Bachelor Thesis

Design and implementation of the Meta Casanova 3 compiler back-end

Douwe van Gijn

supervised by
Dr. Giuseppe Maggiore

2016-06-06

Contents

Abstract

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 program representation from the front-end. In this thesis, we will walk through the back-end 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 Introduction

Games are complex programs that have to do a lot of things in a small timespan. To make writing games easier, a new language was developed: Casanova.

Implementing the Casanova compiler proved difficult. 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 1480 lines long, and became unmaintainable. After a rewrite in MC it was 300 lines [1].

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

Structure

We will first discuss the context of the assignment in section 2. Then we will give a short overview of Meta Casanova in section 3.

Section 4, the main part of is thesis, is next. It presents the main research question and splits it

[1] in `Sources/MarkIII/InterpreterBasicMetacasanova/` at `https://github.com/vs-team/metacompiler`

in sub-questions. Each sub-question is then answered in each subsection.

Section 5 presents the evidence that the requirements of the main research question have been met. This is followed by conclusions in section 6 that summarize the results. After the thesis proper, we give recommendations for the future development of the back-end in section 7. Section 8 is the last part of the thesis, and shows that the Dublin descriptors have been met.

The appendices contain the contact details of the stakeholders and a glossary. The source code is available in online[1].

2 Context

The graduation assignment is carried out at Kenniscentrum Creating 010. *Kenniscentrum Creating 010 is a transdisciplinary design-inclusive Research Center enabling citizens, students and creative industry making the future of Rotterdam* [2].

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

2.1 Research group

The research group is creating the Casanova language. The members of the research group are Francesco di Giacomo[2], Mohamed Abbadi[2], Agostino Cortesi[2], Giuseppe Maggiore[3] and Pieter Spronck[4].

Within the research group is our research team, tasked with the design and implementation of Meta Casanova. The research team is supervised by Giuseppe Maggiore and comprises of three students. Louis van der Burg, responsible for developing the Meta Casanova language, Jarno Holstein, responsible for the front-end of the Meta Casanova compiler, and Douwe van Gijn, responsible for the back-end of the Meta Casanova compiler.

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

2.2 Motive

Kenniscentrum is interested in innovative technologies. Innovative technologies like virtual reality and video games are the fields our research group is researching.

In order to ease the development of virtual reality and video games, the Casanova language was developed. The Casanova language is the subject of the PhD thesis of Francesco.

The complex nature of the Casanova language lead to a complex compiler. To simplify the development of Casanova, the language Meta Casanova was developed.

3 Meta Casanova

It is necessary to understand a subset of Meta Casanova (MC) in order to understand the problem-space of the back-end. Meta Casanova is a functional, declarative language. This section will cover the subset of the language that is relevant for code generation.

3.1 Data

`Data` declarations declare a discriminated union [**algebraic_datastructures**]. For example, we could define an inductive list as:

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

Which 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 a list can be deconstructed in two ways. The programmer will assert which deconstructor is expected, and the

rule does not match if the deconstructor does not match. An example of this is shown later.

Additionally, constructors may be manipulated and partially applied like functions. This allows for greater flexibility, requiring only that function and constructor names be unique in their namespace.

3.2 Functions

Function 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.

3.3 Rules

Meta Casanova uses a syntax similar to that of natural deduction. It allows for multiple implementations of functions. These implementations are called *rules*.

Rules can fail to match. If that happens, the rule will jump ahead to the next rule. This this will continue until a rule succeeds, or no rule matches in which case the program throws a runtime exception.

```
---------------
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 `length xs -> res` 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
2
```

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

3.4 Main

Each program needs an entry point. The entry point of an MC program is the main function.

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

The results of the main function are printed on the console. The previous program would therefore print 2 on the screen.

4 Research

The primary research question of this thesis is:

How to implement a transformation from type-checked Meta Casanova (MC) from the front-end, to executable code within the timeframe of the internship?

Where the transformation must satisfy these requirements:

The correctness requirement: The back-end must in no case produce an incorrect program.

The .NET requirement: The executable must be able to inter-operate with .NET.

The multiplatform requirement: The generated code must run on all the platforms .NET runs on.

The performance requirement: The performance of the generated program should be better than Python.

