Bachelor Thesis

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

*Douwe van Gijn*

supervised by
Dr. Giuseppe Maggiore

2016-05-31

# DRAFT - DO NOT GRADE

## 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 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.

### 1.1 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 inter-operability 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 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.

### 2.3 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
```

2

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 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.
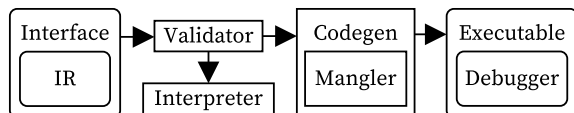


As you can see, the front-end interface contains the IR and goes through the validator. From there, depending on the compiler flags, it either goes to the interpreter or the codegen. In case it goes to the interpreter, the program is directly executed. In case it goes to the codegen, it is translated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

3

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. 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[7], 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#.[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 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[1]

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 perform 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[2].

---

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

[2]see section 3.4

## 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 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.

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[3].

Next are the inputs and output of the rule. Input and outputs consist only of local identifiers. This is because of *normalization*[4].

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.

The typemap contains a map from the local identifiers in a rule to their types. This gives the backend 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 always checks the interface before it is handed to the codegen.

---

[3]see section 3.7
[4]see section 3.3
[5]see section 2.3

## 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[5]. 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.

6

**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 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

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 ahead in subsection *rules*.

**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:
```

8

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[6][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.

---

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

| ! | _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].

**Types**

Global identifiers need type information embedded in the name since the name alone does uniquely identify it. Types can be recursive[7], 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 |

**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, inspectible output code.

### 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.

---

[7]see section 2.1
[8]see section 3.7
[9]see interpreter.fs

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.

**Evolution**

The first design for an interpreter used the continuation monad. This is a complex construct that allowes 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 desireable to have the debugger in the codegen instead of the interpreter[8], 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. While all of the recursion were tail-calls, it still meant near-identical code duplication for each instruction.

**Structure**

The final design uses fold. This eliminated the recursion, making the interpret instruction a straight-line function that executed a single instruction. This interpreter was writen under 100 lines[9].

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**].

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 datastructures and fields are handled the same way.
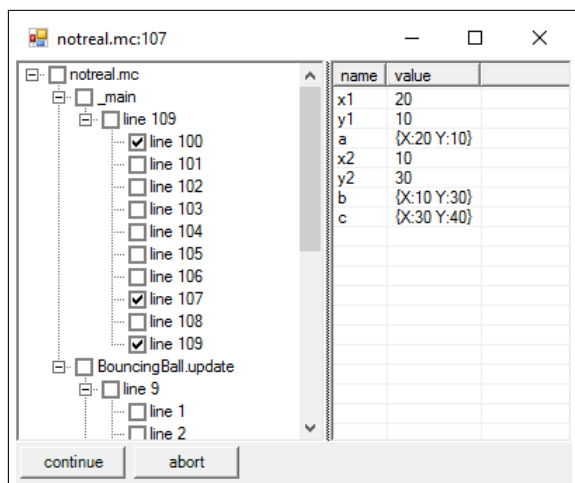
## 3.7  Debugger

The validation of the codegen 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?

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 breakpoint 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 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 sourcecode in the rule. This is different than breaking at every instruction, as normalization often splits single lines into multiple instructions.

11

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

**Debug class**

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

The debugger is defined in a seperate 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 class _DBUG. This class 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 linenumber. 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 performace. Straight arrays with booleans are very fast to index since it only costs one bounds-check, one addition and one dereference.

**The tree representation** of the program is four levels deep. The first level represents the file, the second level represents the funcion, the third level represents the rule and the fourth level represents the premise.

**The breakpoint table** Breakpoints are realised as an array of booleans for each rule in a closure. The class also contains a `bool[][][][]`

**The breakpoint function** This class 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 linenumber. This is to uniquely identify the callsite. The third argument is the symboltable that has been accumulated so far.

**Initialization**

The program tree is a public field of the _DBUG class, and is initialized by the main function.

**Evolution**

First own tree impl. Then 4d node. Now winform tree nodes.

# 4   Results

The result is a working, reliable, performant back-end, with interpreter, validator and embedded debugger.

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

## 4.1   Test programs

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 first program defined a list datastructure 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

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

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

Which when executed prints the following on screen.

```
4
```

This program can also be debugged with the embedded debugger.

**XNA test**

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

## 5  Conclusions

— summary here —

### 5.1  Recommendations

— propose optimization —

### 5.2  Reflection

It went well.

## 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.

## A  Glossary

boilerplate code

— difficult words here —

## B    Source

### B.1    Common.fs

```
1   module Common
2
3   type Position = { File : string; Line : int; Col : int }
4     with
5       member pos.NextLine = { pos with Line = pos.Line + 1; Col = 1 }
6       member pos.NextChar = { pos with Col = pos.Col + 1 }
7       static member FromPath(path:string) = { File = path; Line = 1; Col = 1 }
8       static member Zero = { File = ""; Line = 1; Col = 1 }
9
10  type Bracket = Curly | Round | Square | Lambda | Implicit | Comment
11
12  type Predicate = Less | LessEqual | Equal | GreaterEqual | Greater | NotEqual
13
14  type genericId<'a>= {Namespace:List<string>;Name:'a;}
15  type Id        = genericId<string>
16
17  type Literal = I64 of System.Int64
18               | U64 of System.UInt64
19               | I32 of System.Int32
20               | U32 of System.Int32
21               | F64 of System.Double
22               | F32 of System.Single
23               | String of System.String
24               | Bool of System.Boolean
25               | Void
```

### B.2    CodegenInterface.fs

```
1   module CodegenInterface
2   open Common
3
4   type LambdaId = genericId<int>
5   type TypeId   = genericId<string>
6
7   type Type = DotNetType       of TypeId
8             | McType           of TypeId
9             | TypeApplication of Type*List<Type>
10            | Arrow            of Type*Type
11
12  type local_id = Named of string
13              | Tmp   of int
14
15  type premisse = Literal            of LiteralAssignment // assign literal to local
16              | Conditional          of Conditional       // stops evaluation if condition
       is false
17              | Destructor           of Destructor        // destructs Mc data into its
       constructor arguments
18              | ConstructorClosure of closure<Id>        // assigns mc data constructor
       closure to local
19              | FuncClosure          of closure<Id>        // assigns mc func closure to
       local
20              | LambdaClosure        of closure<LambdaId> // assigns lambda closure to local
21              | Application          of Application        // applies an argument to a local
       closure
22              | ApplicationCall      of ApplicationCall    // applies an argument to a local
       closure and calls it
23              | DotNetCall           of DotNetCall         // calls .Net method and assigns
       result to local
24              | DotNetStaticCall     of DotNetStaticCall   // calls .Net static method and
       assigns result to local
```

14

```
25                        | DotNetConstructor  of DotNetStaticCall  // calls .Net constructor
26                        | DotNetGet          of DotNetGet          // gets field and assigns it to
      local
27                        | DotNetSet          of DotNetSet          // sets field from local
28  and LiteralAssignment = {value:Literal; dest:local_id}
29  and Conditional = {left:local_id; predicate:Predicate; right:local_id}
30  and Destructor  = {source:local_id; destructor:Id; args:List<local_id>}
31  and closure<'a> = {func:'a;dest:local_id}
32  and Application = {closure:local_id; argument:local_id; dest:local_id}
33  and ApplicationCall = {closure:local_id; argument:local_id; dest:local_id; side_effect:
      bool}
34  and DotNetStaticCall = {func: Id; args:List<local_id>; dest:local_id; side_effect:bool}
35  and DotNetCall      = {instance: local_id; func: string; args:List<local_id>; dest:
      local_id; side_effect:bool; mutates_instance:bool}
36  and DotNetGet   = {instance: local_id; field: string; dest:local_id}
37  and DotNetSet   = {instance: local_id; field: string; src:local_id}
38
39  type rule = {
40    side_effect :bool
41    input  :List<local_id>
42    output :local_id
43    premis :List<premisse*int> // linenumber
44    typemap:Map<local_id,Type>
45    definition: Position
46  }
47
48  type data = {
49    args        :List<Type>
50    outputType :Type
51  }
52
53  type CompilerFlags = {debug:bool}
54
55  type fromTypecheckerWithLove = {
56    assemblies : List<string>
57    funcs   : Map<Id,List<rule>*Position>
58    lambdas : Map<LambdaId,rule>
59    datas   : List<Id*data>
60    main    : rule
61    flags   : CompilerFlags
62  }
63
64  let (-->) t1 t2 =
65    Arrow(t1,t2)
66
67  let mutable tmp_index = 0
68
69  let next_tmp () =
70    let current = tmp_index
71    tmp_index <- tmp_index+1
72    Tmp(current)
73
74  let current_tmp () =
75    Tmp(tmp_index-1)
76
77  let reset_tmp () =
78   do tmp_index <- 1
79   Tmp(0)
80
81  let reset () =
82   do tmp_index <- 0
```

