A Purely Functional Combinator Language for Process Management*

Pedro Martins[1], João P. Fernandes[1], and João Saraiva[1]

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

Abstract

Software quality assessment is hot researching topic, further interesting when such assessments are the result of smaller, completely independent processes.

Controlling how such processes are organized to achieve a final result is a hard task, for the manually creation of such flow of inter-processes information would imply a tedious, error prone work which would not be very modular or adaptable to future improvements.

In this work we introduce a combinator library written in Haskell, composed by a set of combinators together with Attribute Grammar-based analysis and code generation mechanisms that transform such task into an easy, intuitive work that is also hugely modular and easily adapted and improved as needed.

1998 ACM Subject Classification D.2.11 Software Architectures, D.4.1 Process Management

Keywords and phrases Process Management, Combinators, Attribute Grammars, Functional Programming

Digital Object Identifier 10.4230/OASIcs.xxx.yyy.p

1 Introduction

Software quality assessment is a relevant research topic, and the implications of quality assessment are even more intricate and interesting when assessing open source software (OSS). With this is mind, the Certification and Re-engineering of Open Source Software (CROSS) project[1] was presented with the objective assessing the quality of OSS software in general.

Observing recent growing integration in various public and industrial organizations, OSS remains a risk as there are no substantial standards or analysis tools that can provide an assertive quantification of the overall quality of such products.

In the context of CROSS and our work, we call Certification to the detailed information of a particular OSS solution. Such Certifications would be able to process an OSS solution and provide a technical analysis of it, decreasing the exposure to risk on the adoption of OSS software in the organizations that use it.

Highly motivated by the OSS certification challenge, the goal of CROSS is four-fold: i) to select a number of specific problems in OSS certification to address; ii) to develop techniques for both code and documentation analysis; iii) to develop an open certification infra-structure for OSS projects; and iv) to frame the whole project into a collaboration with leading IT companies that rely on it.

* This work is funded by the ERDF through the Programme COMPETE and by the Portuguese Government through FCT - Foundation for Science and Technology, project ref. PTDC/EIA-CCO/108995/2008.

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

Conference/workshop/symposium title on which this volume is based on.
Editors: Billy Editor, Bill Editors; pp. 1–16

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The context of this work focuses on the development of an infra-structure, a Web Portal, that works both as a repository of all the tools that implement the analysis techniques for both code and documentation and as a service that allows the easy analysis and certification of OSS software. Such service has to maintain the OSS spirit of heterogeneous and distributed collaboration: the portal has to store all the tools produced and, more important, allow any user to create Certifications by re-arranging the tools in the order he/she finds best serves his/her needs. What's more, every user should be able to create his own Certification, that might represent any arrangement of the existing tools so that each specific Certification produces the exact results and information needed.

The combinator language introduced in this paper aims at allowing an easy configuration of the flow of information among processes (tools) both parallelized or chained to form 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 give 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 is explained in Section 5. In Section 6 we provide an overview of related works, in Section 7 we explain how our work could be further improved and extended and in Section 8 we conclude.

2 Motivation

This work is integrated into CROSS, a project whose aim is to research and develop new program understanding techniques capable of assessing and measuring the degree of excellency of open source software while being able to cope with the its often collaborative, distributed and heterogeneous character. Either individually or combined with others, they result on the production of reports called Certifications (to which we have a specific, XML based definition).

Such techniques are often composed by smaller units capable of communicating among themselves 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 (Certifications). A Component is one of this smaller software units.

A Component is therefore a a bash tool, capable of accessing and producing meta-data via the standard channels (STDIN and STDOUT) and that is capable of also receiving arguments that define the type of the information received via STDIN and the type of the information that is channeled through STDOUT.

The certifications composed by such components are often the result of environments whose philosophy closely resembles that of open source software: they are collaborative in the sense that they are produced by different research teams, who are often distributed themselves and heterogeneous because the motivation for their development can come from completely different sources.

The image in Figure 1 shows the flow of information in order to produce a Certification (called *'Certification 2'*), which is composed by two main sequences of information: one composed by a series of chained software units (*Java Slicer*, *Interface Analysis* and *csv2Report*) and another Certification (*'Certification 1'*). Since these two flows of information are independent, they can be executed in parallel and therefore can be transformed into two distinct sub-processes. Nevertheless, a requirement all certifications must fulfill in our environment is to always return results in our report format, so these two distinct processes

that represent the two flows of information are aggregated by a *'Reports aggregator'*, a component whose only responsibility is to aggregate outputs.

