCallCC' callCC f = StateT $ \ s ->
    callCC $ \ c ->
    runStateT (f (\ a -> StateT $ \ s' -> c (a, s'))) s
{-# INLINE liftCallCC' #-}

-- | Lift a @catchE@ operation to the new monad.
liftCatch :: Catch e m (a,s) -> Catch e (StateT s m) a
liftCatch catchE m h =
    StateT $ \ s -> runStateT m s `catchE` \ e -> runStateT (h e) s
{-# INLINE liftCatch #-}

-- | Lift a @listen@ operation to the new monad.
liftListen :: (Monad m) => Listen w m (a,s) -> Listen w (StateT s m) a
liftListen listen m = StateT $ \ s -> do
    ~((a, s'), w) <- listen (runStateT m s)
    return ((a, w), s')
```

```
{-# INLINE liftListen #-}

-- | Lift a @pass@ operation to the new monad.
liftPass :: (Monad m) => Pass w m (a,s) -> Pass w (StateT s m) a
liftPass pass m = StateT $ \ s -> pass $ do
    ~((a, f), s') <- runStateT m s
    return ((a, s'), f)
{-# INLINE liftPass #-}

{- $examples

Parser from ParseLib with Hugs:

> type Parser a = StateT String [] a
>    ==> StateT (String -> [(a,String)])

For example, item can be written as:

> item = do (x:xs) <- get
>        put xs
>        return x
>
> type BoringState s a = StateT s Identity a
>      ==> StateT (s -> Identity (a,s))
>
> type StateWithIO s a = StateT s IO a
>      ==> StateT (s -> IO (a,s))
>
> type StateWithErr s a = StateT s Maybe a
>      ==> StateT (s -> Maybe (a,s))

-}

{- $counting

A function to increment a counter.
Taken from the paper \"Generalising Monads to Arrows\",
John Hughes (<http://www.cse.chalmers.se/~rjmh/>), November 1998:

> tick :: State Int Int
> tick = do n <- get
>           put (n+1)
>           return n

Add one to the given number using the state monad:

> plusOne :: Int -> Int
> plusOne n = execState tick n

A contrived addition example. Works only with positive numbers:
```

```
> plus :: Int -> Int -> Int
> plus n x = execState (sequence $ replicate n tick) x

-}

{- $labelling

An example from /The Craft of Functional Programming/, Simon
Thompson (<http://www.cs.kent.ac.uk/people/staff/sjt/>),
Addison-Wesley 1999: \"Given an arbitrary tree, transform it to a
tree of integers in which the original elements are replaced by
natural numbers, starting from 0.  The same element has to be
replaced by the same number at every occurrence, and when we meet
an as-yet-unvisited element we have to find a \'new\' number to match
it with:\"

> data Tree a = Nil | Node a (Tree a) (Tree a) deriving (Show, Eq)
> type Table a = [a]

> numberTree :: Eq a => Tree a -> State (Table a) (Tree Int)
> numberTree Nil = return Nil
> numberTree (Node x t1 t2) = do
>     num <- numberNode x
>     nt1 <- numberTree t1
>     nt2 <- numberTree t2
>     return (Node num nt1 nt2)
>   where
>     numberNode :: Eq a => a -> State (Table a) Int
>     numberNode x = do
>         table <- get
>         case elemIndex x table of
>             Nothing -> do
>                 put (table ++ [x])
>                 return (length table)
>             Just i -> return i

numTree applies numberTree with an initial state:

> numTree :: (Eq a) => Tree a -> Tree Int
> numTree t = evalState (numberTree t) []

> testTree = Node "Zero" (Node "One" (Node "Two" Nil Nil) (Node "One"
    ↪ (Node "Zero" Nil Nil) Nil)) Nil
> numTree testTree => Node 0 (Node 1 (Node 2 Nil Nil) (Node 1 (Node 0
    ↪ Nil Nil) Nil)) Nil

-}

    files
    changes
    branches
    issues (4)
    fork
    file changes
    annotate
    download .zip
    darcs get url
    Packs built at 2015-11-20 14:15:46 UTC
```