The correctness requirement exists because the compiler must be reliable. Any program can at most be as reliable as the compiler used to generate it. The .NET requirement exists because of the need for a large library and inter-operability with Unity game engine. This is because the main area of research of the organization is game-related[5]. The multiplatform requirement is because the games are produced for any platform. The performance requirement is there because games have to be fast.

In order to answer the research question, seven sub-questions were formulated.

The language question: In what language should the code generator produce its output?

The interface question: What should the interface be between the front-end and the back-end?

The IR question: What should the intermediate representation of the functions be?

The codegen question: How does the interface map to the output language?

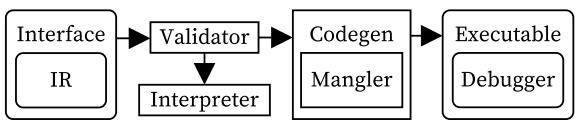
The mangle question: How to generate names so that they comply with the output language?

The validation question: How to validate the code-generator?

The debug question: How to validate the test programs?

Each answer of a sub-question is provided evidence by implementing a part of the back-end. This will in turn provide evidence to answer the main research question.

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



[5] see section 2.2

As you can see, the front-end interface contains the Intermediate Representation (IR) and goes through the validator. From there, depending on the compiler flags, it either goes to the interpreter or the code generator. In case it goes to the interpreter, the program is directly executed. In case it goes to the code generator, 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.

4.1 Choice of 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." [3] 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 4. 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 [4].

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. Transferring data between .NET and native code has a high performance cost [5], 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 [6]. Besides that, non-type-safe operations (the main advantage of C++/CLI) are only allowed on windows [6].

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 [7]. The same large serialization overhead as P/Invoke is present [8].

.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 [**fsharp**]. It would be a natural choice, since the compiler is written in it. It has the advantage of supporting tail-calls, which is unsupported by C#, and since F# already supported algebraic datatypes, it seemed like a viable solution.

C# is an imperative, object-oriented language [**csharp**]. It is the most popular .NET language [9], so the compiler gets the most attention by Microsoft. It is also easy to debug, as it has the most mature debugging tools. C# too seemed like a viable option.

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# [10]. As a result, it makes debugging and verification harder, with little to no gain. It also omits the optimizations of the C# compiler, such as dead-code elimination and stuff[6].

Conclusion

The result of the research was that C# and F# were both viable. To choose an output language, I built a code model for each language.

The F# code model mainly involved switching off indentation-based scoping for F# and using the verbose syntax. Scoping was implemented by making each rule a numbered function that returned an `Option<>`. the numbered rules were then called and appended using the `>>=` of the `Option` monad.

While this model worked, it was cumbersome and slow. The code was hard to inspect, since there was no indentation. It was also relatively slow because[7]:

[6] see section 7

[7] information obtained by inspecting generated CIL code of the microsoft F# compiler `fsc.exe` version 1.0 with `+optimize`

1. It had to wrap each return value in a `Option`. This is particularly costly for value-types, as they have to be boxed and unboxed each time the value is used.
2. It performs monadic operations for each rule attempt. These generic functions do type resolution at run-time, each time they are called.
3. The numbered functions were not inlined, preventing any cross-rule optimization.

The C# code model proved far easier to generate and inspect, as it had braces and local scopes. It is the model that is now used[8].

4.2 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 back-end. This makes testing very easy, as the rest of the back-end only relies on its input.

Interface

The interface is all contained in a single data structure.

```
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 identifier 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.

We can illustrate this by defining a tuple and 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
```

[8] see section 4.4

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 is its premises. These are the instructions that make up the rule. The instruction set is described in section 4.3.

The premise list also contains line numbers for each premise. This is debug information, that is used by the embedded debugger[9].

Next are the inputs and output of the rule. Input and outputs consist only of local identifiers. This is because of *normalization*[10].

In the case that a rule-input or output has an expression instead of a local identifier, the expression is assigned to a new local identifier and the local identifier is substituted.

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 type checker has accumulated.

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

Validator

The first versions of the back-end had no working front-end to test with. Early testing was done by writing the interface data structure 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 code generator.

Evolution

The front end interface went through a lot of iterations, often to simplify and sometimes to add features.

