UCLouvain

École polytechnique de Louvain

Reasoning over generated code

Author: **Antoine Buysse**
Supervisors: **Johan Fabry, Kim Mens**
Reader: **Céline Deknop**
Academic year 2022–2023
Master [120] in Computer Science and Engineering

Abstract

This thesis concerns the calls to the runtime used by the company Raincode in their compilers. Their compiler, written in YAFL, contains data structures called FunctionCalls. These are used by the compiler developers, to access C# libraries at runtime of the compiled code. The structure provides all the information related to the called function.
Human errors can be introduced during compiler development, leading to crash at compilation time of the compiler, or worse: crashes of client code that was compiled with Raincode's compiler.
The thesis work generates the FunctionCall usage and adapts the YAFL implementation to call this generated code. The generation represents the core part of this work, and the refactoring redirects the calls to the newly created FunctionCallBuilder, which defines the produced FunctionCall structure.
The general solution can produce a basic structure, based on the signature of the library function required. More iterations on the solution allow to include variations, that we call patterns. Each pattern requires further information conveyed to the generation program for an accurate generation.
The work is open, as many legacy FunctionCalls still reside in the YAFL code. This thesis is the starting point in order to provide an easier development and maintenance of the FunctionCalls.

Remerciements

Je tiens tout d'abord à remercier mes superviseurs : M. Kim MENS, professeur en sciences informatiques à l'Université catholique de Louvain et M. Johan FABRY, ingénieur logiciel senior chez Raincode, pour leur suivi et leur guidance tout au long de cette année. En particulier, leurs conseils et leurs critiques constructives m'ont aidé à rédiger et ajuster la direction de ce mémoire.

Je remercie également Mme Ynès JARADIN, ingénieure logiciel senior chez Raincode, pour m'avoir fait découvrir les outils nécessaires à la réalisation de ce mémoire. Sa disponibilité et son aide m'ont fourni les pistes utiles pour avancer dans mon travail.

Je tiens aussi à remercier toute l'équipe de Raincode, ainsi que M. Darius BLASBAND, le créateur du compilateur YAFL, pour leur accueil et leur soutien.

Contents

Convention

Several diagrams will be presented in the following chapters. Table 1 describes the conventions used in the diagrams:

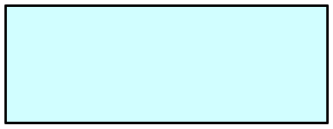
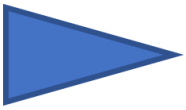
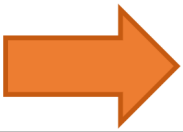
Element	Representation
	Box represents a file
	Oval represents a process
	Triangle represents process flow
	Arrow represents a reference or a call

Table 1: Convention for diagram elements

Inside the text, some words will present a first letter upper case even though the convention would require it to be lower case. This is intentional: the lower case is the general representation of the word, the upper case is Raincode's pendant. For example, "variable" represents a variable in the computer science sense, while "Variable" represents the Variable element as defined in the DotNet module of Raincode's compilers.

Chapter 1

Presentation of thesis

In this part we present Raincode, the company that proposed this thesis, and their compiler. We provide some context on the problems that the current implementation of this compiler poses us, the ideas behind this thesis proposal and finally the proposed solution.

1.1 Preamble

This thesis about "reasoning over generated code" was proposed by Raincode. The company Raincode is located in Brussels, Belgium and specializes in mainframe migration and modernization solutions. The products and services they offer help organizations move their legacy mainframe applications to modern platforms. The legacy source languages they are targeting (COBOL, PL/I, and other mainframe languages) are converted into .dll files executable on machines with a Microsoft Windows operating system.

The compilers of Raincode are vital for the company as their core business revolves around them: to Raincode, they are the product they offer together with additional tools.

The compilers themselves are written in YAFL, a language developed by the founder and CEO of Raincode, Darius Blasband. The YAFL compiler itself exists since 1991.

To better understand the current state of the YAFL compiler, we also have to take a look at some of the source languages targeted:

COBOL (COmmon Business-Oriented Language [1]) is a language used for business applications, as the name implies, as well as for administrative systems. It was one of the first programming language designed for commercial applications, in the late 1950s. Nowadays, COBOL is still used, especially in legacy systems, but is nowhere as popular as it once was, given the rise in popularity of more modern programming languages. The COBOL language characteristics include the use of an English-like syntax, that makes it easier to read and understand for people less versed in programming. It is mainly used to handle large amounts of data, especially when dealing with financial transactions, inventory management, or even payroll systems. Clients with COBOL code are therefore often in charge of sensitive data.

PL/I (Programming Language One [2]), developed in the 1960s by IBM, followed a similar evolution: it was specialized in scientific, business, and systems programming applications but declined in popularity with the advent of C and Java notably.

Even though these languages are less used nowadays, as mentioned, the maintenance of a correct compilation for the programs written with it remains vital in order to keep those programs executable on more recent generations of computers.

The code of the YAFL compiler being of old age as well is not a problem by itself. But as we are going to argue hereunder, the massive amount of constant modifications and additions in the compiler by different developers over the years has led to irregular implementation choices and design, which causes several problems.

1.2 Context

We will introduce the context with an example: the runtime wants the current date. It calls the builtin_date function shown in Figure 1.1 which returns the date of the machine executing it. This builtin_date function is part of a collection of functions found in the *RainCodeLegacyRuntime* library. The functions of this collection can be called anytime the application needs it. Such calls are created in the YAFL files of the compiler, thanks to a FunctionCall structure, as in Figure 1.2, referring to the library function on line 343.

```
53      public static string builtin_date(ExecutionContext ctx, string format)
54      {
55          DateTime n = ctx.Now;
56          if ((format != null) && string.Compare(format, "YYYYMMDD", StringComparison.OrdinalIgnoreCase) == 0)
57              return n.ToString("yyyyMMdd");
58          else if ((format != null) && string.Compare(format, "YYYYDDD", StringComparison.OrdinalIgnoreCase) == 0)
59          {
60              return n.Year.ToString("0000") + n.DayOfYear.ToString("000"); ;
61          }
62          else if ((format != null) && string.Compare(format, "YYDDD", StringComparison.OrdinalIgnoreCase) == 0)
63          {
64              return (n.Year % 100).ToString("00") + n.DayOfYear.ToString("000"); ;
65          }
66          else
67              return n.ToString("yyMMdd");
68      }
```

Figure 1.1: Library code of builtin_date function from the BuiltIn class of *RainCodeLegacyRuntime* library

We take a look at this function call, as illustrated on Figure 1.2. The variable holding it is called FuncCall on line 341 and is the result returned by the YAFL method as seen line 344. Line 343 creates the Functioncall with the reference to the function of the library. As we can see in red, the name of the *RainCodeLegacyRuntime* library and the path to the function are declared before "::", the function requested after them. The call then needs to define the mandatory elements related to the called function : the type returned (line 345), and the arguments with first the context on line 346 and then on line 348 or 350 the second, depending on the value passed by TheFormat variable, the format to template the date.

```
338   METHOD CompileAcceptFromDate (Func: DotNetFunction.Function;
339                                 Scope: AllocationScope): DotNetExpression.Expression;
340   VAR
341     FuncCall: DotNetFunctionCall.FunctionCall;
342   BEGIN
343     FuncCall.CREATE ("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Language.Cobol.BuiltIn::builtin_date");
344     RESULT := FuncCall;
345     FuncCall.SetType (DotNetTypes.NativeStringType);
346     FuncCall.AppendVariableArgument(Func.ExecutionContextVar);
347     IF TheFormat <> VOID THEN
348       FuncCall.AppendArgument(NEW DotNetLiteralExpression.NativeStringExpression (TheFormat));
349     ELSE
350       FuncCall.AppendArgument(NEW DotNetLiteralExpression.NativeStringExpression ("YYMMDD"));
351     END;
352   END CompileAcceptFromDate;
```

Figure 1.2: FunctionCall structure for builtin_date function call

FunctionCalls are the pieces of code the application needs to make calls to the *RainCodeLegacyRuntime* library functions at runtime. They are coded by developers with no help from the editor, as this structure is not recognised before compilation. The String part (in red, on line 343) needs particular caution as any mistake in it can introduce faults in the compiler.

Should an error occur after the execution of the code, it is hard to trace the origin of the error. An incorrect call can be the result of a wrongly written FunctionCall, but its implementation will be tested the earliest at execution time. An error will only arise when the compiler is compiled, or worse, when clients want to compile their code.
Can we change this implementation to cover those problems ? And, at the same time, help the developer working with Functioncalls ?

We will first further explain the compilation process at Raincode.

The Raincode compilation process involves 2 related phases as illustrated in Figure 1.3:

1. The compilation of the compilers, first the YAFL compiler, which produces the COBOL and PL/I ones. We will call this compilation "compiler compilation" and refer to the language it compiles if necessary. As a result of this compilation, we obtain, amongst others, libraries of C# functions, that will be used by the result of the second phase. Source code of an example of such a function is presented in Figure 1.1.

   This function builtin_date will return the current date in the format specified. There are 4 return formats, 3 cases on lines 57, 60 and 62 and a default format on line 67. Since the information it wants to return is the date at runtime of the compiled program of the second phase, it is built to be called at runtime. Therefore, the function is declared in the *RainCodeLegacyRuntime* library and the reference to this function is made via FunctionCalls, created inside the YAFL code of the compilers.

2. The compilation of programs in the target languages (COBOL, PL/I, ...), which we will call "sources compilation". This produces, from the client source code given to the compiled compiler, the byte code executable for Windows.

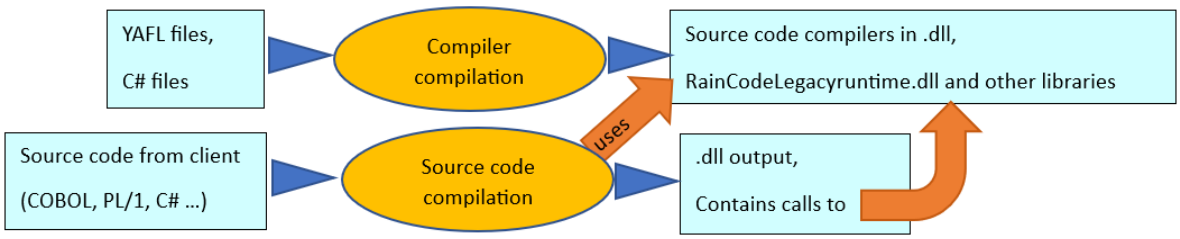


Figure 1.3: Flow of the two compilation processes

In schema 1.3, the color gives an indication of Raincode's compiler process:

- Yellow ovals represent the name of the compilation.
- Blue boxes are the files: the sources and result of the previous (orange box) step.
- Blue triangles indicate the direction of the process.
- Orange arrows are the references: to the call from the source .dll to the library .dll containing functions from box to box, and to the compiler used from oval to box.

Both compilation phases produce .dll files, the first one created being used by the second: the "sources compilation" output files need to call functions previously defined in C#, now compiled in *RainCodeLegacyRuntime*.dll after "compiler compilation".

1.3 Problem

The problem is that the C# functions from the *RainCodeLegacyRuntime*.dll files are not callable before runtime. The default choice has been to create a call that will make sense to the computer at runtime, while keeping it readable by the developer working on the Raincode compiler. As such, we will find, inside the YAFL files, FunctionCall statements provided by the YAFL compiler, that create specific function calls to functions. These statements contain a reference to the called functions in an explicit String format.

The Figure 1.3 presented the compilation processes, we will reduce it to the part that concerns us the most. Figure 1.4 represent the link between the YAFL files and the C# library. In the current implementation, the Raincode developer

created the Functioncall X reference to the function Y they need from the library.

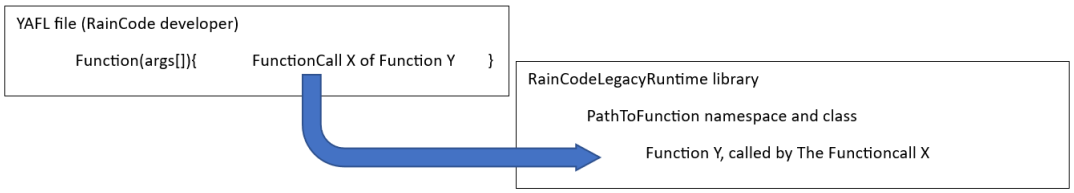


Figure 1.4: Deducted relation from YAFL file to C# library

In this Figure 1.4, we have to assume the Raincode developer is able to look at the C# library function to be used in "compiler compilation". They have to search for this function in order to correctly construct the Functioncall with all the parameters it requires. This is inefficient and can cause human errors, as the FunctionCall is not recognised by the editor.
We can then reason over this code in order to improve it. This schema will be helpful to illustrate the different modifications we planned for the architecture of the YAFL file to C# library relation (see section 1.7 especially).

In the YAFL files of the Raincode compilers, we can consider for example the Figure 1.5, a method from the COBOL source compiler.

```
63      REDEFINE METHOD CompileBody(Func : DotNetFunction.Function; Scope: RCompAllocation.AllocationScope) : DotNetStatement.Statement;
64      VAR
65        Compound: DotNetStatement.CompoundStatement;
66        FCall: DotNetFunctionCall.FunctionCall;
67      BEGIN
68        ASSERT Func.ExecutionContextVar <> VOID;
69        Compound.CREATE;
70        RESULT := Compound;
71        FOR Arg IN TheArgs DO
72          FCall.CREATE("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Core.ExecutionContext::CancelModule");
73          FCall.SetObject(Func.ExecutionContextVar.GetValueExpression);
74          FCall.AppendArgument(Arg.Compile(Func, Scope).ToNativeString(Func, Scope));
75          Compound.AddStatement(FCall);
76        END;
77      END CompileBody;
78
79    END CancelStatement;
80
81  END CancStatement;
```

Figure 1.5: Example taken from the Raincode repository, in the compiler for COBOL programs

In this method named CompileBody, we can spot, in red on line 72, the creation of a .NET FunctionCall with a String related to a C# function from the *RainCodeLegacyRuntime* library. Abbreviated by FCall in the example (N.B.: most of

the time by FuncCall but other name variations exists in the code), it will create a reference to a .dll library. The most common, and the one in the Figure 1.5, is called *RainCodeLegacyRuntime*, but as mentioned in Figure 1.3, other libraries exist.

We can immediately recognize the format of this reference: a String (on line 72, in red so recognized as a String by the editor). This means the information is hand-encoded by the developer. Such a way to pass information as text makes it hard to rely on the YAFL editor to check potential faults introduced: could the error be a string typo or a mistake from the additional information of lines 73 and 74 (respectively SetObject and AppendArgument) ? That additional information actually depends on the called function itself. If any error makes it way in the FunctionCall, a crash will happen at "sources compilation" time.
Additionally, using FunctionCall signifies that a change to any of these implies the greatest precautions. Indeed, if a change in the library of C# function is demanded, it can solely be completed after all its uses as a FunctionCall inside YAFL files are known. Then, after the library function changes, each usage has to be checked for potential faults introduced. Therefore, the update and maintenance of the legacy code of the compiler, mandatory for the compilation, is hindered. This, in result, limits the capacity of the company to expand its code.

But before further developing the FunctionCall structure and the solution proposed, we will look at the compiler in itself.

1.4 The Raincode Compiler

A compiler is a computer program that translates source code written in a programming language into machine code that can be executed by a computer. Compilers, and Raincode's compilers in particular, are considerably massive, therefore we need to identify every step and provide a concrete example for the problem introduced in the section 1.3.

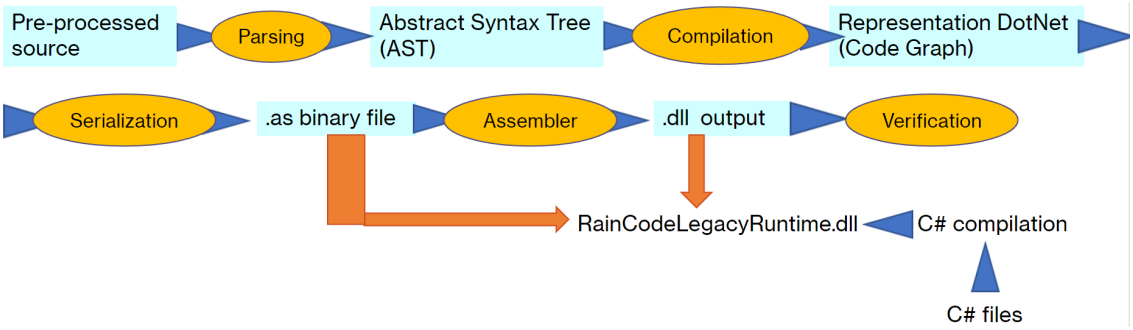


Figure 1.6: Flow of Raincode's compiler process

In the schema of Figure 1.6, the shape and color give an indication of Raincode's compiler process although slightly different than in Figure 1.3:

- Yellow boxes are the different steps of compilation.
- Blue boxes are the result of the previous (yellow box) step, and therefore a file.
- Blue triangles still indicate the direction of the process.
- Orange arrows are the references discussed in section 1.2: the FunctionCall is then an actual reference to the library function and *RainCodeLegacyRuntime* can be accessed from this stage in the compilation process.

As we can spot in the bottom right part of the schema, the compilation process includes a previous "compiler compilation" of C# files, the ones containing the functions called by a FunctionCall. The compilation creates the runtime library named *RainCodeLegacyRuntime*.dll accessible during runtime from *RainCodeLegacyRuntime*.

To clarify Figure 1.6, we will follow the transformation process of this line of code, taken from Figure 1.5:

```
FCall.CREATE("[RainCodeLegacyRuntime]
RainCodeLegacyRuntime.Core.ExecutionContext::CancelModule");
```

String references are used inside YAFL files. As we have seen in section 1.2, and illustrated by Figure 1.4, FunctionCall statements will allow the "sources compilation" result to call functions from the "compiler compilation" result.

