Bachelor Thesis

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

Douwe van Gijn

supervised by
Dr. Giuseppe Maggiore

2016-05-25

Contents

1 Introduction

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

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

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

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

In this document, we will walk through the 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
```

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.

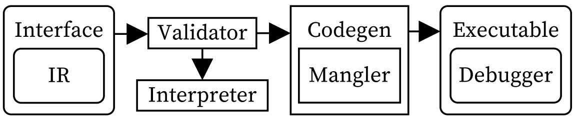
```
        1 ← 1+0
    2 ← 1+1
2
```

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.

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 indentifier may be defined multiple times: once for each type. There is no performance penalty for the back-end, as no lookups by identifier are performed.

Data declarations

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

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

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

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

Where `Type` represents a monomorphic MC type.

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

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

This will appear as the following list in the interface:

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

Rule containers

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

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

Functions in MC can contain multiple rules that implement them.

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

Rules

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

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

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

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

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 #\textit{<function name>}# {
    #\textit{<function arguments>}#
    public #\textit{<return type>}#
    _run(#\textit{<last argument>}#) {
        {
            #\textit{<rule 1 implementation
    >}#
            return #\textit{<local>}#;
        }
      skip1:
        {
            #\textit{<rule 2 implementation
    >}#
            return #\textit{<local>}#;
        }
      skip2:
        #\vdots#
        {
            #\textit{<rule n implementation
    >}#
            return #\textit{<local>}#;
        }
      skip#\textit{n}#:
        throw new #\textit{<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:
```

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

closure	`(+) -> add`	`var add = new _plus();`
application	`add a -> inc`	`var inc = add;` `inc._arg0 = a;`
call	`inc b -> c`	`var c = inc.run(b);`
.NET instr.	**MC**	**C#**
call	`date.toString format -> str`	`var str = date.toString(format);`
static call	`System.DateTime.parse str -> date`	`var date = System.DateTime.parse(str);`
get	`date.DayOfWeek -> day`	`var day = date.DayOfWeek;`
static get	`date.DayOfWeek -> day`	`var day = date.DayOfWeek;`
set	`hr -> System.DateTime.hour`	`System.DateTime.hour = hr;`
static set	`hr -> System.DateTime.hour`	`System.DateTime.hour = hr;`

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.

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.

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

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.

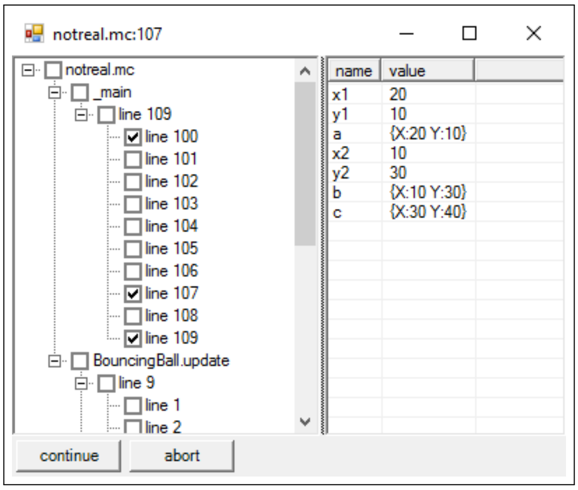
3.7 Debugger

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.

```
...
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.

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.

Two such test programs were written.

4.1 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.

4.2 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

```
module Common

type Position = { File : string; Line : int; Col : int }
  with
    member pos.NextLine = { pos with Line = pos.Line + 1; Col = 1 }
    member pos.NextChar = { pos with Col = pos.Col + 1 }
    static member FromPath(path:string) = { File = path; Line = 1; Col = 1 }
    static member Zero = { File = ""; Line = 1; Col = 1 }

type Bracket = Curly | Round | Square | Lambda | Implicit | Comment

type Predicate = Less | LessEqual | Equal | GreaterEqual | Greater | NotEqual

type genericId<'a>= {Namespace:List<string>;Name:'a;}
type Id       = genericId<string>

type Literal = I64 of System.Int64
             | U64 of System.UInt64
             | I32 of System.Int32
             | U32 of System.Int32
             | F64 of System.Double
             | F32 of System.Single
             | String of System.String
             | Bool of System.Boolean
             | Void
```

B.2 CodegenInterface.fs

```
module CodegenInterface
open Common

type LambdaId = genericId<int>
type TypeId   = genericId<string>

type Type = DotNetType      of TypeId
          | McType          of TypeId
          | TypeApplication of Type*List<Type>
          | Arrow           of Type*Type

type local_id = Named of string
              | Tmp   of int

type premisse = Literal            of LiteralAssignment // assign literal to local
              | Conditional        of Conditional       // stops evaluation if condition is false
              | Destructor         of Destructor        // destructs Mc data into its constructor arguments
              | ConstructorClosure of closure<Id>       // assigns mc data constructor closure to local
              | FuncClosure        of closure<Id>       // assigns mc func closure to local
              | LambdaClosure      of closure<LambdaId> // assigns lambda closure to local
              | Application        of Application       // applies an argument to a local closure
              | ApplicationCall    of ApplicationCall   // applies an argument to a local closure and calls it
              | DotNetCall         of DotNetCall        // calls .Net method and assigns result to local
              | DotNetStaticCall   of DotNetStaticCall  // calls .Net static method and assigns result to local
```