The biggest simplification of the interface was the decision to stop using recursive data structures. Recursive data structures such as trees are more difficult to traverse and modify than lists. The interface used to be defined in by a list of *Scopes*. Each scope would have a list of functions, data declarations and lambdas. The scope would also have a name and a list of scopes that were beneath the current scope in the hierarchy. In this way, it formed a tree of scopes that represented the program structure.

```
type Scope = {
  Name          : String
  Children      : List<Id*Scope>
  FuncDecls     : Map<Id,SymbolDeclaration*
    Type>
  TypeFuncDecls : Map<Id,SymbolDeclaration*
    Type>
  DataDecls     : Map<Id,SymbolDeclaration*
    Type>
  TypeFuncRules : Map<Id,List<Rule>>
  FuncRules     : Map<Id,List<Rule>>
}
```

This evolved to a `Map<List<Id>,scope>`, transferring the nesting of the scope to a list describing the address. Eventually, the contents of the scope were given a global identifier, got put

[9] see section 4.7
[10] see section 4.3

in a single data structure, and was renamed to be the interface we have now.

4.3 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[11]. 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 extensions.

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 [11]. 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. These 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 does not match, the rule does not match 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 useful. The exception is data-constructors. They do not have to be called as they insert their elements in the data structure as they are applied.

.NET extensions

A separate set of instructions are needed to interoperate 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

The IR changed a lot during development. Each iteration it got simpler.

Existing IR 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 code generator to generate the CIL instructions.

[11] see section 3.3

Call 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 Application used to also take the position of the argument that was applied. This was because the back-end 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 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.

4.4 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 code generator 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 jump ahead to the label when they do not match. More on rules in subsection *rules*.

Data declarations

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

This is a pretty straight-forward transformation.

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

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

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

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

class Union{}
class Left :Union {string _arg0;}
class Right:Union {float  _arg0;}
```

The types `Tuple` and `Union` can now be easily be deconstructed. When a premise 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 reached by the code generator, 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)
1 -> _tmp4            (literal)
tmp4 = tmp0           (conditional)
"odd!" -> _tmp5       (literal)
--------------------
evenOrOdd a -> _tmp5


(%) -> _tmp0          (closure)
_tmp0 b -> _tmp1      (application)
2 -> _tmp2            (literal)
_tmp1 _tmp2 -> _tmp3 (call)
0 -> _tmp4            (literal)
tmp4 = tmp0           (conditional)
"even!" -> _tmp5      (literal)
--------------------
evenOrOdd b -> _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;
    <omitted for brevity>
    return _tmp5;
}
_skip1:
```

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

Main function

The main function is the entry point of the program. When the program is run, the main function is called and the result is printed.

The body of the rule is generated like any rule in its own function. This improves readability by separating the main function specific code from the general rule code.

```
class _main{
    static <return type> _body(){
        <main rule>
    }
    static void Main(){
        <debugger initialization>
        System.Console.WriteLine(
            System.String.Format("{0}",
                                 body()));
    }
}
```

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;`

Evolution

The code generator changed little because the code model was developed in the early stages to research if C# was viable. There were only two instances where the code generator changed during development.

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. By overlapping fields in memory, we could achieve the same effect as C union. 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 other structs and *simple types* like integers, floats, and booleans. This was a problem since most of the .Net objects are classes, and only a few of the .Net objects are value-types.

The second change was a simplification that made it much easier to have breakpoints for the embedded debugger. The code generator used to produce a nested structure before we used `goto`. The if-statements were nested for the rest of the rule, for example:

```
if(x>10){
    var foo = 20;
    if(x<100){
        return foo;
    }
}
```

It took a lot more work to traverse this tree-like structure in order to insert breakpoints at the appropriate places, particularly because these breakpoints added a another if-statement and thus another layer of nesting. The size of the code shrunk a lot when the decision was made to use `goto`s, and the code became a lot more readable.

4.5 Mangler

C# has far more limited set of identifiers than MC. Still, each MC identifier must map to a valid C# identifier.

This leads to our fifth sub-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 MC identifier. The mangler is designed to be simple, and produces readable output. Readable output allows reflection and makes it easy to use MC functions and data structures from C#. Readability also eases the verification of 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[12] [12]. 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 '`@`' [12].

Types

Global identifiers need type information embedded in the name since the name alone does uniquely identify it. Types can be recursive[13], 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 separator, `_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`

[12] regex: `[_A-Za-z][_A-Za-z0-9]*`
[13] see section 3.1
[14] see section 4.7

Evolution

The first iteration of the mangler just numbered every identifier. While this was a simple system to generate identifiers with, it was absolutely impossible to inspect the resulting code. Most of the mangler is the result of a desire for readable, inspectable output code.