## B.3  Interpreter.fs

```
1  module Interpreter
2  open Common
3  open CodegenInterface
```

```
4   open ParserMonad
5
6   let print_loc  (x:local_id) = match x with Named x -> x | Tmp x -> sprintf "_%d" x
7   let print_id   (x:Id) = x.Namespace @ [x.Name] |> String.concat "."
8
9   let print_premisse (p:premisse,i:int) :string =
10    let print_lamb (x:LambdaId) = x.Namespace @ [sprintf "lambda%d" x.Name] |> String.concat
        "."
11    match p with
12    | Literal            x -> sprintf "%03d:␣LITR␣%A␣->␣%s" i x.value (print_loc x.dest)
13    | Conditional        x -> sprintf "%03d:␣COND␣%s␣%A␣%s" i (print_loc x.left) x.predicate
        (print_loc x.right)
14    | Destructor         x -> sprintf "%03d:␣DTOR␣%s␣%s␣->␣%s" i (print_id x.destructor) (
        print_loc x.source) (x.args|>List.map print_loc|>String.concat "␣")
15    | ConstructorClosure x -> sprintf "%03d:␣CTOR␣%s␣->␣%s" i (print_id x.func) (print_loc x
        .dest)
16    | FuncClosure        x -> sprintf "%03d:␣FUNC␣%s␣->␣%s" i (print_id x.func) (print_loc x
        .dest)
17    | LambdaClosure      x -> sprintf "%03d:␣LAMB␣%s␣->␣%s" i (print_lamb x.func) (print_loc
        x.dest)
18    | Application        x -> sprintf "%03d:␣APPL␣%s␣%s␣->␣%s" i (print_loc x.closure) (
        print_loc x.argument) (print_loc x.dest)
19    | ApplicationCall    x -> sprintf "%03d:␣CALL␣%s␣%s␣->␣%s␣%s" i (print_loc x.closure) (
        print_loc x.argument) (print_loc x.dest) (if x.side_effect then "{SIDE-EFFECT}" else "
        ")
20    | DotNetConstructor  x -> sprintf "%03d:␣NCON␣%s(%s)␣->␣%s␣%s" i (print_id x.func) (x.
        args|>List.map print_loc|>String.concat "␣") (print_loc x.dest)(if x.side_effect then
        "{SIDE-EFFECT}" else "")
21    | DotNetStaticCall   x -> sprintf "%03d:␣NSCA␣%s(%s)␣->␣%s␣%s" i (print_id x.func) (x.
        args|>List.map print_loc|>String.concat "␣") (print_loc x.dest) (if x.side_effect then
        "{SIDE-EFFECT}" else "")
22    | DotNetCall         x -> sprintf "%03d:␣NDCA␣%s.%s(%s)␣->␣%s␣%s␣%s" i (print_loc x.
        instance) x.func (x.args|>List.map print_loc|>String.concat "␣") (print_loc x.dest) (
        if x.side_effect then "{SIDE-EFFECT}" else "") (if x.mutates_instance then "{MUTATES-
        INSTANCE}" else "")
23    | DotNetGet          x -> sprintf "%03d:␣NGET␣%s.%s␣->␣%s" i (print_loc x.instance) x.
        field (print_loc x.dest)
24    | DotNetSet          x -> sprintf "%03d:␣NSET␣%s.%s␣<-␣%s" i (print_loc x.instance) x.
        field (print_loc x.src)
25
26  type global_context = {assemblies:List<System.Reflection.Assembly>;funcs:Map<Id,List<rule
        >*Position>;lambdas:Map<LambdaId,rule>;datas:List<Id*data>;main:rule;}
27
28  let getClass (assemblies) (name:string) :System.Type =
29    let ts =
30      assemblies |> List.fold (fun (a:List<System.Type>) (v:System.Reflection.Assembly)->
31        let t:System.Type = v.GetType(name,false)
32        if t=null then a else t::a
33      ) []
34    match ts with
35    | [t] -> t
36    | []  -> failwith ("EVAL␣ERROR:␣"+name+"␣not␣found␣in␣assembly")
37    | _   -> failwith ("EVAL␣ERROR:␣"+name+"␣found␣in␣multiple␣assemblies")
38
39  let staticCallNonBuiltin (x:DotNetStaticCall) (symbol_table:Map<local_id,obj>) (assemblies
        :list<System.Reflection.Assembly>) :obj =
40    let t = getClass assemblies (x.func.Namespace|>String.concat ".")
41    let args = x.args |> List.map (fun a->symbol_table.[a]) |> List.toArray
42    let argtypes = System.Type.GetTypeArray(args)
43    let f = t.GetMethod(x.func.Name,argtypes)
44    f.Invoke(null,args)
45
46  let DotNetConstruct (x:DotNetStaticCall) (symbol_table:Map<local_id,obj>) (assemblies:list
        <System.Reflection.Assembly>) :obj =
47    let t = getClass assemblies (x.func.Namespace@[x.func.Name]|>String.concat ".")
48    let args = x.args |> List.map (fun a->symbol_table.[a]) |> List.toArray
49    let argtypes = System.Type.GetTypeArray(args)
```

```
50     let c = t.GetConstructor(argtypes)
51     c.Invoke(args)
52
53  let dynamicCallNonBuiltin (x:DotNetCall) (parent_type:Id) (symbol_table:Map<local_id,obj>)
         (assemblies:list<System.Reflection.Assembly>) :obj =
54     let t = getClass assemblies (parent_type.Namespace@[parent_type.Name]|>String.concat "."
         )
55     let args = x.args |> List.map (fun a->symbol_table.[a]) |> List.toArray
56     let argtypes = System.Type.GetTypeArray(args)
57     let f = t.GetMethod(x.func,argtypes)
58     f.Invoke(symbol_table.[x.instance],args)
59
60  let dotNetGet (x:DotNetGet) (parent_type:Id) (symbol_table:Map<local_id,obj>) (assemblies:
         list<System.Reflection.Assembly>) :obj =
61     let t = getClass assemblies (parent_type.Namespace@[parent_type.Name]|>String.concat "."
         )
62     let field = t.GetField(x.field)
63     field.GetValue(symbol_table.[x.instance])
64
65  let dotNetSet (x:DotNetSet) (parent_type:Id) (symbol_table:Map<local_id,obj>) (assemblies:
         list<System.Reflection.Assembly>) =
66     let t = getClass assemblies (parent_type.Namespace@[parent_type.Name]|>String.concat "."
         )
67     let field = t.GetField(x.field)
68     field.SetValue(symbol_table.[x.instance],symbol_table.[x.src])
69
70  let rec eval_step (p:premisse,i:int)
71                    (global_context:global_context)
72                    (type_map:Map<local_id,Type>)
73                    (symbol_table:Map<local_id,obj>)
74                    :List<Map<local_id,obj>> =
75   (*
76   do symbol_table |> Map.toSeq |> Seq.iter (fun(id,o)->
77       do System.Console.ForegroundColor <- System.ConsoleColor.Magenta
78       do printf "%s\n" (print_loc id)
79       do System.Console.ResetColor()
80       do printf "%A\n" o
81       ()
82     )
83   *)
84     do System.Console.ForegroundColor <- System.ConsoleColor.Yellow
85     do printf "%s\n" (print_premisse (p,i))
86     do System.Console.ResetColor()
87     match p with
88     | Literal x ->
89       let value = match x.value with
90                   | I64 x->box x | U64 x->box x | F64 x->box x
91                   | I32 x->box x | U32 x->box x | F32 x->box x
92                   | String x->box x | Bool x->box x | Void->box()
93       [symbol_table.Add(x.dest,value)]
94     | Conditional x ->
95       let l = symbol_table.[x.left]  :?> System.IComparable
96       let r = symbol_table.[x.right] :?> System.IComparable
97       let f = match x.predicate with Less -> (<) | LessEqual->(<=) | Equal -> (=) |
         GreaterEqual -> (>=) | Greater -> (>) | NotEqual -> (<>)
98       if f l r then [symbol_table] else []
99     | Destructor x ->
100      let id,args = symbol_table.[x.source] :?> Id*List<obj>
101      if x.destructor = id then
102        let additions = args |> List.zip x.args |> Map.ofList
103        [symbol_table |> Map.fold (fun a k v->a|>Map.add k v) additions]
104      else
105        []
106    | FuncClosure x
107    | ConstructorClosure x ->
108      let ret = x.func,List.empty
109      [symbol_table.Add(x.dest,(box ret))]
```