```
              | DotNetConstructor  of DotNetStaticCall  // calls .Net constructor
              | DotNetGet          of DotNetGet         // gets field and assigns it to local
              | DotNetSet          of DotNetSet         // sets field from local
and LiteralAssignment = {value:Literal; dest:local_id}
and Conditional = {left:local_id; predicate:Predicate; right:local_id}
and Destructor  = {source:local_id; destructor:Id; args:List<local_id>}
and closure<'a> = {func:'a;dest:local_id}
and Application = {closure:local_id; argument:local_id; dest:local_id}
and ApplicationCall = {closure:local_id; argument:local_id; dest:local_id; side_effect:bool}
and DotNetStaticCall = {func: Id; args:List<local_id>; dest:local_id; side_effect:bool}
and DotNetCall       = {instance: local_id; func: string; args:List<local_id>; dest:local_id; side_effect:bool; mutates_instance:bool}
and DotNetGet   = {instance: local_id; field: string; dest:local_id}
and DotNetSet   = {instance: local_id; field: string; src:local_id}

type rule = {
  side_effect :bool
  input  :List<local_id>
  output :local_id
  premis :List<premisse*int> // linenumber
  typemap:Map<local_id,Type>
  definition: Position
}

type data = {
  args       :List<Type>
  outputType :Type
}

type CompilerFlags = {debug:bool}

type fromTypecheckerWithLove = {
  assemblies : List<string>
  funcs   : Map<Id,List<rule>*Position>
  lambdas : Map<LambdaId,rule>
  datas   : List<Id*data>
  main    : rule
  flags   : CompilerFlags
}

let (-->) t1 t2 =
  Arrow(t1,t2)

let mutable tmp_index = 0

let next_tmp () =
  let current = tmp_index
  tmp_index <- tmp_index+1
  Tmp(current)

let current_tmp () =
  Tmp(tmp_index-1)

let reset_tmp () =
 do tmp_index <- 1
 Tmp(0)

let reset () =
 do tmp_index <- 0
```

B.3 Codegen.fs

```
module Codegen
open Common
open CodegenInterface
```

```
open Mangle

let mutable flags:CompilerFlags={debug=false};

let fst(x,_)=x
let snd(_,x)=x

let foldi (f:int->'state->'element->'state) (s:'state) (lst:seq<'element>) :'state =
  let fn ((counter:int),(state:'state)) (element:'element) :int*'state =
    counter+1,(f counter state element)
  let _,ret = lst|>Seq.fold fn (0,s)
  ret

let ice () =
    do System.Console.BackgroundColor <- System.ConsoleColor.Red
    do System.Console.Write "INTERNAL␣COMPILER␣ERROR"
    do System.Console.ResetColor()

type NamespacedItem = Ns       of string*List<NamespacedItem>
                    | Data     of string*data
                    | Function of string*List<rule>
                    | Lambda   of int*rule

let construct_tree (input:fromTypecheckerWithLove) :List<NamespacedItem> =
  let rec datatree (s:List<NamespacedItem>) (idx:List<string>,v:string*data) :List<NamespacedItem> =
    match idx with
    | []    -> (Data(v))::s
    | n::ns -> match s |> List.partition (fun x->match x with Ns(n,_)->true | _->false) with
               | [Ns(n,body)],rest -> Ns(n,datatree body (ns,v))::rest
               | [],list           -> Ns(n,datatree []   (ns,v))::list
  let rec functree (s:List<NamespacedItem>) (idx:List<string>,v:string*(List<rule>*Position)) :List<NamespacedItem> =
    match idx with
    | []    -> let (l,(r,_)) = v in (Function(l,r))::s
    | n::ns -> match s |> List.partition (fun x->match x with Ns(n,_)->true | _->false) with
               | [Ns(n,body)],rest -> Ns(n,functree body (ns,v))::rest
               | [],list           -> Ns(n,functree []   (ns,v))::list
  let rec lambdatree (s:List<NamespacedItem>) (idx:List<string>,v:int*rule) :List<NamespacedItem> =
    match idx with
    | []    -> (Lambda(v))::s
    | n::ns -> match s |> List.partition (fun x->match x with Ns(n,_)->true | _->false) with
               | [Ns(n,body)],rest -> Ns(n,lambdatree body (ns,v))::rest
               | [],list           -> Ns(n,lambdatree []   (ns,v))::list
  let go input output state = input |> Seq.map (fun (n,d)->(List.rev n.Namespace),(n.Name,d)) |> Seq.fold output state
  [] |> go (Map.toSeq input.lambdas) lambdatree
     |> go (Map.toSeq input.funcs) functree
     |> go input.datas datatree

let print_literal (lit:Literal) =
  match lit with
  | I64 i    -> sprintf "%dL"  i
  | U64 i    -> sprintf "%uUL" i
  | I32 i    -> sprintf "%u" i
  | U32 i    -> sprintf "%uU" i
  | F64 i    -> sprintf "%f" i
  | F32 i    -> sprintf "%ff" i
  | String s -> sprintf "\"%s\"" s
  | Bool b   -> if b then "true" else "false"
  | Void     -> "void"

let print_predicate (p:Predicate) :string=
```

```
  match p with Less -> "<" | LessEqual -> "<=" | Equal -> "=" | GreaterEqual -> ">=" | Greater -> ">" | NotEqual -> "!="

let field (n:int) (t:Type) :string =
  sprintf "public␣%s␣_arg%d;\n" (mangle_type t) n

let highest_tmp (typemap:Map<local_id,Type>): int =
  typemap |> Map.fold (fun s k _ -> match k with Tmp(x) when x>s -> x | _ -> s) 0

let get_map (a:local_id) (m:Map<local_id,int>) :int*Map<local_id,int> =
  match Map.tryFind a m with None -> 0,(Map.add a 1 m) | Some x -> x,(Map.add a (x+1) m)

