Bachelor Thesis

Developing the back-end of the Meta-Casanova 3 Compiler

Douwe van Gijn

supervised by
Dr. Giuseppe Maggiore

2016-05-24

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.

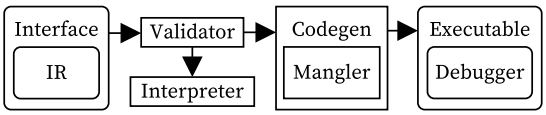
In this document, we will walk through the backend and examine the various parts and their design decisions. In this way, this document aims to be useful to the future developers of the MC compiler.

1.1 Goals

The main goal was to develop a backend so typechecked Meta-Casanova(MC) could be transformed into executable code. In order to do this, several sub-goals were set.

1. decide on the output language.
2. design a front-end interface and an intermediate representation(IR).
3. Implement a program to validate the front-end interface and IR.
4. Implement a Code generator with code mangler.
5. Implement an interpreter to validate the code generator.
6. Implement an embedded debugger to help debug the compiler.

To illustrate how the goals 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 goed 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.

1.2 Requirements

The project had 3 requiremens.

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 faster was better.

The first requirement 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 inter-operability with Unity game engine.

1.3 Organization

— todo: describe organization —

2 Output language

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

In what programming language does 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.

2.1 Unmanaged

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.

Another advantage of unmanaged code is that the fast LLVM C/C++ code generator can be used. — explanation LLVM here — This would mean we get all the optimisations of LLVM with very little effort.

.NET compatibility is also required. 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 (P/invoke) allows managed code to call unmanaged functions that are implemented in a DLL.[1]

large overhead because of marshalling [2]

C++/CLI interop

C++ for the Common Language Infrastructure (C++/CLI) is a programming language designed for interoperability with unmanaged code.

compiler windows-only[3]

not typesafe(not allowed in safe-mode) pure CIL is windows only[3]

only on x86[3]

Hosted runtime

large overhead, no fast jit.

conclusion

none are good enough.

2.2 .NET

Because of the problems go with .NET. A big advantage is stability because everything happends inside the .NET runtime. This has a higher chance of working on non-native platforms than the hybrid unmanaged solutions.

F#

F# would be a natural choice, since the compiler is written in it. However, it is quite slow[source] and resists the imperative style of the generated code.

C#

C# has more advantages. It is the most popular .NET language, and the compiler gets the most attention by Microsoft. It is also easy to debug, as it has the most mature debug tools.

CIL

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] So it makes debugging and verification harder, with little to no gain.

2.3 Conclusion

The debugability together with a lot of controll make C# the best choice in this case.

3 Meta-Casanova

Meta-Casanova is a functional, declarative language. It allows for multiple implementations of functions called *rules*. Rules may fail and the program will continue with the successful ones.

Multiple rules may match at any given time. If this happens, the program execution splits into multiple branches; each branch following one rule. If none of the rules match, the branch dies off.

This mechanism effectively implements lookahead-behavior in programs, and is therefore useful for writing parsers.

3.1 Data

`Data` declarations declare a two-way many-to-1 relation between types. This two-way relationships makes `Data` an *alias*.

```
Data "nil" → list 'a
Data 'a → "::" → list '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.

3.2 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.

3.3 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.

3.4 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`.

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+0 → 1→
 2
```

After which it tells us correctly that the length of the list 1::(2::nil) is indeed 2. TODO: metacasanova type syntax TODO: metacasanova no longer matches multiple statements

4 The front-end interface

The front-end interface is the interface between the front-end and the back-end. All the inputs for the back-end are in this datastructure.

Primarily the lambda and function defenitions, and the data declarations.

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

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, as no lookups by identifier are performed.

4.1 Data declarations

The data declarations are grouped with the identifier of —

```
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 -> int * string
Data "fst" -> int    -> int | string
Data "snd" -> string -> int | string
```

This will appear as the following list in the interface:

identifier	arguments	type
","	int; string	int * string
"fst"	int	int | string
"snd"	string	int | string

4.2 Rule containers

Function and lambda definitions, aswell 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.

4.3 Rules

Rules rules rules.

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

input and output. premises contain instructions, explained in next section. typemap. debug info.

4.4 Validator

The first versions of the backend had no working front-end to test with. So the 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 always checks the interface before it is handed to the codegen.

5 Intermediate Representation

Each rule contains a list of Premises. These premises are normalized to one or more instructions. These instructions are still valid MC.

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

5.1 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. 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.

5.2 .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`

5.3 Evolution

It was briefly considered to have the CIL code be the intermediate-representation, but 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. There would be not difference in the type of the uncalled closure and the called closure. This made type-analysis of the program nearly impossible, so it was decided that call also applies the last argument.

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.

6 Code generator

The codegen is the heart of the back end. It is responsible for generating the C# code.

6.1 Function declarations

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>;
    }
};
```

6.2 Data declarations

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

It is easy to see the pattern with an example.

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

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

Transforms into this.

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

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

6.3 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:
```

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;`
		`if(_tmp0==null){goto _skip0;}`
		`var x  = _tmp0._arg0;`
		`var xs = _tmp0._arg1;`

closure	`(+) -> add`	`var add = new _plus();`
application	`add a -> inc`	`var inc = add;` `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;`

6.4 Evolution

C# unions only work with value-types.

7 Mangler

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.

7.1 C# identifiers

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

Valid C# identifiers are `[_A-Za-z][_A-Za-z0-9]*` [4]. The only valid non-alphanumeric character is an underscore, so that is used to escape with.

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`				

7.2 reserved words

C# allows reserved words to be used as valid identifiers if prefixed with an '@'[4].

7.3 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`

8 Interpreter

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.

8.1 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.

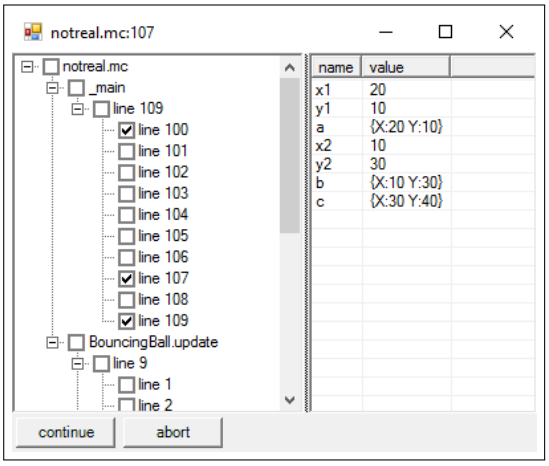
Fold is a standard function in F# and other functional languages that behaves like an accumulator. example: fold (+) 0 [1 2 3 4] evaluates to 10. fold (*) 1 [1 2 3 4] evaluates to 24.

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

8.2 Evolution

Considered continuation monad. Turned out to complicated.

9 Debugger



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.

9.1 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

9.2 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.

9.3 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.

10 Conclusion(TODO)

10.1 Reflection

A Glossary

boilerplate code

— difficult words here —

References

[1] Microsoft. *MSDN Platform Invoke Tutorial*. https://msdn.microsoft.com/en-us/library/aa288468(v=vs.71).aspx.

[2] Microsoft. *MSDN Performance Considerations for Interop (C++)*. https://msdn.microsoft.com/en-us/library/ky8kkddw.aspx.

[3] Alexander Köplinger. *Mono C++/CLI Documentation*. http://www.mono-project.com/docs/about-mono/languages/cplusplus/.

[4] Microsoft. *MSDN 2.4.2: Identifiers*. https://msdn.microsoft.com/en-us/library/aa664670.aspx.