Bachelor Thesis

# Design and implementation of the Meta-Casanova 3 Compiler back-end

*Douwe van Gijn*

supervised by
Dr. Giuseppe Maggiore

2016-05-25

# Contents



# 1 Introduction

This project is about the development of the back-end of the bootstrap compiler for the Meta-Casanova 3 language. The back-end is responsible for generating an executable after receiving the type-checked information from the front-end.

Compilers are complex programs that have to operate on a wide range of inputs. Since compilers have such a large input-space, the chance of a bug hiding somewhere is substantial. But for al their complexity, compilers also have to be bug-free since every program can only be as bug-free as its compiler.

Abstractions can help in this regard. The limits of which were observed when implementing the compiler for the Casanova language in F#. The compiler was 0000 lines long, and became unmaintainable. After a rewrite in MC it was 000 lines[Maggiore].

The primary reason for this was the lack of higher-order type operators. This made abstractions such as monad-transformers impossible, hampering modularity and resulted in a lot of boilerplate code.

In this document, we will walk through the backend and examine the various parts and their design decisions. In this way, this document aims to be useful to the future developers of the MC compiler.

## 1.1 Research questions

*How to transform typechecked Meta-Casanova(MC) into executable code?*

Where the transformation must satisfy these requirements:

1. The backend must in no case produce an incorrect program.
2. The executable must be able to interoperate with .NET.
3. The generated code must run on all the platforms .NET runs on.

An additional soft requirement was that the performance of the generated program should be as high as possible.

The first requirement exists because the compiler must be reliable. Any program can at most be as reliable as the compiler used to generate it.

The second requirement existed because of the need for a large library and interoperability with Unity game engine. This is because the main area of research of the organization is game-related.

In order to answer the research question, seven subquestions were formulated.

1. In what language should the code generator produce its output?

2. What should the interface be between the front-end and the back-end?
3. What should the intermediate representation of the functions be?
4. How does the intermediate representation map to the output language?
5. How to make sure that the generated names comply with the output language?
6. How to validate the code-generator?
7. How to validate the test programs?

These subquestions will be answered in their respective subsections in section 3.

### 1.2 Organization

— todo: describe organization —

## 2 Meta Casanova

Meta Casanova is a functional, declarative language. This section will cover the subset of the language that is relevant for code-generation.

It allows for multiple implementations of functions called *rules*. Rules may fail, in that case the next rule will be attempted. This will continue until a rule succeeds, or no rule matches in which case the program throws an exception.

### 2.1 Data

`Data` declarations declare an algebraic union[**algebraic_datastructures**].

```
Data "nil" -> list<'a>
Data 'a -> "::" -> list<'a> -> list<'a>
```

defines the same structure as this F#-like pseudocode.

```
List<'a> = nil
         | 'a :: List<'a>
```

In this example, the list type is declared with two constructors. They specify that a lists can be constructed in two ways: with `nil` and with `::` surrounded with a term of type `'a`, and a term of type `list<'a>`.

Conversely, they also specify that an list can be destructed in two ways. The programmer will assert which destructor is expected, and the rule fails if the destructor does not match. An example of this is shown later, in subsection “Funcs”.

Additionally, constructors may be manipulated and partially applied like functions. This allows for greater flexibility at the cost that function and constructor names need to be unique in their namespace.

### 2.2 Polymorphism

Polymorphic data structures are supported with the **`is`** keyword.

```
Data "error" -> string -> failableList<'a
    >
failableList 'a is list<'a>
```

This means every constructor of the `list` is also a valid constructor of `failableList`, but not vice-versa.

### 2.3 Funcs

Func declarations specify a new function and its type.

```
Func "length" -> list<'a> -> int
```

As with constructors, functions may be freely manipulated and partially applied, and have the restriction that their name must be unique in their namespace.

### 2.4 Rules

Meta-Casanova uses a syntax similar to that of natural deduction. For each Func declaration, there are one or more rules that define it.

```
---------------
length nil -> 0

length xs -> res
---------------------
length x::xs -> 1+res
```

A rule is comprised of a line with below it on the left of the arrow the input, and on the right the output. The statements above the horizontal line are called *premises*. They can be assignments like in the example above, or conditionals like `a==b` or `c<d`.

