A Purely Functional Combinator Language for Software Quality Assessment*

Pedro Martins¹, João P. Fernandes¹, and João Saraiva¹

1   **HASLab / INESC TEC**
    Universidade do Minho, Portugal
    `{prmartins, jpaulo, jas}@di.uminho.pt`

─── **Abstract** ───

Quality assessment of open source software is becoming an important and active research area. One of the reasons for this recent interest is the consequence of Internet popularity. Nowadays, programming also involves looking for the large set of open source libraries and tools that may be reused when developing our software applications. In order to reuse such open source software artifacts, programmers not only need the guarantee that the reused artifact is certified, but also that independently developed artifacts can be easily combined into a coherent piece of software.

In this paper we describe a domain specific language that allows programmers to describe in an abstract level how software artifacts can be combined into powerful software certification processes. This domain specific language is the building block of a web-based, open-source software certification portal. This paper introduces the embedding of such domain specific language as combinator library written in the Haskell programming language. The semantics of this language is expressed via attribute grammars that are embedded in Haskell, which provide a modular and incremental setting to define the combination of software artifacts.

1   Introduction

Software quality assessment is a relevant research topic, and the implications of quality assessment are even more intricate and interesting when we consider open source software (OSS). With this is mind, the *Certification and Re-engineering of Open Source Software* (CROSS) project¹ was presented with the global goal of assessing the quality of general-purpose OSS software.

While we observe a growing integration of OSS in various public and industrial organizations, the fact is that there are no substantial standards or analysis tools that can provide an assertive quantification of the overall quality of such products. This means that their use still incorporates several risks.

In the context of CROSS and of the work presented in this paper, our general intention is to be able of certifying OSS. We understand a Certification as the process of analyzing a software solution while producing an information report about it. Certifications are expected

---

¹   `http://twiki.di.uminho.pt/twiki/bin/view/Research/CROSS/` [Accessed in 25 March, 2012]

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz; pp. 51–69

to process an OSS solution and provide a technical analysis of it, decreasing the exposure to risk associated to the adoption of OSS.

To be more concrete, the challenges undertaken within CROSS are four-fold: i) to select and address several OSS-specific certification problems; ii) to develop techniques for the analysis of both code and its documentation; iii) to develop a certification infra-structure for OSS projects that is open to contributions and freely available; and iv) to embark in several collaborations with leading IT companies so that the overall results of the project are available for them.

The work described in this paper contributes to goals i) and ii) above. Indeed, we introduce a combinator language that allows users to easily construct tailor made certifications that can actually be the result of gluing together simpler certifications. The language that we propose is being used as a central piece of a more elaborated goal in the lines of iii): we want to develop an infra-structure, a Web Portal, that works both as a repository of software analysis tools and as a service that allows the analysis and certification of OSS. Such service has to maintain the open source spirit of heterogeneous and distributed collaboration: the portal has to store all the tools produced and, more important, allow any user to create new certifications by arranging the tools inputs/outputs in the appropriate order. Also, users should be able of combining already existing certifications into more complex analyses.

The language that we introduce in this paper aims at allowing an easy configuration of the flow of information among processes/tools that run either in parallel or in chain to create certifications, and at the automatic analysis and generation of low-level scripts that implement such configuration.

This paper is organized as follows: In section 2 we provide an overview of the motivation and potential challenges this work faces. In section 3 we introduce our combinator language together with small examples of its usage. In section 4 we present an Attribute Grammar-based type checker mechanism to control the flow of information inter-processes, which also is responsible for the script generation, as it is explained in section 5. In section 6 we provide an overview of related works, in section 7 we discuss opportunities for improving and extending our work and in section 8 we conclude the paper.

2   Motivation

We have already mentioned that the work presented in this paper is integrated within CROSS, a project whose aim is to develop program understanding techniques capable of measuring the degree of quality of open source software while being able to cope with its collaborative, distributed and heterogeneous character.

The techniques for analyzing source code to produce in the context of CROSS should, either individually or combined with others, result in the production of reports called Certifications. In the context of our work in this paper, we have developed an XML-based representation for these reports. Also, certifications are often composed by smaller units that are capable of communicating with each other in order to achieve a state where the overall mechanics of each unit and the flow of information among them is capable of producing quantifiable results. In the remaining of this paper, we will address ourselves to this smaller units that contribute to a general goal as Components.

In detail, a Component is therefore a bash tool, that is capable of accessing and producing meta-data via the standard channels (the standard input, STDIN, and the standard output, STDOUT). Also, a component must be able of receiving arguments that define the type of the information that is received via STDIN and the type of the information that is to

be channelled through STDOUT. Also, components are often developed independently by different, heterogeneous and distributed teams, and their development and their integration in more complex certifications closely follows the philosophy of open source software development itself.

In Figure 1 we sketch the flow of information that has been implemented in order to produce a sample Certification called Certification 2. This is a certification that expects Java programs and that analyzes them according to two distinct sub-processes that are independent with respect to each other and therefore can be executed in parallel. One of these processes chains a series of software units, namely Java Slicer, Interface Analysis and csv2Report. The other, which is itself a certification called Certification 1, implements a Cyclomatic Dependency analysis while producing an information report. Finally, and since all certifications must produce reports that conform to our format, the two distinct flows of information are aggregated by a Reports aggregator, a component whose single responsibility is precisely to aggregate outputs.
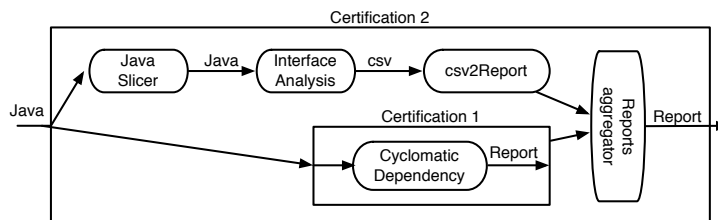
■ **Figure 1** The flow of information implemented in Certification 2.