4.6 Interpreter

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

How to validate the code generator?

The interpreter was built to automatically validate the code generator and later allow constant-folding as a 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 code generator.

Evolution

The first design for an interpreter used the continuation monad. This is a complex construct that allows for arbitrary control flow.

The idea was that during debugging, you could change the line that was executed. It turned out that it was more desirable to have the debugger in the code generator instead of the interpreter[14], so the primary benefit of the construct was lost.

The next design used explicit recursion to walk the list of instructions. This was a huge simplification compared to the continuation-monad, but every instruction still had to explicitly recurse over the instruction list. While all of the recursive calls were tail-calls[**tailcalls**], it still meant near-identical code duplication for each instruction.

Structure

The final design uses `fold`, a specialization of a catamorphism for lists[**catamorphism**]. This eliminated the recursion, making `interpret` a straight-line function that executed a single instruction. This interpreter was written in under 100 lines[15].

`fold` (or `reduce`) is a standard function in F# and other functional languages with the following type signature:

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

It applies a function for each element that takes the element and accumulator and produces a new accumulator. The first argument is that function, the second argument is the starting state and the last is the array. [**realworldhaskellch4**].

For example: `fold (+) 0 [1 2 3 4]` evaluates to 10 and `fold (*) 1 [1 2 3 4]` evaluates to 24.

Using a fold radically simplifies the function, as all the explicit recursion becomes implicit. The function now only takes the state of the program and an instruction, and produces the new state of the program.

.NET instructions

The interpreter has to be able to load .NET libraries on the fly, since the libraries are not known at the time the compiler is compiled.

In the front-end interface, the `assemblies` field contains a list of strings. These strings are the assembly names the program is linked to. When a .NET function is called, the interpreter will open the assemblies one by one and search through it for a function that matches the name and signature of the one called. .NET data structures and fields are handled the same way.

4.7 Debugger

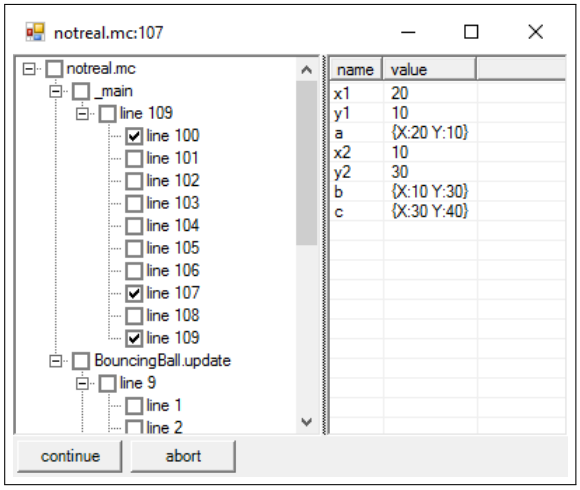
The validation of the code generator lead to another validation-issue. If a test program is not behaving as expected, is there a bug in the test program or in the compiler?

[15] see `interpreter.fs`

In other words:

How to validate the test programs?

The answer to this is an embedded debugger in the target executable.



The program will then trigger a breakpoint 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.

The left pane shows a four level deep tree which sorts the program on file name, function name, rule and line.

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

Program changes

When compiling with the debug flag set, some additions are made to target program.

Local identifier table After each instruction that defines named local identifiers, a new instruction is generated.

```
var foo = 42;
_dbug_symbol_table["foo"] = foo;
```

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 break-

point is hit. A new key-value collection is defined at the start of each rule.

Break points When compiling with the debug-flag set, function closures will have a group of static boolean arrays. One array for each rule in the function.

```
class <function name>{
    <arguments>
    static bool[] _dbug_breakpoints_0;
    static bool[] _dbug_breakpoints_1;
    <return value> _run(<last argument>){
        <body>
    }
}
```

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

```
...
if(_dbug_Breakpoints_1[6]){
    _dbug.breakpoint("filename.mc", 12,
                     _dbug_symbol_table);
}
...
```

Debug struct

The breakpoint function is defined as a public static member of the debug struct.

The debugger is defined in a separate file, `_dbug.cs`, which is imported by the target executable. This is done to keep the program-specific code out of `_dbug.cs`.

`_dbug.cs` contains the struct `_dbug`. This struct contains only the following public static items.