In the case of assignments, they create a *local identifier*. These identifiers are local to the rule they appear in. The input arguments of the rule are also local identifiers.

We can now call the function `length` with an example list:

```
1::(2::nil) -> x
length x    -> res
```

The first premise constructs a list called "x", and the second statement calls length with that list. The program will execute as follows:

```
length 1::(2::nil)
    ~~nil~~
    x::xs → 1+(length 2::nil)
        ~~nil~~
        x::xs → 1+(length nil)
            nil → 0
            ~~x::xs~~
```

After which the function stops calling itself and starts accumulating the result on the way down.

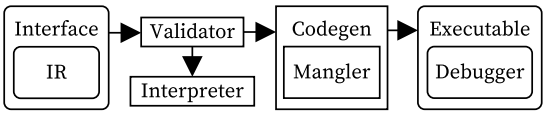
```
      1 ← 1+0
  2 ← 1+1
```

After which it tells us correctly that the length of the list 1::(2::nil) is indeed 2.

## 3 Research

As discussed in section 1.1, the main research question is split in subquestions. Each answer of a subquestion is proven by implementing a part of the backend.

To illustrate how the different parts of the back-end relate to each other, here is a diagram of the dataflow through the backend.




As you can see, the front-end interface contains the IR and goes through the validator. From there, depending on the compiler flags, it either goes to the interpreter or the codegen. In case it goes to the interpreter, the program is directly executed. In case it goes to the codegen, it is translated to the output language. To translate all the identifiers, the mangler is needed. The debugger is optionally embedded in the executable, depending on compiler flags.

### 3.1 Output language

The first research question had the most impact on the project, and was one that was difficult to change later on.

*In what programming language should the code-generator produce its output?*

This may be different than the language the code-generator is written in. The code-generator is written in F#, like the rest of the compiler. The reason we use F#, rather than Meta-Casanova 2, is because Meta Casanova 2 lacks tool support, such as descriptive error messages and debuggers.

#### Unmanaged languages

Since speed was one of the requirements, I first looked at solutions with unmanaged parts.

Unmanaged code is code that is not interpreted by a runtime, but is instead executed directly.

The main advantage of unmanaged code is that the fast LLVM code generator can be used. LLVM is a "collection of modular and reusable compiler and toolchain technologies."[1] Specifically, the LLVM optimizer is valuable. It is used in the Clang, a C/C++ compiler on par with gcc, and with little effort can be use it to optimize our generated code. This would mean we get all the optimisations of LLVM with relative ease. It would however mean that we had to implement a garbage collector, as LLVM does not come with one.

.NET compatibility is also required, as explained in section 1.1. There are a few systems that allow for managed and unmanaged code to communicate. The most viable are P/invoke, C++/CLI interop, and a hosted runtime.

**P/invoke** Platform Invocation Services allows managed code to call unmanaged functions that are implemented in a DLL.[2]

This is the most common form of inter-op, and has great documentation. However, there are two big disadvantages.

1. .NET can only call native functions, not the other way around. This means that the bulk of the control flow happens inside .NET, minimizing the fast native code.
2. Transfering data between .NET and native code has a high performance cost[3], since it has to be serialized. This overhead is so large that we expect it to negate any performance benefit from using native code.

Because of this, P/Invoke was not chosen.

**C++/CLI** C++ for the Common Language Infrastructure is a programming language designed for interoperability with unmanaged code.[**msdn_c++cli**]

While it seems like it does exactly what we need, it has portability issues. The only C++/CLI compiler runs on windows and it only compiles for processors with the x86 architecture[4]. Besides that, non-typesafe operations (the main advantage of C++/CLI) are only allowed on windows.[4]

This means C++/CLI is not cross-platform enough.

**Hosted runtime** It is possible to embed a .NET runtime inside a native program. This would make it so the control flow takes place inside the native part.

This seems like the best solution out of the native hybrids. However it still has two drawbacks. The mono runtime has a different interface than the microsoft .NET api, leading to incompatible programs [5]. The same large serialization overhead as P/Invoke is present[6].

**.NET languages**

None of the inter-op methods offer a satisfactory solution. They all have downsides that outweigh the benefits. It was decided to let go of the LLVM code-generation in favor of a more portable and reliable system.