let overloadableOps:Map<string,string> =
  [
    "op_Equality","=="
    "op_Inequality","!="
    "op_GreaterThan",">"
    "op_LessThan","<"
    "op_GreaterThanOrEqual",">="
    "op_LessThanOrEqual","<="
    "op_BitwiseAnd","&"
    "op_BitwiseOr","|"
    "op_Addition","+"
    "op_Subtraction","-"
    "op_Division","/"
    "op_Modulus","%"
    "op_Multiply","*"
    "op_LeftShift","<<"
    "op_RightShift",">>"
    "op_ExclusiveOr","^"
    "op_UnaryNegation","-"
    "op_UnaryPlus","+"
    "op_LogicalNot","!"
    "op_OnesComplement","~"
    "op_False","false"
    "op_True","true"
    "op_Increment","++"
    "op_Decrement","--"
  ] |> Map.ofList

let print_label (i:int) = sprintf "_skip%d" i

let print_debug_tree (fromTypeChecker:Map<Id,list<rule>*Position>) (main:rule) =
  let tree =
    // first flatten the hierarchy
    let rules = fromTypeChecker |> Map.toSeq |> Seq.map (fun(name,(lst,pos))->lst|>Seq.ofList|>Seq.map(fun rule->name,rule,pos)) |> Seq.concat
    // add main to the flat list
    let rules = rules |> Seq.append ([{Namespace=[];Name="main"},main,main.definition])
    // then group by file (that the rule is defined by)
    let files = rules |> Seq.groupBy (fun(id,rule,pos)->rule.definition.File)
    // within each file, group by func (and strip info out of rule)
    files |> Seq.map (fun(filename,rules)->
      let funcs = rules |> Seq.groupBy (fun(id,rule,pos)->id,pos)
                  |> Seq.map (fun((id,pos),rules)->id,pos,rules|>Seq.map(fun(id,rule,pos)->rule))
      filename,funcs)
  let debug_tree =
    let per_file = tree  |> Seq.mapi (fun filenr (filename,funcs)->
      let per_func = funcs |> Seq.mapi (fun funcnr (funcname,declposition,rules)->
        let per_rule = rules |> Seq.mapi (fun rulenr (rule) ->
          let per_line = rule.premis |> Seq.groupBy snd
          let per_prem = (per_line(*|>Seq.rev|>Seq.tail|>Seq.rev*)) |> Seq.mapi (fun premnr (linenumber,prems)->
            sprintf "_DBUG.program_tree.Nodes[%d].Nodes[%d].Nodes[%d].Nodes.Add(\"line␣%d\");\n" filenr funcnr rulenr linenumber )
          sprintf "_DBUG.program_tree.Nodes[%d].Nodes[%d].Nodes.Add(\"line␣%d\");\n"
```

```
    filenr funcnr (Seq.last per_line|>fst)
          + (String.concat "" per_prem) )
        sprintf "_DBUG.program_tree.Nodes[%d].Nodes.Add(\"%s\");\n" filenr (mangle_id funcname)
        + (String.concat "" per_rule) )
      sprintf "_DBUG.program_tree.Nodes.Add(\"%s\");\n" filename
      + (String.concat "" per_func) )
    per_file |> String.concat ""
  let breakpoint_tree =
    let per_file = tree |> Seq.map (fun(_,funcs)->
      let per_func = funcs |> Seq.map (fun(id,_,rules)->
        let per_rule = rules |> Seq.mapi (fun i _ ->
          sprintf "%s._DBUG_breakpoints_%d" (mangle_id id) i)
        sprintf "␣␣␣␣␣␣␣␣new␣bool[][]{%s}"  (String.concat "," per_rule) )
      sprintf "␣␣␣␣␣␣new␣bool[][][]{\n%s\n␣␣␣␣␣␣}"  (String.concat ",\n" per_func) )
    sprintf "␣␣␣␣new␣bool[][][][]{\n%s\n␣␣␣␣};\n" (String.concat "," per_file)
  ("_DBUG.breakpoints="+breakpoint_tree)+debug_tree

let premisse (p:premisse) (m:Map<local_id,Type>) (app:Map<local_id,int>) (rule_nr:int) =
  match p with
  | Literal x -> app,sprintf "/*LITR*/var␣%s␣=␣%s;\n"
                   (mangle_local_id x.dest)
                   (print_literal x.value)
  | Conditional x -> app,sprintf "/*COND*/if(!(%s␣%s␣%s)){goto␣%s;}\n"
                       (mangle_local_id x.left)
                       (print_predicate x.predicate)
                       (mangle_local_id x.right)
                       (print_label rule_nr)
  | Destructor x ->
    let new_id  = (Tmp(1+(highest_tmp m)))
    app,sprintf "/*DTOR*/var␣%s␣=␣%s␣as␣%s;\nif(%s==null){goto␣%s;}\n%s"
      (mangle_local_id new_id)
      (mangle_local_id x.source)
      (mangle_id       x.destructor)
      (mangle_local_id new_id)
      (print_label rule_nr)
      (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 "")
  | ConstructorClosure x
  | FuncClosure x -> (app|>Map.add x.dest 0),sprintf "/*FUNC*/var␣%s␣=␣new␣%s();\n"
                       (mangle_local_id x.dest)
                       (mangle_id x.func)
  | LambdaClosure x -> (app|>Map.add x.dest 0),sprintf "/*LAMB*/var␣%s␣=␣new␣%s();\n"
                         (mangle_local_id x.dest)
                         (mangle_lambda x.func)
  | DotNetCall x ->
    let isVoid = m.[x.dest]=Type.DotNetType({Namespace=[];Name="void"})
    app,sprintf  "/*NDCA*/%s%s.%s(%s);\n"
      (if isVoid then "" else sprintf "var␣%s␣=␣" (mangle_local_id x.dest))
      (mangle_local_id x.instance)
      x.func
      (x.args |> List.map mangle_local_id|>String.concat ",")
  | DotNetStaticCall x ->
        let isVoid = m.[x.dest]=Type.DotNetType({Namespace=[];Name="void"})
        if overloadableOps.ContainsKey(x.func.Name) then
          let args = x.args |> List.rev
          app,sprintf "/*NSCA*/%s%s␣%s␣%s;\n"
            (if isVoid then "" else sprintf "var␣%s␣=␣" (mangle_local_id x.dest))
            (match x.args.Length with
             | 1 -> ""
             | 2 -> mangle_local_id args.[1])
            overloadableOps.[x.func.Name]
            (mangle_local_id args.[0])
        else
          app,sprintf "/*NSCA*/%s%s(%s);\n"
            (if isVoid then "" else sprintf "var␣%s␣=␣" (mangle_local_id x.dest))
            (x.func.Namespace@[x.func.Name]|>String.concat ".")
```