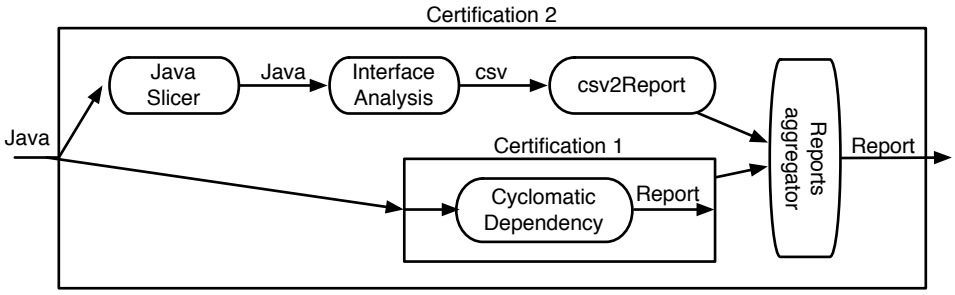

Figure 1 Flow of information for Certification 2.

It is also important to notice that the creator of the certification presented in Figure 1 does not have to worry about the details of the components and certifications it is made of. Indeed, these sub-units already existed in the system, and the user only needs to make sure that along the flow the information matches, both semantically and syntactically. This means that the types must match, and in a flow from component A to B the return type of A is the same as the input type of B, and that the nature of the information makes sense (for example, if a Slicer extracts the interface, it can not feed a component that analyses concurrency requirements or super classes, even if we are talking about the same language). As we shall see in Section 4, our system automatically performs syntactic type checking.

A traditional approach to implement a certification such as the one in Figure 1 would imply the cumbersome task of manually writing a script that is capable of launching processes in sequence and pipe their results, launching processes in parallel, timeout the processes and warn the user if any of them is taking too long to run and warn the user while minimizing the impact in processes that ran without problems, etc. An example of such a (simplified) script is written in Perl and is presented in Appendix A.

Despite being a possible solution to implement a certification such as *'Certification 2'*, creating a script manually has serious disadvantages: it is a tedious and cumbersome, error-prone. Furthermore, the effort increases exponentially to the complexity of the certification (i.e., with number of components and certifications the flow of information is made of). Finally, it is also very hard to analyze the flow of information and to perform tests such as checking the existence of circular dependencies or ensuring the types of inputs and outputs among components/certifications match.

Having an huge set of scripts manually written by random users is not a good practice as well: one can never be sure how well the script was written and how well it handles processing errors. Also, if the target system changes, would have to manually re-write everything to support the new target 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 Certification , Component , and a data-type for the processes work flow. In the

Listing 1 Data types for Certification and Component.

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

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

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

Listing 1 we show the data types for Component and Certification. A Certification has a name (such as *'Certification 1'*, as is in Figure 1) and a definition of a flow of information to achieve the results for that certification, here represented by the data type ProcessingTree.

A Component (also on Listing 1) is represented by a name, a bash call, which is the name from which the respective process shall be called on the system, and a list of inputs and outputs. Components might accept a varying number of input types as well as return a varying number of output types. The list of inputs and outputs have the same data type and represent the arguments the process supports and with which it shall be called in order to accept a particular format and in order to return a specific format as well.

In the context of the Figure 1 for example, the component *'Java Slicer'* might have the bash call *'JSlicer'* and have to be called with the arguments *'-java' '-interface'* in order to slice the interface out of java code, whereas if it was called with *"-java'* and *'-superclasses'* it would slice only the superclasses out of the Java code (and, in the particular example of Figure 1, break the flow of information).

Another important data type in our setting is ProcessingTree, that represents the flow of information. This data type, presented in Listing 2, represents the flow of information to create a certification, similar to the one presented in Figure 1. The simplest processing tree possible is the one composed only by one component, as shown by the constructor ProcessComp. This constructor represents a certification composed only by a component, together with the desired arguments to run the respective program, supplied by the user.

A processing tree can also be composed only by a certification (represented by ProcessCert) and run processes both in sequence (SequenceNode) and in parallel (ParallelNode).

In the case of the constructor ParallelNode, it takes as arguments a processing list and a processing tree (ParallelNode ProcessingList ProcessingTree). The first argument of a parallel computation is a processing list, which represents a list of processing trees which can run in parallel, independently of their complexity. The second argument, ProcessingTree, exists to fulfill a requirement of our specification: all the results of all the parallel computations must be aggregated using a component. Therefore, this processing tree must always be a component (this is checked by our type checking mechanism, as seen in Section 4) that is capable of aggregating information into one uniform, combined output.