The parsing step produce an Abstract Syntax Tree (AST). In this AST, there is a String Literal we can spot in red Figure 1.7. Its value is

"[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Core.ExecutionContext::CancelModule", a reference to the CancelModule function inside the *ExecutionContext* class. It indicates the part of runtime we are seeking at compilation.

```
REDEFINE METHOD CompileBody(Func : DotNetFunction.Function; Scope: RCompAllocation.AllocationScope) :
VAR
  Compound: DotNetStatement.CompoundStatement;
  FCall: DotNetFunctionCall.FunctionCall;
BEGIN
  ASSERT Func.ExecutionContextVar <> VOID;
  Compound.CREATE;
  RESULT := Compound;
  FOR Arg IN TheArgs DO
    FCall.CREATE("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Core.ExecutionContext::CancelModule");
    FCall.SetObject(Func.ExecutionContextVar.GetValueExpression);
    FCall.AppendArgument(Arg.Compile(Func, Scope).ToNativeString(Func, Scope));
    Compound.AddStatement(FCall);
  END;
END CompileBody;
```

Figure 1.7: Example code parsed

The compilation step then constructs the FunctionCall to CancelModule.

The result of the serialization step contains the call, not a object from the .NET representation as previously, with *callvirt* in a .as file (see Figure 1.8). This is the first actual reference call (orange arrow of Figure 1.6) to the concrete library containing CancelModule.

```
stloc.2
ldarg.0
ldloc.2
callvirt instance void [RainCodeLegacyRuntime]RainCodeLegacyRuntime.Core.ExecutionContext::CancelModule(string)
```

Figure 1.8: Example code in the .as file, the "caller"

Next, the assembler step produces the output .dll file (Figure 1.9). This file is in binary format, but we can decompile it with a tool like dotPeek [3]. We can therefore confirm that the function is equivalent to one of the previous steps as illustrated by Figure 1.10.

Finally, the verification step controls the correctness of the .dll file, otherwise it deletes it and informs the developer that an error occured and that the corrupted files were erased. Currently, this step sees the most crashes, but is still within direct range of action from Raincode. In more critical cases, some errors occur when the

```
string str = Convert.cnv_char_string_to_string(v_ec_69, memoryArea);
v_ec_69.CancelModule(str);
```

Figure 1.9: Example code in the binary file

```
// [100 7 - 100 32]
IL_00e6: ldarg.0      // v_ec_69
IL_00e7: ldloc.2      // str
IL_00e8: callvirt     instance void [RainCodeLegacyRuntime]RainCodeLegacyRuntime.Core.ExecutionContext::CancelModule(
```

Figure 1.10: Example code decompiled

compiler runs on a client machine. In this case debugging needs to be done, but as stated in section 1.1, client information can be sensitive[1], hindering the process and making it a heavy procedure. That is the presentation of the worst case situation, even though most of the time errors occur still within range of action of Raincode. The process could, however, be optimized.

The *RainCodeLegacyRuntime* library contains the function called as defined in Figure 1.11.

```
859     public void CancelModule(string name)
860     {
861         Module.Module module = ModuleDictionary.FindModule(name, false);
862         if (module == null) return;
863
864         MemoryArea mem = GetMemoryAddress(module);
865         if (mem.Ofs == 0) return;
866
867         module.initializeStaticVars(this, mem);
868     }
```

Figure 1.11: Example code as defined in *RainCodeLegacyRuntime*.dll

The function CancelModule of the library will try to find a module matching the argument name on line 861. If no module is found, there is nothing to cancel line 862. On line 864 we get the address of the module found in the memory. If there is no offset linked to the memory area, the function stops line 865. On line 867 the module has its static variables initialized to the execution context values in the memory area found on line 864.

This concludes the path of the example code, but we should emphasize the following note. As stated, most crashes occurs at verification phase. This is late in

[1]In the sense that even Raincode engineers will not be given access to it.

the process, and it would be preferable if we could avoid such crashes and detect these errors at a prior step, as illustrated on Figure 1.12.

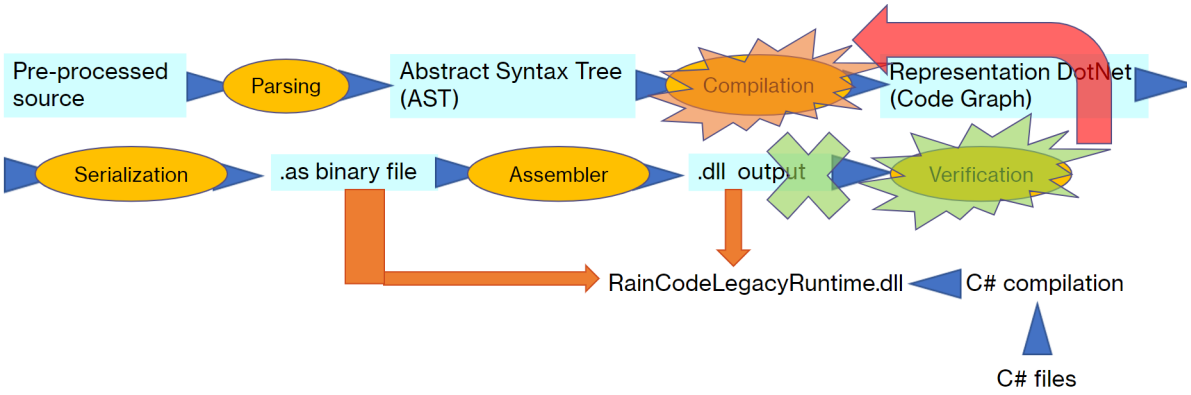


Figure 1.12: Compilation crashes: actual in green, expected improvement in red

1.5 The problem in practice

Now that the context and anatomy of Raincode's compilers have been explained, we discuss some of the concerns raised nowadays with the FunctionCall structure and hand coded String implementation.

FunctionCalls have been used for a long time, Raincode's developers already came across annoying situations throughout the years. As such, they realised the limitations posed by the current implementation in their compilers, and had to find fixes. Traces of past faults can be found in the YAFL files, such as in Figure 1.13: the FuncName String assignation code previously generated errors.
The (normally String) call creation here is an append of a String for the path and various variables from the context of the concerned function. The call wants to retrieve a conversion function, but the name of this function is a combination of the elements "cnv_" + origin format + "_to_" + intermediate qualifier + output format. Each call creation will require 3 previously defined variables to be assembled into one String for the FunctionCall. Given this multiplication of possibilities, it is no surprise that String names with no correspondence in the *RainCodeLegacyRuntime* library could appear. To patch this, developers used assertions, or more precisely assertions of difference, between the result of the String appended and the previously encountered String that caused an failure. Each time a chain of characters

leading to an error was discovered during debugging time, it was added in the list of ASSERT NOT statements already present in the YAFL function. We can also note that this debugging time, before the corresponding assertion line addition, would take a while and be based on developer intuition. Such a way to validate the Strings does not guarantee no occurrence of subsequent problems: it merely limits the already seen wrong values.

```
FuncName := ReplaceAmpersands("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Types.Convert::cnv_" +
                              FromName + "_to_" +
                              LanguageQualifier +
                              ToName);
-- RCompTrace.Debug ("FuncName: " + FuncName);
ASSERT NOT String.Equals (FuncName, "[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Types.Convert::cnv_string_to_fixed_dec");
ASSERT NOT String.Equals (FuncName, "[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Types.Convert::cnv_string_to_struct");
ASSERT NOT String.Equals (FuncName, "[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Types.Convert::cnv_string_to_memory_area");
ASSERT NOT String.Equals (FuncName, "[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Types.Convert::cnv_memory_area_to_string");
ASSERT NOT String.Equals (FuncName, "[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Types.Convert::cnv_int32_to_memory_area");
ASSERT NOT String.Equals (FuncName, "[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Types.Convert::cnv_struct_to_transient_char_
ASSERT NOT String.Equals (FuncName, "[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Types.Convert::cnv_rcstringutf16be_to_transi
```

Figure 1.13: Messy code to counter errors, as often with the converion functions

Another important issue concerns the maintenance of the library of functions in C#. We will take the example entry of Figure 1.14 for simplicity:

```
858         //called from the generated code
            7 references
859         public int SetCurrentModule(Module.Module module)
860         {
861             return theRunUnits.Peek().SetCurrentModule(module);
862         }
            0 references
863         public void CancelModule(string name)
864         {
865             Module.Module module = ModuleDictionary.FindModule(name, false);
866             if (module == null) return;
867
868             MemoryArea mem = GetMemoryAddress(module);
869             if (mem.Ofs == 0) return;
870
871             module.initializeStaticVars(this, mem);
872         }
```

Figure 1.14: Example of source code in the library; no reference is detected by the editor

In this Figure, we can spot two functions from the C# library : on line 859 SetCurrentModule and on line 863 CancelModule, that we already encountered in section 1.3 and followed its way through the "sources compilation" process. The

picture shows the graphical editor interface that tracks the references, changes and authors above each function. In the editor information of CancelModule, it is stated that no reference exists for this function. As we have seen when explaining the Raincode's compiler, this function is actually used inside the YAFL files, therefore, the assumption of the editor program is incorrect, which is an issue. Indeed, if we cannot find the references from the callable function, or even know whether it is referenced somewhere, we cannot easily modify it. We mentioned previously in section 1.3 that forcing a change would mean checking the validity of all YAFL functions using a FunctionCall with a String to this CancelModule and that this would be impractical.
We can also note that the information from the editor cannot be taken for granted, and we cannot clean the library of unused functions if we cannot be certain a function is not used somewhere in the compiler. For the SetCurrentModule library function, the editor does find references, but we cannot be sure it found all of them, as explained above. With this SetCurrentModule function, we discover the comment "called from the generated code" above the amount of references registered. This comment from the developer is present to indicate that the function should not be modified by any means, even if the developer is confident in the inventory of all the references where such function is used inside the YAFL files.

This section illustrates a crucial issue with the legacy code, as it renders future code breakable in case of changes. This slows the development and enhancement of *RainCodeLegacyRuntime* and other libraries, and thus of the compiler itself.

1.6 Objectives

The aim of this thesis is to prevent human errors from hand coded Strings inside FunctionCalls and, if possible, to mark the C# function called from the YAFL code.

To remove the possibility of errors introduction, we switch from the hand coded FunctionCalls structures to a library of generated Builders functions. Each Builder contains its corresponding FunctionCall structure. This automation of the previous human work relies upon the reflection for the functions with annotations and limits the chances of human typos in these FunctionCalls.

1.7 Approach

The first of our objectives is to identify all function calls to the runtime contained in the COBOL and PL/I compilers. As mentioned in section 1.2, the FunctionCall is constructed over several lines of code, but the main line is the String one containing the path and function from the library, the additional information lines are linked to the function called. Therefore, retrieving in the YAFL file all those Strings will give us an idea of the functions from the libraries that will be called.

We then want to annotate the functions in *RainCodeLegacyRuntime* we want to work on like in Figure 1.15. If we can retrieve the annotation, this will provide us a way to automatically get the signature of the called functions. Additionally, we might want to give these annotations more information, for a later phase in the development of our solution (we will provide more details regarding annotations in Chapter 2).

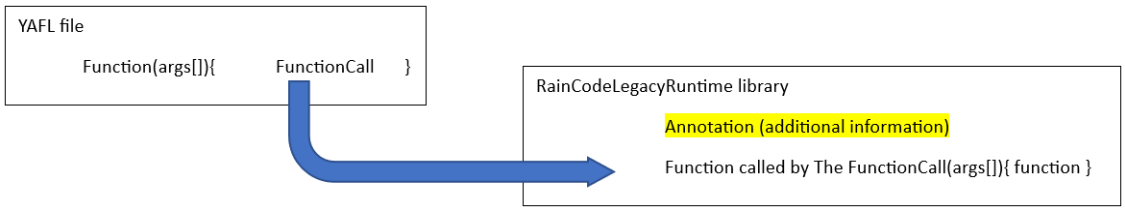


Figure 1.15: YAFL FunctionCall relation to the *RainCodeLegacyRuntime* library with the added annotations

This first part of the thesis will allow us to gather knowledge about the way the YAFL functions creates FunctionCalls. We will analyse this in Chapter 2, thanks to a few representative examples. If we can manage to identify recognizable patterns in the construction of the calls, we could provide the developer more support while limiting the possibilities of error.

A way of providing developer support would be to generate the FunctionCall from the library itself, and compare it to the actual call made inside the legacy YAFL file as in Figure 1.16. The generated FunctionCall cannot be wrong because it gets the information directly from the function signature.

If we take another direction, we might transform the FunctionCalls themselves and provide them with more information. This time, we will, as in Figure 1.17, give a mode parameter and the function reference to a function building the Func-

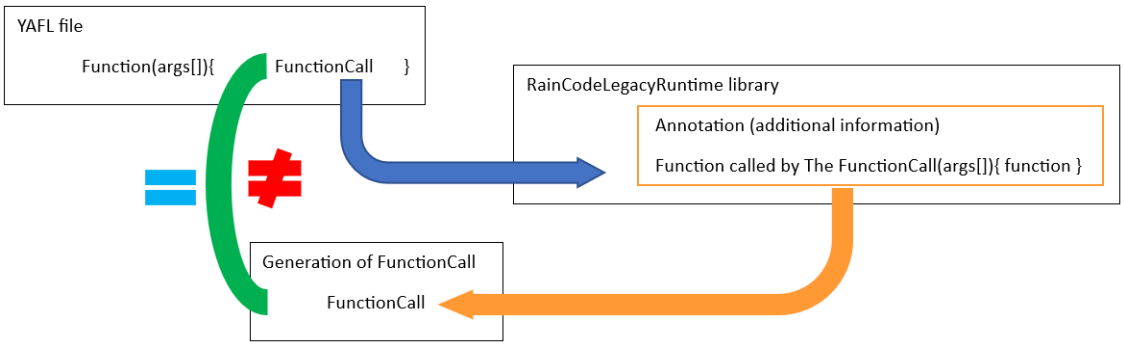


Figure 1.16: Generation of FunctionCall from the library and comparison with the actual FunctionCall

tionCall with respect to the specified mode. We can notice, this implies a rewrite of the YAFL code would be totally accepted if proven worth.

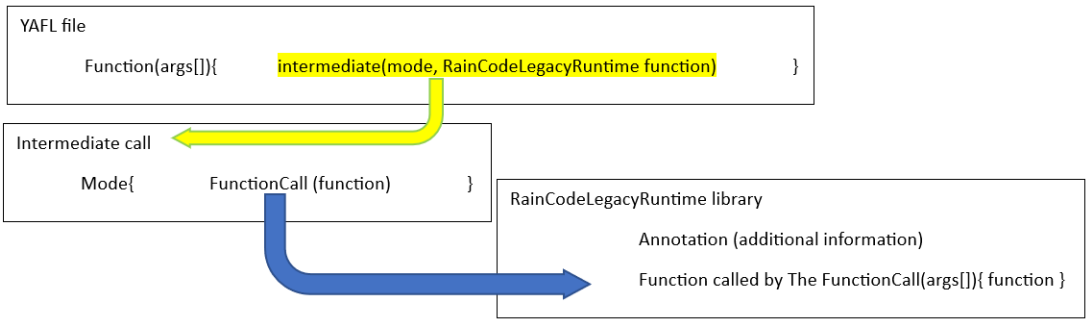


Figure 1.17: Usage of mode in the YAFL file to dictate the FunctionCall implementation around the function to be called

We already presented the concept of two compilation processes in section 1.2, and the fact that a FunctionCall is the way to specify the call from the source compiled .dll to the runtime library. To solve the above-mentioned issues in the use of FunctionCalls, modifying their construction in itself will not be the favored choice. Indeed, the system currently used combines both the viability of a call at runtime and the comprehension of the call by the developer. So we will explore other options, and expand on the objectives presented in section 1.6.

We base our solution on the idea presented in Figure 1.16 where we use the C# function library to generate a FunctionCall structure. As mentioned, generating the FunctionCall for a *RainCodeLegacyRuntime*function will require to retrieve the

signature of the function. To do so, we have to annotate the declaration in the C# library, as in Figure 1.15, and create a piece of code to retrieve these annotations automatically. Once we have the information needed for the generation, we can proceed with the creation of the FunctionCall for the annotated functions.

The part above is not sufficient to provide support to the developer, so in addition to just comparing (Figure 1.16) and leaving the work to the developer, we will adapt the mode parameter introduction of Figure 1.17. The idea to have an intermediate file can be adapted in order for the generated file with correct FunctionCall to take this place, we will call it the FunctionCallBuilder file. Therefore, the former FunctionCalls present in the YAFL file will be replaced with a call to FunctionCallBuilder, containing the generated FunctionCalls, as seen in Figure 1.18. By using this new architecture, we get rid of possibles human errors, since the FunctionCall is generated and therefore correct. Additionally, the FunctionCallBuilder will allow the editor to find the reference before compilation and provide feedback to the developer on the YAFL file adaptation since this modification will be done manually, case by case.

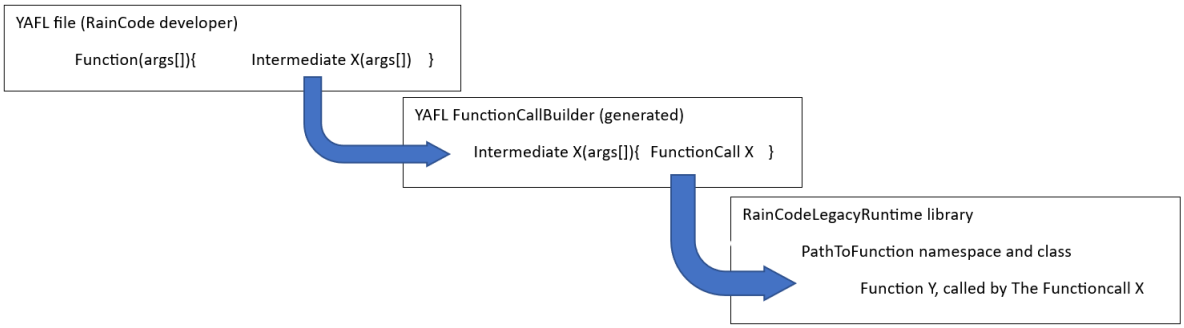


Figure 1.18: Adapted relation from YAFL file to the *RainCodeLegacyRuntime* library

Once the concept presented above is proven working with an "easy" pattern, we iterate to add more and more complex functions.