```
            (x.args |> List.map mangle_local_id|>String.concat ",")
  | DotNetConstructor x -> app,sprintf "/*NCON*/var␣%s␣=␣new␣%s(%s);\n"
                             (mangle_local_id x.dest)
                             (x.func.Namespace@[x.func.Name]|>String.concat ".")
                             (x.args |> List.map mangle_local_id|>String.concat ",")
  | DotNetGet x -> app,sprintf "/*NGET*/var␣%s␣=␣%s.%s;\n"
                     (mangle_local_id x.dest)
                     (mangle_local_id x.instance)
                     x.field
  | DotNetSet x -> app,sprintf "/*NSET*/%s.%s␣=␣%s;\n"
                     (mangle_local_id x.instance)
                     x.field
                     (mangle_local_id x.src)
  | Application x ->
    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))
    (app|>Map.add x.dest (i+1)),sprintf "/*APPL*/var␣%s␣=␣%s;␣%s.%s=%s;\n"
                       (mangle_local_id  x.dest)
                       (mangle_local_id  x.closure)
                       (mangle_local_id  x.dest)
                       (sprintf "_arg%d" i)
                       (mangle_local_id  x.argument)
  | ApplicationCall x ->
    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))
    (app|>Map.add x.dest (i+1)),sprintf "/*CALL*/%s.%s=%s;␣var␣%s␣=␣%s._run();\n"
      (mangle_local_id  x.closure)
      (sprintf "_arg%d" i)
      (mangle_local_id  x.argument)
      (mangle_local_id  x.dest)
      (mangle_local_id  x.closure)

let get_definitions (p:premisse) :List<local_id> =
  match p with
  | Literal            x -> [x.dest]
  | Conditional        _ -> []
  | Destructor         x ->  x.args
  | ConstructorClosure x -> [x.dest]
  | FuncClosure        x -> [x.dest]
  | LambdaClosure      x -> [x.dest]
  | Application        x -> [x.dest]
  | ApplicationCall    x -> [x.dest]
  | DotNetCall         x -> [x.dest]
  | DotNetStaticCall   x -> [x.dest]
  | DotNetConstructor  x -> [x.dest]
  | DotNetGet          x -> [x.dest]
  | DotNetSet          x -> [x.src]

let print_rule (rule_nr:int) (rule:rule) =
  let symbol_table_add id =
    let s=mangle_local_id id
    if id=Named("nil") then "" else sprintf "_DBUG_symbol_table[\"%s\"]=%s;\n" s s
  let linegroups:seq<int*seq<premisse>> =
    rule.premis |> Seq.groupBy (fun(_,x)->x) |> Seq.map (fun(l,p)->l,(p|>Seq.map(fun(x,_)->x)))
  let fn (app:Map<local_id,int>,str:string) (p:premisse) =
    let (a:Map<local_id,int>,s:string) =
      let l,r = premisse p rule.typemap app rule_nr
      if flags.debug then
        let stab_adds = p|>get_definitions|>List.map symbol_table_add |> String.concat ""
        l,(r+stab_adds)
      else l,r
    a,(str+s)
  let lines =
    linegroups |> Seq.mapi
      (fun idx (linenumber,premisses)->
        let breakpoint =
```

```
          if flags.debug then
            sprintf "if(_DBUG_breakpoints_%d[%d]){_DBUG.breakpoint(\"%s\",%d,_DBUG_symbol_table);}\n" rule_nr idx rule.definition.File linenumber
          else ""
        let _,s = ((Map.empty,""),premisses) ||> Seq.fold fn
        s+breakpoint)
  sprintf "{\n%s%s%sreturn %s;}\n%s:\n"
    (if flags.debug then "var _DBUG_symbol_table = new System.Collections.Generic.Dictionary<string, object>();\n" else "")
    (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 "")
    (lines|>String.concat "")
    (mangle_local_id rule.output)
    (print_label rule_nr)

let nr_of_actual_lines (rule:rule):int =
  rule.premis |> Seq.map snd |> Seq.distinct |> Seq.length

let print_rule_bodies (rules:rule list) =
  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
  len |> List.mapi (fun x (a,b)->print_rule x a) |> String.concat ""

let generate_breakpoint_def (funcname:string) (rules:rule seq) =
  let linenumbers = rules |> Seq.map (fun x->x.premis) |> Seq.map ((Seq.map snd)>>Seq.distinct)
  let subarrays = linenumbers |> Seq.mapi (fun i x->
    let s =
      if funcname="_main" then
        x |>Seq.mapi (fun i _->if i=0 then "true" else "false") |> String.concat ","
      else
        x|>Seq.map (fun _->"false") |> String.concat ","
    sprintf "%s._DBUG_breakpoints_%d = new bool[]{%s};\n" funcname i s)
  subarrays |> String.concat ""

let generate_breakpoint_decl (rules:rule seq) = rules |> Seq.mapi (fun i _-> sprintf "public static bool[] _DBUG_breakpoints_%d;\n" i) |> String.concat ""