17

```
110    | LambdaClosure x ->
111      let ret = x.func,List.empty
112      [symbol_table.Add(x.dest,(box ret))]
113    | Application x ->
114      match symbol_table.[x.closure] with
115      | :? (Id*List<obj>) as foo ->
116        let id,args = foo
117        let res = id,(args@[symbol_table.[x.argument]])
118        [symbol_table.Add(x.dest,(box res))]
119      | :? (LambdaId*List<obj>) as foo ->
120        let id,args = foo
121        let res = id,(args@[symbol_table.[x.argument]])
122        [symbol_table.Add(x.dest,(box res))]
123    | ApplicationCall x ->
124      match symbol_table.[x.closure] with
125      | :? (Id*List<obj>) as foo ->
126        let id,args = foo
127        let filled_args = args@[symbol_table.[x.argument]]
128        match global_context.datas |> List.tryFind (fun(x,_)->x=id) with
129        | Some(id,data) ->
130          [symbol_table.Add(x.dest,box filled_args)]
131        | None ->
132          fst global_context.funcs.[id] |> List.map (fun rule -> // for each rule
133              let results = eval_rule rule global_context filled_args
134              results |> List.map (fun v->symbol_table.Add(x.dest,v))
135          )|>List.concat
136      | :? (LambdaId*List<obj>) as foo ->
137        let id,args = foo
138        let filled_args = args@[symbol_table.[x.argument]]
139        let rule = global_context.lambdas.[id]
140        let results = eval_rule rule global_context filled_args
141        results |> List.map (fun v->symbol_table.Add(x.dest,v))
142    | DotNetStaticCall x ->
143      let ret:obj =
144        match x.func.Namespace with
145        | ["System";"Int32"] when x.args.Length=2 ->
146          let l = symbol_table.[x.args.[0]] :?> System.Int32
147          let r = symbol_table.[x.args.[1]] :?> System.Int32
148          match x.func.Name with "+"->box(l+r) | "/"->box(l/r) | "*"->box(l*r) | "%"->box(l%
    r) | "-"->box(l-r) | _ ->staticCallNonBuiltin x symbol_table global_context.assemblies
149        | ["System";"Int64"] when x.args.Length=2 ->
150          let l = symbol_table.[x.args.[0]] :?> System.Int64
151          let r = symbol_table.[x.args.[1]] :?> System.Int64
152          match x.func.Name with "+"->box(l+r) | "/"->box(l/r) | "*"->box(l*r) | "%"->box(l%
    r) | "-"->box(l-r) | _ ->staticCallNonBuiltin x symbol_table global_context.assemblies
153        | ["System";"Single"] when x.args.Length=2 ->
154          let l = symbol_table.[x.args.[0]] :?> System.Single
155          let r = symbol_table.[x.args.[1]] :?> System.Single
156          match x.func.Name with "+"->box(l+r) | "/"->box(l/r) | "*"->box(l*r) | "%"->box(l%
    r) | "-"->box(l-r) | _ ->staticCallNonBuiltin x symbol_table global_context.assemblies
157        | ["System";"Double"] when x.args.Length=2 ->
158          let l = symbol_table.[x.args.[0]] :?> System.Double
159          let r = symbol_table.[x.args.[1]] :?> System.Double
160          match x.func.Name with "+"->box(l+r) | "/"->box(l/r) | "*"->box(l*r) | "%"->box(l%
    r) | "-"->box(l-r) | _ ->staticCallNonBuiltin x symbol_table global_context.assemblies
161        | _ -> staticCallNonBuiltin x symbol_table global_context.assemblies
162      [symbol_table.Add(x.dest,ret)]
163    | DotNetConstructor x ->
164      let ret = DotNetConstruct x symbol_table global_context.assemblies
165      [symbol_table.Add(x.dest,ret)]
166    | DotNetCall x ->
167      match type_map.[x.instance] with
168      | DotNetType t ->
169        let ret = dynamicCallNonBuiltin x t symbol_table global_context.assemblies
170        [symbol_table.Add(x.dest,ret)]
171    | DotNetGet x ->
172      match type_map.[x.instance] with
```

18

```
173        | DotNetType t ->
174          let ret = dotNetGet x t symbol_table global_context.assemblies
175          [symbol_table.Add(x.dest,ret)]
176      | DotNetSet x ->
177        match type_map.[x.instance] with
178        | DotNetType t ->
179          do dotNetSet x t symbol_table global_context.assemblies
180          [symbol_table]
181
182  and eval_rule (rule:rule)
183                (ctxt:global_context)
184                (args:List<obj>) :List<obj> =
185      let input = Seq.zip rule.input args |> Map.ofSeq
186      let called_stabs = ([input],rule.premis) ||> List.fold (fun (stabs:List<Map<local_id,
        obj>>) p -> // for each instruction
187          stabs |> List.map (fun stab -> // for each fork
188            eval_step p ctxt rule.typemap stab
189          ) |> List.concat
190        )
191      let output = called_stabs |> List.map (fun stab-> stab.[rule.output])
192      output
193
194  let load_assemblies (src:List<string>) :List<System.Reflection.Assembly> =
195    src |> List.map (fun str-> System.Reflection.Assembly.LoadFrom(str) )
196
197  let eval_main (src:fromTypecheckerWithLove) =
198    let ctxt:global_context={assemblies=load_assemblies src.assemblies;funcs=src.funcs;datas
        =src.datas;lambdas=src.lambdas;main=src.main}
199    do printf "starting␣interpreting\n"
200    let res = eval_rule ctxt.main ctxt []
201    do printf "results:\n%s" (res|>List.map (sprintf "␣␣%A\n")|>String.concat "")
202    res
```

## B.4   Codegen.fs

```
1   module Codegen
2   open Common
3   open CodegenInterface
4   open Mangle
5
6   let mutable flags:CompilerFlags={debug=false};
7
8   let fst(x,_)=x
9   let snd(_,x)=x
10
11  let foldi (f:int->'state->'element->'state) (s:'state) (lst:seq<'element>) :'state =
12    let fn ((counter:int),(state:'state)) (element:'element) :int*'state =
13      counter+1,(f counter state element)
14    let _,ret = lst|>Seq.fold fn (0,s)
15    ret
16
17  let ice () =
18      do System.Console.BackgroundColor <- System.ConsoleColor.Red
19      do System.Console.Write "INTERNAL␣COMPILER␣ERROR"
20      do System.Console.ResetColor()
21
22  type NamespacedItem = Ns       of string*List<NamespacedItem>
23                      | Data     of string*data
24                      | Function of string*List<rule>
25                      | Lambda   of int*rule
26
27  let construct_tree (input:fromTypecheckerWithLove) :List<NamespacedItem> =
28    let rec datatree (s:List<NamespacedItem>) (idx:List<string>,v:string*data) :List<
        NamespacedItem> =
29      match idx with
30      | []    -> (Data(v))::s
```