Several aspects of constructing certifications as illustrated in Figure 1 deserve further notice. For once, certification developers do not need to worry about concrete details of the sub-components or sub-certifications to use. Indeed, these sub-units must already exist in the certification framework, and the developer only needs to make sure that information flows, both semantically and syntactically, along the global certification process. This means that the output type of a component/certification must match the input type of the component/certification that immediately follows it. In these lines, for example, if a Slicer extracts the interface of a program, it can not feed a component that analyses concurrency requirements or super classes, even if we consider the same language being analyzed. Another thing to notice is that validations of this kind are automatically ensured by our system, as explained in section 4, that performs syntactic type checking.

In contrast to the approach that we propose in this paper, a traditional approach to implement a certification such as the one in Figure 1 would imply the cumbersome task of manually writing a script implementing the same features. Typically, such a script must be capable of launching processes in sequence and pipe their results, launching processes in parallel, timeout the processes and warn users if any of them is taking too long to run while minimizing this impact in processes that ran without problems, for example.

For demonstration purposes, we present in Appendix A an example of a script implementing Certification 2 in Perl. In fact, even being this a long and error-prone solution, it actually shows a simplified version of the script only. Furthermore, the implementation effort would significantly increase with more and more complex certifications (i.e., with certifications with larger numbers of sub-components). Furthermore, in this approach it is also very hard to systematically analyze the flow of information and to perform tests such as checking the existence of circular dependencies or ensuring that the types of inputs and outputs among components/certifications match.

In resume, we believe that having a large set of manually-written scripts is not a good practice: one can never be sure how well the script was written and how well it handles and processes errors. It is also very important to notice that, if the target system changes, this approach would require manually re-writing every script to support the new requirements and specifications.

In this paper we present a purely functional combinator language which allows an easy and intuitive combination of components/certifications and an easy creation of new certifications together with a script that implements them, and also with automatic inter-processes type checking and automatic code generation.

3   A Combinator Language for Certifications

The combinator language that we propose is written in Haskell, and starts by defining the data-types for certifications and components. These data-types are introduced in Listing 1 as Certification and Component, respectively.

◼ **Listing 1** Data types for Certification and Component.

```
data Certification = Certification Name ProcessingTree
data Component = Component Name InputList OutputList BashCall

data Language = Java       -- .java
              | C_Source -- .c
              | C_Header -- .h
              | Cpp       -- .cpp
              | Haskell   -- .hs
              | XML       -- .xml
              | Report    -- Report XML

type Arg       = String
type Name      = String
type BashCall = String
type InputList   = [(Arg, Language)]
type OutputList  = [(Arg, Language)]
```

A Certification has a name (e.g. Certification 1 as in Figure 1) and defines the particular information flow to achieve a desired global analyzis. This flow is represented by data-type ProcessingTree, that we introduced in Listing 3 and that we describe in detail later.

A Component is represented by a name, the list of arguments it receives and the list of results it produces. These lists, that are represented by type synonyms InputList and OutputList, respectively, have similar definitions and consist of varying numbers of arguments and results. The arguments(results) that are defined(expected) for a particular component are then passed to concrete bash calls. This is the purpose of type BashCall, which consists of the name of the process to execute on the system.

In the context of Figure 1, for example, the component Java Slicer may be executed by the bash call JSlicer with arguments -j and -i in order to slice the interface out of a piece of Java code. In a different scenario, executing the same process with arguments -j and -s would result in the slicing of the superclasses of the Java code (actually, if this execution occurred in the particular example of Figure 1, it would break the flow of information due to input/output mismatches).

For a generic Java Slicer component that could also be used in the context of Figure 1, we may define the instance of the Component data-type presented in Listing 2.

■ **Listing 2** A sample instance of the Component data-type.

```
Component "Java Slicer" [("-j", Java)]
          [("-i", Java), ("-s", Java)] "./jSlicer"
```

■ **Listing 3** Data-type for ProcessingTree.

```
data ProcessingTree = RootTree ProcessingTree
                    | SequenceNode ProcessingTree ProcessingTree
                    | ParallelNode ProcessingList ProcessingTree
                    | ProcessCert  Certification
                    | ProcessComp  Component Arg Arg
                    | Input

data ProcessingList = ProcessingList ProcessingTree ProcessingList
                    | ProcessingListNode ProcessingTree
```

In order to represent the flow of information defined for a certification, we have defined the data-type ProcessingTree, which is introduced in Listing 3.

The simplest processing tree that we can construct is the one to define a certification with a single component. This is expressed by constructor ProcessComp, which also expects a name to be associated to the component and the specification of the options to run the component with.

A certification can also be defined by a single sub-certification, here represented by ProcessCert.

In addition to these options, more complex certifications can be constructed by running processes in sequence, using SequenceNode, and in parallel, using ParallelNode. Constructor ParallelNode takes as arguments a processing list and a processing tree. The first argument represents a list of trees whose processes can run in parallel. The second argument is used to fulfill our requirement that all results of all parallel computations must be aggregated using a component. Therefore, this processing tree must always be a component (and this is ensured by our type checking mechanism in a way that we describe in detail in section 4) that is capable of aggregating information into one uniform, combined output.

The ProcessingTree data-type can be used, for example, to implement the global information flow of Certification 1 presented in Figure 1, which results in the instance presented in Listing 4.