1. the program tree
2. the breakpoint tree
3. the breakpoint function

The program builds up the program tree and the breakpoint tree in the main function, before the first user-written line starts. The trees are both four-levels deep and sorted on filename, function name, rule and line number. The breakpoint function is called when the program hits a breakpoint.

This was chosen because breakpoint checks happen every few instructions, so they have a huge effect on debug performance. Straight arrays with booleans are very fast to index since it only costs one bounds-check, one addition and one dereference. The dereference can even be done speculatively, due to branch prediction [**branchprediction**].

The tree representation of the program is four levels deep. The first level represents the file, the second level represents the function, the third level represents the rule and the fourth level represents the premise. This tree representation is initialized in the main function, before the user code begins.

The breakpoint table Each closure has its group of breakpoints. To easily index all the breakpoint arrays from one central point, the `_dbug` struct has a breakpoint table. The breakpoint table is a static four-dimensional array (`bool[][][][]`) that points to the static breakpoint arrays in the closures. This way, the arrays are quick to index from the closure, while being available in a tree-like from for the debugger.

The breakpoint function The `_dbug` struct contains a static method `breakpoint` that will pause the execution of the program and present the GUI. When the user presses 'continue' or 'abort', the GUI will close and the `breakpoint` method will return control back to the program.

The first two arguments to `_dbug.breakpoint` are the filename and the line number to uniquely identify the call site. The third argument is the symbol table that has been accumulated so far.

Evolution

The debugger has changed little, since it is a relatively simple construct. The only significant changes were the decision to go from a single boolean array per closure to one per rule. The book-keeping involved in having the boolean arrays packed was inelegant, because there was now a dependency between the rules: the latter breakpoint number depends on the breakpoints in the former rule.

5 Results

After answering all the sub-questions, one thing is left: proving the main research question. The main research question and its requirements were:

How to implement a transformation from type-checked Meta Casanova (MC) from the front-end, to executable code within the timeframe of the internship?

Where the transformation must satisfy these requirements:

The correctness requirement: The back-end must in no case produce an incorrect program.

The .NET requirement: The executable must be able to inter-operate with .NET.

The multiplatform requirement: The generated code must run on all the platforms .NET runs on.

The performance requirement: The performance of the generated program should be better than Python.

5.1 Correctness

Unfortunately, since the front-end was incomplete, it is not possible to compile source files. It *is* however possible to write the front-end interface by hand.

Data test

The first test was developed to test the Data declarations. It is equivalent to the following MC code.

```
Data int -> "::" -> List -> List
Data "nil" -> List

---------
main -> 0
```

List length test

The list length program defined a list data structure and a program to compute its length. This was used since it uses each basic instruction at least once, as well as matching. It is equivalent to the following MC code:

```
Data int -> "::" -> List -> List
Data "nil" -> List

Func "length" -> List -> int

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

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

--------------------------------
main -> length (1::2::3::4::nil)
```

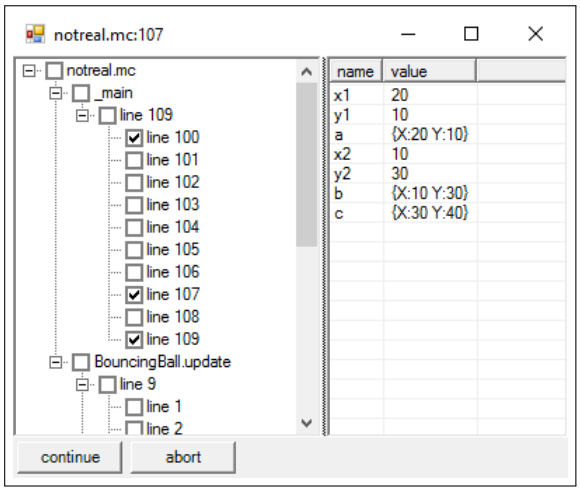
Which when executed prints 4 on screen.

5.2 .NET

The second test program was to test the .NET functionality. It consists of a simple program that modifies XNA data structures, specifically the `Vector2`.

```
20.0 -> x1
10.0 -> y1
Microsoft.Xna.Framework.Vector2 x1 y1 -> a
20.0 -> x2
10.0 -> y2
Microsoft.Xna.Framework.Vector2 x2 y2 -> b
Microsoft.Xna.Framework.Vector2.+ a b -> c
Microsoft.Xna.Framework.Vector2.Normalize c
y2 -> v.x
v.x -> ret
-----------------------------------------
run -> ret
```

Which returns `20`. This is especially interesting to debug, because we can see the values change.