Having defined these data types we can show, in Listing 3, a preview of our combinator

Listing 2 Data type for the ProcessingTree

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

Listing 3 An example of a Certification using our Combinator Language

```
Input >-  (jSlicer,"-j","-i") >-
          (iAnalysis,"-i","-csv")  >|
Input >- certification1            >|> (aggregator,"-r","-r")
```

language in the form of the certification defined in the previous section. We believe that our language is easy to understand, even without the combinators themselves presented. The flow of information between components/certification, where the information enters the chain of processes (Input), which parts correspond to parallel processes and how everything is aggregated.

By writing the small piece of code in Listing 3, the user is able to create a script with the exact functionalities such as the one presented in Section 2, but which will also be statically analyzed and type checked, and for which the code will be automatically generated.

The Sequence Processing Combinator.

For sequencing operations, we start by defining the sequence combinator: $>-$. This combinator defines processes that are supposed to run as a chain, i.e, where the output of the first is the input of the second and so on. In this case, if one of these processes fails the whole chain fails.

The combinator $>-$ must always start with the constructor Input, which is mandatory and used to make sure the user knows that is the exact place the flow of information starts. After the constructor Input, the user can write as many components and certifications as he wants, as long as they are connected by $>-$.

In Listing 4 we show an example of a chain of events. The combinator $>-$ can take as input certifications, components and other processing trees defined by other combinators. If the input is a certification (as is the example of *'certify'*), the combinator connects the processes before it to the processing tree of the certification, and makes sure the result of this processing tree is then channeled to the processes that appears after.

When the input is a component, the user has to supply both the component and the arguments it must take in order to both receive and return a specific format. In the particular example of Listing 4, *'(jSlicer,"-j","-i")'* has the argument *"-j"*, which ensures the process *'jSlicer'* accepts java code as input and the argument *"-i"*, which ensures it slices the interfaces out of the java code (instead of the super classes, for example, which in case were supported would need a different argument).

The arguments introduced with components are very important to allow the flow of information inter-processes, as the input and output types must match when the information is channeled. In the cases when certifications are the arguments of $>-$, the user is constrained by the input and output types chosen by whoever created the certification, and must be careful to ensure the types match.

Listing 4 An example of Sequence of Processes using the Combinator >−.

```
Input >- (jSlicer,"-j","-i") >- (iAnalysis,"-i","-csv") >- certif4
```

Listing 5 An example of Parallelization of Processes using the Combinators >| and >|>.

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

The result of a sequence of >- is a processing tree that implements the combination of processes. This tree is untested in the sense that this combinator does not perform any kind of analysis on such tree (such as type checking).

The Parallel Processing Combinator.

The next feature we introduce is the parallel composition of processes, that is actually supported by two combinators, >| and >|>. The first one is responsible for launching an infinite number of processes in parallel, while the second one is mandatory after a sequence of >| and chains all the outputs of the processes to a component that is capable of aggregating them.

In the Listing 5 we show an example of how this combinators work together. The combinator >| takes either a processing tree, a component, a certification or a set of the other combinators. Nevertheless, the arguments of >| must always start with the constructor Input, to give a clear idea to the user of the flow of information: in this case it is very easy to see where the information enters in this parallelization.

The combinator >|> is mandatory in the end of a parallelization and aggregates all the outputs of all the child processes into a standard output. It always combines the result of an infinite number of parallel processes using a component. An interesting note is that the use of just the combinator >|>, as in 'Input >|> (aggr,"-x","-r")' is not only possible, but also exactly the same as running 'Input >- (aggr,"-x","-r")', as they both channel the input to the component aggr. The combinator >| can never be used alone, due to the constrain that all parallel processes must be aggregated.

A Combinator to Create Certifications.

Another interesting combinator is +>. This combinator always combines a processing tree, in the left, with a String, in the right, creating a certification. In fact, if we ended both codes of Listings 4 and 5 with +> "cert4" we would, in both cases, create a certification with the name ″cert4".

An interesting feature of the combinator +> though is that is analyses the processing tree and makes sure all his types check, which means analyzing all the parallelized and sequenced process for their input and output types, and see wether they break or maintain the flow of information, and making sure that, in the end, the processing tree produces a report, which is mandatory for a certification in our system.

This combinator ensures that, whenever a certification is created, it's corresponding processing tree is necessarily valid. In Section 4 we explain in detail what this type analysis means and how it is performed.

Listing 6 Using the combinator #>>.

```
Input >- (comp1,"-j","-x") >- (comp2,"-k","-o") #>> 't'
Input >- cert1 >- cert5 #>> 's'
```

The 'Finalize' Combinator.