let print_main (input:fromTypecheckerWithLove) =
  let rule = input.main
  let return_type = mangle_type rule.typemap.[rule.output]
  let body = sprintf "static %s body(){\n%sthrow new System.MissingMethodException();\n}" return_type (print_rule_bodies [rule])
  let debug_init =
    if flags.debug then
      let foo = input.funcs |> Map.toSeq |> Seq.map (fun(k,(rules,pos))-> generate_breakpoint_def (mangle_id k) rules) |> String.concat ""
      foo+(generate_breakpoint_def "_main" [rule])+(print_debug_tree input.funcs input.main)
    else ""
  let main = sprintf "static void Main() {\n%s\nSystem.Console.WriteLine(System.String.Format(\"{0}\", body()));\n}" debug_init
  sprintf "class _main{\n%s%s%s}\n" (if flags.debug then generate_breakpoint_decl [rule] else "") body main

let rec print_tree (lookup:fromTypecheckerWithLove) (ns:List<NamespacedItem>) :string =
  let build_func (name:string) (rules:rule list) =
    let breakpoints = if flags.debug then generate_breakpoint_decl rules else ""
    let rule = List.head rules
    let args = rule.input |> Seq.mapi (fun nr id-> sprintf "public %s _arg%d;\n" (mangle_type rule.typemap.[id]) nr) |> String.concat ""
    let ret_type = mangle_type rule.typemap.[rule.output]
    let rules = print_rule_bodies rules
    sprintf "class %s{\n%s%spublic %s _run(){\n%sthrow new System.MissingMethodException();\n}\n}\n" (CSharpMangle name) args breakpoints ret_type rules
  let print_base_types (ns:List<NamespacedItem>) =
    let types = ns |> List.fold (fun types item -> match item with Data (_,v) -> v.
```

```
    outputType::types | _ -> types) [] |> List.distinct
    let print t = sprintf "public␣class␣%s{}\n" (t|>remove_namespace_of_type|>mangle_type)
    List.map print types
  let go ns =
    match ns with
    | Ns(n,ns)      -> sprintf "namespace␣%s{\n%s}\n" (CSharpMangle n) (print_tree lookup ns)
    | 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 "")
    | Function(name,rules) -> build_func name rules
    | Lambda(number,rule)  -> build_func (sprintf "_lambda%d" number) [rule]
  (print_base_types ns)@(ns|>List.map go)|>String.concat "\n"

let get_locals (ps:(premisse*int) list) :local_id list =
  ps |> List.collect (fun (p,_) ->
    match p with
    | Literal              x -> [x.dest]
    | Conditional          x -> [x.left;x.right]
    | Destructor           x -> x.source::x.args
    | LambdaClosure        x -> [x.dest]
    | FuncClosure          x -> [x.dest]
    | DotNetCall           x -> [x.dest]
    | DotNetStaticCall     x -> [x.dest]
    | DotNetConstructor    x -> [x.dest]
    | DotNetGet            x -> [x.dest]
    | DotNetSet            x -> [x.src]
    | ConstructorClosure   x -> [x.dest]
    | Application          x -> [x.dest]
    | ApplicationCall      x -> [x.closure;x.dest;x.argument] )

let validate (input:fromTypecheckerWithLove) :bool =
  let print_local_id (id:local_id) = match id with Named(x)->x | Tmp(x)->sprintf "temporary(%d)" x
  let print_id (id:Id) = String.concat "^" (id.Name::id.Namespace)
  let check_typemap (id:Id) (rule:rule) :bool =
    let expected = (get_locals rule.premis) @ (rule.output::rule.input) |> List.distinct |> List.sort
    let received  = rule.typemap |> Map.toList |> List.map (fun (x,_)->x) |> List.sort
    if expected = received then true
    else
      let missing = expected |> List.filter (fun x -> received |> List.exists (fun y->x=y) |> not)
      let extra   = received |> List.filter (fun x -> expected |> List.exists (fun y->x=y) |> not)
      do ice()
      do printf "␣incorrect␣typemap␣in␣rule␣%s:\n␣␣missing␣%A\n␣␣extra:␣%A\n" (print_id id) missing extra
      false
  let check_dest_constness (id:Id) (rule:rule) (success:bool):bool =
      let per_premisse (statementnr:int) (set:Set<local_id>,success:bool) (premisse:premisse,i:int) :Set<local_id>*bool =
        let check (set:Set<local_id>,success:bool) (local:local_id) =
          if set.Contains(local) then
            do ice()
            do printf "␣%s␣assigned␣twice␣in␣rule␣%s,␣statement␣%d␣on␣line␣%d\n" (print_local_id local) (print_id id) statementnr i
            set,false
          else set.Add(local),success
        match premisse with
        | Literal x              -> check (set,success) x.dest
        | Conditional _          -> set,success
        | Destructor x           -> x.args |> Seq.fold check (set,success)
        | ConstructorClosure x   -> check (set,success) x.dest
        | FuncClosure  x         -> check (set,success) x.dest
        | LambdaClosure x        -> check (set,success) x.dest
        | DotNetCall x           -> check (set,success) x.dest
        | DotNetStaticCall x     -> check (set,success) x.dest
```