19

```
31      | n::ns -> match s |> List.partition (fun x->match x with Ns(n,_)->true | _->false)
        with
32              | [Ns(n,body)],rest -> Ns(n,datatree body (ns,v))::rest
33              | [],list            -> Ns(n,datatree []   (ns,v))::list
34   let rec functree (s:List<NamespacedItem>) (idx:List<string>,v:string*(List<rule>*
        Position)) :List<NamespacedItem> =
35      match idx with
36      | []    -> let (l,(r,_)) = v in (Function(l,r))::s
37      | n::ns -> match s |> List.partition (fun x->match x with Ns(n,_)->true | _->false)
        with
38              | [Ns(n,body)],rest -> Ns(n,functree body (ns,v))::rest
39              | [],list            -> Ns(n,functree []   (ns,v))::list
40   let rec lambdatree (s:List<NamespacedItem>) (idx:List<string>,v:int*rule) :List<
        NamespacedItem> =
41      match idx with
42      | []    -> (Lambda(v))::s
43      | n::ns -> match s |> List.partition (fun x->match x with Ns(n,_)->true | _->false)
        with
44              | [Ns(n,body)],rest -> Ns(n,lambdatree body (ns,v))::rest
45              | [],list            -> Ns(n,lambdatree []   (ns,v))::list
46   let go input output state = input |> Seq.map (fun (n,d)->(List.rev n.Namespace),(n.Name,
        d)) |> Seq.fold output state
47   [] |> go (Map.toSeq input.lambdas) lambdatree
48      |> go (Map.toSeq input.funcs) functree
49      |> go input.datas datatree
50
51 let print_literal (lit:Literal) =
52   match lit with
53   | I64 i    -> sprintf "%dL"  i
54   | U64 i    -> sprintf "%uUL" i
55   | I32 i    -> sprintf "%u" i
56   | U32 i    -> sprintf "%uU" i
57   | F64 i    -> sprintf "%f" i
58   | F32 i    -> sprintf "%ff" i
59   | String s -> sprintf "\"%s\"" s
60   | Bool b   -> if b then "true" else "false"
61   | Void     -> "void"
62
63 let print_predicate (p:Predicate) :string=
64   match p with Less -> "<" | LessEqual -> "<=" | Equal -> "=" | GreaterEqual -> ">=" |
        Greater -> ">" | NotEqual -> "!="
65
66 let field (n:int) (t:Type) :string =
67   sprintf "public %s  _arg%d;\n" (mangle_type t) n
68
69 let highest_tmp (typemap:Map<local_id,Type>): int =
70   typemap |> Map.fold (fun s k _ -> match k with Tmp(x) when x>s -> x | _ -> s) 0
71
72 let get_map (a:local_id) (m:Map<local_id,int>) :int*Map<local_id,int> =
73   match Map.tryFind a m with None -> 0,(Map.add a 1 m) | Some x -> x,(Map.add a (x+1) m)
74
75 let overloadableOps:Map<string,string> =
76   [
77     "op_Equality","=="
78     "op_Inequality","!="
79     "op_GreaterThan",">"
80     "op_LessThan","<"
81     "op_GreaterThanOrEqual",">="
82     "op_LessThanOrEqual","<="
83     "op_BitwiseAnd","&"
84     "op_BitwiseOr","|"
85     "op_Addition","+"
86     "op_Subtraction","-"
87     "op_Division","/"
88     "op_Modulus","%"
89     "op_Multiply","*"
90     "op_LeftShift","<<"
```

```
 91        "op_RightShift",">>"
 92        "op_ExclusiveOr","^"
 93        "op_UnaryNegation","-"
 94        "op_UnaryPlus","+"
 95        "op_LogicalNot","!"
 96        "op_OnesComplement","~"
 97        "op_False","false"
 98        "op_True","true"
 99        "op_Increment","++"
100        "op_Decrement","--"
101     ] |> Map.ofList
102
103  let print_label (i:int) = sprintf "_skip%d" i
104
105  let print_debug_tree (fromTypeChecker:Map<Id,list<rule>*Position>) (main:rule) =
106    let tree =
107      // first flatten the hierarchy
108      let rules = fromTypeChecker |> Map.toSeq |> Seq.map (fun(name,(lst,pos))->lst|>Seq.
         ofList|>Seq.map(fun rule->name,rule,pos)) |> Seq.concat
109      // add main to the flat list
110      let rules = rules |> Seq.append ([{Namespace=[];Name="main"},main,main.definition])
111      // then group by file (that the rule is defined by)
112      let files = rules |> Seq.groupBy (fun(id,rule,pos)->rule.definition.File)
113      // within each file, group by func (and strip info out of rule)
114      files |> Seq.map (fun(filename,rules)->
115        let funcs = rules |> Seq.groupBy (fun(id,rule,pos)->id,pos)
116                   |> Seq.map (fun((id,pos),rules)->id,pos,rules|>Seq.map(fun(id,rule,pos)
         ->rule))
117        filename,funcs)
118    let debug_tree =
119      let per_file = tree  |> Seq.mapi (fun filenr (filename,funcs)->
120        let per_func = funcs |> Seq.mapi (fun funcnr (funcname,declposition,rules)->
121          let per_rule = rules |> Seq.mapi (fun rulenr (rule) ->
122            let per_line = rule.premis |> Seq.groupBy snd
123            let per_prem = (per_line(*|>Seq.rev|>Seq.tail|>Seq.rev*)) |> Seq.mapi (fun
         premnr (linenumber,prems)->
124               sprintf "_DBUG.program_tree.Nodes[%d].Nodes[%d].Nodes[%d].Nodes.Add(\"line %d
         \");\n" filenr funcnr rulenr linenumber )
125             sprintf "_DBUG.program_tree.Nodes[%d].Nodes[%d].Nodes.Add(\"line %d\");\n"
         filenr funcnr (Seq.last per_line|>fst)
126           + (String.concat "" per_prem) )
127         sprintf "_DBUG.program_tree.Nodes[%d].Nodes.Add(\"%s\");\n" filenr (mangle_id
         funcname)
128           + (String.concat "" per_rule) )
129       sprintf "_DBUG.program_tree.Nodes.Add(\"%s\");\n" filename
130       + (String.concat "" per_func) )
131      per_file |> String.concat ""
132    let breakpoint_tree =
133      let per_file = tree |> Seq.map (fun(_,funcs)->
134        let per_func = funcs |> Seq.map (fun(id,_,rules)->
135          let per_rule = rules |> Seq.mapi (fun i _ ->
136            sprintf "%s._DBUG_breakpoints_%d" (mangle_id id) i)
137          sprintf "        new bool[][]{%s}"  (String.concat "," per_rule) )
138        sprintf "      new bool[][][]{\n%s\n      }"  (String.concat ",\n" per_func) )
139      sprintf "    new bool[][][][]{\n%s\n    };\n" (String.concat "," per_file)
140    ("_DBUG.breakpoints="+breakpoint_tree)+debug_tree
141
142  let premisse (p:premisse) (m:Map<local_id,Type>) (app:Map<local_id,int>) (rule_nr:int) =
143    match p with
144    | Literal x -> app,sprintf "/*LITR*/var %s = %s;\n"
145                     (mangle_local_id x.dest)
146                     (print_literal x.value)
147    | Conditional x -> app,sprintf "/*COND*/if(!(%s %s %s)){goto %s;}\n"
148                     (mangle_local_id x.left)
149                     (print_predicate x.predicate)
150                     (mangle_local_id x.right)
151                     (print_label rule_nr)
```