5.3 Multiplatform

All test programs via the Microsoft .NET runtime on Windows, and on mono everywhere else. Because no platform-specific code is used, the code runs on all platforms where mono is suported[13].

Currently supported operating systems are:

- Linux
- Mac OS X, iOS, tvOS, watchOS
- Sun Solaris
- BSD - OpenBSD, FreeBSD, NetBSD
- Microsoft Windows
- Nintendo Wii
- Sony PlayStation 3
- Sony PlayStation 4

5.4 Performance

Performance was mesured with the *list length* test program. A Python version was written to match the MC version as closely as possible:

```
Nil = None

def cons(x,xs):
    return (x,xs)

def length(xs):
    if xs is Nil:
        return 0
    else:
        return 1+length(xs[1])
```

Optimizations were turned off on both the C# compiler and python, to make sure the optimizer did not take advantage of optimizations that were only applicable in this case. This decision makes the results reflect an average program, as recursive calls are often used in MC.

The Python program counted the length of a 10^4 element list. To get the same precision, the MC program counted the length of a 10^6 element list. Both programs were executed 10^3 times, the results below is the minimum, maximum and average time taken to count a single element.

	Python	MC	Python/MC
min	3442μs	22.82μs	151
max	3627μs	49.79μs	72.8
avg	3534μs	36.30μs	97.3

The mesurements show that MC is two orders of magnitude faster than the equivalent python code. This illustrates the strength of the MC language, the user-defined data structures are very fast.

Even if we compare it to the built-in python `List.count`, the naïvely implemented list outperforms it.

```
lst = [1]*1000000

length=lst.count(1)
```

	Python	MC	Python/MC
min	39.00μs	22.82μs	1.71
max	52.86μs	49.79μs	1.06
avg	41.95μs	36.30μs	1.16

This shows that the performance requirement has been met.

6 Conclusions

The result is a working, reliable, performant back-end, with interpreter, validator and embedded debugger. All finished within the time frame of the internship.

To prove it works, three test programs were written. All are correctly generated, all run in the interpreter, and all can be debugged.

7 Further work

The back-end is feature complete. Because the performance requirement was already met, optimizations were not necessary.

Inlining

The most important optimization is inlining [**inlining**]. Inlining is the process of replacing a function call with the function body. While this saves a function call, the greatest benefit is that it enables other optimizations.

When the function body is copied in the larger context of the call site, some input values may be identified as compile-time constants, enabling a whole array of optimizations.

Inlining is not always desirable. If a large function is called from many different places, the size of the program increases, increasing the cache-misses on the instruction cache, reducing performance.

The choice for inlining a call should consider:

1. If the function recurses in any way, even indirect recursion. This makes inlining impossible.
2. The size of the function. The smaller the function, the greater the inlining benefit.
3. The amount of times the function is called. If the function is only called once, inlining has no disadvantages. For each additional time, the size of the program increases.

Tail call optimization

Some recursive functions can be transformed into loops. This has the advantage that no new stack frames will be allocated, preventing stack-overflows and increasing performance.

If the function returns right after the recursive call, there is no need to save the state of the function, since it will be thrown away right after the call returns. In these cases, it is safe to replace the recursion with a modification of the input arguments and a jump to the top of the function.

These constructs can be implemented in C# using `goto`. Alternatively, the `tail call` CIL instruction can be generated.

Constant folding

Constant folding is done when an expression involving constants can be computed to another constant. For example the expression `3+5` has only constants in it and can on compile-time be substituted with `8`.

C# does constant folding in very limited conditions. The C# language specification[16] states that this is only done on *simple type constants*. Simple types are types like `int`, `float`, `bool`, `byte` and the like. Simple types do not include any compound types like structs or classes. The constant expression can also only be with operators defined by the simple types. No user-defined function can therefore be constant folded.

The MC compiler could constant fold a lot more. Everything in a rule that is not related to its inputs can be constant folded.

[16] second bullet point of section 11.1.4, page 110 of [14]

8 Evaluation

In this section I will show that I have the competences associated with computer science according to Rotterdam University of applied sciences.