Stability is a big advantage because everything happends inside the .NET runtime. This has a higher chance of working on non-native platforms than the hybrid solutions.

**F#** is a functional/declarative language in the .Net family[**.**] It would be a natural choice, since the compiler is written in it. However, it is quite slow[**fsharp_slow**] and resists the imperative style of the generated code. The programmer has also less control of the program execution, as F# is a more higher-level language[**fsharp**]

A version of the code generator was made that used F#, but it proved too cumbersome since there was no mechanism to simply skip rules if they failed.

**C#** is an imperative, object-oriented language[**csharp**]. It is the most popular .NET language[7], so the compiler gets the most attention by Microsoft. It is also easy to debug, as it has the most mature debugging tools.

When implementing the code-generator in this style, it was possible to write the code-generator so that the output-code is straight-line code. This is opposed to F#, where I had to generate a tree-structure as an output. This greatly improved debugability and simplicity of implementation.

**CIL** (Common Intermediate Language) is the bytecode that all the languages are compiled to. Since it is typed, it has the same restrictions as C#.[source] As a result, it makes debugging and verification harder, with little to no gain. It also ommits the optimizations of the C# compiler, such as dead-code elimination and stuff[**csharp_optimizations**].

#### Conclusion

The debugability together with a lot of control make C# the best choice for code generation.

### 3.2 The front-end interface

The second research question is about the specification of the front-end interface.

*What should the interface between the front-end and the back-end be?*

The front-end interface contains all the input for the backend. This makes testing very easy, as the rest of the backend only relies on its input.

#### Interface

```
type Interface = {
  datas      : List<Id*Data>
  funcs      : List<Id*List<rule>>
  lambdas    : List<LambdaId*rule>
  main       : rule
  flags      : CompilerFlags
  assemblies : List<string>
}
```

As you can see, the interface contains the data declarations, function definitions, lambda definitions and a main function.

The design principles for this interface were simplicity and minimalism. There should be as few ways as possible to represent the same program. This makes testing easier and minimizes bugs that appear only in certain representations of the same program.

All the symbols in the descriptions are provided with monomorphic types by the front-end. Functions with generic types are made concrete by the front-end.

The reason that datas, funcs and lambdas are defined as a list of key-value pairs instead of as a Map, is that the keys are not guaranteed to be unique. Since MC allows polymorphic types, one indentifier may be defined multiple times: once for each type. There is no performance penalty for the back-end, as no lookups by identifier are performed.

#### Data declarations

The data declarations are grouped with the identifier of the constructor.

```
datas : List<Id*Data>
```

Where `Data` is simply a list of input types and output types.

```
type Data = {
  args       : List<Type>
  outputType : Type
}
```

Where `Type` represents a monomorphic MC type.

To illustrate, let's define a tuple an a union in MC.

```
Data int -> "," -> string -> Tupple<int
    string>
Data "fst" -> int    -> Union<int string>
Data "snd" -> string -> Union<int string>
```

This will appear as the following list in the interface:

| identifier | arguments | type |
|---|---|---|
| "," | int; string | Tuple<int string> |
| "fst" | int | Union<int string> |
| "snd" | string | Union<int string> |

#### Rule containers

Function and lambda definitions, as well as the main function contain rules.

```
funcs   : List<Id*List<rule>>
lambdas : List<LambdaId*rule>
main    : rule
```

Functions in MC can contain multiple rules that implement them.

The entry point of the program is defined by a single rule, here called `main`. It is not a full function since full functions can have multiple rules. This was done to make the entry-point as simple as possible.

#### Rules

Functions are defined with of one or more *rules*. This is how they are represented in the interface.

```
type rule = {
  premises    : List<premise*linenr>
  input       : List<local_id>
  output      : local_id
  typemap     : Map<local_id,Type>
  declaration : Position
  definition  : Position
}
```

The main component of rules are their premises. They are the instructions that make up the rule. The instruction set is described in section 3.3.

The premise list also contains line numbers for each premisse. This is debug information, that is used by the embedded debugger (section 3.7).

Next are the inputs and output of the rule. Input and outputs consist only of local identifiers. This is because of *normalization*.

In the case that a rule-input or output has an expression instead of a local identifer, the expression is assigned to a new local identifier and the local identifier is substituted. More on normalization in section 3.3.