21

```
152      | Destructor x ->
153        let new_id  = (Tmp(1+(highest_tmp m)))
154        app,sprintf "/*DTOR*/var␣%s␣=␣%s␣as␣%s;\nif(%s==null){goto␣%s;}\n%s"
155          (mangle_local_id new_id)
156          (mangle_local_id x.source)
157          (mangle_id       x.destructor)
158          (mangle_local_id new_id)
159          (print_label rule_nr)
160          (x.args|>List.mapi(fun nr arg->sprintf "var␣%s=%s._arg%d;\n" (mangle_local_id arg) (
           mangle_local_id new_id) nr)|>String.concat "")
161      | ConstructorClosure x
162      | FuncClosure x -> (app|>Map.add x.dest 0),sprintf "/*FUNC*/var␣%s␣=␣new␣%s();\n"
163                            (mangle_local_id x.dest)
164                            (mangle_id x.func)
165      | LambdaClosure x -> (app|>Map.add x.dest 0),sprintf "/*LAMB*/var␣%s␣=␣new␣%s();\n"
166                            (mangle_local_id x.dest)
167                            (mangle_lambda x.func)
168      | DotNetCall x ->
169        let isVoid = m.[x.dest]=Type.DotNetType({Namespace=[];Name="void"})
170        app,sprintf  "/*NDCA*/%s%s.%s(%s);\n"
171          (if isVoid then "" else sprintf "var␣%s=␣" (mangle_local_id x.dest))
172          (mangle_local_id x.instance)
173          x.func
174          (x.args |> List.map mangle_local_id|>String.concat ",")
175      | DotNetStaticCall x ->
176          let isVoid = m.[x.dest]=Type.DotNetType({Namespace=[];Name="void"})
177          if overloadableOps.ContainsKey(x.func.Name) then
178            let args = x.args |> List.rev
179            app,sprintf "/*NSCA*/%s%s␣%s␣%s;\n"
180              (if isVoid then "" else sprintf "var␣%s=␣" (mangle_local_id x.dest))
181              (match x.args.Length with
182               | 1 -> ""
183               | 2 -> mangle_local_id args.[1])
184              overloadableOps.[x.func.Name]
185              (mangle_local_id args.[0])
186          else
187            app,sprintf "/*NSCA*/%s%s(%s);\n"
188              (if isVoid then "" else sprintf "var␣%s=␣" (mangle_local_id x.dest))
189              (x.func.Namespace@[x.func.Name]|>String.concat ".")
190              (x.args |> List.map mangle_local_id|>String.concat ",")
191      | DotNetConstructor x -> app,sprintf "/*NCON*/var␣%s␣=␣new␣%s(%s);\n"
192                                (mangle_local_id x.dest)
193                                (x.func.Namespace@[x.func.Name]|>String.concat ".")
194                                (x.args |> List.map mangle_local_id|>String.concat ",")
195      | DotNetGet x -> app,sprintf "/*NGET*/var␣%s␣=␣%s.%s;\n"
196                        (mangle_local_id x.dest)
197                        (mangle_local_id x.instance)
198                        x.field
199      | DotNetSet x -> app,sprintf "/*NSET*/%s.%s␣=␣%s;\n"
200                        (mangle_local_id x.instance)
201                        x.field
202                        (mangle_local_id x.src)
203      | Application x ->
204        let i = match app|>Map.tryFind x.closure with Some(x)->x | None-> failwith (sprintf "
           Application␣failed:␣%s␣is␣not␣a␣closure." (mangle_local_id x.closure))
205        (app|>Map.add x.dest (i+1)),sprintf "/*APPL*/var␣%s␣=␣%s;␣%s.%s=%s;\n"
206                        (mangle_local_id  x.dest)
207                        (mangle_local_id  x.closure)
208                        (mangle_local_id  x.dest)
209                        (sprintf "_arg%d" i)
210                        (mangle_local_id  x.argument)
211      | ApplicationCall x ->
212        let i = match app|>Map.tryFind x.closure with Some(x)->x | None-> failwith (sprintf "
           ApplicationCall␣failed:␣%s␣is␣not␣a␣closure." (mangle_local_id x.closure))
213        (app|>Map.add x.dest (i+1)),sprintf "/*CALL*/%s.%s=%s;␣var␣%s␣=␣%s._run();\n"
214          (mangle_local_id  x.closure)
215          (sprintf "_arg%d" i)
```

```
216            (mangle_local_id  x.argument)
217            (mangle_local_id  x.dest)
218            (mangle_local_id  x.closure)
219
220  let get_definitions (p:premisse) :List<local_id> =
221    match p with
222    | Literal              x -> [x.dest]
223    | Conditional          _ -> []
224    | Destructor           x ->  x.args
225    | ConstructorClosure x -> [x.dest]
226    | FuncClosure          x -> [x.dest]
227    | LambdaClosure        x -> [x.dest]
228    | Application          x -> [x.dest]
229    | ApplicationCall      x -> [x.dest]
230    | DotNetCall           x -> [x.dest]
231    | DotNetStaticCall     x -> [x.dest]
232    | DotNetConstructor  x -> [x.dest]
233    | DotNetGet            x -> [x.dest]
234    | DotNetSet            x -> [x.src]
235
236  let print_rule (rule_nr:int) (rule:rule) =
237    let symbol_table_add id =
238      let s=mangle_local_id id
239      if id=Named("nil") then "" else sprintf "_DBUG_symbol_table[\"%s\"]=%s;\n" s s
240    let linegroups:seq<int*seq<premisse>> =
241      rule.premis |> Seq.groupBy (fun(_,x)->x) |> Seq.map (fun(l,p)->l,(p|>Seq.map(fun(x,_)
         ->x)))
242    let fn (app:Map<local_id,int>,str:string) (p:premisse) =
243      let (a:Map<local_id,int>,s:string) =
244        let l,r = premisse p rule.typemap app rule_nr
245        if flags.debug then
246          let stab_adds = p|>get_definitions|>List.map symbol_table_add |> String.concat ""
247          l,(r+stab_adds)
248        else l,r
249      a,(str+s)
250    let lines =
251      linegroups |> Seq.mapi
252        (fun idx (linenumber,premisses)->
253          let breakpoint =
254            if flags.debug then
255              sprintf "if(_DBUG_breakpoints_%d[%d]){_DBUG.breakpoint(\"%s\",%d,
         _DBUG_symbol_table);}\n" rule_nr idx rule.definition.File linenumber
256            else ""
257          let _,s = ((Map.empty,""),premisses) ||> Seq.fold fn
258          s+breakpoint)
259    sprintf "{\n%s%s%sreturn␣%s;}\n%s:\n"
260      (if flags.debug then "var␣_DBUG_symbol_table␣=␣new␣System.Collections.Generic.
         Dictionary<string,␣object>();\n" else "")
261      (rule.input|>List.mapi (fun i x->sprintf "var␣%s=_arg%d;\n%s" (mangle_local_id x) i (
         if flags.debug then symbol_table_add x else "")) |> String.concat "")
262      (lines|>String.concat "")
263      (mangle_local_id rule.output)
264      (print_label rule_nr)
265
266  let nr_of_actual_lines (rule:rule):int =
267    rule.premis |> Seq.map snd |> Seq.distinct |> Seq.length
268
269  let print_rule_bodies (rules:rule list) =
270    let len = rules |> List.scan (fun s r->s+(nr_of_actual_lines r)-1) 0 |> List.rev |> List
         .tail |> List.rev |> List.zip rules
271    len |> List.mapi (fun x (a,b)->print_rule x a) |> String.concat ""
272
273  let generate_breakpoint_def (funcname:string) (rules:rule seq) =
274    let linenumbers = rules |> Seq.map (fun x->x.premis) |> Seq.map ((Seq.map snd)>>Seq.
         distinct)
275    let subarrays = linenumbers |> Seq.mapi (fun i x->
276      let s =
```

```
277        if funcname="_main" then
278          x |>Seq.mapi (fun i _->if i=0 then "true" else "false") |> String.concat ","
279        else
280          x|>Seq.map (fun _->"false") |> String.concat ","
281      sprintf "%s._DBUG_breakpoints_%d␣=␣new␣bool[]{%s};\n" funcname i s)
282    subarrays |> String.concat ""
283
284  let generate_breakpoint_decl (rules:rule seq) = rules |> Seq.mapi (fun i _-> sprintf "
        public␣static␣bool[]␣_DBUG_breakpoints_%d;\n" i) |> String.concat ""
285
286  let print_main (input:fromTypecheckerWithLove) =
287    let rule = input.main
288    let return_type = mangle_type rule.typemap.[rule.output]
289    let body = sprintf "static_%s␣body(){\n%sthrow␣new␣System.MissingMethodException();\n}"
        return_type (print_rule_bodies [rule])
290    let debug_init =
291      if flags.debug then
292        let foo = input.funcs |> Map.toSeq |> Seq.map (fun(k,(rules,pos))->
        generate_breakpoint_def (mangle_id k) rules) |> String.concat ""
293        foo+(generate_breakpoint_def "_main" [rule])+(print_debug_tree input.funcs input.
        main)
294      else ""
295    let main = sprintf "static␣void␣Main()␣{\n%s\nSystem.Console.WriteLine(System.String.
        Format(\"{0}\",␣body()));\n}" debug_init
296    sprintf "class␣_main{\n%s%s%s}\n" (if flags.debug then generate_breakpoint_decl [rule]
        else "") body main
297
298  let rec print_tree (lookup:fromTypecheckerWithLove) (ns:List<NamespacedItem>) :string =
299    let build_func (name:string) (rules:rule list) =
300      let breakpoints = if flags.debug then generate_breakpoint_decl rules else ""
301      let rule = List.head rules
302      let args = rule.input |> Seq.mapi (fun nr id-> sprintf "public␣%s␣␣_arg%d;\n" (
        mangle_type rule.typemap.[id]) nr) |> String.concat ""
303      let ret_type = mangle_type rule.typemap.[rule.output]
304      let rules = print_rule_bodies rules
305      sprintf "class␣%s{\n%s%spublic␣%s␣␣_run(){\n%sthrow␣new␣System.MissingMethodException()
        ;\n}\n}\n" (CSharpMangle name) args breakpoints ret_type rules
306    let print_base_types (ns:List<NamespacedItem>) =
307      let types = ns |> List.fold (fun types item -> match item with Data (_,v) -> v.
        outputType::types | _ -> types) [] |> List.distinct
308      let print t = sprintf "public␣class␣%s{}\n" (t|>remove_namespace_of_type|>mangle_type)
309      List.map print types
310    let go ns =
311      match ns with
312      | Ns(n,ns)      -> sprintf "namespace␣%s{\n%s}\n" (CSharpMangle n) (print_tree lookup
        ns)
313      | Data(n,d)     -> sprintf "public␣class␣%s:%s{\n%s}\n" (CSharpMangle n) (d.outputType
        |>remove_namespace_of_type|>mangle_type) (d.args|>List.mapi field|>String.concat "")
314      | Function(name,rules) -> build_func name rules
315      | Lambda(number,rule)  -> build_func (sprintf "_lambda%d" number) [rule]
316    (print_base_types ns)@(ns|>List.map go)|>String.concat "\n"
317
318  let get_locals (ps:(premisse*int) list) :local_id list =
319    ps |> List.collect (fun (p,_) ->
320      match p with
321      | Literal            x -> [x.dest]
322      | Conditional        x -> [x.left;x.right]
323      | Destructor         x -> x.source::x.args
324      | LambdaClosure      x -> [x.dest]
325      | FuncClosure        x -> [x.dest]
326      | DotNetCall         x -> [x.dest]
327      | DotNetStaticCall   x -> [x.dest]
328      | DotNetConstructor  x -> [x.dest]
329      | DotNetGet          x -> [x.dest]
330      | DotNetSet          x -> [x.src]
331      | ConstructorClosure x -> [x.dest]
332      | Application        x -> [x.dest]
```

```
333        | ApplicationCall     x -> [x.closure;x.dest;x.argument] )
334
335  let validate (input:fromTypecheckerWithLove) :bool =
336    let print_local_id (id:local_id) = match id with Named(x)->x | Tmp(x)->sprintf "
         temporary(%d)" x
337    let print_id (id:Id) = String.concat "^" (id.Name::id.Namespace)
338    let check_typemap (id:Id) (rule:rule) :bool =
339      let expected = (get_locals rule.premis) @ (rule.output::rule.input) |> List.distinct
         |> List.sort
340      let received = rule.typemap |> Map.toList |> List.map (fun (x,_)->x) |> List.sort
341      if expected = received then true
342      else
343        let missing = expected |> List.filter (fun x -> received |> List.exists (fun y->x=y)
         |> not)
344        let extra  = received |> List.filter (fun x -> expected |> List.exists (fun y->x=y)
         |> not)
345        do ice()
346        do printf " incorrect typemap in rule %s:\n  missing %A\n  extra: %A\n" (print_id id
         ) missing extra
347        false
348    let check_dest_constness (id:Id) (rule:rule) (success:bool):bool =
349      let per_premisse (statementnr:int) (set:Set<local_id>,success:bool) (premisse:
         premisse,i:int) :Set<local_id>*bool =
350        let check (set:Set<local_id>,success:bool) (local:local_id) =
351          if set.Contains(local) then
352            do ice()
353            do printf " %s assigned twice in rule %s, statement %d on line %d\n" (
         print_local_id local) (print_id id) statementnr i
354            set,false
355          else set.Add(local),success
356        match premisse with
357        | Literal x               -> check (set,success) x.dest
358        | Conditional _           -> set,success
359        | Destructor x            -> x.args |> Seq.fold check (set,success)
360        | ConstructorClosure x    -> check (set,success) x.dest
361        | FuncClosure  x          -> check (set,success) x.dest
362        | LambdaClosure x         -> check (set,success) x.dest
363        | DotNetCall x            -> check (set,success) x.dest
364        | DotNetStaticCall x      -> check (set,success) x.dest
365        | DotNetConstructor x     -> check (set,success) x.dest
366        | DotNetGet x             -> check (set,success) x.dest
367        | DotNetSet x             -> set,success
368        | Application x           -> check (set,success) x.dest
369        | ApplicationCall x       -> check (set,success) x.dest
370      let _,ret = rule.premis |> foldi per_premisse (Set.empty,success)
371      ret
372    (true,input.funcs) ||> Map.fold (fun (success:bool) (id:Id) (rules,position)->
373      if rules.IsEmpty then
374        do ice()
375        do printf " empty rule: %s\n" (print_id id)
376        false
377      else
378        (true,input.main::rules) ||> List.fold (fun (success:bool) (rule:rule) -> if
         check_typemap id rule then (check_dest_constness id rule success) else false))
379
380  let failsafe_codegen(input:fromTypecheckerWithLove) :Option<string>=
381    do flags <- input.flags
382    if validate input then
383      let foo = input |> construct_tree |> print_tree input
384      let dbug = if flags.debug then System.IO.File.ReadAllText("_DBUG.cs.txt") else ""
385      dbug+foo+(print_main input) |> Some
386
387    else None
```