Having in hand the data-types that we have defined so far, we could already create, in a manual way, certifications with all the capabilities that we propose to offer. As an example of this, Certification 2 of the previous section could be defined as presented in Listing 5.

■ **Listing 4** A sample instance of the ProcessingTree data-type.

```
RootTree
   ProcessComp
       Component "Cyclomatic Dependency"
                 [("-j",Java)]
                 [("-r",Report)]
                 "./exec"
       "-j"
       "-r"
```

Nevertheless, expressing certifications in this manual approach would be of impractical use. This is precisely the main motivation to develop a language where simple components can be combined into more complex ones, which themselves can grow as large as needed in order to implement extensive certifications. So, instead of the certification implementation shown above, we suggest the equivalent representation that is introduced in Listing 6.

◼ Listing 6 An example of a Certification using our Combinator Language.

```
javaSlicer =
    Component
        "Java Slicer"
        [("-j",Java)]
        [("-i", Java), ("-s", Java)]
        "./jSlicer"
interfaceAnalysis =
    Component
        "Interface Analysis"
        [("-j",Java)]
        [("-csv",CSV)]
        "./iAnalysis"
csv2Report =
    Component
        "csv2Report"
        [("-csv",CSV)]
        [("-r",Report)]
        "./csv2Report"
cyclomaticDependency =
    Component
        "Cyclomatic Dependency"
        [("-j",Java)]
        [("-r",Report)]
        "./cDepend"
reportsAggregator =
    Component
        "aggregator"
        [("-r",Report)]
        [("-r",Report)]
        "./csv2Report"

certification2 =
    Input >- (javaSlicer,"-j","-i")
          >- (interfaceAnalysis,"-j","-csv")
          >- (csv2Report,"-csv","-r")                       >|
    Input >- Input >- (cyclomaticDependency,"-j","-r")
                  +> "Certification 1"                      >|>
             (reportsAggregator,"-r","-r")
          +> "Certification 2"
```

With the addition of this code being smaller, elegant and easy to read and understand, the fact is that it will also be statically analyzed and type checked, and the script code that actually implements the certification it defines will be automatically generated. These features will be introduced in the remaining of this paper, together with the combinators that we use in our language, that we present in detail next.

■ **Listing 5** A sample instance of the Certification data-type.

```
Certification
   "Certification 2"
   ParallelNode
     ProcessingList
          SequenceNode
              SequenceNode
                  ProcessComp
                     Component
                         "Java Slicer"
                         [("-j",Java)]
                         [("-i", Java), ("-s", Java)]
                         "./jSlicer"
                     "-j"
                     "-i"
                  ProcessComp
                     Component
                         "Interface Analysis"
                         [("-j",Java)]
                         [("-csv",CSV)]
                         "./iAnalysis"
                     "-j"
                     "-csv"
              ProcessComp
                 Component
                     "scv2Report"
                     [("-csv",CSV)]
                     [("-r",Report)]
                     "./csv2Report"
                 "-csv"
                 "-r"
          ProcessingListNode
              ProcessCert
                 Certification
                     "Certification 1"
                     ProcessComp
                         Component
                             "Cyclomatic Dependency"
                             [("-j",Java)]
                             [("-r",Report)]
                             "./cDepend"
                         "-j"
                         "-r"
     ProcessComp
         Component
             "aggregator"
             [("-r",Report)]
             [("-r",Report)]
             "./csv2Report"
         "-r"
         "-r"
```

The Sequence Processing Combinator

For sequencing operations, we define the combinator `>-`. This combinator defines processes that are to be executed in a chain, i.e, where the output of a process serves as input to the process that follows it. When sequencing processes, it is also the case that if one of process in the chain fails the entire chain will also fail.

The use of combinator `>-` must always be preceded by the use of constructor Input, which signals the beginning of an information flow. Then, as many components and certifications as needed can be used, as long as they again connected by `>-`. In Listing 7 we show an example of a chain of events defined using `>-`.

■ **Listing 7** An example of Sequence of Processes using the Sequence Combinator.

```
Input >- (jSlicer ,"-j","-i") >- (iAnalysis ,"-i","-csv") >- certif
```

Combinator `>-` can be used to sequence certifications, components and other processing trees that are defined using the remaining combinators of our language. When it encounters a certification, the combinator connects the processes before it to the processing tree of the certification, and ensures that the result of this processing tree is then channeled to the processes that follow it. This is the case of the sub-certification called `certif` in Listing 7. When sequencing components, users need to supply to `>-` both the component and its input/output parameters. In the particular example of Listing 7, `jSlicer` is to be called with input parameter `-j` to state that the process accepts Java code as input and with output argument `-i` to state that it slices the interfaces out of that code.

It is worthwhile to notice that the arguments that are specified within components are very important in that they allow checking the flow of information inter-processes for correctness, as the input and output types must match when the information is channeled. When using `>-` to channel certifications, the user is constrained by the input and output types that were associated to it, and this is an information that must be carefully observed to ensure that the involved types do match.

Finally, the result of a sequence defined using `>-` is a processing tree that implements the combination of processes.

The Parallel Processing Combinator

Now, we introduce the combinator that enables the parallel composition of processes, This type of composition is actually supported by two combinators, `>|` and `>|>`. The first one is responsible for launching a varying number of processes in parallel, while the second is mandatory after a sequence of `>|` uses and chains all outputs of all processes to a component that is capable of aggregating them. In Listing 8 we show an example of how these combinators work together.