The typemap contains a map from the local identifiers in a rule to their types. This gives the back-end all the information that the typechecker has accumulated.

The last two members are `declartion` and `definition`. These represent the position that the function was declared and the position that it was defined. This information is used by the debugger.

#### Validator

The first versions of the backend had no working front-end to test with. Early testing was done by writing the interface datastructure by hand. Because that was error-prone, I implemented an automatic checker for the interface to check the invariants.

The validator asserts the following:

- Each local identifier is defined only once.
- Each local identifier has a type in the typemap.
- Each function has at least one rule.

The validator was initially only for validating hand-written interfaces, but it proved to be very good in catching errors that slipped through the front-end. The validator now al-

ways checks the interface before it is handed to the codegen.

## 3.3 Intermediate Representation

While the intermediate representation(IR) of the functions is part of the interface, it is complex enough to have its own research question.

*What should the intermediate representation of the functions be?*

Each rule contains a list of premises (see section 2.4). These premises represent the executable code in each rule.

To minimize the number of representations of the same program, all compound premises are split into multiple premises that do only one operation each. This process is called *normalization*.

The instruction set exists in two parts: the base instructions and the .NET extentions.

### Base instructions

The instruction set was designed to minimize the number of representations of the same program. This happens to coincide with a small orthogonal instruction set.

The instruction set is in *static single assignment* (SSA) form. This means the local identifiers are constant and can not be redefined.

Base instructions fall in one of two groups. The first maps a global identifier to a local identifier. This are the Literal and Closure instructions. The second operates on local identifiers. The Conditional, Deconstructor, Application and Call instructions belong to this group.

**Literal** (`42 -> x`) assigns a string-, boolean-, integer- or floating-point literal to a local identifier.

**Conditional** (`x < y`) asserts that a comparison between local identifiers is true. The comparisons can be <,<=,==,>=,> or `!=`. If the assertion fails, the rule fails and the next rule in the function is attempted.

**Deconstructor** (`lst -> x::xs`) disassembles a local identifier constructed by a data declaration.

**Closure** (`((+) -> add`) assigns a closure of a global function to a local identifier. The closure can hold a function, lambda or data-constructor.

**Application** (`add a -> inc`) applies a local identifier to a closure in another local identifier.

**Call** (`inc b -> c`) applies a local identifier and calls the closure. All closures need to be called eventually to be usefull. The exception is data-constructors. They do not have to be called as they insert their elements in the datastructure as they are applied.

### .NET extentions

A seperate set of instructions are needed to inter-operate with .NET. This is because unlike MC, .NET objects are mutable, and the functions can be overloaded on the number and types of arguments.

| instruction | MC example |
|---|---|
| call | `System.DateTime d m y -> date` |
| static call | `System.DateTime d m y -> date` |
| get | `System.DateTime d m y -> date` |
| static get | `System.DateTime d m y -> date` |
| set | `System.DateTime d m y -> date` |
| static set | `System.DateTime d m y -> date` |

### Evolution

It was briefly considered to use an existing intermediate representation, like CIL or LLVM-IR. However, it would mean over 100 instructions and the front-end would do most of the work. It would also mean the front-end needed its own codegen to generate the CIL instructions.

Call did not used to apply an argument, but it caused inconsistencies in the type-checker. There would be not difference in the type of the uncalled closure and the called closure, resulting in an extra bit of information being required with the type. This caused special-cases all over the codebase, so it was decided to make application take an argument, like in lambda-calculus.

Application used to also take the position of the argument that was applied. This was because the backend did not care in what order the closures were applied. But since the MC language only allows for in-order closure application, the decision was made to make the position of the argument implicit to limit the program representations.

Comparisons could first only take a boolean local identifier. It was changed to a predefined set of comparisons because of two reasons. Firstly, it makes the language-agnostic base instructions depend on .NET Booleans. Secondly, by restricting the inputs to only a predefined set of comparisons, we restrict the number of representations for the same program.

## 3.4 Code generator

The fourth research question gets at the heart of the back-end.

*How does the intermediate representation map to the output language?*

The codegen is in many ways the heart of the back end, as it is responsible for generating the C# code.

