Adventures in Meta-Casanova

How usable is Meta-Casanova

Rotterdam, 2015–2016

Written by

LOUIS VAN DER BURG
Hogeschool Rotterdam
Rotterdam

2015

Abstract

MC has not been tested in the field yet and we want to know how it fares. We will put it to the test and see that it still needs some development, but has promise.

Contents

1 Introduction **5**

2 The Problem **6**
2.1 MC 6

3 The Idea **7**

4 Details **9**
4.1 The basics 9
4.2 Standard Library 11
4.3 Monads implemented 13
4.4 Put to the test 14

5 Conclusion **16**

6 References **17**

1 Introduction

MC aims to be a strongly typed language to easily build compilers in.

Right now Meta Casanova (MC) has no base of usage and has not been tested in any practical environment. This means we have no way of knowing how practical MC actually is. We will set up basic test criteria to find out how good MC is in a practical environment. After which we explain MC in detail and put it to the test. Our findings will then lead to a conclusion as to what state MC is in.

1. Firstly we explain why we want MC to be tested for it's practical use.
2. Then we will explain on what criteria we will judge MC.
3. We will then dive into MC, explain the language, go over the criteria and see how it holds up.
4. Lastly we will conclude how good and useful MC really is at this stage and how to proceed from here.

2 The Problem

2.1 MC

The fact that there are very few strongly typed languages, stems from their complexity and usability in most practical forms.

MC aims to be the first strongly typed language which is also practical.

MC is made for creating other languages. This might sound odd at first, but when you look at Lisp, it doesn't sound too strange. Lisp has been made with the principle of bottom up programming [1]. Most experienced Lisp programmers first create the language they need with Lisp, after which they use this created language to program the actual application.

MC takes this quite literally and wants to be a language to easily build compilers in. It does this by using monads and monad transformers. You can easily combine monads via monad transformers to create a parser monad. A parser monad is quite adapt for parsing, hence its name. MC uses it as a basis for the compilers we can build with it.

As of this moment the language is not yet complete and has no parser monad, but all the building blocks are there.

We first want to test if MC is a usable language up to now. Then we can improve it to make it fit the purpose it aims to be.

3 The Idea

For a programming language to be practically usable it needs to be tested on the following criteria[3, 2]:

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

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

Read- & 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 it.

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

Documentation makes up a basic part of read- & writability. But to really be able to make a program readable, even after a prolonged absence, the language needs to be it's own documentation. This can be done via a manner of ways:

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

These points will be explained later on with examples off MC in section 4.4.1.

Simplicity of a language is decided by it's features. These should be easy to learn and remember. To make this more provable we will make use of these sub points:

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

We will touch on these briefly with concern towards MC. 4.4.2 But the main concept through which simplicity is achieved, is through basing the language around a few simple well-defined concepts. [3]

Definiteness makes it clear 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 tho language.

Orthogonality means that the language should be predictable. When one user programs the same as another user, the result should be the same. This implies that if the user understands the basics of the language, he can then 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.

Expressiveness goes hand in hand with abstraction. The language should be able to show the user the program in a way that matches with his thoughts.

Machine language is hard to understand for a human, it is too basic in it's abilities. 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. Therefor the language should be similar to the way the

user thinks. This makes it easier for the user to express his thoughts in code.

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

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 a long time.

Time is something everything needs to build momentum and a stable user base, even programming languages can't escape this. For now we won't be able to really test this, as MC is still in it's development stages. For this reason we will leave this out.

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

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.

Redesign enables the evolution of a program written in the language. This makes it easier to go from a rough prototype to a fully featured program.

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 doesn't 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% provable. Thus we will try to discuss the most objective parts for provability's sake.

First we will explain MC in more detail, after which it can be put to the test. We will see how far it already has the above described criteria and in which areas it still needs some improvement.