■ **Listing 8** An example of Parallelization of Processes using the Parallel Combinators.

```
Input >- cert3                           >|
Input >- cert1 >- cert5                  >|
Input >- (jSlicer ,"-j","-x") >- cert8 >|> (aggr ,"-x","-r")
```

Combinator `>|` takes either a processing tree, a component, a certification or a set of processes constructed using the other combinators. The arguments of `>|` must always begin by constructor Input, to give a clear idea of the flow of information. In the case of this listing it is indeed easy to spot where the information enters a parallel distribution.

As for combinator `>|>`, it is mandatory for it to appear in the end of a parallelized set of processes. It is used to aggregate all the outputs of all the child processes into a single standard output, and it is able of combining varying numbers of parallel processes using an aggregation component.

It is worthwhile to explain further the relationship between the parallel combinators `>|` and `>|>`. In the trivial cases, `>|>` can actually replace the use of `>|`. This is the case of the process `Input >|> (aggr,"-x","-r")`, which is equivalent to `Input >- (aggr,"-x","-r")`, as both processes channel the input to the aggregation component `aggr`. Finally, combinator `>|` can never appear alone in a certification, due to the constraint that all parallel processes must be aggregated.

A Combinator to Create Certifications

Our combinator language includes also a combinator to create certifications and to associate names to them. This is precisely the purpose of combinator `+>`, which always combines a processing tree, given as its left argument, with a String, given to its right. It then creates a certification associating the name with the processing tree.

As an illustration of the use of `+>`, consider again the process implementations of Listings 7 and 8. In both cases, the result of running the implemented code is a processing tree (i.e., and element of type ProcessingTree), that needs to be given a name to become a certification (i.e., an element of type Certification). By simply appending, for example, `+> "certification"` to the end of both codes, we would precisely be creating certifications named certification with the respective trees of processes. In the case of Listing 7, this would result in the code show in Listing 9.

■ **Listing 9** Using a Combinator to construct a Certification.

```
Input >- (jSlicer ,"-j","-i")
      >- (iAnalysis ,"-i","-csv")
      >- certif
      +> "certification"
```

An important remark about `+>` is that it analyzes the processing tree that it receives as argument and checks its correctness. This includes testing whether the implied types match, which means analyzing all the parallelized and sequenced processes for their input and output types, and see whether or not they respect the flow of information. Also, it is ensured that the processing tree produces a report which is a mandatory feature for a certification in our system.

Later in this paper, in section 4, we explain in more detail the features that are implemented by our type analysis and how they are actually implemented under our framework.

The 'Finalize' Combinator

The last combinator of our language is `#>>`, that combines a processing tree with a flag instructing it to either produce a script implementing that tree or to simply check its types for correctness. The examples presented in Listing 10 show the two possible uses for this combinator.

■ **Listing 10** Using the finalize combinator.

```
Input >- (comp1 ,"-j","-x") >- (comp2 ,"-k","-o") #>> 't'
Input >- cert1 >- cert2 #>> 's'
```

In the first case, we are demanding a check on the types of running component `comp1` after component `comp2`. This means that we are interested in knowing whether the return type of `comp1` is the same as the input type of `comp2`.

With the second case, we are asking for the script that implements chaining certification `cert2` after certification `cert1` and also checks if the types match. In section 5 we explain in more detail how the script generation is achieved.

An Example Scenario

The examples that we presented so far were used to illustrate in simple ways the use of our combinator language. Still, we believe to have already demonstrated the simplicity that is involved in its use to create more and more complex certifications. In this section, we explore this argument further by introducing and describing the certification process of Listing 11.

Listing 11 Example of a parallel process in the middle of a sequence of processes.

```
Input >- (comp1,"-s","-ast") >- parallel >-
                                    (comp2,"-h","-r") >- cert

parallel  = Input >- (comp3,"-ast","-x") >|
            Input >- (comp4,"-ast","-x") >|
            Input >- (comp5,"-ast","-x") >|> (compAggr,"-x","-h")
```

In this example we have introduced a parallel computation in the middle of a sequence of processes. An example of where such a scenario could be useful is in the case of having a set of processes to analyze an Abstract Syntax Tree (AST), but having an input as source code. This code then needs to be converted to an AST using, in our illustration, `comp1`.

Following a manual approach to implementing a script for this scenario would lead to a complex development process. Indeed, one would have to manually edit it to make sure the process corresponding to `comp1` feeds each process on the parallel computation (that in this case is composed by 3 sub-processes but that could easily grow further).

Furthermore, imagine that we do not want the results of the parallel computation, but rather we want them to be compared against a repository of results to analyze the characteristics of our AST. For this, a certification `cert` has been implemented, but it does not take as input the same format that is returned by our parallel processes. A possible solution using our combinator language is to channel the result of the parallel processes to an auxiliary component `comp2` that converts the formats so the information can be fed to the certification. But this is something that is not easily implemented by hand.

The overall proposal of performing everything manually would be considerably difficult and error-prone. One would have to mess with legacy scripts, potentially built by different programmers, to understand them, and to create the correct chain of information. And further difficulties still need to be resolved if one wants to ensure script robustness, and that the processes are controlled in terms of processing times and failures, for example. And this significative effort of manually building scripts is furthermore severely compromised considering script evolution. Indeed, small changes in particular certification sub-processes may lead to severe overall changes being required.

We believe that our combinator approach has the advantage of not only making it easier to create flows of information among process, that can be easily edited, but also of being highly modular. Indeed, our combinators receive as arguments small fragments that can be edited, managed and transformed in simple ways. Also, it does not require a significant effort

for a programmer to change a particular certification, making it able of producing different results or improving it by introducing new safe processing mechanisms.