The conversion function append seen in Figure 1.13, for example, will require more work given the complexity of its implementation. Such difficulty might involve a rebuilding of the YAFL function rather than just the adaptation mentioned.

1.8 Validation

Once the solution proposed has been implemented, we will want to test it. For their compiler, Raincode already has complete tests. The local tests can detail the successes and failures, but also compare the results with the previous testing run and inform on the progression - or regression - on the passing of those tests.

The client side validation will be impossible, this is a long term objective. But we can deduce future amelioration on shorter term, with the "compiler compilation" and the execution of regression tests. If no test regressed, then the compiler is supposed at least as correct as the former implementation. In addition, we got rid of the previous problems: the possible introduction of human errors and the absence of editor feedback.

The first step is to set the YAFL environment and rebuild via the `make` command. Then, we run the command `make new` on the three compilers: COBOL, PL/I and the assembler one. There, we can spot errors introduced that crash the compilation. Once the compilations finish without fail, we can launch the local test suite, callable from command line. We run the test suite covering what could be impacted by our work, to receive the feedback in Figure 1.19 on successes, failures, progressions and regressions. The main concern will be for regression, as a test that do not pass anymore will possibly require a fix in the code of the solution.

"Possibly" does not seem precise enough: allow us to develop this reasoning. Some regressed tests may result from a part of the compilation untouched by us, and for which we do not have access to the related tools. For example, the database of SQL tests produced the 22 regressions in Figure 1.19, the IBM Db2 related tests are another cause of regression. We will not take these into account as it does not directly relate to our work. The concrete idea of the validation is to verify no test from the suite regressed for any other reason than the understandable causes.

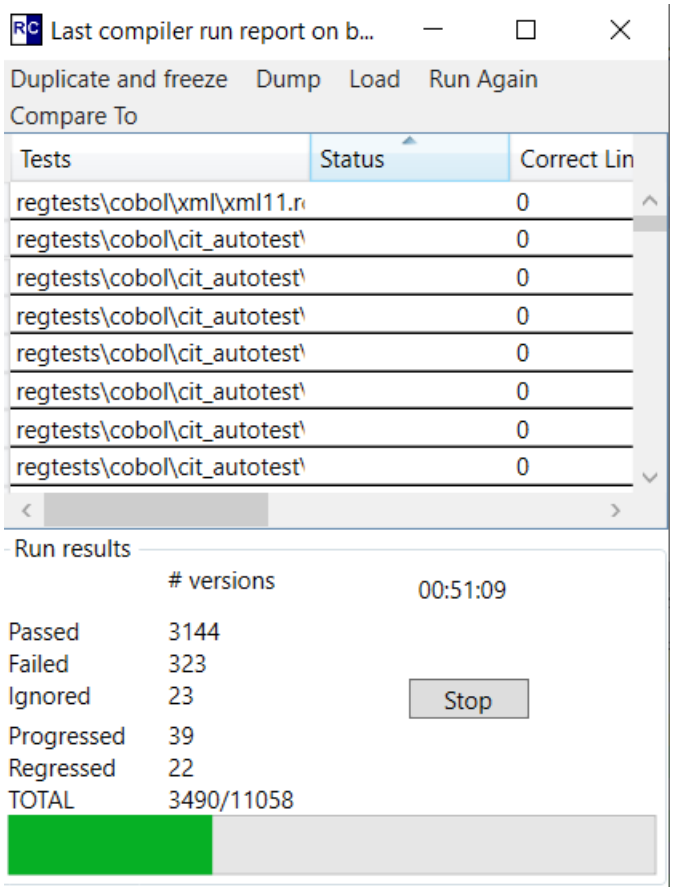


Figure 1.19: Compiler tester interface

1.9 Roadmap

This chapter presented the context of the thesis, what problem needs to be solved.

The second chapter will show the current hand coded implementations of FunctionCalls inside the YAFL code and link it with the corresponding C# functions. The design and tools needed of the solution will also be discussed.

The third chapter introduces our solution with a basic case from its generation to the refactoring of YAFL code. The issues encountered and how we overcame them are explained. Additionally, the implemented templates of solution are presented with their characteristics.

The fourth chapter validates the results obtained and their place inside Raincode's compiler. The validation shows the covered replacement of FunctionCalls

and the reason some still remains inside the compilers.

The fifth and final chapter closes this document with a discussion regarding the choice of patterns. Lastly, the plan for future work is presented.

Chapter 2

Legacy code and solution design

In this chapter, we take a look at the current state of the Raincode compiler and the different ways the FunctionCalls are inserted and built by developers inside the YAFL files. We will mainly focus on the possible patterns found for such FunctionCall creation. Those patterns will be relevant for the modification of the YAFL files holding the FunctionCall creation to calling a generated function.

The essence of our solution are the FunctionCallBuilder which will be detailed in Section 2.2. It contains the corresponding FunctionCall and is generated from the *RainCodeLegacyRuntime* library. This removes the need for developers to write FunctionCalls, and therefore removes the potential introduction of human errors. The generation process requires additional tools, we will present the ones used and their properties. Once the generation is accurate, we need to modify the legacy YAFL code to remove the legacy FunctionCalls and make the code use the correct FunctionCallBuilder. This modification to the YAFL file will be presented with the continuous example that illustrated the generation.

2.1 Introduction to YAFL

YAFL is an modern object-oriented language with polymorphism and single inheritance used in the COBOL and PL/I compilers. Darius Blasband, CEO of Raincode wrote an article on the YAFL programming language [4].

This section aims to provide the basic knowledge of the YAFL language in order to understand the fragments of code that we will present later in this chapter.

We take the example of Figure 2.1 to explain the main components of the YAFL language.

```
1  IMPLEMENTATION MODULE RuntimeAPI;
2   FROM RCompAllocation  IMPORT AllocationScope;
3
4    ONCE CLASS Intermediate_calls;
5
6      METHOD builtin_date_FunctionCallBuilder(Func: DotNetFunction.Function;
7                                              Scope: AllocationScope;
8                                              format: DotNetExpression.Expression):
9                                              DotNetStatement.Statement;
10     VAR
11       FuncCall: DotNetFunctionCall.FunctionCall;
12     BEGIN
13       FuncCall.CREATE("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Language.Cobol.BuiltIn::builtin_date");
14       FuncCall.AppendVariableArgument(Func.ExecutionContextVar);
15       ASSERT format.GetType.GetUnderlyingType = DotNetTypes.NativeStringType;
16       FuncCall.AppendArgument(format);
17       FuncCall.SetType(DotNetTypes.NativeStringType);
18       RESULT := FuncCall;
19     END builtin_date_FunctionCallBuilder;
20
21   END Intermediate_calls;
22 END RuntimeAPI;
```

Figure 2.1: YAFL code example

A YAFL program is composed of MODULEs (line 1). We can use FROM Module IMPORT Object to include an Object of some Module into another module (line 2).

Modules are composed of CLASSes, their constructor is CREATE (as in line 13, for the instance of FunctionCall). A ONCE class is a singleton, only one instance of this class can exist (line 4).

Inside the classes, we find METHODs (line 6) with the list of PARAMETER: TYPE between parentheses. Similarly, the return type for the method is indicated after the ":" following the list (line 9). In VAR (line 10), all local variables of the method must be defined, with their type after ":". BEGIN starts the code (line 12), until END with the name of the method (line 19). Classes and modules ending also specify their respective name (lines 21 and 22). The RESULT (line 18) is assigned by ":=" to FuncCall and acts as the return value of the method. The RESULT element is a variable, we can give it a value or apply methods on it.

We add the following information to cover the code presented in the various examples:

- Comments are preceded by "–" and take a single line.
- The key word REDEFINE preceding a method allows a method from the parent to be overloaded in the child class.

- The condition instruction follows the template IF condition THEN action; (optional ELSE action;) END. The ELSIF element defines an ELSE containing a IF.
- The for loop instruction follows the template FOR argument (optional TO argument) IN array or list DO action; END. The argument does not need to be defined in the VAR part.

The origin behind the creation of the method builtin_date_FunctionCallBuilder in Figure 2.1 will be explained in section 3.3.

2.2 Different FunctionCalls

A FunctionCall is the way to call a *RainCodeLegacyRuntime* function at runtime. It is implemented in the YAFL files and needs to contain all the information defining the called function signature.

As mentioned in the introduction of Chapter 2, we will take a look at the current implementations of how C# functions are called inside the Raincode compilers. This legacy code contains several differences in the way the FunctionCalls are created. For each of the examples presented, both the YAFL implementation and the definition of the corresponding function inside *RainCodeLegacyRuntime* will be illustrated. This list of examples is not to be taken as an absolute list, but the coverage of common possibilities thanks to few selected examples illustrating most cases present in the code. You will note that the usage of String to create a FunctionCall is common and was, in fact, a choice made since the early years. It was kept because it was a good viable solution, but as discussed in Chapter 1, this now poses limitations to the further enhancement of the compiler. A change towards a more solid system for those FunctionCalls is sought.

The first chapter of this thesis took a look inside the Raincode compilers. A few elements concern us: the COBOL and PL/I compilers and the .NET and RComp modules. In the files composing these elements, the FunctionCall always has a String line specifying the path to the C# function. Given the explicit String, the inventory of FunctionCalls was done manually by performing a grep, a command used for searching and matching text files expressions. We use a program for Windows called grepWin, an open-source tool created by Stefan Kueng [5]. We launch an analysis on the YAFL files of the different modules with the expression *"[RainCode*. The result of this search can be seen in Figure 2.2, the idea is to get all references to Raincode libraries, not only *RainCodeLegacyRuntime*.

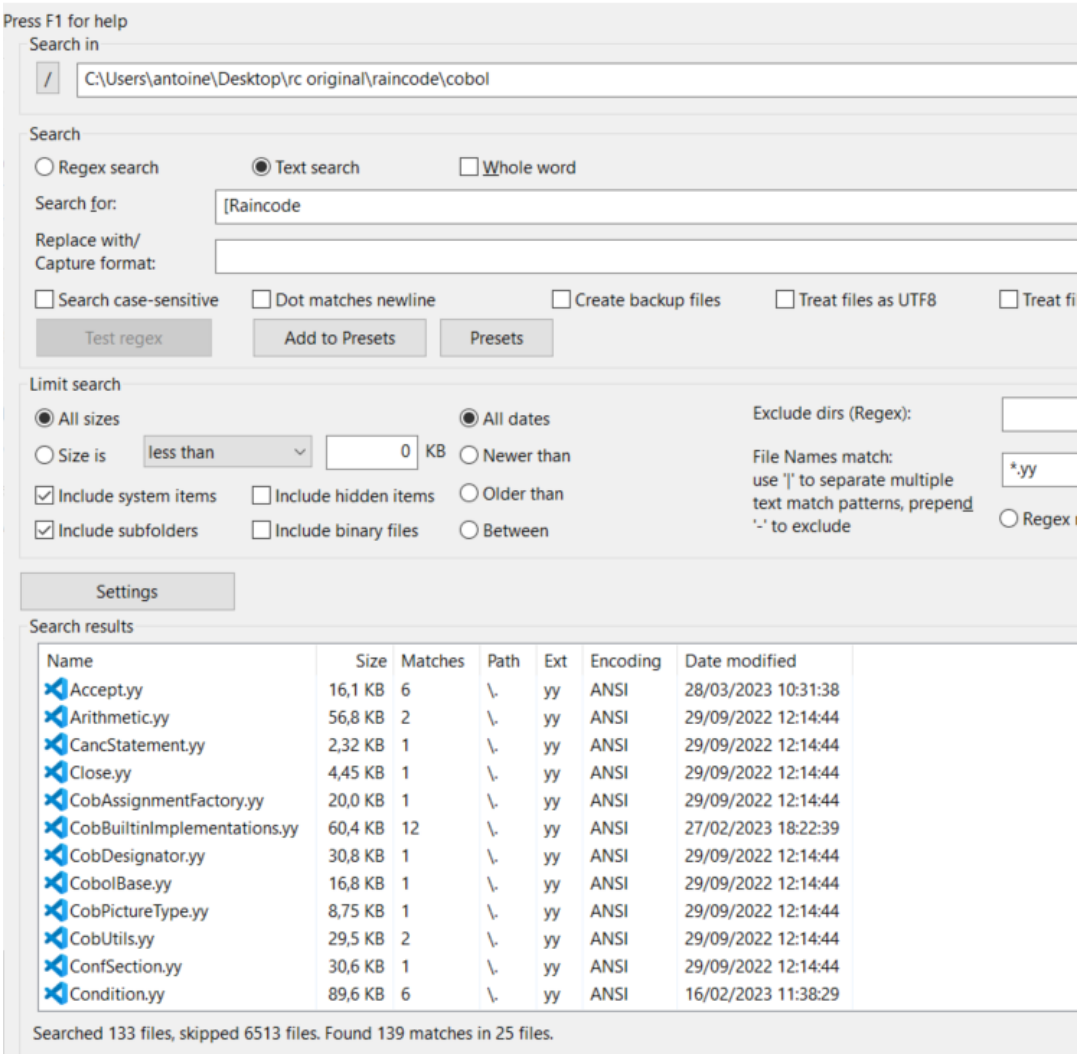


Figure 2.2: Grep search inside the COBOL compiler, on YAFL files for matches of *"[RainCode*

To summarize the results, the grep command found the following results for the *"[RainCode* entry in different modules, as seen in Table 2.1:

Above 22% of the YAFL files have at least one matching expression, and for those files, the average number of expressions found is just under 7 per file. A large amount of references exists in the YAFL files, our focus will be on the COBOL repository, even though an example from RComp is also present in this chapter.

module	# of matches	# of YAFL files flagged	total # of YAFL files
COBOL	139	25	133
.NET	217	16	28
RComp	63	16	66
PL/I	151	25	140
total	570	82	367

Table 2.1: Results for the expression *"[RainCode* in the 4 modules concerned

We will now start the presentation of the selected examples of the legacy code, mainly from the COBOL compiler:

2.2.1 Two close functions in the library

First we take a look at the CloseStatement class from the *Close* module of the COBOL compiler in Figure 2.3. We can notice the appearance of an IF statement containing two FuncCall.AppendArgument on lines 66 and 67. These methods construct the call by adding those arguments to the FunctionCall created on line 64. The String on line 59 indicate the class reference class RecordBasedFile, in which we have two close functions present in Figure 2.4 (on lines 2091 and 2101 respectively). The existence of the IF in Figure 2.3 serves the following purpose: differentiating between closing the file or handling the CLOSE operation as detailed in the respective comments on Figure 2.4. The difference is made thanks to the nature and amount of arguments in the FunctionCall created.

```
51  REDEFINE METHOD CompileBody (Func: DotNetFunction.Function;
52                               Scope: AllocationScope): DotNetStatement.Statement;
53  VAR
54    Compound: DotNetStatement.CompoundStatement;
55    FileRef: DotNetVariables.Variable;
56    FuncCall: DotNetFunctionCall.FunctionCall;
57    prefix: STRING;
58  BEGIN
59    prefix := "[RainCodeLegacyRuntime]RainCodeLegacyRuntime.IO.RecordBasedFile::";
60    Compound.CREATE;
61    RESULT := Compound;
62    FOR f IN TheArgs DO
63      FileRef := f.GetIdent!GetFDEntry.CompileGetFileRef(Func, Scope, Compound, TRUE);
64      FuncCall.CREATE (prefix + "close");
65      IF f.CloseMode THEN
66        FuncCall.AppendArgument(DotNetFactory.MakeBoolExpression(f.Reel));
67        FuncCall.AppendArgument(CloseEnumType.GetEnumValue(f.CloseOption));
68      END;
69      FuncCall.SetObject(FileRef.GetValueExpression);
70      Compound.AddStatement(CobUtils.CompileIOExceptionHandlers(Func, Scope, FuncCall,
71                                                                VOID, VOID, f.GetIdent, TRUE, VOID));
72    END;
73  END CompileBody;
```

Figure 2.3: YAFL code for *close.yy*, example 1

On a side note, we can also notice in Figure 2.3 that the first part of the String is assigned on line 59 and the called function name only appears line 64. This way of dividing the String of the Functioncall creation in two parts: respectively, the path to the class of *RainCodeLegacyRuntime*and the actual function requested for the task, will be reused for another purpose as we will see in the next example in section 2.2.2. Concretely:
"[RainCodeLegacyRuntime]RainCodeLegacyRuntime.PathToFunction::FunctionCalled" constitutes the String format from the append of lines 59 and 64.

```
2091        public void close()
2092        {
2093            close(false, CloseType.DetailNotSpecified);
2094        }
2095
2096        /// <summary>
2097        /// Handles the CLOSE operation
2098        /// Section 13.5.1 of http://supportline.microfocus.com/Documentation/books/oc41books/fhexfh.htm
2099        /// </summary>
2100
            1 reference
2101        public void close(bool unit, CloseType detail)
2102        {
2103            Context.Hooks.OnFilePreClose(LLFile);
2104            // closing
```

Figure 2.4: Declarations for the two close methods inside *RainCodeLegacyRuntime*