```
        | DotNetConstructor x     -> check (set,success) x.dest
        | DotNetGet x             -> check (set,success) x.dest
        | DotNetSet x             -> set,success
        | Application x           -> check (set,success) x.dest
        | ApplicationCall x       -> check (set,success) x.dest
      let _,ret = rule.premis |> foldi per_premisse (Set.empty,success)
      ret
  (true,input.funcs) ||> Map.fold (fun (success:bool) (id:Id) (rules,position)->
    if rules.IsEmpty then
      do ice()
      do printf " empty rule: %s\n" (print_id id)
      false
    else
      (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))

let failsafe_codegen(input:fromTypecheckerWithLove) :Option<string>=
  do flags <- input.flags
  if validate input then
    let foo = input |> construct_tree |> print_tree input
    let dbug = if flags.debug then System.IO.File.ReadAllText("_DBUG.cs.txt") else ""
    dbug+foo+(print_main input) |> Some

  else None
```

B.4 Balltest.fs

```
module balltest
open Common
open CodegenInterface

#nowarn "0058" // silences indentation warnings

let ball_func =
  let vec2_t:Type = DotNetType({Namespace=["Microsoft";"Xna";"Framework"];Name="Vector2"})
  let int_t:Type = DotNetType({Namespace=["System"];Name="Int32"})
  let float_t:Type = DotNetType({Namespace=["System"];Name="Single"})
  let ball_t:Type = McType({Namespace=["BouncingBall"];Name="Ball"})
  let ball_id:Id  = {Namespace=["BouncingBall"];Name="ball"}

  let ball_data:data =
    {
      args=[vec2_t;vec2_t];
      outputType=ball_t;
    }

  let void_t:Type = DotNetType({Name="void";Namespace=[]})

  // update
  let update_id:Id = {Namespace=["BouncingBall"];Name="update"}
  let update_t:Type = float_t --> (ball_t --> ball_t)
  let update_fall_down:rule = {
    side_effect = false
    definition  = { File="notreal.mc"; Line=9001; Col=1}
    input = [Named("dt");Named("b")]
    output = Named("out")
    premis =
      [
        // b -> ball(position velocity)
        Destructor({source=Named("b");destructor=ball_id;args=[Named("position");Named("velocity")]}),1

        // gety() -> y
        DotNetGet({field="Y"
                   instance = Named("position")
                   dest=Named("y") }),2

```

```
      // y >= 0
      Literal({value=F32(0.0f);dest=Named("zero")}),3
      Literal({value=F32(500.0f);dest=Named("ground")}),3
      Conditional({left=Named("y");predicate=LessEqual;right=Named("ground")}),3

      // Vector2(0,9.81)
      Literal({value=F32(98.1f);dest=Named("g")}),4
      DotNetConstructor({func={Namespace=["Microsoft";"Xna";"Framework"];Name="Vector2"}
                         args=[Named("zero");Named("g")]
                         dest=Named("v2")
                         side_effect=false}),4

      // dotproduct
      DotNetStaticCall({func={Namespace=["Microsoft";"Xna";"Framework";"Vector2"];Name="op_Multiply"}
                        args=[Named("v2");Named("dt")]
                        dest=Named("outerProduct")
                        side_effect=false}),5

      // sum
      DotNetStaticCall({func={Namespace=["Microsoft";"Xna";"Framework";"Vector2"];Name="op_Addition"}
                        args=[Named("velocity");Named("outerProduct")]
                        dest=Named("updatedVelocity")
                        side_effect=false}),6

     // dotproduct
      DotNetStaticCall({func={Namespace=["Microsoft";"Xna";"Framework"];Name="op_Multiply"}
                        args=[Named("updatedVelocity");Named("dt")]
                        dest=Named("outerProduct2")
                        side_effect=false}),7

      // sum
      DotNetStaticCall({func={Namespace=["Microsoft";"Xna";"Framework"];Name="op_Addition"}
                        args=[Named("position");Named("outerProduct2")]
                        dest=Named("updatedPosition")
                        side_effect=false}),8

      // Ball (updated_position, updated_velocity)
      ConstructorClosure({func=ball_id;dest=next_tmp()}),9
      Application({closure=current_tmp(); argument=Named("updatedPosition"); dest=next_tmp()}),9
      Application({closure=current_tmp(); argument=Named("updatedVelocity"); dest=Named("out")}),9
    ]
  typemap=
    [
      Named("dt"),float_t
      Named("b"),ball_t
      Named("out"),ball_t
      Named("y"),float_t
      Named("zero"),float_t
      Named("ground"),float_t
      Named("velocity"),vec2_t
      Named("position"),vec2_t
      Named("g"),float_t
      Named("v2"),vec2_t
      Named("updatedPosition"),vec2_t
      Named("updatedVelocity"),vec2_t
      Named("outerProduct"),vec2_t
      Named("outerProduct2"),vec2_t
      reset_tmp(),(vec2_t --> (vec2_t --> ball_t))
      next_tmp(),(vec2_t --> ball_t)
    ] |> Map.ofSeq
}
```

```
do reset()
let update_bounce:rule = {
  side_effect = false
  definition  = { File="notreal.mc"; Line=9001; Col=2}
  input = [Named("dt");Named("b")]
  output = Named("out")
  premis =
    [
      // b -> ball(position velocity)
      Destructor({source=Named("b");destructor=ball_id;args=[Named("position");Named("velocity")]}),10

      // gety() -> y
      DotNetGet({field="Y"
                 instance = Named("position")
                 dest=Named("y") }),11

      // gety() -> x
      DotNetGet({field="X"
                 instance = Named("position")
                 dest=Named("x") }),12

      // y >= 0
      Literal({value=F32(500.0f);dest=Named("ground")}),13
      Conditional({left=Named("y");predicate=Greater;right=Named("ground")}),13

      // Vector2(pos.x,zero)
      DotNetConstructor({func={Namespace=["Microsoft";"Xna";"Framework"];Name="Vector2"}
                         args=[Named("x");Named("ground")]
                         dest=Named("updatedPosition")
                         side_effect=false}),14


      DotNetStaticCall({func={Namespace=["Microsoft";"Xna";"Framework";"Vector2"];Name="op_Subtraction"}
                        args=[Named("velocity")]
                        dest=Named("updatedVelocity")
                        side_effect=false}),15