Administering

Analyzing

I have split the back-end problem up in managable chunks.

Advising

This thesis also serves as a documentation of the back-end.

Designing

The parts of the back-end are modular and communicate with each other through well-defined interfaces. This made it easy to respond to changes in the language, as changes in one part have no effect on the other parts.

Realizing

I managed to to realize a working compiler back-end within the allocated time.

References

[1] Francesco Di Giacomo, Pieter Spronck, and Giuseppe Maggiore. “Building game scripting DSL’s with the Metacasanova metacompiler”. In: 2015.

[2] *Creating 010 - Kenniscentrum Creating 010*. `http://creating010.com/en/`. Retrieved: 2016-03-18.

[3] *LLVM official site*. https://llvm.org. Retrieved: 2016-06-01.

[4] Microsoft. *MSDN Platform Invoke Tutorial*. https://msdn.microsoft.com/en-us/library/aa288468(v=vs.71).aspx. Retrieved: 2016-06-01.

[5] Microsoft. *MSDN Performance Considerations for Interop (C++)*. https://msdn.microsoft.com/en-us/library/ky8kkddw.aspx. Retrieved: 2016-06-01.

[6] Alexander Köplinger. *Mono C++/CLI Documentation*. http://www.mono-project.com/docs/about-mono/languages/cplusplus/. Retrieved: 2016-06-01.

[7] Alexander Köplinger. *Mono Embedding Documentation*. http://www.mono-project.com/docs/advanced/embedding/. Retrieved: 2016-06-01.

[8] Microsoft. *MSDN .NET Framework 4 Hosting Interfaces*. https://msdn.microsoft.com/en-us/library/dd380851(v=vs.100).aspx. Retrieved: 2016-06-01.

[9] 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.

[10] ECMA International. *ECMA-335: Common Language Infrastructure (CLI)*. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf. June 2012.

[11] Peter Lee, Frank Pfenning, and André Platzer. "Static Single Assignment". In: ().

[12] Microsoft. *MSDN 2.4.2: Identifiers*. https://msdn.microsoft.com/en-us/library/aa664670.aspx. Retrieved: 2016-06-01.

[13] Alexander Köplinger. *Mono Platform Documentation*. http://www.mono-project.com/docs/about-mono/supported-platforms/. Retrieved: 2016-06-07.

[14] ECMA International. *ECMA-334: C# Language Specification*. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf. June 2006.

A Glossary

boilerplate code expl

polymorphic can have multiple types

B Benchmarks

B.1 list.cs

```
class List { };
class Cons : List { public int _arg0; public List _arg1;}
class Nil  : List {}

class length {
  public int _run(List _arg0) {
    {
      var lst = _arg0;

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

      var _tmp1 = new length();

      var len = _tmp1._run(xs);

      var res = len + 1;

      return res;
    }
_skip0:
    {
      var lst = _arg0;

      var _tmp0 = lst as Nil;
      if (_tmp0 == null) { goto _skip1; }

      var res = 0;

      return res;
    }
_skip1:
    throw new System.Exception();
  }
}

class _main {
  static List makelist(int x, List l) {
    if (x == 0) {
      return l;
    } else {
      Cons c = new Cons();
      c._arg0 = x;
      c._arg1 = l;
      return makelist(x - 1, c);
    }
  }
  static void Main(string[] args) {
    var list = makelist(100000, new Nil());
    for (int i = 0; i < 1000; i++) {
      var stopwatch = new System.Diagnostics.Stopwatch();
      var l = new length();
      stopwatch.Start();
      var result = l._run(list);
      stopwatch.Stop();
      System.Console.WriteLine(stopwatch.Elapsed);
    }
  }
}
```

B.2 list.py

```
import sys
sys.setrecursionlimit(10003)
import time

Nil = None

def cons(x,xs):
    return (x,xs)

def length(xs):
    if xs is Nil:
        return 0
    else:
        return 1+length(xs[1])

def makelist(i,xs):
    if(i==0):
        return xs
    else:
        return makelist(i-1,cons(i,xs))

lst = makelist(10000,Nil)
for i in range(0,1000):
    start = time.clock()
    val = length(lst)
    end = time.clock()
    print(end-start)
```

B.3 native.py

```
import time

lst = [1]*1000000

for i in range(0,1000):
    start=time.clock()
    length=lst.count(1)
    end=time.clock()
    print(end-start)
```