2.2.2 Redefinition of Trim inside YAFL

Inside the same COBOL compiler, the Trim class of the *CobBuiltinImplementations* module (Figure 2.5) contains a different String creation. On line 736, the called function part of the String is a variable that can possibly take different values. In our example, only "trim" is shown in line 746, but other classes from the same YAFL module can inherit this function and replace the GetBuiltinName method starting line 744. For example, further down the file in Figure 2.6, TrimLeft and TrimRight will inherit the class Trim and redefine the method GetBuiltinName (starting on lines 851 and 861).

```
725   REDEFINE METHOD Compile(Func : DotNetFunction.Function; Scope : AllocationScope) : DotNetExpression.Expression;
726   VAR
727     Arg1, Arg2: DotNetExpression.Expression;
728     FCall: DotNetFunctionCall.FunctionCall;
729   BEGIN
730     Arg1 := GetParam(0).Compile(Func, Scope);
731     IF Arity = 2 THEN
732       Arg2 := GetParam(1).Compile(Func, Scope);
733     ELSE
734       Arg2 := NEW DotNetLiteralExpression.NativeStringExpression (" ");
735     END;
736     FCall.CREATE("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Language.Cobol.BuiltIn::builtin_" + GetBuiltinName);
737     FCall.AppendVariableArgument(Func.ExecutionContextVar);
738     FCall.AppendArgument(Arg1.ToRawString(Func, Scope));
739     FCall.AppendArgument(Arg2.ToRawString(Func, Scope));
740     FCall.SetType(GetRCompType.ToValueType.ToAbstractType);
741     RESULT := FCall;
742   END Compile;
743
744   METHOD GetBuiltinName : STRING;
745   BEGIN
746     RESULT := "trim";
747   END GetBuiltinName;
```

Figure 2.5: YAFL code for the trim FunctionCall, example 2

Inside the *RainCodeLegacyRuntime* library, we can retrieve three different definitions in Figure 2.7: one for each class presented in this paragraph. As we can notice from the declarations on lines 303, 309 and 315, the functions all have the same arguments. This property is what makes the solely redefinition of the called function part inside the YAFL FunctionCall String possible. The AppendVariableArgument from line 737 and the AppendArgument from lines 738 and 739 in Figure 2.5 will take care of the same type of argument for the three functions: the context on line 737 and the two following arguments cast inside the YAFL function as raw Strings on lines 738 and 739.

```
851  CLASS TrimRight;
852    INHERITS Trim;
853
854    REDEFINE METHOD GetBuiltinName : STRING;
855    BEGIN
856      RESULT := "trim_right";
857    END GetBuiltinName;
858
859  END TrimRight;
860
861  CLASS TrimLeft;
862    INHERITS Trim;
863
864    REDEFINE METHOD GetBuiltinName : STRING;
865    BEGIN
866      RESULT := "trim_left";
867    END GetBuiltinName;
```

Figure 2.6: YAFL code for the redefinition inside the classes TrimRight/trimLeft

```
303      public static RCString builtin_trim(ExecutionContext ctx, RCString s, RCString c)
304      {
305          if (s == null) return RCString.Empty;
306          return s.Trim(ctx, c.Bytes);
307      }
308
                0 references
309      public static RCString builtin_trim_left(ExecutionContext ctx, RCString s, RCString c)
310      {
311          if (s == null) return RCString.Empty;
312          return s.TrimStart(ctx, c.Bytes);
313      }
314
                0 references
315      public static RCString builtin_trim_right(ExecutionContext ctx, RCString s, RCString c)
316      {
317          if (s == null) return RCString.Empty;
318          return s.TrimEnd(ctx, c.Bytes);
319      }
```

Figure 2.7: *RainCodeLegacyRuntime* library entries of the various trim

2.2.3 Three builtin_abs vary in their argument type

We stay in the same *CobBuiltinImplementations* module but for another call: the builtin_abs of the Abs class on line 49 in Figure 2.8. This Figure shows the use of GetRCompType on lines 50 and 52, which means the type is dependent on the context it is called. These lines define the type to be converted on line 52 and the output Type on line 50.

```
44  REDEFINE METHOD Compile (Func: DotNetFunction.Function;
45                           Scope: AllocationScope): DotNetExpression.Expression;
46  VAR
47    FCall: DotNetFunctionCall.FunctionCall;
48  BEGIN
49    FCall.CREATE("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Language.Common.BuiltIn::builtin_abs");
50    FCall.SetType(GetRCompType);
51    FCall.AppendVariableArgument(Func.ExecutionContextVar);
52    FCall.AppendArgument(GetParam(0).Compile(Func, Scope)!CastTo(Func, Scope, GetRCompType!ToAbstractType));
53    RESULT := FCall;
54  END Compile;
```

Figure 2.8: YAFL code for the creation of the *builtin_abs* FunctionCall, example 3

If we take the *RainCodeLegacyRuntime* library, we can spot 3 different builtin_abs functions in Figure 2.9. The lines 21, 26 and 31, with the different declarations, show a change in the type of the output and the second argument, which suits what we just emphasized above.

```
21      0 references
        public static FixedDecValue builtin_abs(ExecutionContext ctx, FixedDecValue d)
22      {
23          return d.abs(ctx);
24      }
25
26      0 references
        public static FixedBinValue builtin_abs(ExecutionContext ctx, FixedBinValue d)
27      {
28          return d.abs();
29      }
30
31      0 references
        public static double builtin_abs(ExecutionContext ctx, double d)
32      {
33          return Math.Abs(d);
34      }
35
36      static Random rnd = new Random();
37
```

Figure 2.9: The 3 declarations of the different *builtin_abs* from *RainCodeLegacyRuntime*

It is also interesting to take this example from *RainCodeLegacyRuntime* and compare it with the first one presented in Figure 2.4, where the two close functions differ in the argument number. In reality, the type of the arguments for the called function is the information verified to choose the correct builtin_abs from the *RainCodeLegacyRuntime* in Figure 2.9.

2.2.4 SetCursorX FunctionCall argument type is more precise than the library function

Again from the COBOL compiler, we can take a look at the DisplayStatement class of the *Display* module in Figure 2.10. There is a subtlety with the String construction in the FunctionCall of this YAFL function compared to the abstract format presented in section 2.1.1. The ConsoleControle C# file does not exist, the function SetCursorX(int32) on line 106 in YAFL Figure 2.10 is stored in the ConsoleControle class (in Figure 2.11 on line 102 for the class and line 105 for the function) from the Console file in the library. This indicates that the "[RainCodeLegacyRuntime]RainCodeLegacyRuntime.
PathToFunction::FunctionCalled" String of section 2.1.1 must be refined to "[RainCodeLegacyRuntime]RainCodeLegacyRuntime.
PathToFunction.FunctionLocation::FunctionCalled". This FunctionLocation will be given its complete meaning with section 2.1.5.

```
103     METHOD MoveCursor;
104     BEGIN
105       IF X <> VOID THEN
106         FuncCall.CREATE ("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.IO.Streams.ConsoleControle::SetCursorX(int32)");
107         FuncCall.SetType (DotNetTypes.VoidType);
108         FuncCall.AppendArgument(X.Compile(Func,Scope).CastTo(Func, Scope, DotNetTypes.AbstractIntType));
109         Compound.AddStatement (FuncCall);
110         X := VOID;
111       END;
112       IF Y <> VOID THEN
113         FuncCall.CREATE ("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.IO.Streams.ConsoleControle::SetCursorY(int32)");
114         FuncCall.SetType (DotNetTypes.VoidType);
115         FuncCall.AppendArgument(Y.Compile(Func,Scope).CastTo(Func, Scope, DotNetTypes.AbstractIntType));
116         Compound.AddStatement (FuncCall);
117         Y := VOID;
118       END;
119     END MoveCursor;
```

Figure 2.10: YAFL FunctionCalls for SetCursorX and SetCursorY

The FunctionCall String in Figure 2.10 possesses the particularity of containing information in parentheses on line 106. The int32 defines the amount of bits taken for the integer as the argument of the function, but we can notice in Figure 2.11 no such restriction is present in the function definition nor body from line 105 to 115. If you refer to the previous FunctionCalls explained, not once did the String (of FunctionCalled in the format) contain a type specification in parentheses, this is a novelty. Since this information is passed in the runtime and not checked in the function called, we will have to keep this limitation during the refactoring phase of the YAFL file.

```
101     // Control cursor and console windows on standard output
        0 references
102     public static class ConsoleControle
103     {
104         // X and Y start at 1 in COBOL
            0 references
105         public static void SetCursorX(int x)
106         {
107             try
108             {
109                 System.Console.CursorLeft = x - 1;
110             }
111             catch (Exception e)
112             {
113                 Logger.Log(IORuntime.LogSource, e);
114             }
115         }
116
            0 references
117         public static void SetCursorY(int y)
118         {
119             try
120             {
121                 System.Console.CursorTop = y - 1;
122             }
123             catch (Exception e)
124             {
125                 Logger.Log(IORuntime.LogSource, e);
126             }
127         }
```

Figure 2.11: Corresponding library entries for SetCursorX and SetCursorY in the ConsoleControl class

2.2.5 Each VaryingStringTypeDescriptor has its own Functioncall

From the *Rcomp* module this time, in the *RCompStringType* file, the class Utils shows yet another way of handling several similarly named functions.
In the second case (*CobBuiltinImplementations*, Figures 2.5 and 2.6), each builtin_ function specification was inside a redefined method (understand here: the same method but with an adaptable variable on line 736 in Figure 2.5. The methods on lines 854 and 861 of Figure 2.6 modify the value returned, for an adapted call in the corresponding class).
In the third case (*CobBuiltinImplementations*, Figure 2.8), one builtin_abs method is present in the YAFL code on line 49, but three entries in the *RainCodeLegacyRuntime* library in Figure 2.9 in lines 21, 26 and 31.

For the Get functions of the current example, on lines 916, 930 and 944 in Figure 2.12, on the other hand, different methods coexist inside the Utils class. The choice made in the YAFL file was not to construct the call with an append, as we have seen previously. Rather, a different method for each get was created on lines 916, 930 and 944, the same way all these Get are in separated classes (and files) inside the *RainCodeLegacyRuntime* library: line 25 in Figure 2.13 of class Vary-

ingCharacterStringType, line 20 in Figure 2.14 of class VaryingZCharacterStringType and line 15 in Figure 2.15 of class VaryingBinaryStringType.

```
913   METHOD VaryingCharacterStringTypeDescriptor_DynSize(size: DotNetExpression.Expression) : DotNetExpression.Expression;
914   VAR FCall: DotNetFunctionCall.FunctionCall;
915   BEGIN
916     FCall.CREATE("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Descriptor.VaryingCharacterStringType::Get");
917     FCall.SetType(DotNetDescriptorTypes.VaryingCharacterStringTypeDescriptorType);
918     FCall.AppendArgument(size);
919     RESULT := FCall;
920   END VaryingCharacterStringTypeDescriptor_DynSize;
921
922   METHOD VaryingCharacterStringTypeDescriptor(size: INTEGER) : DotNetExpression.Expression;
923   BEGIN
924     RESULT := VaryingCharacterStringTypeDescriptor_DynSize(DotNetFactory.MakeIntExpression(size));
925   END VaryingCharacterStringTypeDescriptor;
926
927   METHOD VaryingZCharacterStringTypeDescriptor_DynSize(size: DotNetExpression.Expression) : DotNetExpression.Expression;
928   VAR FCall: DotNetFunctionCall.FunctionCall;
929   BEGIN
930     FCall.CREATE("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Descriptor.VaryingZCharacterStringType::Get");
931     FCall.SetType(DotNetDescriptorTypes.VaryingZCharacterStringTypeDescriptorType);
932     FCall.AppendArgument(size);
933     RESULT := FCall;
934   END VaryingZCharacterStringTypeDescriptor_DynSize;
935
936   METHOD VaryingZCharacterStringTypeDescriptor(size: INTEGER) : DotNetExpression.Expression;
937   BEGIN
938     RESULT := VaryingZCharacterStringTypeDescriptor_DynSize(DotNetFactory.MakeIntExpression(size));
939   END VaryingZCharacterStringTypeDescriptor;
940
941   METHOD VaryingBinaryStringTypeDescriptor_DynSize(size: DotNetExpression.Expression) : DotNetExpression.Expression;
942   VAR FCall: DotNetFunctionCall.FunctionCall;
943   BEGIN
944     FCall.CREATE("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Descriptor.VaryingBinaryStringType::Get");
945     FCall.SetType(DotNetDescriptorTypes.TypeDescriptorType);
946     FCall.AppendArgument(size);
947     RESULT := FCall;
948   END VaryingBinaryStringTypeDescriptor_DynSize;
```

Figure 2.12: YAFL code for the different Get of example 5

As started in section 2.1.1 and refined with section 2.1.4, the String takes the form of "[RainCodeLegacyRuntime]RainCodeLegacyRuntime.PathToFunction.FunctionLocation::FunctionCalled". We will use the current example to illustrate how the FunctionLocation can be the same in the FunctionCall for various library functions of different files. For some of the calls, VaryingCharacterStringType (Figure 2.13) and VaryingZCharacterStringType (Figure 2.14) are separated from Descriptor by an additional sub-module called *CharacterString*, contrary to the third, VaryingBinaryStringType (Figure 2.15). This is made possible by defining, in *RainCodeLegacyRuntime*, the namespace (line 9 in Figure 2.13, line 9 in Figure 2.14 and line 3 in Figure 2.15 . With this explanation, the FunctionCall String defined as something similar to "[RainCodeLegacyRuntime]RainCodeLegacyRuntime.PathToFunction::FunctionCalled" in section 2.1.1 have to be changed to

"[RainCodeLegacyRuntime]RainCodeLegacyRuntime.
NamespaceDefinedInLibraryFile.ClassName::FunctionCalled".

```
9   namespace RainCodeLegacyRuntime.Descriptor
10  {
        13 references
11      public class VaryingCharacterStringType : AbstractCharacterStringTypeDescriptor
12      {
13          /// <summary>
14          /// Indicates whether this varying char type is the last element in the enclosing structure
15          /// </summary>
            3 references
16          public bool lastInStructure { get; set; }
17
            1 reference
18          public VaryingCharacterStringType(int maxLength)
19          {
20              MaxStringLength = maxLength;
21          }
22
23          private static readonly DescriptorDic<int, VaryingCharacterStringType> map = new DescriptorDic<int, VaryingCharacterStringType>();
24
            1 reference
25          public static VaryingCharacterStringType Get(int maxLength)
26          {
27              return map.Get(maxLength, () => new VaryingCharacterStringType(maxLength));
28          }
29
```

Figure 2.13: Get functions entry for VaryingCharacterStringType inside the *RainCodeLegacyRuntime* library

```
9   namespace RainCodeLegacyRuntime.Descriptor
10  {
        8 references
11      public class VaryingZCharacterStringType : AbstractCharacterStringTypeDescriptor
12      {
            1 reference
13          public VaryingZCharacterStringType(int maxLength)
14          {
15              MaxStringLength = maxLength;
16          }
17
18          private static readonly DescriptorDic<int, VaryingZCharacterStringType> map = new DescriptorDic<int, VaryingZCharacterStringType>();
19
            0 references
20          public static VaryingZCharacterStringType Get(int maxLength)
21          {
22              return map.Get(maxLength, () => new VaryingZCharacterStringType(maxLength));
23          }
24
```

Figure 2.14: Get functions entry for VaryingZCharacterStringType inside the *RainCodeLegacyRuntime* library

```
3   namespace RainCodeLegacyRuntime.Descriptor
4   {
        8 references
5       public class VaryingBinaryStringType : AbstractTypeDescriptor
6       {
7           private readonly int maxLength;
            1 reference
8           public VaryingBinaryStringType(int maxLength)
9           {
10              this.maxLength = maxLength;
11          }
12
13          private static readonly DescriptorDic<int, VaryingBinaryStringType> map = new DescriptorDic<int, VaryingBinaryStringType>();
14
            1 reference
15          public static ITypeDescriptor Get(int maxLength)
16          {
17              return map.Get(maxLength, () => new VaryingBinaryStringType(maxLength));
18          }
19
            17 references
```

Figure 2.15: Get functions entry for VaryingBinaryStringType inside the *RainCodeLegacyRuntime* library

2.2.6 Compare changes the object method and returns a FunctionCall

From the COBOL compiler, in the *Condition* module, the function MakeOptimizedDecimalDesignatorComparaison in Figure 2.16 presents several deviations from the other examples. The first comes on line 504 where the output of the function is a FunctionCall. We can see from lines 512 to 517 (+ line 519) that the FunctionCall methods are directly applied on the RESULT element of the YAFL function. Secondly, the SetObject on line 514 uses the GetStaticDescriptor method, contrary to the previous GetValueExpression we could see in section 2.1.1. Additionally, this GetStaticDescriptor method is also used for the AppendArgument on line 519.

```
498  METHOD MakeOptimizedFixedDecimalDesignatorComparison (Func: DotNetFunction.Function;
499                                                        Scope: AllocationScope;
500                                                        Anchor: RCompBase.BaseNonTerminalInterface;
501                                                        ALeft: Designator; Left: DotNetExpression.Expression;
502                                                        LeftType: RCompType.FixedDecimalType;
503                                                        Op: CHAR;
504                                                        ARight: Expression; Right: DotNetExpression.Expression): DotNetFunctionCall.FunctionCall;
505  VAR
506    RightType: RCompType.FixedDecimalType;
507  BEGIN
508    WHAT ARight OF
509      IN Designator:
510        RightType := Arithmetic.Utils.AsOptimizableIntegerDecimal (TAG.GetRCompType);
511        IF (RightType <> VOID) AND (LeftType.GetScale = RightType.GetScale) THEN
512          RESULT.CREATE ("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Descriptor.FixedDecimalDescriptor::Compare");
513          RESULT.SetType ( DotNetTypes.IntType );
514          RESULT.SetObject (LeftType.GetStaticDescriptor);
515          RESULT.AppendVariableArgument(Func.ExecutionContextVar);
516          RESULT.AppendReferenceArgument (Left);
517          RESULT.AppendReferenceArgument (Right);
518          IF LeftType <> RightType THEN
519            RESULT.AppendArgument(RightType.GetStaticDescriptor);
520          END;
521        END;
522      END;
523      IN IntegerLiteral:
524        Anchor.LazyInternalError (20150713); -- handled by MakeIntegerLiteralComparison
525      END;
526    ELSE
527    END;
528  END MakeOptimizedFixedDecimalDesignatorComparison;
```

Figure 2.16: YAFL code for the Compare of example 6

Similarly to the *close* example of section 2.1.1, two *Compare* functions from of the *RainCodeLegacyRuntime* library can be called from this YAFL function. Figure 2.17 shows the first one, with three parameters on line 1394. The second one in figure 2.18 has 4 parameter, the fourth argument being the one using the GetStaticDescriptor method.