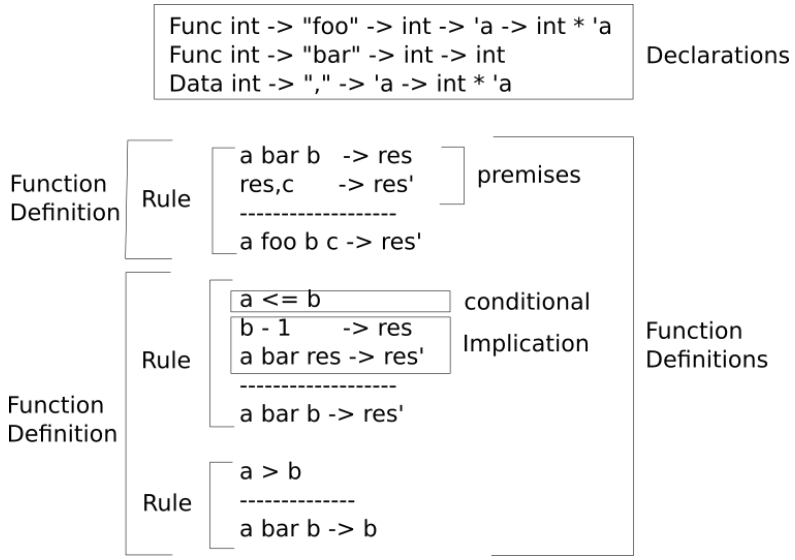
4 Details

We will now look at MC itself and explain the syntax and the built-in features by looking at examples from the standard library.

First we will look at the basics of MC.

4.1 The basics

Let's start with giving a simple example of some MC source code:



Now let's go over the different parts:

Declarations enable the user to define their own keywords. The keywords are in parenthesis and the arrows make up the order of the declaration. As shown above it is possible to have functions with variables on the left side of the declaration. The variable after the final arrow is the return value of the function.

Function definitions occur after the declarations. The rule starts with the input and output below the line of dashes, called the bar. On the left of the arrow the input is stated and on the right the output. Above the line are where the function actually does the work. It tests the premises or conditionals to see if the rule can be executed.

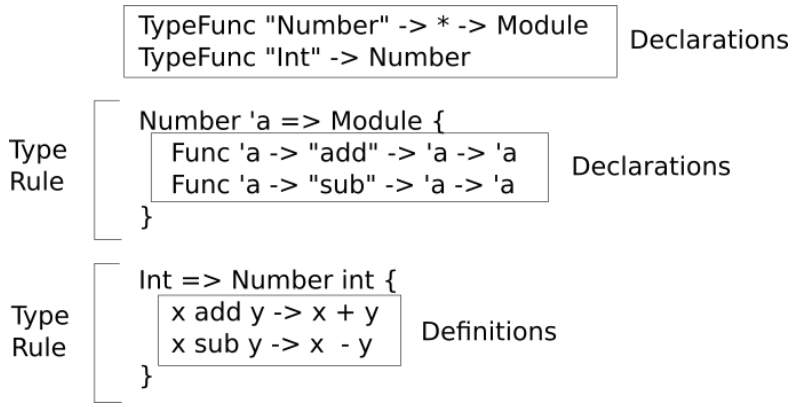
We can define multiple rules per function and they will be simultaneously executed. The program will split and rules that don't match will stop.

Data and Func are both used to declare. Func defines a function and Data defines a data type. The major difference between them is that Data is two-way traversable and Func only one-way. This means that when we have a variable which has a type created by Data we can extract the start values. We

cannot do this with Func. Func also has the ability to curry, but both however define terms by using types.[1]

4.1.1 Going higher

Take a look at the following source code:



Here we see a similarity with the previous code example. There are again declarations and rules, this time however we also see a declaration in a rule. Let's go over the new keywords one by one.

TypeFunc is similar to Func in that it declares. However, as the name suggests, TypeFunc defines a type and not a term. And it uses kinds to do this.[2] These types can than be implemented by Func as we can see in the example.

Another difference are the arrows used. Instead of using the normal arrow (->) the => arrow is used in TypeFunc declarations. This is to add clarity as to what is being declared.

Module is used to create a collection of declarations. In this case two Func declarations are used. We can also use the TypeFunc or Data declarations within a module.

When we take the example module Number, we can see it takes an argument. As we can see in the Func declarations it is implied that this argument is a type and not a variable. These variables are called type-variables and are distinguishable by an apostrophe at start of their name.

The declaration of "add" and "sub" within Int assume that the + and - operators are already defined. We will later see how to use built-in and system operators in section 4.2.1. Because Int makes use of the module Number by feeding it a type-variable, there is no need for declarations of "add" and "sub." They exist within the module Number and can thereby be used by any definition of the Number.