## B.5  Mangle.fs

```
1  module Mangle
```

```fsharp
2  open Common
3  open CodegenInterface
4
5  let genericMangle (name:string) :string =
6    let readables =
7      Map.ofArray <| Array.zip "!#$%&'*+,-./\\:;<>=?@^_`|~"B [|
8        "bang";"hash";"cash";"perc";"amp";"prime";"star";"plus";"comma";
9        "dash"; "dot";"slash";"back";"colon";"semi";"less";"great";
10       "equal";"quest";"at";"caret";"under";"tick";"pipe";"tilde"|]
11   let mangleChar c =
12     if (c>='A'&&c<='Z') || (c>='a'&&c<='z') || (c>='0'&&c<='9') then
13       sprintf "%c" c
14     else
15       let lookup = readables |> Map.tryFind (System.Convert.ToByte c)
16       match lookup with
17       | None   -> failwith <| sprintf "ERROR(codegen):␣expecting␣printable␣ASCII␣
     character,␣got␣(0x%04X)" (System.Convert.ToUInt16(c))
18       | Some x -> sprintf "_%s" x
19    name |> String.collect mangleChar
20
21 let CSharpMangle (name:string) :string =
22   let keywords = Set.ofArray [| "abstract" ; "as" ; "base" ; "bool" ;
23     "break" ; "byte" ; "case" ; "catch" ; "char" ; "checked" ; "class" ; "const" ;
24     "continue" ; "decimal" ; "default" ; "delegate" ; "do" ; "double" ; "else" ;
25     "enum" ; "event" ; "explicit" ; "extern" ; "false" ; "finally" ; "fixed" ;
26     "float" ; "for" ; "foreach" ; "goto" ; "if" ; "implicit" ; "in" ; "int" ;
27     "interface" ; "internal" ; "is" ; "lock" ; "long" ; "namespace" ; "new" ;
28     "null" ; "object" ; "operator" ; "out" ; "override" ; "params" ; "private" ;
29     "protected" ; "public" ; "readonly" ; "ref" ; "return" ; "sbyte" ; "sealed" ;
30     "short" ; "sizeof" ; "stackalloc" ; "static" ; "string" ; "struct" ;
31     "switch" ; "this" ; "throw" ; "true" ; "try" ; "typeof" ; "uint" ; "ulong" ;
32     "unchecked" ; "unsafe" ; "ushort" ; "using" ; "virtual" ; "void" ;
33     "volatile" ; "while" |]
34   let name = genericMangle name
35   if keywords.Contains(name) then sprintf "@%s" name else name
36
37 let rec mangle_type_suffix(t:Type):string=
38   match t with
39   | DotNetType (id) -> id.Namespace@[id.Name] |> Seq.map genericMangle |> String.concat "
     _ns"
40   | McType     (id) -> id.Namespace@[id.Name] |> Seq.map genericMangle |> String.concat "
     _ns"
41   | TypeApplication (fn,lst) -> (mangle_type_suffix fn)+"_of"+(lst |> List.map
     mangle_type_suffix |> String.concat "_t")
42
43 let rec remove_namespace_of_type(t:Type):Type=
44   match t with
45   | DotNetType (id) -> DotNetType({id with Namespace=[]})
46   | McType     (id) -> McType({id with Namespace=[]})
47   | TypeApplication (fn,lst) -> TypeApplication((remove_namespace_of_type fn),lst)
48
49 let rec mangle_type(t:Type):string=
50   match t with
51   | DotNetType (id) -> id.Namespace@[id.Name] |> Seq.map CSharpMangle |> String.concat "."
52   | McType     (id) -> id.Namespace@[id.Name] |> Seq.map (fun x->if x="System" then "
     _System" else CSharpMangle x) |> String.concat "."
53   | TypeApplication (fn,lst) -> (mangle_type fn)+"_of"+(lst|>List.map mangle_type_suffix|>
     String.concat "_t")
54
55 let mangle_local_id n = match n with Named x -> CSharpMangle x | Tmp x -> sprintf "_tmp%d"
     x
56 let mangle_id (id:Id) =
57   if id.Namespace =[] && id.Name="main" then "_main"
58   else (id.Name::id.Namespace) |> List.rev |> List.map (fun x->if x="System" then "_System
     " else CSharpMangle x) |> String.concat "."
59 let mangle_lambda (id:LambdaId) = sprintf "%s._lambda%d" (id.Namespace|>List.rev|>String.
     concat ".") id.Name
```