#### Functions

Every function was implemented as a closure. In C# this means a class with a public field for each function argument and a `_run` function that takes the last argument and executes the function.

```
class <function name> {
    <function arguments>
    public <return type>
    _run(<last argument>) {
        {
            <rule 1 implementation>
            return <local>;
        }
      skip1:
        {
            <rule 2 implementation>
            return <local>;
        }
      skip2:
        ⋮
        {
            <rule n implementation>
            return <local>;
        }
      skipn:
        throw new <exception>;
    }
};
```

The `_run` function opens a local scope followed by a goto-label for each rule in the function. This allows rules to easily fail by jumping ahead to the label. More on rules in section 3.4.

#### Data declarations

Data declarations are implemented with inheritance. The declared type is represented by an empty baseclass and all the constructors inherit from it.

This is a pretty straight-forward transformation.

```
Data string -> "," -> int -> string * int

Data "Left"  -> string -> string | float
Data "Right" -> float  -> string | float
```

The above MC code transforms into the following C# code.

```
class _star {};
class _comma { string _arg0; int _arg1;}

class _pipe {};
class _Left :_pipe {string _arg0;};
class _Right:_pipe {float  _arg0;};
```

The types `_star` and `_pipe` can now be easily be deconstructed. When a premis deconstructs a datatype, it asserts that a type is constructed by a specific constructor. This is done by simply casting the base-class to a subclass, and checking if the cast succeeded. If the cast failed, the rule does not match and the rule is skipped.

### Rules

Each rule defines its own name for each input argument. These names do not have to be the same, for example:

```
Func "evenOrOdd" -> int -> string

a%2 = 1
-----------------
evenOrOdd a -> "odd!"

b%2 = 0
------------------
evenOrOdd b -> "even!"
```

Of course, by the time the code has arived by the codegen, it would already have been normalized. So the rules actually look more like this:

```
(%) -> _tmp0          (closure)
_tmp0 a -> _tmp1      (application)
2 -> _tmp2            (literal)
_tmp1 _tmp2 -> _tmp3  (call)
0 -> _tmp4            (literal)
tmp4 = tmp0           (conditional)
"even" -> _tmp5       (literal)
--------------------
evenOrOdd a -> _tmp5


(%) -> _tmp0          (closure)
_tmp0 a -> _tmp1      (application)
2 -> _tmp2            (literal)
_tmp1 _tmp2 -> _tmp3  (call)
1 -> _tmp4            (literal)
tmp4 = tmp0           (conditional)
"odd" -> _tmp5        (literal)
--------------------
evenOrOdd a -> _tmp5
```

The first job of the rule is to translate the input arguments to their name and return the output.

```
{
    var a = _arg0;
    ...
    return _tmp5;
}
_skip0:
{
    var b = _arg0;
    ...
    return _tmp5;
}
_skip1:
```

Then each instruction is generated.

```
{
    var a = _arg0;
    // closure
    var _tmp0 = new _plus();
    // application
    var _tmp1 = add;
    _tmp1._arg0 = a;
    // literal
    var _tmp2 = 2;
    // call
    var _tmp3 = _tmp1.run(_tmp2);
    // literal
    var _tmp4 = 1;
    // conditional
    if(!(_tmp3=_tmp4)){goto _skip0;}
    // literal
    "odd!" -> _tmp5;
    return _tmp5;
}
_skip0:
{
    var b = _arg0;
    ...
    return _tmp5;
}
_skip1:
```

See the figure 1 on the next page for an overview of instruction generation.

### Evolution

Before using inheritance, the plan was to use overlapping memory like C unions. Using `System.Runtime.InteropServices`, it was possible to set the specific offset of struct members. While this was multiplatform and worked well, it only worked with structs. This was a major limitation, because structs can only hold value-types. And the only value types are integers, floats, booleans and other structs.