Lambdas are defined by:

```
(\ variables ->
do stuff) -> input\_for\_lambda
```

(TODO)

[1] For more information about terms and types see [4]
[2] For more information about kinds see [4]

4.2 Standard Library

Now let's have a look at what is in the standard library and explain some more functionalities of MC. The standard library consists of the following:

1. StandardLibrary
   (a) boolean
   (b) monad
   (c) tryableMonad
   (d) number
   (e) newnumber
   (f) match
   (g) prelude
   (h) record
2. BasicMonads
   (a) either
   (b) id
   (c) list
   (d) option
   (e) result
   (f) state

4.2.1 System-links

System-links refer to the .NET libraries which are implemented within MC. When we take a look at boolean we can see how system-links are implemented within MC:

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

First we specify which system-links we want to implement after which we tell MC it is a system-links. Using this manner of loading system-types we don't have to write everything from scratch and quickly build upward.

Of course this only tells MC that true and false are of type boolean. MC still doesn't understand which value they have. For this we simply implement the Boolean Module, as is done in prelude:

```
import boolean

Boolean => Boolean {
   True -> TrueBoolean^system
   False -> FalseBoolean^system
}
```

First we import the Boolean Module from the file boolean. Then we implement Boolean and tell which term True and False will get. Because of scoping we con use the same name for Boolean as the Module name for Boolean.

When we look at Monad and TryableMonad we see a few new keywords as well as a more complex practical application of MC.

4.2.2 Monad

```
import prelude

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

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

   Func "MCons" -> *
   MCons -> 'M

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

   Func "lift" -> ('a -> 'b ) -> 'M 'a -> 'M 'b
   a >>= a'
   --
   lift f a -> return(f a')

   Func "lift2" -> ('a -> 'b -> 'c) -> 'M 'a -> 'M 'b -> 'M 'c
   a >>= a'
   b >>= b'
   --
   lift2 f a b -> return(f a' b')

   TypeFunc "liftM" => (* => *) => * => *
   N >>= a
   f a -> b
   lift^N(return^N b) >>= res
   --
   liftM f N -> return res
}
```

In Monad the first thing we see is a TypeFunc which tells MC what exactly a monad is. It takes as an argument "(* => *)" and returns a Module. As we have explained in [**?**] a Module is used as a container. With this in mind a Monad is nothing but a collection of declarations which takes an argument.

When we look at the argument it takes we notice it is a function of some sort which takes a type and returns a type. Which in kinds are described as *.

The 'M stands for a monad it takes. This might be confusing if you have any knowledge about monads, since monads are not always a function. That's because Monad is actually a basis for monad transformers. As we will see in [**?**], it comes in very handy when working with monads.

But let's first look at the new keywords we see in Monad.

Import does what most programmers would expect, it imports the functions from the file specified after "import."

ArrowFunc is an abbreviation of Func. It creates a function which can take arguments placed on the right of the "operator", from the line below it.

It gives an error if the number of right arguments is not met.

At the end of ArrowFunc we see another type of arrow, #>, the priority arrow. It gives the possibility to give it a priority and say if it is left- or right associative. ArrowFunc is standard right associativity, so there is no need to specify that other than clarity.

Comments are a simple but essential feature. They enable the programmer to explain what is happening when the code is complex. In MC a single comment line starts with $$ and when we want to comment a block we use $* to start and *$ to end the block.

4.2.3 TryableMonad

```
import prelude
import monad

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

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

   Data "e" -> String

   Func "getMonad" -> MCons^'M
   getMonad -> 'M

   Func "Tryable" -> 'a -> 'a
   Tryable -> a -> a
}
```

Now let's look at what's new in TryableMonad

Inherit has a similar functionality as import, but it imports the functions of a variable. Note that this is only possible if the variable is based on a module, as only a module can contain function definitions and declarations.

4.3 Monads implemented

When we look at a basic implementation of the monads in MC, we will see how using monad transformers is much easier in the end.

We will first take a look at the ID monad transformer and then see how this will make the transformers usable.

```
import prelude
import monad

TypeAlias "Id" => * => *
Id 'a => 'a

TypeFunc "id" => Monad

id 'M a => Monad(Id) {
   inherit 'M

   x >>= k -> k x
   return x -> x
}
```

TypeAlias creates a data type on kind level. This way we can use Id as a signature to implement the "id" monad. Without TypeAlias there would be no way of actually implementing a monad.