## B.6 _DBUG.cs

```
1   struct _DBUG {
2     public static System.Windows.Forms.TreeView program_tree = new System.Windows.Forms.
       TreeView();
3     public static bool[][][][] breakpoints;
4     public static void breakpoint(string filename, int line, System.Collections.Generic.
       Dictionary<string, object> stab) {
5       // table
6       var table = new System.Windows.Forms.ListView();
7       table.GridLines = true;
8       table.Dock = System.Windows.Forms.DockStyle.Fill;
9       table.View = System.Windows.Forms.View.Details;
10      table.Columns.Add("name", -2, System.Windows.Forms.HorizontalAlignment.Left);
11      table.Columns.Add("value", -2, System.Windows.Forms.HorizontalAlignment.Left);
12      foreach (var x in stab) {
13        table.Items.Add(new System.Windows.Forms.ListViewItem(new string[] { x.Key, System.
       String.Format("{0}", x.Value) }));
14      }
15
16      var tablepanel = new System.Windows.Forms.Panel();
17      tablepanel.Dock = System.Windows.Forms.DockStyle.Fill;
18      tablepanel.Controls.Add(table);
19
20      // tree
21      program_tree.CheckBoxes = true;
22      program_tree.Dock = System.Windows.Forms.DockStyle.Fill;
23      program_tree.ExpandAll();
24      for (int filenr = 0; filenr<breakpoints.Length; filenr++) {
25        for (int funcnr = 0; funcnr < breakpoints[filenr].Length; funcnr++) {
26          for (int rulenr = 0; rulenr < breakpoints[filenr][funcnr].Length; rulenr++) {
27            for (int premnr = 0; premnr < breakpoints[filenr][funcnr][rulenr].Length; premnr
       ++) {
28              program_tree.Nodes[filenr].Nodes[funcnr].Nodes[rulenr].Nodes[premnr].Checked =
       breakpoints[filenr][funcnr][rulenr][premnr];
29            }
30          }
31        }
32      }
33
34      var treepanel = new System.Windows.Forms.Panel();
35      treepanel.Dock = System.Windows.Forms.DockStyle.Fill;
36      treepanel.Controls.Add(program_tree);
37
38      // split
39      var split = new System.Windows.Forms.SplitContainer();
40      split.SplitterDistance = 100;
41      split.Dock = System.Windows.Forms.DockStyle.Fill;
42      split.Panel1.Controls.Add(treepanel);
43      split.Panel2.Controls.Add(tablepanel);
44
45      var splitform = new System.Windows.Forms.Form();
46      splitform.Controls.Add(split);
47      splitform.MinimumSize = new System.Drawing.Size(100, 100);
48      splitform.Text = filename + ":" + line;
49
50      var buttons = new System.Windows.Forms.Button[4];
51      buttons[0] = new System.Windows.Forms.Button();
52      buttons[0].Text = "continue";
53      buttons[0].DialogResult = System.Windows.Forms.DialogResult.Ignore;
54      buttons[1] = new System.Windows.Forms.Button();
55      buttons[1].Text = "abort";
56      buttons[1].DialogResult = System.Windows.Forms.DialogResult.Abort;
57      /*buttons[2] = new System.Windows.Forms.Button();
58      buttons[2].Text = "step in";
59      buttons[2].DialogResult = System.Windows.Forms.DialogResult.OK;
60      buttons[3] = new System.Windows.Forms.Button();
```

```
61      buttons[3].Text = "step over";
62      buttons[3].DialogResult = System.Windows.Forms.DialogResult.Yes;*/
63
64      var collection = new System.Windows.Forms.FlowLayoutPanel();
65      collection.Dock = System.Windows.Forms.DockStyle.Bottom;
66      collection.Controls.AddRange(buttons);
67      collection.AutoSize = true;
68      collection.AutoSizeMode = System.Windows.Forms.AutoSizeMode.GrowAndShrink;
69
70      splitform.Controls.Add(collection);
71
72      var result = splitform.ShowDialog();
73
74      for (int filenr = 0; filenr<breakpoints.Length; filenr++) {
75        for (int funcnr = 0; funcnr < breakpoints[filenr].Length; funcnr++) {
76          for (int rulenr = 0; rulenr < breakpoints[filenr][funcnr].Length; rulenr++) {
77            for (int premnr = 0; premnr < breakpoints[filenr][funcnr][rulenr].Length; premnr
    ++) {
78              breakpoints[filenr][funcnr][rulenr][premnr] = program_tree.Nodes[filenr].Nodes
    [funcnr].Nodes[rulenr].Nodes[premnr].Checked;
79            }
80          }
81        }
82      }
83
84      switch (result) {
85        /*case System.Windows.Forms.DialogResult.OK: // step in
86          return;
87        case System.Windows.Forms.DialogResult.Yes: // step over
88          return; */
89        case System.Windows.Forms.DialogResult.Ignore: // continue
90          return;
91        case System.Windows.Forms.DialogResult.Abort:
92          System.Environment.Exit(0);
93          return;
94      }
95    }
96  }
```

## B.7  Listtest.fs

```
1  module CodegenTest
2
3  open Common
4  open CodegenInterface
5
6  let test_data:fromTypecheckerWithLove=
7    let int_t:Type    = DotNetType({Namespace=["System"];Name="Int32"})
8    let float_t:Type = DotNetType({Namespace=["System"];Name="Single"})
9    let star_t:Type   = TypeApplication((McType({Namespace=["test";"mc"];Name="star"})),[
       int_t;float_t])
10   let pipe_t:Type   = TypeApplication((McType({Namespace=["test";"mc"];Name="pipe"})),[
       int_t;float_t])
11   let comma_id:Id  = {Namespace=["test";"mc"];Name="comma"}
12   let left_id:Id    = {Namespace=["test";"mc"];Name="left"}
13   let right_id:Id  = {Namespace=["test";"mc"];Name="right"}
14   let comma_data:data =
15     {
16       args=[int_t; float_t; ];
17       outputType=star_t;
18     }
19   let left_data:data =
20     {
21       args=[int_t;];
22       outputType=pipe_t;
23     }
24   let right_data:data =
```

```
25          {
26            args=[float_t;];
27            outputType=pipe_t;
28          }
29     let datas = [comma_id,comma_data; left_id,left_data; right_id,right_data]
30     let main = {input=[];output=Tmp(0);premis=[];typemap=Map.empty;side_effect=true}
31     {funcs=Map.empty;lambdas=Map.empty;datas=datas;main=main;assemblies=[]}
32
33   let list_test:fromTypecheckerWithLove =
34     let int_t:Type    = DotNetType({Namespace=["System"];Name="Int32"})
35     let add_t:Type = Arrow(int_t,Arrow(int_t,int_t))
36     let add_id:Id   = {Namespace=["builtin"];Name="add"}
37
38     // data "nil" -> List
39     let list_t:Type  = TypeApplication((McType({Namespace=["mc";"test"];Name="List"})),[
          int_t])
40     let nil_id:Id     = {Namespace=["test";"mc"];Name="nil"}
41     let nil_data:data =
42       {
43         args=[];
44         outputType=list_t;
45       }
46
47     // data Int -> "::" -> List -> List
48     let append_id:Id = {Namespace=["test";"mc"];Name="::"}
49     let append_data:data =
50       {
51         args=[int_t; list_t];
52         outputType=list_t;
53       }
54
55     // length nil -> 0
56     let length_t:Type= Arrow (list_t,int_t)
57     let length_id:Id = {Namespace=["test";"mc"];Name="length"}
58     let length_nil:rule =
59       {
60         side_effect=false
61         input=[Tmp(0)]
62         premis=[Destructor({source=Tmp(0); destructor=nil_id; args=[]})
63                 Literal    ({value=I32(0);dest=Tmp(1)}) ]
64         output=Tmp(1)
65         typemap=Map.ofSeq [Tmp(0),list_t
66                            Tmp(1),int_t ]
67       }
68
69     // length xs -> r
70     // -----------------------------
71     // length x::xs -> add^builtin r 1
72     let length_append:rule =
73       {
74         side_effect=false
75         input=[Tmp(0)]
76         premis=[Destructor({source=Tmp(0); destructor=append_id; args=[Named("x");Named("xs"
      )]})
77                 FuncClosure({func=length_id; dest=Tmp(1)})
78                 ApplicationCall({closure=Tmp(1); argument=Named("xs"); dest=Named("r");
      side_effect=false})
79                 Literal({value=I32(1); dest=Tmp(2)})
80                 DotNetStaticCall({func={Name="+";Namespace=["System";"Int32"]};args=[Named("
      r");Tmp(2)];dest=Tmp(3); side_effect=false})
81                 ]
82         output=Tmp(3)
83         typemap=Map.ofSeq [Tmp(0),list_t
84                            Named("x"),int_t
85                            Named("xs"),list_t
86                            Tmp(1),Arrow(list_t,int_t)
87                            Named("r"),int_t
```

```
88                              Tmp(2),int_t
89                              Tmp(3),int_t ]
90        }
91
92    let datas = [nil_id,nil_data; append_id,append_data]
93    let Funcs = Map.ofSeq <| [length_id,[length_nil;length_append]]
94    let main =
95        {
96          input=[]
97          output=Tmp(7)
98          premis=[ConstructorClosure({func=nil_id;dest=Named("end")})
99                  Literal({value=I32(2);dest=Named("second")})
100                 ConstructorClosure({func=append_id;dest=Tmp(0)})
101                 Application({closure=Tmp(0); argument=Named("second"); dest=Tmp(1)})
102                 Application({closure=Tmp(1); argument=Named("end"); dest=Tmp(2)})
103                 Literal({value=I32(1);dest=Named("first")})
104                 Conditional({left=Named("first");predicate=Less;right=Named("second")})
105                 ConstructorClosure({func=append_id;dest=Tmp(3)})
106                 Application({closure=Tmp(3); argument=Named("first"); dest=Tmp(4)})
107                 Application({closure=Tmp(4); argument=Tmp(2); dest=Tmp(5)})
108                 FuncClosure({func=length_id; dest=Tmp(6)})
109                 ApplicationCall({closure=Tmp(6); argument=Tmp(5); dest=Tmp(7); side_effect=
      false})]
110        typemap=Map.ofSeq [Named("end"),list_t
111                           Named("second"),int_t
112                           Named("first"),int_t
113                           Tmp(0),Arrow(int_t,(Arrow(list_t,list_t)))
114                           Tmp(1),Arrow(list_t,list_t)
115                           Tmp(2),list_t
116                           Tmp(3),Arrow(int_t,(Arrow(list_t,list_t)))
117                           Tmp(4),Arrow(list_t,list_t)
118                           Tmp(5),list_t
119                           Tmp(6),Arrow(list_t,int_t)
120                           Tmp(7),int_t]
121        side_effect=true
122        }
123    {funcs=Funcs;datas=datas;lambdas=Map.empty;main=main;assemblies=[]}
```