      // Ball (updated_position, updated_velocity)
      ConstructorClosure({func=ball_id;dest=next_tmp()}),16
      Application({closure=current_tmp(); argument=Named("updatedPosition"); dest=next_tmp()}),16
      Application({closure=current_tmp(); argument=Named("updatedVelocity"); dest=Named("out")}),16
    ]
  typemap=
    [
      Named("dt"),float_t
      Named("b"),ball_t
      Named("out"),ball_t
      Named("y"),float_t
      Named("x"),float_t
      Named("ground"),float_t
      Named("velocity"),vec2_t
      Named("position"),vec2_t
      Named("updatedPosition"),vec2_t
      Named("updatedVelocity"),vec2_t
      reset_tmp(),(vec2_t --> (vec2_t --> ball_t))
      next_tmp(),(vec2_t --> ball_t)
    ] |> Map.ofSeq
}
let Funcs = Map.ofSeq <| [update_id,([update_fall_down;update_bounce],{File="decl.mc";Line=1337;Col=2})]
let main = {
  definition  = { File="notreal.mc"; Line=9001; Col=2}
```

```
    input=[]
    output=Named("ret")
    premis=[
        Literal({dest=Named("x1");value=F32(20.0f)}),100
        Literal({dest=Named("y1");value=F32(10.0f)}),101
        DotNetConstructor({dest=Named("a");func={Namespace=["Microsoft";"Xna";"Framework"];Name="Vector2"};args=[Named("x1");Named("y1")];side_effect=false}),102
        Literal({dest=Named("x2");value=F32(10.0f)}),103
        Literal({dest=Named("y2");value=F32(30.0f)}),104
        DotNetConstructor({dest=Named("b");func={Namespace=["Microsoft";"Xna";"Framework"];Name="Vector2"};args=[Named("x2");Named("y2")];side_effect=false}),105
        DotNetStaticCall({dest=Named("c");func={Namespace=["Microsoft";"Xna";"Framework";"Vector2"];Name="op_Addition"};args=[Named("a");Named("b")];side_effect=false}),106
        DotNetCall({dest=Named("nil");instance=Named("c");func="Normalize";args=[];side_effect=false;mutates_instance=true;}),107
        DotNetSet({src=Named("y2");instance=Named("c");field="X"}),108
        DotNetGet({dest=Named("ret");instance=Named("c");field="X"}),109
      ]
    typemap=Map.ofList <| [
      Named("x1"),float_t
      Named("y1"),float_t
      Named("x2"),float_t
      Named("y2"),float_t
      Named("a"),vec2_t
      Named("b"),vec2_t
      Named("c"),vec2_t
      Named("nil"),void_t
      Named("ret"),float_t
     ]
    side_effect=true
   }
  {funcs=Funcs;datas=[ball_id,ball_data];lambdas=Map.empty;main=main;assemblies=["Microsoft.Xna.Framework.dll"];flags={debug=true}}
```

B.5 Mangle.fs

```
module Mangle
open Common
open CodegenInterface

let genericMangle (name:string) :string =
  let readables =
    Map.ofArray <| Array.zip "!#$%&'*+,-./\\:;<>=?@^_`|~"B [|
      "bang";"hash";"cash";"perc";"amp";"prime";"star";"plus";"comma";
      "dash"; "dot";"slash";"back";"colon";"semi";"less";"great";
      "equal";"quest";"at";"caret";"under";"tick";"pipe";"tilde"|]
  let mangleChar c =
    if (c>='A'&&c<='Z') || (c>='a'&&c<='z') || (c>='0'&&c<='9') then
      sprintf "%c" c
    else
      let lookup = readables |> Map.tryFind (System.Convert.ToByte c)
      match lookup with
      | None   ->  failwith <| sprintf "ERROR(codegen): expecting printable ASCII character, got (0x%04X)" (System.Convert.ToUInt16(c))
      | Some x -> sprintf "_%s" x
  name |> String.collect mangleChar

let CSharpMangle (name:string) :string =
  let keywords = Set.ofArray [| "abstract" ; "as" ; "base" ; "bool" ;
    "break" ; "byte" ; "case" ; "catch" ; "char" ; "checked" ; "class" ; "const" ;
    "continue" ; "decimal" ; "default" ; "delegate" ; "do" ; "double" ; "else" ;
    "enum" ; "event" ; "explicit" ; "extern" ; "false" ; "finally" ; "fixed" ;
    "float" ; "for" ; "foreach" ; "goto" ; "if" ; "implicit" ; "in" ; "int" ;
    "interface" ; "internal" ; "is" ; "lock" ; "long" ; "namespace" ; "new" ;
    "null" ; "object" ; "operator" ; "out" ; "override" ; "params" ; "private" ;
    "protected" ; "public" ; "readonly" ; "ref" ; "return" ; "sbyte" ; "sealed" ;
    "short" ; "sizeof" ; "stackalloc" ; "static" ; "string" ; "struct" ;
```

```
    "switch" ; "this" ; "throw" ; "true" ; "try" ; "typeof" ; "uint" ; "ulong" ;
    "unchecked" ; "unsafe" ; "ushort" ; "using" ; "virtual" ; "void" ;
    "volatile" ; "while" |]
  let name = genericMangle name
  if keywords.Contains(name) then sprintf "@%s" name else name

let rec mangle_type_suffix(t:Type):string=
  match t with
  | DotNetType (id) -> id.Namespace@[id.Name] |> Seq.map genericMangle |> String.concat "_ns"
  | McType     (id) -> id.Namespace@[id.Name] |> Seq.map genericMangle |> String.concat "_ns"
  | TypeApplication (fn,lst) -> (mangle_type_suffix fn)+"_of"+(lst |> List.map mangle_type_suffix |> String.concat "_t")

let rec remove_namespace_of_type(t:Type):Type=
  match t with
  | DotNetType (id) -> DotNetType({id with Namespace=[]})
  | McType     (id) -> McType({id with Namespace=[]})
  | TypeApplication (fn,lst) -> TypeApplication((remove_namespace_of_type fn),lst)