The last combinator of our language is #>>. This combinator combines a processing tree to a flag instructing it to either produce a script or just check for types.

The example of Listing 6 shows the two possible uses for this combinator. In the first case, we are checking if the types of running the component comp1 after the component comp2 match, i.e, if the returning type of comp1 is the same as the input type of comp2. The second example creates the script that implements chaining the certification cert5 after the certification cert1 (in Section 5 we explain in detail how this generation is made) and also checks if the types match.

An Example Scenario

In the examples presented before we have shown simple examples of how our combinator language can be used, and how much easier the process of controlling processes and creating certification (together with the script that implements them) becomes.

Even though those examples already prove the simplicity of our approach compared to the boring, error-prone manual task of writing of script, they are introductory and their aim was simply to introduce the combinators. This library can be used to intuitively create complex certifications without much effort but whose script would be exponentially more complex.

An example of such a script is on Listing 7. In this example we have introduced a parallel computation in the middle of a sequence of processes. An example of where such task would be useful is if we have a set of processes to analyze an Abstract Syntax Tree (AST) but the input is source code, so we need to first convert the source code using, in this case, the component *'comp2'*.

Without our combinator language one would have to manually edit it the script and make sure the process corresponding to *comp2* would feed each process on the parallel computation (that in this case is composed by 3 sub-processes but could be composed by 20). That by itself would imply a big effort from the user, but the task is not done yet.

Imagine now that we don't want the results of this parallel computation, but rather we want it to be compared against a repository of results to compare the characteristics of our AST. Someone else created a certification to do so ('cert2') but this certification does not take as input the same format our parallel processes return. The solution would be to channel the result of the parallel processes to an auxiliary component (in this case 'comp1') that would convert the formats so the information can be fed to the certification.

The overall task of performing everything manually would be huge. One would have to mess with existing scripts, potentially built by different people, understand them, and create the correct chain of information. And all this without even thinking about the script robustness, by ensuring that the processes are controlled in terms of processing times and failures for example.

And the enormous effort of building a script could not necessarily pay of. What if someone suddenly decided that they would want to change something in the parallel process, that he wants, for example, to add two more tasks. Would the user have to re-write and edit several

Listing 7 Example of a parallel process in the middle of a sequence of processes.

```
Input >- (comp2,"-s","-ast") >- parallel >-
                                          (comp1,"-h","-h") >- cert2

parallel  = Input >- (comp5,"-j","-x") >|
            Input >- (comp9,"-c","-x") >|
            Input >- (compL,"-c","-x") >|> (compAggr,"-x","-x")
```

parts of the script? Eventually, yes.

Our combinator approach has the advantage of not only making it incredibly easy to create flows of information among process, that can be easily edited, but also of being incredibly modular. The combinators's arguments are small pieces that can be easily edited, managed and transformed. Without much effort the programmer can easily change a certification, making it able of producing different results or improving it by introducing new safe processing mechanisms or any other possible features.

4 Type Checking on Combinators

We have introduced combinators in the previous section, and shown how they can be used to elegantly and easily create certifications which, in our setting, are flows of information among processes, implemented in a script.

We have also shown how easily such certifications can be created, but also how their definition in our setting is sufficiently manageable so that one can easily edit, maintain and extend such certifications to cope with the introduction of new features or the introduction of different analysis.

Such combinators match perfectly and ensure the programmer that anytime he writes a certification the order in which the combinators appear is natural to the definition of certifications on our setting, and that a wrong usage of combinations, typically by altering their natural order, is simply not possible. Such is supported by Haskell's powerful type system. By relying on both the advanced type mechanism provided and modern compilers such as ghc, we have the guarantee, for free, that the combinators are written in the write order because otherwise the compiler would detect a types mismatch between combinators and would simply generate an error - sometimes even suggesting possible fixes.

The analysis of the order of the combinators though is not the only test we want to perform. We also want to analyze if the types match in the flow of information. Returning to the first example on this paper (Figure 1), we can see that whenever information passes form process A to process B types must match. For example, the returning type of *'Interface Analysis'* must be the same as the input type of *'csv2Report'*, in this case, csv.

In our setting we perform such tests by transforming the combinators into a variable on the ProcessingTree data type (remember Listing 1) and by analyzing such tree in an Attribute Grammar (AG) nature. We chose an AG-based mechanisms because these are declarative in nature and make analysis and transformation of tree based structures (such as our ProcessingTree data type) extremely easy an intuitive.