Few YAFL functions return a FunctionCall type in the legacy code, but this is a possible output that we will cover in our solution. Creating this pattern is useful when the FunctionCall is used as parameter of another YAFL function. We do not want to change the parameter Type of working functions, changing the output type will offer us this stability.

The SetObject line of this example will require additional information to be correctly generated. As we discussed, the GetStaticDescriptor method is new, we only encountered GetValueExpression in the previous examples. This need will see the use of a specific GenerationMode which we will develop in section 2.2.1.

```
1393        [CompilerCalled(GenerationMode.Create | GenerationMode.StaticDescSO | GenerationMode.FCreturn, "base")]
            0 references
1394        public int Compare(ExecutionContext ctx, ref MemoryArea m, ref MemoryArea v)
1395        {
1396            long l = 0;
1397            long r = 0;
1398
1399            if (!RequiresValidityCheck)
1400            {
1401                bool littleEndian = IsComp5 && ctx.Comp5LittleEndian;
1402                switch (m.Size)
1403                {
1404                    case 1:
1405                        l = m.Asp.Mem[m.Ofs];
1406                        r = v.Asp.Mem[v.Ofs];
1407                        break;
1408                    case 2:
1409                        if (littleEndian)
1410                        {
1411                            l = m.LoadInt16();
1412                            r = v.LoadInt16();
1413                        }
1414                        else
1415                        {
1416                            l = m.LoadMainframeInt16();
1417                            r = v.LoadMainframeInt16();
1418                        }
1419                        break;
1420                    case 4:
1421                        if (littleEndian)
1422                        {
1423                            l = m.LoadInt32();
1424                            r = v.LoadInt32();
1425                        }
1426                        else
1427                        {
1428                            l = m.LoadMainframeInt32();
1429                            r = v.LoadMainframeInt32();
1430                        }
1431                        break;
1432                    case 8:
1433                        if (littleEndian)
1434                        {
1435                            l = m.LoadInt64();
1436                            r = v.LoadInt64();
1437                        }
1438                        else
1439                        {
1440                            l = m.LoadMainframeInt64();
1441                            r = v.LoadMainframeInt64();
1442                        }
1443                        break;
1444                }
1445                if (l < r)
1446                    return -1;
1447                else if (l > r)
1448                    return 1;
1449                else
1450                    return 0;
1451            }
1452            else
1453            {
1454                Int10 vm = GetInt10Value(ctx, ref m);
1455                Int10 vval = GetInt10Value(ctx, ref v);
1456                return Int10.compare(ref vm, ref vval);
1457            }
1458        }
```

Figure 2.17: Compare function entry without descriptor parameter inside the *RainCodeLegacyRuntime* library

```
1462        [CompilerCalled(GenerationMode.Create | GenerationMode.StaticDescSO | GenerationMode.FCreturn, "descriptor")]
            0 references
1463        public int Compare(ExecutionContext ctx, ref MemoryArea m, ref MemoryArea v, FixedDecimalDescriptor desc)
1464        {
1465            long l = 0;
1466            long r = 0;
1467
1468            if (!RequiresValidityCheck&& !desc.RequiresValidityCheck)
1469            {
1470                bool littleEndianL = IsComp5 && ctx.Comp5LittleEndian;
1471                bool littleEndianR = desc.IsComp5 && ctx.Comp5LittleEndian;
1472                switch (m.Size)
1473                {
1474                    case 1:
1475                        l = m.Asp.Mem[m.Ofs];
1476                        break;
1477                    case 2:
1478                        l = littleEndianL? m.LoadInt16() : m.LoadMainframeInt16();
1479                        break;
1480                    case 4:
1481                        l = littleEndianL? m.LoadInt32() : m.LoadMainframeInt32();
1482                        break;
1483                    case 8:
1484                        l = littleEndianL? m.LoadInt64() : m.LoadMainframeInt64();
1485                        break;
1486                    default:
1487                        System.Diagnostics.Debug.Assert(false, "Unsupported comp size: " + m.Size);
1488                        break;
1489                }
1490                switch (v.Size)
1491                {
1492                    case 1:
1493                        r = v.Asp.Mem[v.Ofs];
1494                        break;
1495                    case 2:
1496                        r = littleEndianR? v.LoadInt16() : v.LoadMainframeInt16();
1497                        break;
1498                    case 4:
1499                        r = littleEndianR? v.LoadInt32() : v.LoadMainframeInt32();
1500                        break;
1501                    case 8:
1502                        r = littleEndianR? v.LoadInt64() : v.LoadMainframeInt64();
1503                        break;
1504                    default:
1505                        System.Diagnostics.Debug.Assert(false, "Unsupported comp size: " + m.Size);
1506                        break;
1507                }
1508                if (l < r)
1509                    return -1;
1510                else if (l > r)
1511                    return 1;
1512                else
1513                    return 0;
1514            }
1515            else
1516            {
1517                Int10 vm = GetInt10Value(ctx, ref m);
1518                Int10 vval = desc.GetInt10Value(ctx, ref v);
1519                return Int10.compare(ref vm, ref vval);
1520            }
1521        }
```

Figure 2.18: Compare function entry with the descriptor parameter inside the *RainCodeLegacyRuntime* library

2.2.7 The String of **Next** is generated and a local variable takes the object position

In the COBOL compiler, the *Statement* module possesses a redefinition of the function CompileBody. Inside this redefinition, a FunctionCall is used for the C#

function Next. As we detailed in section 2.1.5, the String for CREATE takes the form of "[RainCodeLegacyRuntime]RainCodeLegacyRuntime.
NamespaceDefinedInLibraryFile.ClassName::FunctionCalled". We can see on line 1998 in Figure 2.19 the DotNetFactory.RuntimeCall function that returns the CREATE part for a FunctionCall in the format "[RainCodeLegacyRuntime]completeNamespace.class::member". The elements composing the String are referred as follows:

- *[RainCodeLegacyRuntime]* is the Dynamic Link Library (dll) associated with RuntimeCall, an equivalent function exists for *[RainCodeLegacyQixInterface]*.
- *"Language.Cobol"* is the namespace that will be transformed into *"RainCodeLegacyRuntime.Language.Cobol"*, taking the place of the completeNamespace in the format presented above.
- *"XmlParse"* is the class of the namespace.
- *"Next"* is the member called.

This implementation is a previous answer to the hand coded String present in the FunctionCalls. It automates the creation of the correct String based on the namespace, the class and the member parameters. RuntimeCall aims to reduce the risk of human error being introduced during the creation of the FunctionCall. In Figure 2.20, we can recognize, at the top, the same separation of dll, completeNamespace + class, and member as presented above.

```
1995        LET Loc := Func.GetTemporaryLocal (XmlParseFunctionType, "XmlParse");
1996        Loc.Lock;
1997        Compound.AddStatement (RCompAssignmentFactory.Utils.One.CompileExprToVariable (Func, Scope, AFCall, Loc));
1998        LET FCall := DotNetFactory.RuntimeCall("Language.Cobol", "XmlParse", "Next");
1999        FCall.SetObject (Loc.GetValueExpression);
2000        FCall.SetType (DotNetTypes.BooleanType);
```

Figure 2.19: YAFL code for the Next of example 7

Another variation of this example is the SetObject on line 1999 in Figure 2.19. The Loc.GetValueExpression parameter can be divided between the already encountered GetValueExpression method, often used inside the SetObject line, and a Local Variable named Loc, defined on line 1999. If we refer only to the FunctionCall, the Loc and FileRef of section 2.1.1 are actually quite similar, as both are referenced as *DotNetVariables.Variable*, on which is applied the GetValueExpression method. However, for clarity, we will keep them separated, the same way developers gave an explicit name for both of them: the FileRef is a reference to a file, while the Loc is a local variable often created close to the FunctionCall it is used in.

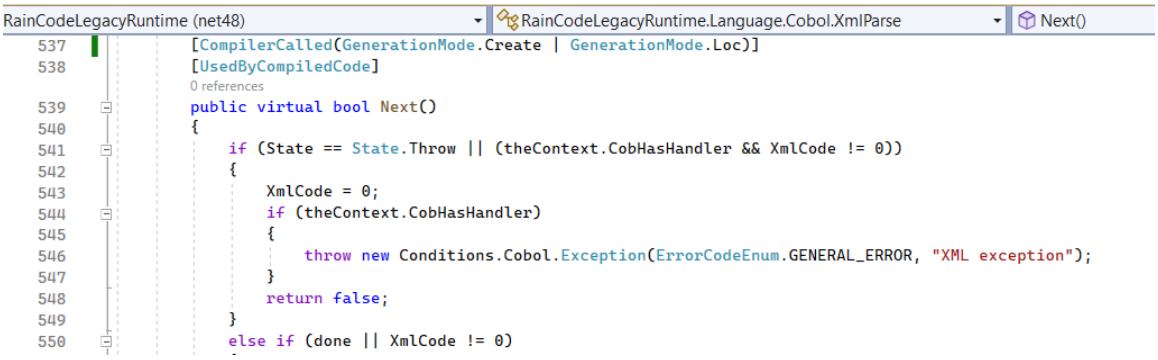


Figure 2.20: Part of Next function, with the path to the function at the top of the picture

2.2.8 Summary of the previous sections examples

This section will summarize the information of the examples seen in sections 2.1.1 to 2.1.5.

1. Two close functions in the library: the first takes no argument, the second takes two. Both are called in the same YAFL method depending on the value of a checked variable.

2. Redefinition of Trim inside YAFL: the function part of the FunctionCall creation String is a variable redefined for the TrimRight and TrimLeft classes that inherit from Trim. The three functions have the same signature since the only change in the FunctionCall is the name of the function called.

3. Three builtin_abs vary in their argument type: three similarly named functions exists in the library. The FunctionCall specifies the one it needs by checking the type at runtime and passing this information in the FunctionCall to the second argument type. The type of the second argument (and return Type) is the difference in the signatures of the three functions.

4. The SetCursorX FunctionCall is more precise than the library function: the FunctionCall holds the (int32) information. This is not a limitation of the function called, but we will need to keep this information after the refactoring of the YAFL file.

5. Each VaryingStringTypeDescriptor has its own FunctionCall: three descriptors are called, each by its implementation of FunctionCall. It would be possible to make a single one and get the information on the specific function the runtime needs. The choice made here was to keep them separated, probably for clarity.

6. Compare changes the parameter method of SetObject and returns a FunctionCall: the FunctionCall methods are directly called on the RESULT element of the YAFL function. Additionally, we see the use of GetStaticDescriptor method inside the SetObject parameter, contrary to the previous GetValueExpression.

7. The String of Next is constructed by a function and a local variable takes the object position: a function called RuntimeCall construct the CREATE(String) part of the FunctionCall. Additionally, the parameter of SetObject uses a local variable and present similarities with the parameter construction of the SetObject of the close example.

These examples are representative of the FunctionCall implementations inside YAFL files. Our objective is to turn these into usable patterns that can be used

to generate the FunctionCalls presented. Once the result is abstract enough to represent a broad implementation design, we refactor the YAFL files of the examples to use the generation.

2.3 Design

In order to present the changes to the legacy FunctionCalls in the YAFL files, we first have to understand the rationale behind the design of the solution. Our idea is to get rid of the hand coded FunctionCall (and the design of Figure 2.21) for an automatic generation of the data structure. The YAFL files will then refer to the generated FunctionCallBuilder file (as presented in the design of Figure 2.22) instead of the previous direct FunctionCall implementation. Such organization will allow the developer to have more feedback when they need a FunctionCall; as the generated file will provide them an already constructed FunctionCall inside a callable function. In the YAFL editor, the developer will have a visual notice of error should they give too many arguments or a wrong intermediate function name.

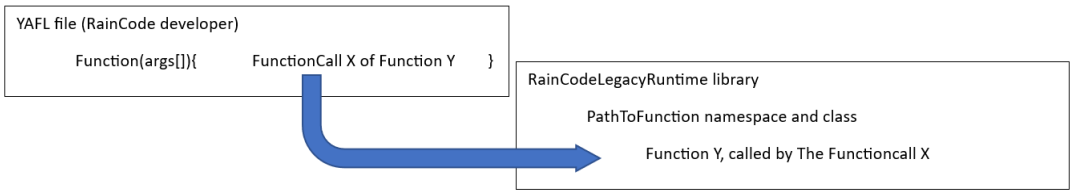


Figure 2.21: Current implementation of a Functioncall X

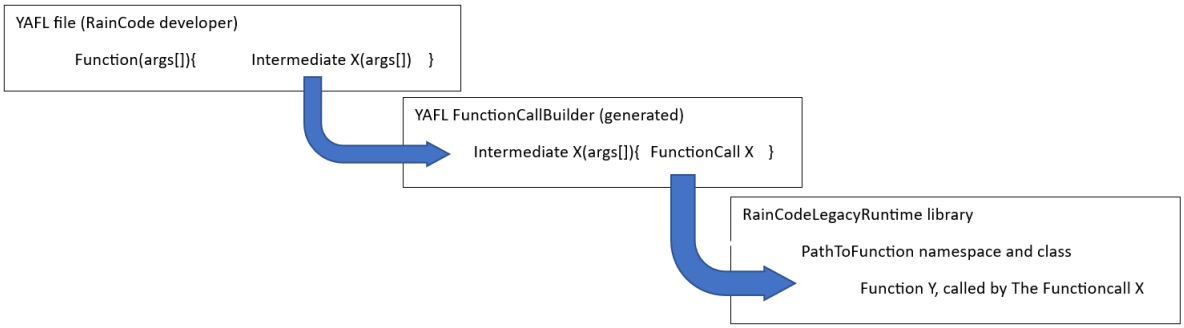


Figure 2.22: Solution idea: generating the Functioncall

The complete explanation will be organized in a few parts that will be presented hereunder, each in their own section.

First, the tools needed to implement this solution. Then the design of the solution files will be constructed. In Chapter 3, an example will be given covering the hand coded implementation, the generation idea and the modification applied in YAFL. This illustrates the adaptation process from the previous FunctionCall to a call towards the FunctionCallBuilder.

2.3.1 Attributes and reflection

In C#, an attribute is a declarative flag put above the element it is linked to [6], similarly to the concept of annotations in Java. In this text, the term annotation will also be used to refer to the attribute declarations.

The .NET framework allows us to create our own custom attribute [6], which we will name CompilerCalledAttribute for the proposed solution. The definition of this attribute is located in *RainCodeLegacyRuntime*, the function library we want the attribute for. In the definition, we can set the object the attribute will be associated with. In our case classes and methods can be annotated. We also indicate the list of modes we use for the generation and the optional information each instance of the CompilerCalledAttribute attribute can have.

The linked element is, in our case, a *RainCodeLegacyRuntime* function as we have already seen, explicitly named in YAFL with a String template similar to: "[RainCodeLegacyRuntime]RainCodeLegacyRuntime.PathToFunction::FunctionCalled". The annotations are directly specified in the library above the function called without the need to precise the path (or rather, the namespace). When retrieving the attributes, all linked functions in the *RainCodeLegacyRuntime* library will be found. The presence of the attribute and mode(s) will indicate the information we want to retrieve for the generation, that we will base on the previous manual FunctionCall in YAFL files.
When adding the attribute, we can set different *GenerationMode* modes. We use different modes to pass different kinds of information: the (basic) fact that we want to create a FunctionCall with the String, the fact that this function has arguments to be appended in the FunctionCall, or as explained later in section 2.2.4, the presence of the FuncCall.SetType(Type) line in the FunctionCall. The information we want to convey with modes will refer to complementary data rather than separated implementations: the FunctionCall template outline will stay consistent, but the information inside it changes with the function signature and mode called. Not every piece of detail can be known by reflection, therefore the modes will pass the additional information needed at runtime.

The concept of attributes was already used in the compiler, however with limited

intentions. The purpose, before we introduced our CompilerCalledAttribute, was not to have a specific warning when compiling the compiler ("compiler compilation").

The attributes system allows us to use reflection [7] on annotated functions. Reflection provide access at runtime to the signature information of the functions. This list of elements includes the name of the function in the C# library, the types of both its parameters and its output and even its visibility. These elements makes it possible to store the required data and organise the template (see section 2.3.3) for the generation program.

2.3.2 BackendGenerator

We created a new solution in Microsoft Visual Studio named *BackendGenerator*. This contains several files we will talk about: Generation_from_annotated_functions, List_of_FunctionCall_from_signature and Intermediate_calls, related as in the architecture of Figure 2.23.

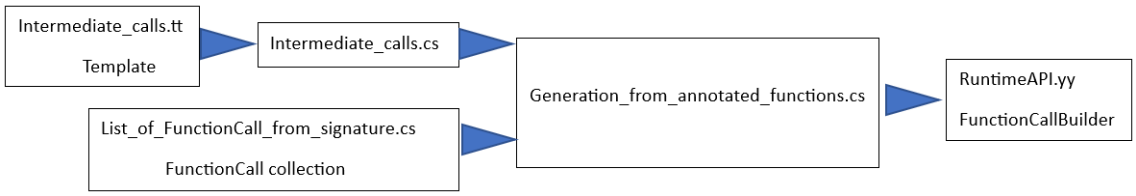


Figure 2.23: *BackendGenerator* architecture from the three initial files to the generated result with FunctionCalls

List_of_FunctionCall_from_signature holds the elements and containers to put the information we find by reflection on the attributes. For the moment, the information contained consists in a collection called *FunctionCall* with the various elements of a FunctionCall for each marked function entry of the list. This includes the name of the function, a list of arguments and their type, the need of a *Variable* argument, of an object and the type of the output. In Generation_from_annotated_functions, we retrieve reflection information on the annotated functions and fill the collection with the elements retrieved. Intermediate_calls.tt is the template for the FunctionCallBuilder file and therefore, may not contain extensive computational logic.