4   Type Checking on Combinators

In the previous section, we have introduced our combinator language for the development of software certifications. In this section, we introduce a set of validations that are automatically guaranteed to the users of our system.

For once, and since we have chosen to use Haskell in our implementation, we inherit the advanced features of both the language and its compilers. In particular, the powerful type system of ghc helps us providing some static guarantees on the certifications that are developed. Indeed, the order in which the combinators of our language are applied within a certification is not arbitrary, and the uses that do not respect it will automatically be flagged by the compiler. The simplest example of this situation is the attempt to construct a processing tree without explicitly using constructor Input, but of course more realistic examples are also detected, e.g., not wrapping up a set of parallel computations with the use of `>|>` as well as the application of an aggregator.

Apart from static analyzes that are ensured by the compiler, we have also implemented some dynamic ones. Indeed, we also want to analyze if the types match in the flow of information defined for a certification. This means that the input type of a process must match the output type of the process feeding it. Taking the example on Figure 1, the output type of Interface Analysis must be the same as the input type of csv2Report, which in this case is csv.

In our setting, we perform such tests on elements of type ProcessingTree, that we use our combinators to construct. These elements are then analyzed using validations that are expressed as attribute grammars (AGs) [8]. The reasons for choosing this approach are essentially of two different natures: i) for once, we are analyzing tree-based structures, for which the AG formalism is particularly suitable; ii) secondly, because AGs have a declarative nature which in our context contributes to intuitive implementations that are easy to reason about and to further extend. In fact, we believe that implementing in our framework advanced AG-based and well studied techniques such as the detection of circular dependencies [5] and the use of higher-order attributes [17].

The analyzes that we have implemented in an AG-style rely heavily in the concept of functional zippers [6]. Indeed, every element that we may want to analyze (i.e., upon which we define analyzing attributes) first needs to be wrapped up inside a Zipper. A more detailed description of Zippers and of how we use them can be found in section 6.

As we have already mentioned, our type checking is performed on trees of the type of ProcessingTree and, because we used an AG-based approach, this analysis is broken down into tree nodes which, in our case, are represented by the Haskell constructors of the ProcessingTree data type. The type checking is computed as the value of an attribute called typeCheck. This attribute, of type Boolean, indicates whether or not the analyzed types are correct. Apart from this attribute two other are involved: input and output, that support typeCheck. These attributes are of type Language (see Listing 1) and for each tree node give the input and the output types of that subtree.

Next, we will explain how these three attribute are calculated in each tree node.

Type Checking Component nodes

Components are the simplest units of our processing trees, and represent simple processes without any actual flow of information. This means that their type is always correct, and that the value produced for attribute typeCheck is always True.

As for the attributes input and output, they are computed analyzing the component and its associated element of the Component data type, against the invocation that is made for it in a ProcessingTree. Indeed, whenever a component is associated to a certification, an element such as ProcessComp comp inp out is defined. But comp itself is an element of type Component, i.e., has the form Component name inplist outlist call. So, our validation starts by detecting whether or not inp (respectively out) is an option of inplist (respectively outlist). In case it is, attribute input (respectively output) returns the Language option associated with inp (out). Otherwise an error is raised.

Type Checking Certification nodes

Certifications are, similarly to components, simple nodes of a processing tree.

In our setting, an interesting feature about certifications is that they must always be created using combinator +>. Therefore, everytime this combinator is employed, as in `tree +> name`, we automatically inspect the value of attribute typeCheck that is computed for tree. If this value is True, we create the certification Certification name tree; otherwise, no certification is constructed and an error is raised.

Actually, it is not only necessary that a processing tree type checks in order for us to be able of producing a certification out of it. Indeed, a requirement of our system is that certifications must always produce a report within our format. Therefore, we also check if the output attribute computed for tree is Report prior to the creation of an actual certification.

Finally, when we consider sub-certifications within a certification, again we use the fact that they need to be created using +>. Indeed, if the construction of a sub-certification succeeds, then it is always true that the tests so far described have also succeeded. Therefore, for such certifications, i.e., for certifications under ProcessCert nodes, we can always ensure that it type checks. Also, because we use an AG based analysis the attributes input and output are very simple to implement: we return the value of the same attribute that is synthesized at the sub-tree of ProcessCert nodes.

Type Checking Sequence nodes

The sequence node construction is useful whenever we have a processing tree that is followed by another. This node channels the information from the first processing tree into the second one and returns the result of this second processing tree.

The input and output attributes for this node are very simple to compute. In fact, input is the input type of the first processing tree, and output is the output type of the second processing tree.

Similarly, the typeCheck attribute is also very simple to determine: apart from checking if the output attribute of the first tree is equal to the input attribute of the second one, our AG-based implementation also demands the typeCheck attribute on each sub tree individually and checks whether both have value True.

Type Checking Parallel nodes

Parallel nodes are the hardest to type check. The reason for this is that, in a parallel processing definition, all its sub processes must have the same input type and the same output type, and this output type must the same as the input type of the component that aggregates all the results that are computed in parallel.

Parallel nodes always have two children: the second child, of type ProcessingTree, is a component that aggregates all the results of the processes that run in parallel, which are given as the first child of the node, of type ProcessingList.

The first validation we perform is to check whether the output value of the ProcessingList matches the input value of the ProcessingTree. If it does not, an error is raised; otherwise, the processes within the ProcessingList need to be type checked.

Now, type checking processing lists is more complex in that it needs to analyze all the inputs of all the sub processes, which can be components, certifications or processing trees and see if they match, and do the exact same thing for the outputs. In an AG setting this implementation can be achieved as follows: we use the equality of the input and output attributes for the current element of the list and for the subsequent elements. By type definition the processing list can never be empty and must always contain at least one element so the attribute will always return a value.