let rec mangle_type(t:Type):string=
  match t with
  | DotNetType (id) -> id.Namespace@[id.Name] |> Seq.map CSharpMangle |> String.concat "."
  | McType     (id) -> id.Namespace@[id.Name] |> Seq.map (fun x->if x="System" then "_System" else CSharpMangle x) |> String.concat "."
  | TypeApplication (fn,lst) -> (mangle_type fn)+"_of"+(lst|>List.map mangle_type_suffix|>String.concat "_t")

let mangle_local_id n = match n with Named x -> CSharpMangle x | Tmp x -> sprintf "_tmp%d" x
let mangle_id (id:Id) =
  if id.Namespace =[] && id.Name="main" then "_main"
  else (id.Name::id.Namespace) |> List.rev |> List.map (fun x->if x="System" then "_System" else CSharpMangle x) |> String.concat "."
let mangle_lambda (id:LambdaId) = sprintf "%s._lambda%d" (id.Namespace|>List.rev|>String.concat ".") id.Name
```

B.6 _DBUG.cs

```
struct _DBUG {
  public static System.Windows.Forms.TreeView program_tree = new System.Windows.Forms.TreeView();
  public static bool[][][][] breakpoints;
  public static void breakpoint(string filename, int line, System.Collections.Generic.Dictionary<string, object> stab) {
    // table
    var table = new System.Windows.Forms.ListView();
    table.GridLines = true;
    table.Dock = System.Windows.Forms.DockStyle.Fill;
    table.View = System.Windows.Forms.View.Details;
    table.Columns.Add("name", -2, System.Windows.Forms.HorizontalAlignment.Left);
    table.Columns.Add("value", -2, System.Windows.Forms.HorizontalAlignment.Left);
    foreach (var x in stab) {
      table.Items.Add(new System.Windows.Forms.ListViewItem(new string[] { x.Key, System.String.Format("{0}", x.Value) }));
    }

    var tablepanel = new System.Windows.Forms.Panel();
    tablepanel.Dock = System.Windows.Forms.DockStyle.Fill;
    tablepanel.Controls.Add(table);

    // tree
    program_tree.CheckBoxes = true;
    program_tree.Dock = System.Windows.Forms.DockStyle.Fill;
    program_tree.ExpandAll();
    for (int filenr = 0; filenr<breakpoints.Length; filenr++) {
```

```
  for (int funcnr = 0; funcnr < breakpoints[filenr].Length; funcnr++) {
    for (int rulenr = 0; rulenr < breakpoints[filenr][funcnr].Length; rulenr++) {
      for (int premnr = 0; premnr < breakpoints[filenr][funcnr][rulenr].Length; premnr++) {
        program_tree.Nodes[filenr].Nodes[funcnr].Nodes[rulenr].Nodes[premnr].Checked = breakpoints[filenr][funcnr][rulenr][premnr];
      }
    }
  }
}

var treepanel = new System.Windows.Forms.Panel();
treepanel.Dock = System.Windows.Forms.DockStyle.Fill;
treepanel.Controls.Add(program_tree);

// split
var split = new System.Windows.Forms.SplitContainer();
split.SplitterDistance = 100;
split.Dock = System.Windows.Forms.DockStyle.Fill;
split.Panel1.Controls.Add(treepanel);
split.Panel2.Controls.Add(tablepanel);

var splitform = new System.Windows.Forms.Form();
splitform.Controls.Add(split);
splitform.MinimumSize = new System.Drawing.Size(100, 100);
splitform.Text = filename + ":" + line;

var buttons = new System.Windows.Forms.Button[4];
buttons[0] = new System.Windows.Forms.Button();
buttons[0].Text = "continue";
buttons[0].DialogResult = System.Windows.Forms.DialogResult.Ignore;
buttons[1] = new System.Windows.Forms.Button();
buttons[1].Text = "abort";
buttons[1].DialogResult = System.Windows.Forms.DialogResult.Abort;
/*buttons[2] = new System.Windows.Forms.Button();
buttons[2].Text = "step in";
buttons[2].DialogResult = System.Windows.Forms.DialogResult.OK;
buttons[3] = new System.Windows.Forms.Button();
buttons[3].Text = "step over";
buttons[3].DialogResult = System.Windows.Forms.DialogResult.Yes;*/

var collection = new System.Windows.Forms.FlowLayoutPanel();
collection.Dock = System.Windows.Forms.DockStyle.Bottom;
collection.Controls.AddRange(buttons);
collection.AutoSize = true;
collection.AutoSizeMode = System.Windows.Forms.AutoSizeMode.GrowAndShrink;

splitform.Controls.Add(collection);

var result = splitform.ShowDialog();

for (int filenr = 0; filenr<breakpoints.Length; filenr++) {
  for (int funcnr = 0; funcnr < breakpoints[filenr].Length; funcnr++) {
    for (int rulenr = 0; rulenr < breakpoints[filenr][funcnr].Length; rulenr++) {
      for (int premnr = 0; premnr < breakpoints[filenr][funcnr][rulenr].Length; premnr++) {
        breakpoints[filenr][funcnr][rulenr][premnr] = program_tree.Nodes[filenr].Nodes[funcnr].Nodes[rulenr].Nodes[premnr].Checked;
      }
    }
  }
}

switch (result) {
  /*case System.Windows.Forms.DialogResult.OK: // step in
    return;
  case System.Windows.Forms.DialogResult.Yes: // step over
```

```
        return; */
      case System.Windows.Forms.DialogResult.Ignore: // continue
        return;
      case System.Windows.Forms.DialogResult.Abort:
        System.Environment.Exit(0);
        return;
    }
  }
}
```