Generation_from_annotated_functions is the core of this solution: it creates the *FunctionCall* collection by retrieving the elements linked with the *CompilerCalled* attribute and mode(s) from the C# classes of *RainCodeLegacyRuntime*. After that, it initiates the generation according to the template (see next paragraph

for more detailed explanation on this non C# file) and stores the result in a YAFL file.

2.3.3 T4

I already presented the concept of attributes, its main use is to convey information to the runtime about the behavior required for the linked element. We need an additional tool to help us with the coding of the generation for each function:

T4, for the initials TTTT, abbreviation of Text Template Transformation Toolkit, will allow us to write an template file in T4 that generate the C# file *Intermediate_calls.cs* (see architecture of the *Backendgenerator* in Figure 2.23), which generates the YAFL file with FunctionCallBuilders, called RuntimeAPI.yy. T4 is a template-based text generation tool directly available in Microsoft Visual Studio, described as a "mixture of text blocks and control logic that can generate a text file" [8]. As written, it will allow us to run code from inside a .tt extension file and only create one general function template with adaptable parts. Those parts will be replaced by the function data in the C# file List_of_FunctionCall_from_signature, according to the *GenerationMode* modes and the accessible reflection information. The C# code correspondence is quite unclear and redundant, where coding with T4 is concise and cleaner.

The *Intermediate_calls.cs* file would be quite redundant and annoying to write, as the functions will follow the same principles of definition between each other. Having to repeat C# lines of Write(...) for each line of text that will appear in the generated file could lead to errors. It would also render this C# file hard to read for a developer and complex to maintain. Additionally, we want to automate it, therefore we will use T4 to process the file *Intermediate_calls.tt* as in the example Figure 2.24 which generates the *Intermediate_calls.cs* file, which is the one generating the output file, *RuntimeAPI.yy*, with the generated functions (or FunctionCallBuilders). As you will see in the example, the .tt content is close to the output and therefore, easier to understand.

The output for the generation of the builtin_date FunctionCall is given in Figure 2.26 and is based on the signature of the function in Figure 2.25.

```
IMPLEMENTATION MODULE RuntimeAPI;
 FROM RCompAllocation  IMPORT AllocationScope;
 IMPORT DotNetTypes;
 IMPORT DotNetFunctionCall;

  ONCE CLASS Intermediate_calls;

<#
    foreach(var m in Collection.List) { //begin for Methods
    var ccAttr = m.Reflect.GetCustomAttributes(typeof(RainCodeLegacyRuntime.BackendAnnotations.CompilerC
    if((ccAttr.GenerationMode & RainCodeLegacyRuntime.BackendAnnotations.GenerationMode.Create) != 0) {
#>
    METHOD <#=m.Builder#>_FunctionCallBuilder(Func: DotNetFunction.Function; Scope: AllocationScope<#
    if(m.Args != null){//append C# function parameters
    foreach(var p in m.Args){#>;
                                                <#=p.Name#>: DotNetExpression.Expression<#}#>): <#
    if(m.Reflect.ReturnType == typeof(void) ){ // inf output known type, expression
#>
DotNetExpression.Expression;
<#
    }else{ // if output void, statement
#>
DotNetStatement.Statement;
<#
    }}
#>
    VAR
      FuncCall: DotNetFunctionCall.FunctionCall;
    BEGIN
      FuncCall.CREATE("[RainCodeLegacyRuntime]<#=m.Reflect.DeclaringType#>::<#=m.Name#>");
<#
    if(!m.Reflect.IsStatic) { // begin setobject
#>
      FuncCall.SetObject(Func.<#=m.Object#>);
<#  } // end setobject
    if(m.Var == "Func.ExecutionContextVar"){ // begin AppendVariable
#>
      FuncCall.AppendVariableArgument(Func.ExecutionContextVar);
<#
    } // end AppendVariable
    if(m.Args != null){
    foreach(var p in m.Args){ //begin for arguments
    if(p.Ref){#>
      ASSERT <#=p.Name#>.GetType.GetUnderlyingType = <#=p.Type#>;
      FuncCall.AppendReferenceArgument(<#=p.Name#>);
<#;}else{#>
      ASSERT <#=p.Name#>.GetType.GetUnderlyingType = <#=p.Type#>;
      FuncCall.AppendArgument(<#=p.Name#>);
<#  }
    }
    } // end for arguments
      FuncCall.SetType(<#=m.Type#>);
      RESULT := FuncCall;
    END <#=m.Builder#>_FunctionCallBuilder;

<#
    } // end if gen for this function
    } // end for Methods #>
  END Intermediate_calls;
END RuntimeAPI;
<#+
   public Collection Collection {get; set;}
#>
```

Figure 2.24: T4 template code for FunctionCallBuilder

We detail here the code of Figure 2.24 for the builtin_date signature of Figure 2.25. On line 14, we can spot the for loop to use the template following this line on each function registered in the List_of_Functioncall_from_signature. Above this line are the imports (lines 7 to 9) that will be used by the FunctionCalls of the class Intermediate_calls declaration (line 11). We then begin to create specific FunctionCall: line 18, <#=m.Builder#> will be replaced by the value of name for the current function of the loop, we do not just use <#=m.Name#> as we will explain in section 3.1.4. Hence in Figure 2.26, on line 6, the name of the function becomes builtin_date_FunctionCallBuilder (for the builder construction pattern [9]). The third parameter is an expression named format from the builtin_date declaration on line 56 in Figure 2.25. We find this information in the collection thanks to line 20 in Figure 2.24. Lines 24 and 28 are the two possible outputs possible of the FunctionCall: builtin_date return a string so we will obtain an *Expression* as seen in Figure 2.26 on the end of line 8.

Regarding T4 characteristics, we can deduce that all parts inside <# #> are computational logic to be executed. When there is an equal, as mentioned above for <#=m.Name#>, this part will be replaced by the value found from the collection and written in the output.

We now continue our explanation with the body of the builder function. The first element generated is the CREATE method with the String parameter. The T4 code can be seen on line 35 in Figure 2.24 and is present with the correct modifications in Figure 2.25 on line 13.
In Figure 2.24, on line 37, the template retrieves the static information from the list. Static is defined in the signature of the function in Figure 2.25: reflection can get the information and pass it to the list.
In Figure 2.24, on line 46, the code checks the presence of data for arguments of the current function generated. If not, there is no argument to append in the FunctionCall, and lines 43, 49-50 and 52-53 are not needed in the output of this function. On lines 49 and 52, the generation makes sure that the type of the argument matches the one needed by the function, ensuring that no error is introduced during the creation of the FunctionCall. The lines of argument will be repeated for each parameter found for the current function thanks to the for loop declared on line 47. For builtin_date in Figure 2.26, we get the lines 14 and 15-16 with the content in parentheses taking the value found from the equal template logic.
The output Type of the FunctionCall is specified on line 57. By default, the Type is assumed to be void, therefore the line is unnecessary for void return type functions, but all generated FunctionCalls will have their type explicitly written. In Figure

2.26 on line 17, the type correspond to the return type of the signature on line 56 of figure 2.25.
The Method is closed on line 59, with the same replacement as for the declaration on line 18 in Figure 2.24. We end the class and module on lines 64 and 65 respectively. We retrieve these lines in Figure 2.26 on lines 19, 21 and 22 respectively.
We want the output to be a proper YAFL file. We want to have the methods with the FunctionCall callable from the legacy YAFL files of the compiler. The line 67 in Figure 2.24 informs the template file about the collection it should use for the replacements of the equal template logic in T4.

```
55          [CompilerCalled( GenerationMode.Create )]
            1 reference
56          public static string builtin_date(ExecutionContext ctx, string format)
57          {
```

Figure 2.25: *RainCodeLegacyRuntime* entry of the function builtin_date, with its attribute

```
1  IMPLEMENTATION MODULE RuntimeAPI;
2   FROM RCompAllocation  IMPORT AllocationScope;
3
4    ONCE CLASS Intermediate_calls;
5
6      METHOD builtin_date_FunctionCallBuilder(Func: DotNetFunction.Function;
7                                              Scope: AllocationScope;
8                                              format: DotNetExpression.Expression):
9                                              DotNetStatement.Statement;
10     VAR
11       FuncCall: DotNetFunctionCall.FunctionCall;
12     BEGIN
13       FuncCall.CREATE("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Language.Cobol.BuiltIn::builtin_date");
14       FuncCall.AppendVariableArgument(Func.ExecutionContextVar);
15       ASSERT format.GetType.GetUnderlyingType = DotNetTypes.NativeStringType;
16       FuncCall.AppendArgument(format);
17       FuncCall.SetType(DotNetTypes.NativeStringType);
18       RESULT := FuncCall;
19     END builtin_date_FunctionCallBuilder;
20
21   END Intermediate_calls;
22 END RuntimeAPI;
```

Figure 2.26: Result of the template for the annotated function builtin_date

To summarize, we need *Template.cs* to generate what we want, but writing it directly is problematic so we generate it thanks to T4, present in Microsoft Visual Studio, processing the file *Template.tt*. This .tt file is the one that will be modified according to the behaviour we want the program to have: in there, we can retrieve the *GenerationMode* information and create a proper template reflecting the mode(s) associated with the functions of the library we generate the FunctionCall for.

2.4 Conclusion

This concludes the sections covering the design of this chosen solution.

In summary, in this chapter, we presented different FunctionCall constructions from the legacy YAFL files of the compiler. We detailed the various implementations we can encounter. We then presented the solution proposed, with the creation of a FunctionCallBuilder intermediate between the legacy code and the C# library function that contains the FunctionCall. This FunctionCall is generated, and we need to annotate with attributes the functions we want to produce a FunctionCallBuilder method for. Once annotated, we use reflection to store the signature information of these functions in a list. This list will be used by the template code written in T4. Finally, the legacy code has to be refactored to call the functions of the generated FunctionCallBuilders file.

Chapter 3

Implementation

Second chapter presented the idea behind the FunctionCallBuilder and the various tools needed. In this section, an extensive example presents the generation of the FunctionCallBuilder and the modifications to the legacy code. It shows the various decisions made for the generation and explain the purpose of the lines constituting the FunctionCallBuilder. We will discuss more advanced design choices, either to enhance the solution or to resolve a problem. Lastly, other patterns solution will be presented and compared to the initial solution.

3.1 Solution process of the first pattern

After the design of the process and the explanation of the tools we use, we take time to talk about the design of the generated FunctionCalls inside the FunctionCall-Builder file. This file will contain the FunctionCalls needed for the library functions. Each FunctionCall will be inside a YAFL function, callable by the legacy compiler code. This legacy code will see its previous FunctionCall structure be transformed into a call to the FunctionCallBuilder containing the generated Functioncalls (more information about this phase in section 3.2).

As mentioned, we list the functions data in a defined structure and associate them with the GenerationMode specified for each. For the output, we then iterate over those functions following the T4 template. Several decisions regarding this template and what has been found inside the compiler have been made. Consider the following example from function builtin_date in Figures 3.1 and 3.2:

```
338   METHOD CompileAcceptFromDate (Func: DotNetFunction.Function;
339                                 Scope: AllocationScope): DotNetExpression.Expression;
340   VAR
341     FuncCall: DotNetFunctionCall.FunctionCall;
342   BEGIN
343     FuncCall.CREATE ("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Language.Cobol.BuiltIn::builtin_date");
344     RESULT := FuncCall;
345     FuncCall.SetType (DotNetTypes.NativeStringType);
346     FuncCall.AppendVariableArgument(Func.ExecutionContextVar);
347     IF TheFormat <> VOID THEN
348       FuncCall.AppendArgument(NEW DotNetLiteralExpression.NativeStringExpression (TheFormat));
349     ELSE
350       FuncCall.AppendArgument(NEW DotNetLiteralExpression.NativeStringExpression ("YYMMDD"));
351     END;
352   END CompileAcceptFromDate;
```

Figure 3.1: Legacy YAFL code with builtin_date

```
54    [CompilerCalled( GenerationMode.Create )]
      1 reference
55    public static string builtin_date(ExecutionContext ctx, string format)
56    {
57        DateTime n = ctx.Now;
58        if ((format != null) && string.Compare(format, "YYYYMMDD", StringComparison.OrdinalIgnoreCase) == 0)
59            return n.ToString("yyyyMMdd");
60        else if ((format != null) && string.Compare(format, "YYYYDDD", StringComparison.OrdinalIgnoreCase) == 0)
61        {
62            return n.Year.ToString("0000") + n.DayOfYear.ToString("000"); ;
63        }
64        else if ((format != null) && string.Compare(format, "YYDDD", StringComparison.OrdinalIgnoreCase) == 0)
65        {
66            return (n.Year % 100).ToString("00") + n.DayOfYear.ToString("000"); ;
67        }
68        else
69            return n.ToString("yyMMdd");
70    }
```

Figure 3.2: Library entry for the builtin_date example

In Figure 3.1 we can see the creation of a FunctionCall for Builtin_date on line 343, followed by the setter of the return type on line 345, the append of the context argument on line 346, and a regular argument on line 348 or 350. From the declaration of the library function of Figure 3.2 on line 52, we have the static specificity, the type output, a String, and the two arguments. From this declaration, we generate a YAFL file with the intermediate function (see Figure 2.22) using the corresponding created FunctionCallBuilder in Figure 3.3:

```
1  IMPLEMENTATION MODULE RuntimeAPI;
2  FROM RCompAllocation  IMPORT AllocationScope;
3
4  CLASS SimpleMethods;
5
6  METHOD builtin_date_NAME(Func: DotNetFunction.Function;
7                           Scope: AllocationScope;
8                           formatString: DotNetExpression.Expression):
9                           DotNetExpression.Expression;
10 VAR
11     FuncCall: DotNetFunctionCall.FunctionCall;
12 BEGIN
13     FuncCall.CREATE("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Language.Cobol.BuiltIn::builtin_date");
14     FuncCall.AppendVariableArgument(Func.ExecutionContextVar);
15     FuncCall.AppendArgument(formatString);
16     ASSERT formatString.GetType.GetUnderlyingType = DotNetTypes.NativeStringType;
17     FuncCall.SetType(DotNetTypes.NativeStringType);
18     RESULT := FuncCall;
19 END builtin_date_NAME;
20
21 END SimpleMethods;
22 END RuntimeAPI;
```

Figure 3.3: Example of output we aim to generate from the example builtin_date

3.1.1 Function signature

We now break down the general template used and refer to the example of Figure 3.3:

It shows that we first import AllocationScope from RCompAllocation for the whole YAFL file on line 2. This is a common import for the YAFL functions[1]. To continue with this AllocationScope on line 7, the first line of all the functions generated will take the form:
METHOD Example_NAME (Func: DotNetFunction.Function; Scope: AllocationScope; Example_PARAMETERS): DotNetTypeOutput.TypeOutput; where Func and Scope are common among the functions found. In Figure 3.3, each arguments has its line, from line 6 to 8, for clarity. Additionally, some functions require more parameters, which justify the plurality of the Example_PARAMETERS above. In our example we have the formatString on line 8, but other *RainCodeLegacyRuntime* functions may request more arguments or only need the first two. One last note is the absence of mention regarding the output type declaration in this paragraph despite its place at the end of the function declaration in YAFL on line 9. We have to detail this element in the next paragraph.

The output can be of 3 types: FunctionCall, Expression or Statement. FunctionCall comes from the implementation choices made during the creation of the YAFL function, if it does not use an intermediate variable and the FunctionCall methods are cast on the RESULT element.
Expression and Statement take place if the function follows the same principle

[1]We may note the possibility of using the path RCompAllocation.AllocationScope directly in the function declaration instead of the importation proposed here.

we will use in the example template: a FuncCall named variable takes the call construction and RESULT is assigned afterward. The difference between Expression and Statement comes from the output "type" of the called *RainCodeLegacyRuntime* function itself, respectively a known Type or Void. For our example, line 9 is an Expression because the output of builtin_date is a String, as seen in line 55 of Figure 3.2. This concludes the function signature.

3.1.2 Function body

In YAFL, variables are introduced in their respective VAR space. This is where the FuncCall variable is instanced, on line 11 in Figure 3.3. The choice made to use an intermediate variable for the RESULT part is respected.

The next step is code of this Builder. The main idea behind the thesis is to make sure the FunctionCall is properly constructed: we need to make sure all FuncCall.methods() are correct. We begin with the FuncCall.CREATE(...) which requires the String of the correct path (see section 2.2 for more context) and the name of the *RainCodeLegacyRuntime* function. As this FunctionCall is generated using reflection, it ensures the correct name and path for the String part.
Then, the order of the FunctionCall methods as seen in Figure 3.3 will follow the information we seek from the call.
First, SetObject() appears if the function is not defined static. Its role is to provide additional information to the function, not necessarily accessible before the FuncCall is made, e.g. the object of the method called by the Functioncall. Our example does not need this line since the builtin_date function Figure 3.2 is defined as static. Lines 14 to 16 show how we handle the function arguments, and this information is obtained from the data list. To fill this need, 3 different Append methods exists:

- `AppendArgument`, which is the most basic form, indicates the object to take as an argument,
- `AppendReferenceArgument` which indicates the reference to the object to take as an argument,
- `AppendVariableArgument` which indicates the context and is placed in the first place(s) of the arguments order

This `AppendVariableArgument` might be complex to grasp, but is actually easier to code as it will, for the cases that interest us, always be defined with the `Func.ExecutionContextVar` parameter, as seen on line 14 of Figure 3.3. Func comes from the first argument of the YAFL function and the ExecutionContextVar

is being explicitly named: it gives the FunctionCall the execution context in which it is called as a variable. In our example, we have such setting line 14. Regarding the other two Appends, we will have to make sure the type generated corresponds to the one requested inside *RainCodeLegacyRuntime*. To do so, we use an ASSERT comparing the formats for each of those arguments (line 16 in the output example, for the argument of line 15).