As we can see, having a basic understanding of MC, the id monad does nothing with the input. This makes it pass the functionality of the input monad transformer directly as output, thus creating the monad itself.

Without having an id monad there would be no way of using the monad transformers, making them pointless.

4.4 Put to the test

Now that we have a better understanding of MC, we can put it to the test using the criteria described in section 3.

4.4.1 Read- & Write ability

Keywords are few and clear in what they do. Furthermore we can define our own keywords by using the different declaration methods.[**?**] Which means we can basically build our own language which is strongly typed and tuned to our needs.

Abbreviations and concise notation is quite clear in MC. It has almost no shorthand functionality, which makes it clear what is what. The notation is also quite clear and kept to a minimum, which does exactly what it should and nothing more. This is a quick way to recognize the code structure and it's meaning.

Comments are implemented in a proper way as shown in [**?**].

Layout or format of programs is mostly a fixed layout, like most modern functional programming languages. This is a price for having a strong type system.

No overuse of notation rarely comes into play. At most we will see the bars fairly often and declarations quite often.

4.4.2 Simplicity

Structure is one of the stronger points of MC. Declarations and there implementation make it so there is a good structure to be made in the source code. We can group the declarations together and the implementations or group the declarations together with their implementations.

The number of features are basic but very expandable. This makes it easy to understand even complex programs, when we see how they are build on top of the basics.

Multiple ways of specification are achievable through declaring them. Though the basics of the language is quite strict, we can build almost anything with them. So if we want we can declare multiple ways of doing the same thing.

Multiple ways of expressing are achievable by overloading the standard operators or declared operators. This can make it very obscure what is actually done, but gives us the freedom to create and tune our own operators.

4.4.3 Definiteness

Definiteness is definitely there, but not completely realised yet. At least the parser monad should be implemented in the standard library. And to be really sure an example compiler would be handy or some proper documentation.

All the basics are there to realize the goal of MC, it just needs to be implemented.

4.4.4 Orthogonality

Orthogonality is probably one of the things MC aims to achieve the hardest with it's type system. This has definitely been achieved. The strong type system which makes use of three levels: terms, types and kinds. Makes sure the language is similar in predictability as languages such as Ada.

4.4.5 Expressiveness

Expressiveness is quite high because of the high level of abstraction in MC. This enables us to express very complex ideas with very little and clear code.

4.4.6 Efficiency

Efficiency is something we can't really test without a fully functional compiler. For we cannot test this. For more on this subject see [**?**]

4.4.7 Libraries

Libraries are definitely lacking at this moment in time. This only concerns the StandardLibrary, the link with .NET makes the libraries considerably stronger.

4.4.8 Hackability

Hackability is a big part of MC, we can create the language we want. MC itself can be quite powerful by itself, but it really shines when we build a compiler for the language we had in mind.

This might be quite a big step for a lot of programmers, but it can be very productive if they try it. It can save them time in the end.

4.4.9 Succinctness

Succinctness is something MC falls short. We have to declare every function we will ever make, which is anything but succinct. But as with Ada and Haskell, we have compromise something for a strong type system.

4.4.10 Redesign

Redesign is also a critical point in MC. Because of the declarations needed, we will have to redo those entirely when rewriting or redesigning. On the other hand, we can edit the implementation of a certain declaration any way we want, ass long as the input and output work the same way.

MC definitely has not been made for evolutionary programming. It's more of a think-before-you type language.

4.4.11 External Factors

External seem to be promising for MC. The main advantage MC has is the interface with the .NET libraries. This enables native compatibility on the Microsoft Windows platforms and via Mono it, can also work on the GNU/Linux platforms.

5 Conclusion

So far we have learned what makes a programming language practically usable, how MC works and how practically usable MC is, in it's current state.

MC still needs some development before it has reached it's goal. There is no complete compiler for MC itself and the StandardLibrary still needs some additions. Mainly the parser monad.

6 References

[1] Paul Graham. *On lisp*. Prentice Hall, 1994.

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

[3] Uday P Khedker. What makes a good programming language. Technical report, Technical Report TR-97-upk-1, Department of Computer Science University of Pune, 1997.

[4] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.