figure 1: *an overview of instruction generation.*

| **instruction** | **MC** | **C#** |
|---|---|---|
| literal | `42 -> x` | `var x = 42;` |
| conditional | `x > 40` | `if(!(x>40)){goto skip0;}` |
| deconstructor | `lst -> x::xs` | `var _tmp0 = lst as _colon_colon;`<br>`if(_tmp0==null){goto _skip0;}`<br>`var x  = _tmp0._arg0;`<br>`var xs = _tmp0._arg1;` |
| closure | `(+) -> add` | `var add = new _plus();` |
| application | `add a -> inc` | `var inc = add;`<br>`inc._arg0 = a;` |
| call | `inc b -> c` | `var c = inc.run(b);` |
| **.NET instr.** | **MC** | **C#** |
| call | `date.toString format -> str` | `var str = date.toString(format);` |
| static call | `System.DateTime.parse str -> date` | `var date = System.DateTime.parse(str);` |
| get | `date.DayOfWeek -> day` | `var day = date.DayOfWeek;` |
| static get | `date.DayOfWeek -> day` | `var day = date.DayOfWeek;` |
| set | `hr -> System.DateTime.hour` | `System.DateTime.hour = hr;` |
| static set | `hr -> System.DateTime.hour` | `System.DateTime.hour = hr;` |

## 3.5 Mangler

The Mangler could be seen as part of the codegen, but the decisions are intresting enough to get its own research question.

*How to generate the identifiers so they comply with the output language?*

The mangler is responsible for generating a unique C# identifier for every instance of an MC identifier. The mangler is designed to be simple, and produce readable output. Readable output makes it easy to verify both the mangler and the generated code.

There are two kinds of identifier: global identifiers and local identifiers. Global identifiers have a fully-qualified name with type information, where as local identifiers only have the simple name.

#### C# identifiers

Since there are more valid MC identifier names than C# identifier names, some characters have to be escaped.

Valid C# identifiers must start with an alphabetic character or an underscore and the trailing characters must be alphanumeric or underscore[1][8]. The only valid non-alphanumeric character is an underscore, so using it to escape with was a logical choice.

The first iteration of the code mangler just replaced all non-numeric characters with an underscore followed with the two-digit hexadecimal number. This generated correct identifiers but was very unreadable, `>>=` would translate to `_3E_3E_3D`. To remedy this, every ascii symbol gets a readable label.

| | | | | | |
|---|---|---|---|---|---|
| `!` | `_bang` | `-` | `_dash` | `=` | `_equal` |
| `#` | `_hash` | `.` | `_dot` | `?` | `_quest` |
| `$` | `_cash` | `/` | `_slash` | `@` | `_at` |
| `%` | `_perc` | `\` | `_back` | `^` | `_caret` |
| `&` | `_amp` | `:` | `_colon` | `_` | `_under` |
| `'` | `_prime` | `;` | `_semi` | `` ` `` | `_tick` |
| `*` | `_amp` | `<` | `_less` | `\|` | `_pipe` |
| `+` | `_plus` | `>` | `_great` | `~` | `_tilde` |
| `,` | `_comma` | | | | |

#### reserved words

C# allows reserved words to be used as valid identifiers if prefixed with an '@'[8].

[1] regex: `[_A-Za-z][_A-Za-z0-9]*`

#### types

Global identifiers need type information embedded in the name since the name alone does uniquely identify it (see thingy). Types can be recursive (see types), so the system for embedding types must be able to represent tree structures. We use the same syntax as the front-end but with `_S` as seperator, `_L` for the left angle bracket and `_R` for the right angle bracket.

| type | mangled |
|---|---|
| `array<int,3>` | `array_Lint_S3_R` |
| `list<list<int>>` | `list_Llist_Lint_R_R` |

## 3.6 Interpreter

The sixth research question lead to the implementation of an interpreter.

*How to validate the code generator?*

The interpreter was built to automaticly validate the codegen and later allow constant-folding as an compiler optimization.

The automatic validation would be done by comparing the results of test programs between the interpreter and the compiler. If they mismatch, there is either a bug in the interpreter or more likely a bug in the codegen.

#### Structure

The interpreter is structured in the simplest possible way to minimize the possibility of bugs.

At the heart of the interpreter is a function that evaluates a single instruction. This function is defined to be used in a fold.