Lastly, the entry might have an output of a known Type or Void. We already discussed this point in section 3.1.1, it results in generating an output type of either Expression or either Statement. SetType() will allow us to be more precise: in case of a known type SetType() will appear with the Type concerned. In our example, we have an Expression output indicated on line 9, and the SetType() will inform about the actual output Type, on line 17 in Figure 3.3.
Some YAFL functions of the legacy code will have a SetType(DotNetTypes.VoidType) inside their body as in Figure 3.4 on line 421. The default Type if the line is not present in the FunctionCall is already SetType(DotNetTypes.VoidType). Since we are generating the FunctionCall, we can explicitly indicate the return type for all function annotated. This is the choice we made and each generated builder will contain the SetType method.

```
414  METHOD CompileAcceptFromFile(Func: DotNetFunction.Function;
415                               Scope: AllocationScope;
416                               Target: Designator): DotNetStatement.Statement;
417  VAR
418    FuncCall: DotNetFunctionCall.FunctionCall;
419  BEGIN
420    FuncCall.CREATE("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.IO.IORuntime::AcceptFrom");
421    FuncCall.SetType(DotNetTypes.VoidType);
422    FuncCall.SetObject(Func.IOFacilityLocal.GetValueExpression);
423
424    FuncCall.AppendArgument(NEW DotNetLiteralExpression.NativeStringExpression(TheFromId.GetImage));
425    FuncCall.AppendArgument(Target.CompileAddress(Func, Scope));
426    RESULT := FuncCall;
427  END;
```

Figure 3.4: Legacy FunctionCall with a SetType(Void) for AcceptFrom

Finally, we close the function on line 19 in Figure 3.3 and may start another function. At the end of the file, we have to close the class on line 21 and the module on line 22.

3.1.3 Type conversions

For the arguments, argument references and output inside the FunctionCallBuilder, their associated type is based on conversions from the type requested by the C# function to the YAFL correspondence. At generation time, since the "compiler compilation" did not occur, the YAFL DotNetTypes do not exist yet. However, they are the ones needed for the Builder file. The conversion process checks the type found and provide to the template the correct DotNetType needed. For that, we use a dictionary where the key is the type found from the C# signature and the value is the corresponding DotNetType in a String format. This DotNetType as a String is the value used in the generation. We may note that not every utilisation of the FunctionCall in the YAFL files requires the same converted type. In that case, the most general type for all usages will be the one generated. This choice is made because such occurrences are rather scarce in the code: the function has to be used in both the COBOL and PL/I compilers.

The Object definition element get the same treatment. However, its conversion only occurs for the basic pattern. Other patterns covering specific Object definition needs will be further detailed in sections 3.3.1, 3.3.2 and 3.3.4

3.1.4 Name mangling

In C#, it is possible for two functions to share the same name, as we could see with the two close functions of section 2.1.1. YAFL does not allow such liberty. As such, we have to modify the generation to differentiate between two similarly named functions.

One idea would be to take advantage of the parameters of the functions. In section 2.1.3, the three Builtin_abs functions illustrate the most common case of name repetition in the C# library: the functions differ only in their last parameter type. The example builtin_abs has one function for each of a decimal or binary fixed value or a double. The library contains various example of such functions.

We could set an indication on the parameter that differ in type, but it would render the code less clear. The indication would be put in the middle of the function signature and hinder the comprehension. Such implementation is also less flexible: the information transmitted would be the varying parameter type. The thesis work aims to facilitate the maintenance and evolution of the company code, so we want the generation to be easily understood. As such, another choice was made: the developer working with the generation can provide additional information in the

form of a string after the GenerationMode(s) of the attribute.

An example of such annotation can be seen in Figure 3.5.

```
[CompilerCalled(GenerationMode.Create | GenerationMode.FileRef , "base")]
18 references
public void close()
{
```

Figure 3.5: Library signature of the example close builder, with its GenerationModes (Create and FileRef) and the additional information for its generated name

The String is then used by the generation and put in the Builder function name in the format : "CalledFunctionName_AttributeString_FunctionCallBuilder". As such, the name of the close builder is adapted in Figure 3.6.

```
METHOD close_base_FunctionCallBuilder(Func: DotNetFunction.Function; Scope: AllocationScope;
                                      FileRef: DotNetVariables.Variable): DotNetStatement.Statement;
```

Figure 3.6: Signature generated for the example close builder

This implementation choice makes the developer responsible for the naming of the functions he will use. It will allow changes to be made more easily than surcharging the signature of the function.

3.2 Refactoring the YAFL files

We covered the generation part of the solution, which needed a lengthy explanation as a variety of tools are used. However, the output generated is not enough alone. It is an intermediate library that we will call from the YAFL files instead of the previous FunctionCalls and String reference used (see the architecture change from Figures 2.21 to 2.22). This part of the work will be manual as patterns cannot adapt to each and every possibility. We will use the same example as the previous section for the refactoring [10] of Figure 3.7

```
333   METHOD CompileAcceptFromDate (Func: DotNetFunction.Function;
334                                 Scope: AllocationScope): DotNetExpression.Expression;
335   BEGIN
336     IF TheFormat <> VOID THEN
337       RESULT := Intermediate_calls.builtin_date_FunctionCallBuilder(Func, Scope, NEW DotNetLiteralExpression.NativeStringExpression(TheFormat));
338     ELSE
339       RESULT := Intermediate_calls.builtin_date_FunctionCallBuilder(Func, Scope, NEW DotNetLiteralExpression.NativeStringExpression("YYMMDD"));
340     END;
341   END CompileAcceptFromDate;
```

Figure 3.7: Example of refactoring we aim to replace the legacy FunctionCall to builtin_date with

As we can see from the refactoring example, the former method from Figure 3.1 is changed to Figure 3.7. The modified code contains a call to the FunctionCallBuilder from Figure 3.3.
The two principles for the refactoring in this thesis are:

- The declaration of the upper function do not change, only its body do. The lines 333-334 in Figure 3.7 correspond to the lines 338-339 in Figure 3.1.

- No reference to the previous FunctionCall is left, only the new reference to the FunctionCallBuilder function, on lines 337 or 339 in Figure 3.7, due to the IF statement on line 347 of Figure 3.1.

The former calls are being generated in a Builder and therefore correct. The calls from the YAFL files can be changed to calls to the generated function, providing direct feedback at development time. For example, consider what happens when listing too many arguments: with the previous way, the amount of append(s) is not checked. A crash happens at compile time of client code, and even then the report will not explicitly identify the error location. Here, we will get rid of the whole manual FuncCall creation and instead make a call to another YAFL file function which will be checked at compilation time of the compiler.

As mentioned, the refactoring work will have to be repeated for each legacy call of the YAFL files. We can encounter more complex functions that might need reformatting and reordering. But while doing so, we will now have the possibility to spot errors in the call directed to the FunctionCallBuilder. To retake a previously mentioned example , having too much arguments will now raise a visual warning in the editor.

3.3 Generation patterns

The thesis aims to create a collection of patterns corresponding to the various needs of the FunctionCalls present in Raincode's compiler. The solution presented in section 3.1 covered the majority of needs a basic FunctionCall requires.

More specific patterns are needed for more diverse implementations.

3.3.1 Second pattern: Object parameter is a reference to a file

The second pattern aims to solve the difficulty raised by the SetObject of several functions, like the close of Figure 2.3. The generated FunctionCall of Figure 3.8 will help us identify the differences with the first pattern. In this FunctionCall, the basic SetObject(Func.[...].GetValueExpression) on line 79 would not be valid. The FunctionCallBuilder of close needs an additional parameter called *FileRef*[2]

[2]This name comes directly from the YAFL implementation: this variable name is always used for the reference to a file inside a FunctionCall.

passed on line 74. This *FileRef* is a Variable used as the object for the method GetValueExpression. This combination creates the SetObject() parameter on line 79.

```
73  METHOD close_base_FunctionCallBuilder(Func: DotNetFunction.Function; Scope: AllocationScope;
74                                        FileRef: DotNetVariables.Variable): DotNetStatement.Statement;
75  VAR
76    FuncCall: DotNetFunctionCall.FunctionCall;
77  BEGIN
78    FuncCall.CREATE("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.IO.RecordBasedFile::close");
79    FuncCall.SetObject(FileRef.GetValueExpression);
80    RESULT := FuncCall;
81  END close_base_FunctionCallBuilder;
```

Figure 3.8: Generated FunctionCallBuilder for the example close

In section 3.1, the content of a FunctionCall could be deducted from the signature of the C# function. Since the generation of the FunctionCallBuilder of close requires more information than the reflection can provide us, this is our second pattern. We can resolve the information trasmission problem with a GenerationMode. This GenerationMode take the name *FileRef* of the parameter that could be seen in Figure 3.8. The proposed *FileRef* pattern holds no variation: we add the Variable *FileRef* in the FunctionCallBuilder parameters, on line 74 in Figure 3.8, and fix the Object method to SetObject(FileRef.GetValueExpression) on line 79.

The GenerationMode purpose is to convey the information not accessible from reflection alone. Since no modification is required among the different functions needing the second pattern, every function marked with *FileRef* will follow the changes mentioned above, with no variation.

3.3.2 Third pattern: Object parameter is a Descriptor

Again this pattern will modify the SetObject line with a third variation. The two previous patterns both used the GetValueExpression method. This new one will cover the use of the GetStaticDescriptor method.

This method is used for the SetObject but also in AppendArgument lines as we could see in the *Compare* function of section 2.1.6. As we stated in section 3.3.2, we consider a new pattern if it needs a GenerationMode to transmit information. Only the SetObject line will modified according to the attribute, the AppendArgument is covered via the conversion of section 3.1.3.

The *Compare* functions example of Figures 2.17 and 2.18 are covered by the newly added GenerationMode named *StaticDescSO* on lines 1393 and 1462 respectively. Similarly to the second pattern, it adds a parameter to the FunctionCallBuilder signature as in Figure 3.9 on line 154. The GetStaticDescriptor method is cast on this argument called *StaticDescSO* . The parameter of SetObject on line 160 becomes StaticDescSO.GetStaticDescriptor.

```
153   METHOD Compare_descriptor_FunctionCallBuilder(Func: DotNetFunction.Function; Scope: AllocationScope;
154                                                 StaticDescSO: RCompType.Type;
155                                                 m: DotNetExpression.Expression;
156                                                 v: DotNetExpression.Expression;
157                                                 desc: DotNetExpression.Expression): DotNetFunctionCall.FunctionCall;
158   BEGIN
159     RESULT.CREATE("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Descriptor.FixedDecimalDescriptor::Compare");
160     RESULT.SetObject(StaticDescSO.GetStaticDescriptor);
161     RESULT.AppendVariableArgument(Func.ExecutionContextVar);
162     ASSERT m.GetType.GetUnderlyingType = DotNetTypes.MemoryAreaType;
163     RESULT.AppendReferenceArgument(m);
164     ASSERT v.GetType.GetUnderlyingType = DotNetTypes.MemoryAreaType;
165     RESULT.AppendReferenceArgument(v);
166     ASSERT desc.GetType.GetUnderlyingType = DotNetDescriptorTypes.FixedDecimalDescriptorType;
167     RESULT.AppendArgument(desc);
168     RESULT.SetType(DotNetTypes.IntType);
169   END Compare_descriptor_FunctionCallBuilder;
```

Figure 3.9: Generated FunctionCallBuilderfor the example of the second Compare, labeled with "descriptor"

The third pattern ends here, surprisingly, as not all difference between Figure 3.9 and the basic solution are presented. To completely cover the Compare example taken, we need to explain the fourth pattern.

3.3.3 Fourth Pattern: returning a FunctionCall

In section 3.1.1, we talked about the signature of function in YAFL displaying the return type. Out of the three possibilities, we covered the Expression and Statement ones. The third type mentioned was FunctionCall, which is the type returned by the *Compare* FunctionCallBuilder example on line 157 in Figure 3.9.

The reason such output exist comes from the RESULT element. In the case of the *Compare* example, the method .CREATE is called on RESULT. This assigns RESULT to the FunctionCall Type, and returns a FunctionCall Type output.

In the legacy code such implementation requires the FunctionCall methods to be called directly on the RESULT element. This element specify to the YAFL function what needs to be returned. In our previous patterns, the Variable FuncCall handle

the methods and then RESULT is assigned to the result of FuncCall. We need to modify the generation code in order to apply the methods directly on RESULT and get rid of the assignation line: this introduces the fourth pattern. The GenerationMode is called FCreturn, and will be used for the generation of the Compare functions.

We can see in Figure 2.18 the presence of the third and fourth GenerationMode: *StaticDescSO* and *FCreturn* on line 1462. The generated function of Figure 3.9 apply both changes. First, The SetObject on line 160 uses the .GetStaticDescriptor method on the additional parameter StaticDescSO passed on line 154. Then, all FunctionCall methods are applied to the RESULT element of the generated Builder, as we can observe from lines 159 to 168 (minus the ASSERT on lines 162, 164 and 166). We can also note the correct absence of the RESULT assignation line. The *Compare* example combines the third and fourth patterns.

3.3.4 Fifth Pattern: Local variable as Object parameter

In the YAFL code, the SetObject parameter used in section 2.2.7 consists in a local Variable and the GetValueExpression method. The local Variable named Loc on line 1995 in Figure 2.19 is defined near the FunctionCall creation. This pattern presents similarities with the second one using *FileRef* in section 3.3.1. The generation requires an additional parameter in the Builder signature, this parameter is used in the SetObject with the GetValueExpression method. The GenerationMode is named after the name of the Variable Loc and an example of generated Builder can be seen in Figure 3.10.

```
74    METHOD Next_FunctionCallBuilder(Func: DotNetFunction.Function; Scope: AllocationScope;
75                                    Loc: DotNetVariables.Variable): DotNetExpression.Expression;
76    VAR
77      FuncCall: DotNetFunctionCall.FunctionCall;
78    BEGIN
79      FuncCall.CREATE("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Language.Cobol.XmlParse::Next");
80      FuncCall.SetObject(Loc.GetValueExpression);
81      FuncCall.SetType(DotNetTypes.BooleanType);
82      RESULT := FuncCall;
83    END Next_FunctionCallBuilder;
```

Figure 3.10: Generated FunctionCallBuilderfor Next with the Local Variable Loc

The proposed *Loc* pattern holds no variation: we add the Variable *Loc* in the FunctionCallBuilder parameters, on line 75 in Figure 3.10, and fix the Object method to SetObject(Loc.GetValueExpression) on line 80.

3.3.5 Sixth Pattern: Type indicated in the Builder parameters for the last argument and the return Type

The previous patterns heavily focused on the Object definition line. We consider the Type definition line for this pattern.

In section 2.2.3, we presented the three possibilities of builtin_abs in C# from one FunctionCall in YAFL. We can keep the same mechanism with the sixth pattern.

```
44  REDEFINE METHOD Compile (Func: DotNetFunction.Function;
45                           Scope: AllocationScope): DotNetExpression.Expression;
46  VAR
47    FCall: DotNetFunctionCall.FunctionCall;
48  BEGIN
49    FCall.CREATE("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Language.Common.BuiltIn::builtin_abs");
50    FCall.SetType(GetRCompType);
51    FCall.AppendVariableArgument(Func.ExecutionContextVar);
52    FCall.AppendArgument(GetParam(0).Compile(Func, Scope)!CastTo(Func, Scope, GetRCompType!ToAbstractType));
53    RESULT := FCall;
54  END Compile;
```

Figure 3.11: Legacy YAFL code for the creation of the *builtin_abs* FunctionCall

The legacy structure used the GetRCompType method on lines 50 and 52 in Figure 3.11 in order to choose the correct function. Similarly, if the pattern allows us to specify the Type output, we can recreate this usage inside the generated Builder. We annotate any of the three C# function signatures with *GenerationMode.returnType*. The generation of the FunctionCall structure will include the returnType in the signature of the Builder on line 13 in Figure 3.12. This parameter will allow us to use one Builder for the three C# functions. The pattern supposes we change the Type of the last AppendArgument and the setType according to the returnType value. The AppendArgument Type verification becomes a comparison of Abstract Types in Figure 3.12 on line 20. The setType argument is fixed to the returnType value on line 22.

By convention, the changing argument is the last of the parameters. The generation work suppose this convention in the C# functions is respected for all functions using the returnType GenerationMode, otherwise the pattern will require further modifications.

```
12  METHOD builtin_abs_FunctionCallBuilder(Func: DotNetFunction.Function; Scope: AllocationScope;
13                                         returnType: RCompType.Type;
14                                         d: DotNetExpression.Expression): DotNetExpression.Expression;
15  VAR
16    FuncCall: DotNetFunctionCall.FunctionCall;
17  BEGIN
18    FuncCall.CREATE("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Language.Common.BuiltIn::builtin_abs");
19    FuncCall.AppendVariableArgument(Func.ExecutionContextVar);
20    ASSERT d.GetType = returnType.ToAbstractType;
21    FuncCall.AppendArgument(d);
22    FuncCall.SetType(returnType);
23    RESULT := FuncCall;
24  END builtin_abs_FunctionCallBuilder;
```

Figure 3.12: Generated FunctionCallBuilderfor builtin_abs with the modified assertion and Type lines

3.3.6 Summary of the GenerationModes

This section will summarize the information of the generated patterns seen in sections 3.3.1 to 3.3.5 with the name of their associated attribute mode.