## B.8   Balltest.fs

```
1    module balltest
2    open Common
3    open CodegenInterface
4
5    #nowarn "0058" // silences indentation warnings
6
7    let ball_func =
8      let vec2_t:Type = DotNetType({Namespace=["Microsoft";"Xna";"Framework"];Name="Vector2"})
9      let int_t:Type = DotNetType({Namespace=["System"];Name="Int32"})
10     let float_t:Type = DotNetType({Namespace=["System"];Name="Single"})
11     let ball_t:Type = McType({Namespace=["BouncingBall"];Name="Ball"})
12     let ball_id:Id  = {Namespace=["BouncingBall"];Name="ball"}
13
14     let ball_data:data =
15        {
16          args=[vec2_t;vec2_t];
17          outputType=ball_t;
18        }
19
20     let void_t:Type = DotNetType({Name="void";Namespace=[]})
21
22     // update
23     let update_id:Id = {Namespace=["BouncingBall"];Name="update"}
24     let update_t:Type = float_t --> (ball_t --> ball_t)
25     let update_fall_down:rule = {
26       side_effect = false
27       definition  = { File="notreal.mc"; Line=9001; Col=1}
```

```
28      input = [Named("dt");Named("b")]
29      output = Named("out")
30      premis =
31        [
32          // b -> ball(position velocity)
33          Destructor({source=Named("b");destructor=ball_id;args=[Named("position");Named("
        velocity")]}),1
34
35          // gety() -> y
36          DotNetGet({field="Y"
37                      instance = Named("position")
38                      dest=Named("y") }),2
39
40          // y >= 0
41          Literal({value=F32(0.0f);dest=Named("zero")}),3
42          Literal({value=F32(500.0f);dest=Named("ground")}),3
43          Conditional({left=Named("y");predicate=LessEqual;right=Named("ground")}),3
44
45          // Vector2(0,9.81)
46          Literal({value=F32(98.1f);dest=Named("g")}),4
47          DotNetConstructor({func={Namespace=["Microsoft";"Xna";"Framework"];Name="Vector2"}
48                           args=[Named("zero");Named("g")]
49                           dest=Named("v2")
50                           side_effect=false}),4
51
52          // dotproduct
53          DotNetStaticCall({func={Namespace=["Microsoft";"Xna";"Framework";"Vector2"];Name="
        op_Multiply"}
54                           args=[Named("v2");Named("dt")]
55                           dest=Named("outerProduct")
56                           side_effect=false}),5
57
58          // sum
59          DotNetStaticCall({func={Namespace=["Microsoft";"Xna";"Framework";"Vector2"];Name="
        op_Addition"}
60                           args=[Named("velocity");Named("outerProduct")]
61                           dest=Named("updatedVelocity")
62                           side_effect=false}),6
63
64          // dotproduct
65          DotNetStaticCall({func={Namespace=["Microsoft";"Xna";"Framework"];Name="
        op_Multiply"}
66                           args=[Named("updatedVelocity");Named("dt")]
67                           dest=Named("outerProduct2")
68                           side_effect=false}),7
69
70          // sum
71          DotNetStaticCall({func={Namespace=["Microsoft";"Xna";"Framework"];Name="
        op_Addition"}
72                           args=[Named("position");Named("outerProduct2")]
73                           dest=Named("updatedPosition")
74                           side_effect=false}),8
75
76          // Ball (updated_position, updated_velocity)
77          ConstructorClosure({func=ball_id;dest=next_tmp()}),9
78          Application({closure=current_tmp(); argument=Named("updatedPosition"); dest=
        next_tmp()}),9
79          Application({closure=current_tmp(); argument=Named("updatedVelocity"); dest=Named(
        "out")}),9
80        ]
81      typemap=
82        [
83          Named("dt"),float_t
84          Named("b"),ball_t
85          Named("out"),ball_t
86          Named("y"),float_t
87          Named("zero"),float_t
```

```
 88              Named("ground"),float_t
 89              Named("velocity"),vec2_t
 90              Named("position"),vec2_t
 91              Named("g"),float_t
 92              Named("v2"),vec2_t
 93              Named("updatedPosition"),vec2_t
 94              Named("updatedVelocity"),vec2_t
 95              Named("outerProduct"),vec2_t
 96              Named("outerProduct2"),vec2_t
 97              reset_tmp(),(vec2_t --> (vec2_t --> ball_t))
 98              next_tmp(),(vec2_t --> ball_t)
 99          ] |> Map.ofSeq
100     }
101
102     do reset()
103     let update_bounce:rule = {
104       side_effect = false
105       definition  = { File="notreal.mc"; Line=9001; Col=2}
106       input = [Named("dt");Named("b")]
107       output = Named("out")
108       premis =
109         [
110           // b -> ball(position velocity)
111           Destructor({source=Named("b");destructor=ball_id;args=[Named("position");Named("
      velocity")]}),10
112
113           // gety() -> y
114           DotNetGet({field="Y"
115                      instance = Named("position")
116                      dest=Named("y") }),11
117
118           // gety() -> x
119           DotNetGet({field="X"
120                      instance = Named("position")
121                      dest=Named("x") }),12
122
123           // y >= 0
124           Literal({value=F32(500.0f);dest=Named("ground")}),13
125           Conditional({left=Named("y");predicate=Greater;right=Named("ground")}),13
126
127           // Vector2(pos.x,zero)
128           DotNetConstructor({func={Namespace=["Microsoft";"Xna";"Framework"];Name="Vector2"}
129                          args=[Named("x");Named("ground")]
130                          dest=Named("updatedPosition")
131                          side_effect=false}),14
132
133
134           DotNetStaticCall({func={Namespace=["Microsoft";"Xna";"Framework";"Vector2"];Name="
      op_Subtraction"}
135                          args=[Named("velocity")]
136                          dest=Named("updatedVelocity")
137                          side_effect=false}),15
138
139           // Ball (updated_position, updated_velocity)
140           ConstructorClosure({func=ball_id;dest=next_tmp()}),16
141           Application({closure=current_tmp(); argument=Named("updatedPosition"); dest=
      next_tmp()}),16
142           Application({closure=current_tmp(); argument=Named("updatedVelocity"); dest=Named(
      "out")}),16
143         ]
144       typemap=
145         [
146           Named("dt"),float_t
147           Named("b"),ball_t
148           Named("out"),ball_t
149           Named("y"),float_t
150           Named("x"),float_t
```

```
151            Named("ground"),float_t
152            Named("velocity"),vec2_t
153            Named("position"),vec2_t
154            Named("updatedPosition"),vec2_t
155            Named("updatedVelocity"),vec2_t
156            reset_tmp(),(vec2_t --> (vec2_t --> ball_t))
157            next_tmp(),(vec2_t --> ball_t)
158          ] |> Map.ofSeq
159    }
160    let Funcs = Map.ofSeq <| [update_id,([update_fall_down;update_bounce],{File="decl.mc";
         Line=1337;Col=2})]
161    let main = {
162      definition  = { File="notreal.mc"; Line=9001; Col=2}
163      input=[]
164      output=Named("ret")
165      premis=[
166          Literal({dest=Named("x1");value=F32(20.0f)}),100
167          Literal({dest=Named("y1");value=F32(10.0f)}),101
168          DotNetConstructor({dest=Named("a");func={Namespace=["Microsoft";"Xna";"Framework"
         ];Name="Vector2"};args=[Named("x1");Named("y1")];side_effect=false}),102
169          Literal({dest=Named("x2");value=F32(10.0f)}),103
170          Literal({dest=Named("y2");value=F32(30.0f)}),104
171          DotNetConstructor({dest=Named("b");func={Namespace=["Microsoft";"Xna";"Framework"
         ];Name="Vector2"};args=[Named("x2");Named("y2")];side_effect=false}),105
172          DotNetStaticCall({dest=Named("c");func={Namespace=["Microsoft";"Xna";"Framework";"
         Vector2"];Name="op_Addition"};args=[Named("a");Named("b")];side_effect=false}),106
173          DotNetCall({dest=Named("nil");instance=Named("c");func="Normalize";args=[];
         side_effect=false;mutates_instance=true;}),107
174          DotNetSet({src=Named("y2");instance=Named("c");field="X"}),108
175          DotNetGet({dest=Named("ret");instance=Named("c");field="X"}),109
176        ]
177      typemap=Map.ofList <| [
178        Named("x1"),float_t
179        Named("y1"),float_t
180        Named("x2"),float_t
181        Named("y2"),float_t
182        Named("a"),vec2_t
183        Named("b"),vec2_t
184        Named("c"),vec2_t
185        Named("nil"),void_t
186        Named("ret"),float_t
187        ]
188      side_effect=true
189    }
190  {funcs=Funcs;datas=[ball_id,ball_data];lambdas=Map.empty;main=main;assemblies=["
       Microsoft.Xna.Framework.dll"];flags={debug=true}}
```