Fold is a standard function in F# and other functional languages that behaves like an accumulator. example: fold (+) 0 [1 2 3 4] evaluates to 10. fold (*) 1 [1 2 3 4] evaluates to 24.

```
fold : (s->a->a) -> s -> [a] -> [a]
```

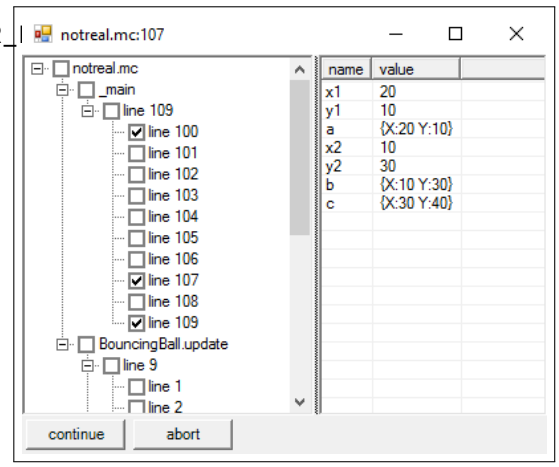
#### Evolution

Considered continuation monad. Turned out to complicated.

## 3.7 Debugger

The validation of the codegen lead to another validation-issue.

*How to validate the test programs?*




The backend can also embed an interactive debugger in the codegen. The program will then break on the first instruction and launch the debugger GUI. From the GUI, more breakpoints can be set with the check-boxes. When the user presses 'continue' or 'abort', the gui will close and appear again on the next breakpoint.

On the left pane is a 4-level deep tree which sorts the program on file name, function name, rule and line.

On the right pane shows a table with the name and value of the local identifiers defined up to the current breakpoint.

#### debug class

A seperate file, `_DBUG.cs` contains the class `_DBUG`. This class contains only the following public static items.

1. the tree representation of the program
2. a breakpoint table
3. a breakpoint function

#### Usage

The class also contains a `bool[][][][]`

The breakpoints are generated at each line of sourcecode in the rule. This is different than breaking at every instruction, as normalisation often splits single lines into multiple instructions.

Breakpoints are realised as an array of booleans for each rule in a closure.

```
class <function name>{
    <arguments>
    static bool[] _DBUG_breakpoints_0;
    static bool[] _DBUG_breakpoints_1;
    <return value> _run(<last argument>){
        <body>
    }
}
```

This was chosen because breakpoint checks happen every few instructions, so it has a big performance impact. Straight arrays with booleans are very fast to index since it only costs one addition and one dereference.

```
...
if(_DBUG_Breakpoints_1[6]){
    _DBUG.breakpoint("filename.mc", 12,
                     _DBUG_symbol_table);
}
...
```

The first two arguments to `_DBUG.breakpoint` are the filename and the linenumber. This is to uniquely identify the callsite. The third argument is the symboltable that has been accumulated so far.

After each assignment to a named local identifier, the named identifier and the value are recorded in a key-value collection. This key-value collection will be passed to the debugger when a breakpoint is hit.

#### Initialization

The `_DBUG` class is initialized in the main function. This is done to keep the program-specific code out of `_DBUG.cs`. The program tree is a public field of the `_DBUG` class, and is initialized by the main function.

# 4 Results

result: a working backend. test programs.

# 5 Conclusions

— summary here —

## 5.1 Recommendations

— propose optimization —

## 5.2 Reflection

It went well.

# A Glossary

boilerplate code

— difficult words here —

# References

[1] *LLVM official site*. `https://llvm.org`.

[2] Microsoft. *MSDN Platform Invoke Tutorial*. https://msdn.microsoft.com/en-us/library/aa288468(v=vs.71).aspx.
[3] Microsoft. *MSDN Performance Considerations for Interop (C++)*. https://msdn.microsoft.com/en-us/library/ky8kkddw.aspx.
[4] Alexander Köplinger. *Mono C++/CLI Documentation*. http://www.mono-project.com/docs/about-mono/languages/cplusplus/.
[5] Alexander Köplinger. *Mono Embedding Documentation*. http://www.mono-project.com/docs/advanced/embedding/.
[6] Microsoft. *MSDN .NET Framework 4 Hosting Interfaces*. https://msdn.microsoft.com/en-us/library/dd380851(v=vs.100).aspx.
[7] Leo A. Meyerovich and Ariel S. Rabkin. "Empirical Analysis of Programming Language Adoption". In: *SIGPLAN Not.* 48.10 (Oct. 2013), pp. 1–18. ISSN: 0362-1340. DOI: 10.1145/2544173.2509515. URL: http://doi.acm.org/10.1145/2544173.2509515.
[8] Microsoft. *MSDN 2.4.2: Identifiers*. https://msdn.microsoft.com/en-us/library/aa664670.aspx.