1. Basic: the solution requires only the C# function signature to generate the base structure of a FunctionCall: the creation of the reference to the C# function, the Object if the reference is a method, the Arguments and the return Type. The first pattern is described in subsections of section 3.1 and serves as the basis for the next patterns.

2. FileRef: the SetObject line is modified to a fixed SetObject(FileRef.GetValueExpression), matching the FileRef Variable used in the legacy YAFL code to store a reference to a file. This FileRef Variable is passed in the FunctionCallBuilder signature.

3. StaticDescSO: the SetObject line is modified to a fixed SetObject(StaticDescSO.GetStaticDescriptor). The variable StaticDescSO is passed in the FunctionCallBuilder signature. We may note that it contains the information of a Type defined in the RComp module contrary to the DotNetTypes associated with the GetValueExpression method.

4. FCreturn: most FunctionCalls use an intermediate parameter to cast the FunctionCall methods on and then return this constructed Variable: the return type is either a Statement or an Expression. To return a FunctionCall Type, the methods are directly cast on the RESULT element of the YAFL function.

5. Loc: Similarly to FileRef, the SetObject line is modified to a fixed SetObject(Loc.GetValueExpression), matching the Loc Variable used in the YAFL code to pass information with a Local Variable to the FunctionCall. This Loc Variable is passed in the FunctionCallBuilder signature.

6. returnType: the SetType line is modified to SetType(returnType) and the last argument Type must match returnType. The returnType parameter is passed in the FunctionCallBuilder signature. Such implementation allows us to combine multiple functions with identical name. The changing Type argument and the output Type are specified in the Builder signature, we can narrow the C# function wanted with the YAFL code.

Chapter 4

Validation

Chapter 3 presented the solution proposed and the various patterns implemented. In this section, we take a look at the impact these patterns have inside the compilers.

4.1 Validation method

We already presented the method used to validate the correctness of the work done during this thesis in section 1.8: rely on Raincode's test suite to avoid regressions. The regression tests are a complete verification of the correct behaviour for the compilers and their associated tools. We did not add any new functional test ourselves, as the company has complete confidence in the suite they provide.

4.2 Validation data

To assess the utility of our work, allow us to reuse the Table 4.1 already seen in section 2.2. This table reports the matches found for the expression *"[RainCode* inside the 4 modules.

module	# of matches	# of YAFL files flagged	total # of YAFL files
COBOL	139	25	133
.NET	217	16	28
RComp	63	16	66
PL/I	151	25	140 (before compilation)
total	570	82	367

Table 4.1: Results for the expression *"[RainCode* in the 4 modules concerned at the beginning of the thesis

After our work, the refactoring of the code, the result Table 4.2, shows that the COBOL compiler got the most changes. It is due to the fact that the module was the basis for pattern creation.
Similarly, the second compiler, for PL/I, saw a good reduction in the number of hand coded FunctionCalls. The lower number of results can be explained by the focus on the COBOL compiler.
In the RComp module, the lesser amount of refactoring is due to the structure difference. For example, some AppendArgument lines are replaced with a YAFL method whose output does the role of an AppendArgument line. This makes the refactoring process for this module slower than for the compilers
The .NET module is a special case: the value reported in Table 4.2 does not take the generated Strings of the solution into account. Should we include the 156 matches of our generated file, the number of actual FunctionCall would explode to 364. Very few changes has been made. A large portion of the figures reported comes from classes definition and not the actual data structures we aim to replace.

module	# of matches	# of YAFL files flagged	total # of YAFL files
COBOL	54 (-61%)	17 (-32%)	133
.NET	208 (-4%)	16 (-0%)	28
RComp	49 (-22%)	12 (-25%)	66
PL/I	90 (-40%)	19 (-24%)	155 (after compilation)
total	401 (-30%)	64 (-22%)	367

Table 4.2: Results for the expression *"[RainCode* in the 4 modules concerned at the end of the thesis

4.3 Analysis of the results

Not every matching expression found could be adapted due to various reasons we will try to list here after.

4.3.1 Research limitations

- The grep command used for the Tables 4.1 and 4.2 returns the matches for the expression searched, but it offers no guarantee the found Strings are part of a FunctionCall. We could consider entry such as in Figure 4.1 as noise data from the inventory method. The .NET module, in particular is flooded with definition matching the expression we used for the grep command, as 115 of the 217 results for this module concern classes and not functions for FunctionCalls structures.

```
480     METHOD BaseModuleClassName : STRING;
481     BEGIN
482       RESULT := "[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Module.Language.CobolModule";
483     END BaseModuleClassName;
```

Figure 4.1: False positive entry caught by grep command

- Some FunctionCall definitions refer to a variable getter as seen in Figure 4.2. This differs from a regular C# function entry: in Figure 4.3, the underlined attribute on line 58 indicates that CompilerCalled cannot mark a C# variable. As such, we do not currently include them in the generation.

```
333         FuncCall.CREATE("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Types.RCString::get_Length");
334         FuncCall.SetType (DotNetTypes.IntType);
335         FuncCall.SetObject (exp);
336         RESULT := FuncCall;
```

Figure 4.2: FunctionCall getter for the Length Variable of the RCString class

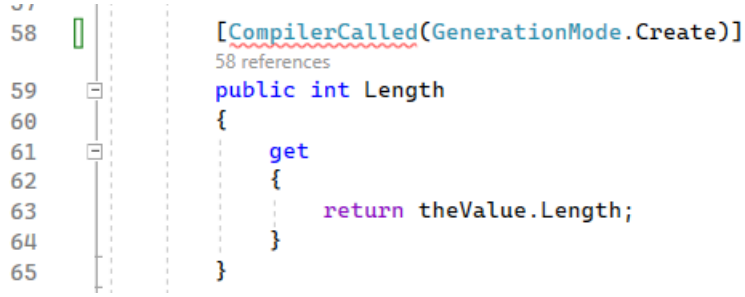


Figure 4.3: Length Variable entry in the RCString class

4.3.2 Current patterns

Some functions cannot be produced based on the patterns presented alone. These will require other iterations of reasoning and implementation.

- The patterns were based on the FunctionCalls mainly present in the COBOL compiler. As the patterns were applied to PL/I, it became apparent an additional pattern could solve a large number of the FunctionCalls exclusive to this compiler. For example, the SetVirtual method on line 997 in Figure 4.4 is never seen in COBOL, but 21 times in PL/I.

```
994 FuncCall.CREATE ("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Core.ExecutionContext::get_IOOperation");
995 FuncCall.SetObject (ExecutionContextVar.GetValueExpression);
996 FuncCall.SetType (IOFacilityType);
997 FuncCall.SetVirtual;
```

Figure 4.4: SetVirtual method inside the PL/I compiler

- A method like AppendVariableArgument is currently fixed in all patterns to the line AppendVariableArgument(Func.ExecutionContexVar);. Some YAFL methods deviate from this parameter as the one in Figure 4.5, we need another pattern or to modify the parameter of line 76 before generating this FunctionCall.

```
73 FuncCall := DotNetFactory.RuntimeCall("Core", "ExecutionContext", "getMemoryAddress");
74 FuncCall.SetType(MemoryAreaType);
75 FuncCall.SetObject(TheExecutionContextVar.GetValueExpression);
76 FuncCall.AppendVariableArgument(GetClass.GetSingletonAttr);
```

Figure 4.5: AppendVariableArgument with an irregular parameter

- Additional methods casted on FunctionCall structure exist in the YAFL code. For example, CastTo on line 135 in Figure 4.6 is a function used in YAFL for type casting.

```
129   REDEFINE METHOD Compile (Func: DotNetFunction.Function;
130                            Scope: AllocationScope): DotNetExpression.Expression;
131   BEGIN
132     LET fc := DotNetFactory.RuntimeCall("Language.Cobol", "BuiltIn", FnName);
133     fc.AppendArgument(GetParam(0).Compile(Func, Scope).CastTo(Func, Scope, DotNetTypes.AbstractIntType));
134     fc.SetType(DotNetTypes.AbstractIntType);
135     RESULT := fc.CastTo(Func, Scope, GetRCompType.ToValueType.ToAbstractType);
136   END Compile;
```

Figure 4.6: FunctionCall casted to another return type

4.3.3 Refactoring slow-down

- The creation of the String for several functions inside the same class can be done with a variable as we can see on line 448 of Figure 4.7. Some YAFL methods take advantage of this, but as no common convention exists for this mechanism, it slows our progress. It forces us to track and verify for longer periods of time, as the refactoring is more prone to test regression.

```
438   REDEFINE METHOD Compile(Func : DotNetFunction.Function; Scope : AllocationScope) : DotNetExpression.Expression;
439   VAR
440     p: RCompExpression.Expression;
441     Arg: DotNetExpression.Expression;
442     FCall: DotNetFunctionCall.FunctionCall;
443     ResType: DotNetTypes.AbstractType;
444   BEGIN
445     p := GetParam(0);
446     ResType := GetRCompType.ToValueType.ToAbstractType;
447     Arg := p.Compile(Func, Scope);
448     FCall.CREATE ("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.Language.Cobol.BuiltIn::builtin_floor_" + ResType.GetName);
449     FCall.AppendVariableArgument(Func.ExecutionContextVar);
450     FCall.AppendArgument(Arg.CastTo(Func, Scope, p.GetRCompType.ToValueType.ToAbstractType));
451     FCall.AppendArgument(TruncateP);
452     FCall.SetType (ResType);
453     RESULT := FCall;
454   END Compile;
```

Figure 4.7: Non explicit FunctionCall, slowing the replacement process

In our example of Figure 4.7, the ResType parameter of lines 443 and 448 contains the information to deduce the library function to be generated. To have this information, we have to find where the redefined method COMPILE is used and track the value of the ResType parameter further in the YAFL code.

Chapter 5

Closing thoughts

Chapter 4 presents the results obtained by the thesis and discusses the factors influencing them. This section aims to close the present thesis with a discussion on the work done and its place inside Raincode 's compilers.

5.1 Contribution

This thesis provided the generation of FunctionCallBuilders corresponding to the patterns described in detail in Chapter 3. The YAFL code with the former FunctionCalls is now refactored to replace them with calls to the generated Builders.

Six Patterns have been created:

1. Basic: this solution covers the basic template a FunctionCall can have with its (possible) object, its parameter(s) and its output type. The information required for this pattern are taken directly from the signature of the library function through reflection.

2. FileRef: it concerns a method call for a file whose reference is known as FileRef. The FileRef Variable is the parameter of the SetObject line inside the FunctionCallBuilder.

3. StaticDescSO: the basic template and most FunctionCall use the GetValue-Expression method for the SetObject parameter. This pattern will use the GetStaticDescriptor method instead.

4. FCreturn: the output type of the FunctionCall is a FunctionCall. It is achieved by using the RESULT element of YAFL as the object of the FunctionCall methods.

5. Loc: Similarly to FileRef, it concerns a method call for a Local Variable called Loc. The Loc Variable is the parameter of the SetObject line inside the FunctionCallBuilder.

6. returnType: Similarly named functions exists inside the library. The difference between them is the type of the output and the last parameter. In the legacy code, a getter for these two places allowed the developers to use any of the similarly named functions with only one FunctionCall coded. This pattern reproduces this mechanism. The type of the output and the last parameter is taken as parameter of the FunctionCallBuilder.

This work removes the need for developers to encode the FunctionCalls by hand, and allows them to get rid of possible human errors. Additionally, the C# functions used in YAFL are now annotated with attributes corresponding to the pattern used in the generation. Having the function used marked will facilitate the maintenance and enhancement of the *RainCodeLegacyRuntime* library.

5.2 Pattern choices

The patterns presented are the consequences of our vision for this work. Changes can be made and be equally valid. The subsections aim to discuss the choices kept and the possible implications or changes available.

5.2.1 GetValueExpression patterns

The base solution, second and fifth patterns all use the same GetValueExpression method for their SetObject defining line. The choice made for the solution presented was to ease the work of the developer for the base pattern: the line is generated from the reflection data alone. The second and fifth patterns, on the other hand, need to pass a Variable (respectively named FileRef and Loc) as parameter of the generated Builder function. This Variable is then used in the SetObject line for these pattern.

Since the GetValueExpression method of the base solution is also cast on a Variable, another possible choice would have been to pass the Variable in the list of parameters for the base pattern. Doing so would reduce the number of patterns, as all three have the same needs: a Variable and the GetValueExpression method in the SetObject line. All three would gather under the base pattern. The drawback of this change is an increase in the amount of information the developer needs to

provide to the Builder function for the base solution.

```
1  METHOD AcceptFrom_FunctionCallBuilder(Func: DotNetFunction.Function; Scope: AllocationScope;
2                                        ObjectArg: DotNetVariables.Variable
3                                        fd: DotNetExpression.Expression;
4                                        mem: DotNetExpression.Expression): DotNetStatement.Statement;
5      VAR
6        FuncCall: DotNetFunctionCall.FunctionCall;
7      BEGIN
8        FuncCall.CREATE("[RainCodeLegacyRuntime]RainCodeLegacyRuntime.IO.IORuntime::AcceptFrom");
9        FuncCall.SetObject(ObjectArg.GetValueExpression);
10       ASSERT fd.GetType.GetUnderlyingType = DotNetTypes.NativeStringType;
11       FuncCall.AppendArgument(fd);
12       ASSERT mem.GetType.GetUnderlyingType = DotNetTypes.MemoryAreaType;
13       FuncCall.AppendArgument(mem);
14       FuncCall.SetType(DotNetTypes.VoidType);
15       RESULT := FuncCall;
16     END AcceptFrom_FunctionCallBuilder;
```

Figure 5.1: Default FunctionCall if we combine the three patterns

In Figure 5.1, we illustrate the impact of such change. On line 9 the parameter of the SetObject method uses a Variable defined on line 2, called ObjectArg. This Variable can easily take the place of the FileRef and Loc patterns, also using variables with the GetValueExpression method. Since the basic solution would now use the same method than the other two, we can reduce the amount of GenerationModes. For the base pattern this implies another change: use the new ObjectArg parameter instead of the automatic conversion proposed in section 3.1.3.

Taking a final stance on this question requires to analyze nearly every FunctionCall. For the time being, the choice was made to consider them as different patterns. The situation could change as the removal of legacy code progresses.

5.2.2 *returnType* generation mode

The sixth pattern presented a solution to cover several similarly named C# functions. As stated in section 2.1.3, those functions often differ in one parameter only, therefore one legacy FunctionCall can handle the reference to multiple functions. In section 3.3.5, the solution offered, the *returnType* pattern, kept this mechanism. The assertion line of the varying argument has to be modified to check the abstract Type. We use the abstract type as the comparison occurs between Types from different modules of the compiler: the returnType is defined as RCompType from RComp, and the variable is defined as DotNetExpression from .NET. Having a less specific Type in the assertion and whether this is deemed a sufficient verification are the points of attention.

Another approach to this pattern could be to divide it into two distinct ones: the setter of the setType method parameter and the last argument variation. If we

call the first setType and the second Variation, we can replace more FunctionCalls due to the new versatility acquired.

The *returnType* pattern might actually be too precise: the last argument type and return type have to been exactly the same. Additionally, some library functions only require the setType part for their generation. As stated in section 3.3.5, the idea behind this pattern was to replicate the mechanism used in the legacy code. Whether this pattern should be kept as it or adapted needs further analysis of the YAFL FunctionCalls.

5.3 State of the solution

In Table 4.2, we can see the COBOL compiler has seen the most changes. The reason is that COBOL was the basis module for the patterns creation. As other modules were not the focus, less of their FunctionCalls structures can be generated with the patterns proposed.

The functions that could be generated have been tested with Raincode's test suites. The choice to push them on the working branch for developers to use in the current state is Raincode's decision.

5.4 Future work

As mentioned, FunctionCalls still reside in the code and require additional work in order to be generated. For some the refactoring needs to be refined. We discuss here the future options considered.

To continue the work, we can improve the range of solutions with the addition of patterns based on each module, especially for the second compiler, PL/I. If the main templates used in each module can be generated, the figures of reduction will improve for the corresponding line of Table 4.2.

After this step, we should go further in the code and design around more problematic entries in the YAFL code. The difficulty will come from either inter-procedural calls for the creation of a FunctionCall or either the generation not following a predictable template. For the first, the worker need to spent more time reasoning over the legacy code. For the second, changes to the generation or tricks

inside the annotation system has to be proposed.

We mentioned the complete confidence Raincode has in its test suite in section 4.1. However, the weakness of this reasoning comes down to the fact the tests were designed before the creation of the generation. To be completely thorough, structural testing adapted to the new builders should be implemented. We did not, in this thesis, create any work in this direction. It would require a more refined understanding of the COBOL and PL/I language, as well as the code behind the compiler tester that Raincode provides.

Bibliography

[1] Wikipedia. *COBOL*. 2007. URL: https://en.wikipedia.org/wiki/COBOL.

[2] Wikipedia. *PL/I*. 2008. URL: https:/en.wikipedia.org/wiki/PL/I.

[3] JetBrains. *dotPeek*. URL: https://www.jetbrains.com/decompiler/.

[4] Darius Blasband. "The YAFL Programming Language". In: *J. Object Oriented Program.* 8.7 (1995), pp. 42–49.

[5] Stefans Tools. *grepWin*. URL: https://tools.stefankueng.com/grepWin.html (visited on 03/05/2023).

[6] Microsoft. *Reflection and Attributes*. 2023. URL: https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/reflection-and-attributes/.

[7] Microsoft. *Reflection in .NET*. 2021. URL: https://learn.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/reflection.

[8] Microsoft. *Code Generation and T4 Text Templates*. 2023. URL: https://learn.microsoft.com/en-us/visualstudio/modeling/code-generation-and-t4-text-templates?view=vs-2022.

[9] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional, 1994. ISBN: 0201633612. URL: http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1.

[10] Martin Fowler et al. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, June 1999. ISBN: 0201485672. URL: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike04-20%7B%5C&%7Dpath=ASIN/0201485672.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**