The input and output attributes for parallel nodes are simple to compute: the input is the input of one of the elements of the processing list, as they are all the same, and the output is the output of the aggregator component. One important note is that, due to the importance of all the types within a processing list being correct, we have followed a safe approach: the input and output attributes for a processing list are always given only after the type checking for the entire list is performed.

5 Script Generation

We have shown a how set of processes can elegantly be combined into a certification, either in a sequence, in parallel or in any combinations of these two. We have also shown how this combinators are easy to read, understand and modify, and how we implemented a supporting type checking mechanism that guarantees a correct match of types throughout the processing chain.

In this section we describe how we can generate Perl scripts that implement the certifications that are created using our combinators. These scripts can be seen as the low level representation of our certifications: they describe the processing chain, handle the individual processes for their completeness and manage the flow of information throughout all the processes the certification is made of.

The script generation follows the same AG-based strategy that we have applied to type check our certifications. The basic idea behind it is that each tree node, that represents a part of the processing tree, generates the corresponding sub-piece of the global script, and that the overall meaning of the AG is the entire, fully working, script.

In Listing 12 we present an example of a script that was generated by the following combination: `Input >- (comp1 ,"-j","-x") >- (comp2 ,"-k","-o")`.

In this script both components are executed via the system call command `capture_exec`, their existence is verified and their STDERR output is checked for problems. Afterwards, their results are channelled to the process that comes next, which in the case of the first component is the second component, and in the case of the second component is the STDOUT of the script since the computations ended.

◼ **Listing 12** Example of a script.

```perl
#!/usr/bin/perl
use strict;
use warnings;
use IO::CaptureOutput qw/capture_exec/;
use IO::File;
$| = 1;
my $stdout0 = $ARGV[0]; # This is actually stdin

my $cmd1 = "./comp1 -j -x";
my ($stdout1, $stderr1, $success1, $exit_code1)
        = capture_exec( $cmd1 . "<<END\n" . $stdout0 . "\nEND" );
if ($?) { die "** The process $cmd1 does not exist! **"; }
if (not $success1)
        { die "** The process $cmd1 failed with msg: $stderr1 ! **";}

my $cmd2 = "./comp2 -k -o";
my ($stdout2, $stderr2, $success2, $exit_code2)
        = capture_exec( $cmd2 . "<<END\n" . $stdout1 . "\nEND" );
if ($?) { die "** The process $cmd2 does not exist! **"; }
if (not $success2)
        { die "** The process $cmd2 failed with msg: $stderr2 ! **";}

print $stdout2;
```

The scripts that we generate perform process control and scheduling of computations both on chained and in parallel flows of information while still being readable and understandable. Nevertheless, constructing such scripts manually is still an error-prone task even for small certifications with small number of processes. A larger certification (with, e.g, over a dozen sub-processes) would imply a significant amount of time to be implemented and debugged, just to name some phases of the development process.

In fact, this situation would further deteriorate if we were considering integrating in our framework more advanced scripting features. We could, for example, be interested in time-outing the processes independently to ensure they do not go past a certain time frame, in controlling better the input and output information from processes (checking, for example, if input information is able to be processed though STDIN, i.e, respects the specific implementations on different programming languages and environments[2], etc) or in ensuring that, anytime an error occurs, the script actually creates a small report that is integrated in the final certification instead of just showing information through the standard STDERR.

We believe, however, that following an AG-based approach similar to our own would facilitate the integration of these features in structured and simple ways as one-of tasks that once achieved become automatically available for any certification, old and new. Furthermore, because tree nodes are modular units of an AG implementation, it is even easier to upgrade small parts of the script as needed. For example, implementing timeout features on parallel processes would imply changing only the corresponding attribute on the desired tree nodes.

Generating scripts automatically presents several difficulties that are orthogonal to any generation mechanism, including to our own AG-based setting. Code translation is challenged

---

[2] `http://en.wikipedia.org/wiki/Here_document` [Accessed in 25 March, 2012]

by the usual concerns of assuring that the result is both syntactically and semantically perfect, and that all constructors/primitives/declarations of the target language are correctly declared and used. Implementing multiple processes, for example, implies a tight control on the variables that carry their results and their inputs. In a chain of processes from A to B, the variable that stores the information produced by A must be the one that feeds B, and all the variables must have different names (and if they do not, they must be used in different execution contexts). Also, this mechanism is even harder to implement within parallel processing, where all the outputs are aggregated into one single process (remember Listing 3, where a parallel tree node has always a processing list and a component that aggregates information).

To implement the generation of a script we have followed a simple rule for the variables scope: their name follows the order in which their corresponding process appears in the tree. So, for example, if the script implements two processes, where the process A receives the input, processes it and sends it to process B, whose output is the certification report, all the variables related to A (first process) end in 1, and all the variables related to B (second process) end in 2. The name of the variables is always the same (except for the last character that is the number) and in this way we guarantee that variables remain exclusive to the process they are related to.

If the certification is composed by a sequence from a processing tree to another processing tree, than the names of the variables on the second processing tree start with 1 plus the number of variables on the first processing tree. Such mechanism is very easy to implement in our AG setting: we have created a very simple attribute whose meaning is the number of sub-processes per tree node. So if the first processing tree of a sequence has four sub-processes, then the variables on that processing tree are named from 1 to 4, and on the second processing tree the variable names start in 5.

For parallel computations this mechanism is very simple, except for a few requisites. First, we must ensure that whatever comes before the parallel computation feeds all its sub-processes. To do so, we ensure that any information is first assigned to a variable from which all the sub processes read their input. Also, the results from all sub-processes are channelled to a variable that aggregates them. It is important to notice that the outputs are not combined: they are simply channelled to a variable that feeds the aggregator component of the parallel computation, and it is the responsibility of such aggregator to read various inputs via STDIN, instead of reading a single instance that is the aggregation of all the results. We do so to preserve the information instead of risking changing it by combining functions.

On the sub-processes of a parallel computation the exclusivity of the names of the variables names is not important, since these are different processes with different execution environments. Nevertheless, it is important to preserve exclusivity inside the processes themselves, which we easily do simply by recursively calling the attributes that were responsible for creating the script in the first place.

After defining the scope rules for variables, the script generation via the attribute toScript is easy to perform, once again thanks to our AG-based mechanism. The code generation follows the syntactic rules of the target language (in this case Perl) and ensures that the constructors/primitives/parenthesis are written and in the correct form.

The attribute toScript on the root of the processing tree creates the headings necessary to the script (such as declaring global variables or importing Perl libraries) and in each tree node we generate the corresponding Perl code, which implies defining system calls by composing processes.

6   Related Work

In [2] an implementation of the orchestration language Orc [7] is introduced as an embedded domain specific language in Haskell. In this work, Orc was realized as a combinator library using the lightweight threads and the communication and synchronization primitives of the Concurrent Haskell library [11]. Despite the similarities on the use of combinators written in Haskell, this paper differentiates from ours because we do not rely on any existing orchestration language. Rather, we generate low level Perl scripts from combinators whose inputs are direct references to system processes (components). Also, our processes management does not rely on Concurrent Haskell, but rather on the parallelization features of the target system via system calls on the script.

The use of attribute grammars as the natural setting to express the embeding of DSLs in Haskell is proposed in [12, 13, 14, 15]. These embeddings use powerful circular, lazy functional programs to execute DSLs. Such circular, lazy evaluators are a simple target implementation of AGs used by several systems [9, 16]. To make such implementations more efficient we have adapted well-know AG [5] and program calculation [3, 4] techniques to refactor circular programs into strict ones.

Zippers were originally conceived by [6] to represent a tree together with a subtree that is the focus of attention, where that focus may move left, right, up or down within the tree. By providing access to the parent and child elements of a structure, zippers are very convenient in our setting: attributes are often defined by accessing other attributes in parent/children nodes. In our work we have used the zipper library of [1]. This library is generic, in that it works for both homogeneous and heterogeneous datatypes, as any data-type for which an instance of the Data [10] type class is available can be immediately traversed using this library.

7   Future Work

Further improvements in the presented library would be specially important and have greater impact in the combinators analysis mechanisms.

A possible improvement is on the type checking mechanism. This mechanism is already on a solid state and perfectly checks for all kinds of types mismatches along the flow of information, but their output is still a simple error to the user, warning him/her about the fact that, somewhere along the chain of processes, something somewhere has a type error.

With huge certifications it becomes very difficult to localize a type error. One improvement we are looking into is on the warnings produced by the typeCheck attribute. Such warning could display valuable information such as showing until which part of the process the types are ok or actually indicating the specific line along the combinators where the type validity breaks.

Another important improvement would be in the number of AG-based analysis we perform. In this state, the library only checks for types errors and generates a script, but the world of AGs is old and heavily studied, and there are a huge amount of works with algorithms that when implemented would, almost for free, greatly improve our library. A perfect example of a well-known AG-based algorithm we are looking forward to implement are circularity checks along the processing tree, where an interdependency between the process A and the process B could actually lead to a deadlock on the final script.

The script itself could also be the subject of some improvements, such as performing individual timeouts on processes, do a better control on the inputs and outputs and improve the warnings to the user.

8   Conclusion

In this paper we have presented a combinator library that supports the scheduling of processes, both chained or as parallel computations. Together with the combinators, we have presented multiple examples of how computations can be elegantly rearranged into new process work flows, and how such combinators relate with each other to easily create complex certifications.

We have also introduced AG-based, purely functional mechanisms, that are not only capable of performing automatic type checking on processes work flows but also of automatically generating scripts.

Implemented in an AG environment, our type checking system automatically guarantees that the input and output types of each sub-processes are right to the definition of a process work flow and of a certification, and do not break such definitions, while also managing to automatically create low-level implementations of certifications in the form of Perl scripts.

We believe the advantages of our system are two fold: first, the combinators create an intuitive and simple yet powerful environment to create not only certifications but also processes work flows in general, while ensuring their validation.

Secondly, our AG approach can be easily transformed with any change that both the definitions of certifications or of processes work flows needs to support. This mechanism is modular, easily extensible and upgrades on code generation or type checking in the form of new features and functionalities are easy to design and implement in a modular and concise way.

The combinator library, together with built-in definitions of certifications and components, an example of a script, one of a component and a README, are available at `http://wiki.di.uminho.pt/twiki/bin/view/Personal/PedroMartins/CombinatorLibrary`.

A   Perl script for Certification Management

```perl
#!/usr/bin/perl
use strict;
use warnings;
use IO::CaptureOutput qw/capture_exec/;
use IO::File;
$| = 1;
my $stdout0 = $ARGV[0]; # This is actually stdin

my $cmd1 = "./script+1.pl -c -r";
my ($stdout1, $stderr1, $success1, $exit_code1) =
capture_exec( $cmd1 . "<<END\n" . $stdout0 . "\nEND" );
if ($?) { die "**** The process $cmd1 does not exist! ****"; }
if (not $success1) { die "**** The process $cmd1 failed with msg: $stderr1 ! ****"; }

#Before starting a parallel computationg we default the stdout to 0
$stdout0 = $stdout1;

my $handle2 = new IO::File();
my $pid2 = open($handle2, "-|");
if ($pid2 == 0) {
my $cmd1 = "./script+1.pl -r -x";
my ($stdout1, $stderr1, $success1, $exit_code1) =
capture_exec( $cmd1 . "<<END\n" . $stdout0 . "\nEND" );
if ($?) { die "**** The process $cmd1 does not exist! ****"; }
```

```
if (not $success1) { die "**** The process $cmd1 failed with msg: $stderr1 ! ****"; }
print $stdout1;
exit;
}

my $handle3 = new IO::File();
my $pid3 = open($handle3, "-|");
if ($pid3 == 0) {
my $cmd1 = "./script+1.pl -r -x";
my ($stdout1, $stderr1, $success1, $exit_code1) =
capture_exec( $cmd1 . "<<END\n" . $stdout0 . "\nEND" );
if ($?) { die "**** The process $cmd1 does not exist! ****"; }
if (not $success1) { die "**** The process $cmd1 failed with msg: $stderr1 ! ****"; }
print $stdout1;
exit;
}

my $handle4 = new IO::File();
my $pid4 = open($handle4, "-|");
if ($pid4 == 0) {
my $cmd1 = "./script+1.pl -r -x";
my ($stdout1, $stderr1, $success1, $exit_code1) =
capture_exec( $cmd1 . "<<END\n" . $stdout0 . "\nEND" );
if ($?) { die "**** The process $cmd1 does not exist! ****"; }
if (not $success1) { die "**** The process $cmd1 failed with msg: $stderr1 ! ****"; }
print $stdout1;
exit;
}


# Lets grab all the results of the parallel processes
my $stdout4 = <$handle2>. " " . <$handle3>. " " . <$handle4>;
my $cmd5 = "./script+1.pl -x -x";
my ($stdout5, $stderr5, $success5, $exit_code5) =
capture_exec( $cmd5 . "<<END\n" . $stdout4 . "\nEND" );
if ($?) { die "**** The process $cmd5 does not exist! ****"; }
if (not $success5) { die "**** The process $cmd5 failed with msg: $stderr5 ! ****"; }

print $stdout5;
```

References

1 Michael D. Adams. Scrap your zippers: a generic zipper for heterogeneous types. In *Proceedings of the 6th ACM SIGPLAN workshop on Generic programming*, WGP '10, pages 13–24, New York, NY, USA, 2010. ACM.

2 Marco Devesas Campos and L. S. Barbosa. Implementation of an orchestration language as a haskell domain specific language. *Electron. Notes Theor. Comput. Sci.*, 255:45–64, November 2009.

3 João Paulo Fernandes, João Saraiva, Daniel Seidel, and Janis Voigtländer. Strictification of circular programs. In *PEPM'11: Procs. of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 131–140. ACM, 2011.

4 João Paulo Fernandes, Alberto Pardo, and João Saraiva. A shortcut fusion rule for circular program calculation. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, pages 95–106, New York, NY, USA, 2007. ACM Press.

5 João Paulo Fernandes and João Saraiva. Tools and Libraries to Model and Manipulate Circular Programs. In *PEPM'07: Proceedings of the ACM SIGPLAN 2007 Symposium on Partial Evaluation and Program Manipulation*, pages 102–111. ACM Press, 2007.

6 Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

7 David Kitchin, Adrian Quark, William Cook, and Jayadev Misra. The orc programming language. In *Proceedings of the Joint 11th IFIP WG 6.1 International Conference FMOODS '09 and 29th IFIP WG 6.1 International Conference FORTE '09 on Formal Techniques for Distributed Systems*, FMOODS '09/FORTE '09, pages 1–25, Berlin, Heidelberg, 2009. Springer-Verlag.

8 Donald Knuth. Semantics of Context-free Languages. *Mathematical Systems Theory*, 2(2), June 1968. *Correction: Mathematical Systems Theory* 5 (1), March 1971.

9 Matthijs Kuiper and João Saraiva. Lrc - A Generator for Incremental Language-Oriented Tools. In Kay Koskimies, editor, *7th International Conference on Compiler Construction*, volume 1383 of *LNCS*, pages 298–301. Springer-Verlag, April 1998.

10 Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '03, pages 26–37, New York, NY, USA, 2003. ACM.

11 Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 295–308, New York, NY, USA, 1996. ACM.

12 João Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, December 1999.

13 João Saraiva and Doaitse Swierstra. Data Structure Free Compilation. In Stefan Jähnichen, editor, *8th International Conference on Compiler Construction, CC/ETAPS'99*, volume 1575 of *LNCS*, pages 1–16. Springer-Verlag, March 1999.

14 João Saraiva and Doaitse Swierstra. Generic Attribute Grammars. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 185–204, Amsterdam, The Netherlands, March 1999. INRIA Rocquencourt.

15 Doaitse Swierstra, Pablo Azero, and João Saraiva. Designing and Implementing Combinator Languages. In Doaitse Swierstra, Pedro Henriques, and José Oliveira, editors, *Third Summer School on Advanced Functional Programming*, volume 1608 of *LNCS Tutorial*, pages 150–206. Springer-Verlag, September 1999.

16 Doaitse Swierstra, Arthur Baars, and Andres Löh. The UU-AG attribute grammar system, 2004. `http://www.cs.uu.nl/groups/ST`.

17 Doaitse Swierstra and Harald Vogt. Higher order attribute grammars. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 48–113. Springer-Verlag, 1991.