What's more, our AG setting is purely functional and does not break the Haskell advantages shown previously and because AGs are an old formalism they are hugely studied and it is easy to find algorithms to support any feature and extensions we might want our system to support, such as detecting circular dependencies or supporting high order attributes for tree transformations.

The AG implementation we propose in this paper relies heavily in the concept of functional zippers. In fact, every element that we may want to analyze (i.e., upon which we define attributes) first needs to be wrapped up inside a Zipper (see Section 6 for more information about Zippers).

Type checking in our system is performed on a tree of the type of ProcessingTree and, because we used an AG 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 responsibility of an attribute called typeCheck. This attribute of type Boolean indicates if wether the types are correct or if there is anything wrong with them. Apart from this attribute there are also more two: input and output, that support typeCheck.

The attributes input and output have the type Language (remember 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 for Components node

Components are the simpler unit of our processing tree. These represent only a simple process without any flow of information and therefore it's type is always correct. The input and output attributes though are calculated by analyzing the Component data type and checking, for the arguments given, what are the corresponding types.

Whenever the user inserts a component into a certification, it is actually a triplet composed by the component together with the two arguments the user wants that process to be called with. This information is used together with the Inputs and Outputs variables presented on the components data type to check for the input and output types. If the arguments are not supported by the component our system will generate the corresponding error.

Type Checking for the Certifications node

Certifications are, similarly to components, simple nodes in a processing tree. An interesting feature of certifications though (and an important requirement of our system) is that they must always be created with the combinator +>. Therefore, anytime the user creates a certification we automatically check if it's sub tree types match, denying it's creation if such is not true. We do this easily since in our setting all AG attributes are also first order Haskell functions, so part of the implementation of +> is to check the result of the attribute typeCheck for the processing tree that constitutes the certification.

It is important to be aware of the rule that a certification must always return a result in the format of our report data type. Therefore, we must also check if the output attribute of the subtree of a certification corresponds to a report.

On the processing tree, every time a node with the type ProcessCert appears we are sure it relates to a certification previously created with +>, so we can always say it is true that it's types match. And because we use an AG based analysis the attributes input and output are very simple to implement, we just have to ask whatever their meaning is in the ProcessCert constructor child, that itself represents another process tree.

Type Checking for the Sequence node

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

The input and output attributes for this node are very simple: the input is whatever the input type of the first child is (the first processing tree), and the output is whatever the type the second child tree returns (the second processing tree).

The typeCheck attribute is also very simple: apart from checking if the output attribute of the first child is equal to the input attribute of the second child tree, the AG also asks for the typeCheck attribute on each sub tree individually.

Type Checking for the Parallel node

This node is the most complicated to perform type checking. The reason is due to the parallel processing definition that all the sub processes must have the same input type and the same output type, which must be the same as the input type of the component that aggregates all the results.

This node has always two children: the second child is a component that, as said before, aggregates all the components. The first child is a list of processes that run in parallel. The type checking for the parallel node itself is quite simple: check the output of the first element of the processing list (remember, all the processes that run in parallel have necessarily the same input and output types) and see if it is equal to the input type of the aggregating component. If it isn't, we have an error. If it is, we just have to type check the processing list.

The typeCheck attribute for the processing list is a bit more complicated. 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. I an attribute grammar setting though this is not very hard to implement: one just have to 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 something.

The input and output attributes for the parallel node are very simple: the input is the same as the input of one element of the processing list (they are all equal), and the output is the same as the output of the component. One important note thought is that, due to the importance of all the types of the processing list to be correct, we have chosen 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 be pleasantly 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 a supporting type checking mechanism was implemented, that guarantees a correct match of types throughout the processing chain.

In this Section we shall introduce the script generation that creates a Perl script that implements our combinators. The script is the low level, complicated representation of our combinators: it describes the processing chain, handles the individual processes for their completeness and manages the flow of information throughout all the processes the certification is made of.

The script generation follows the type checking mechanism idea of using an AG-based strategy. The basic idea is that each tree node, that represents a part of the processing tree,

Listing 8 Example of a script.

```
#!/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;
```

generates the corresponding piece of the script, and the overall meaning of the AG is the whole, fully working script.

In the Listing 8 we present an example of an 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 ensured and their STDERR is checked for problems. Following this checks, their results are channeled to whatever 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 because the computations ended.

The script generated by or system could be further improved. We could timeout the processes independently to ensure they do not go past a certain time slot, do a better job at controlling 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 ensure 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, etc.

We show a simpler implementation of such script on purpose: it is simple to read and simple to understand, which fulfills the expectations of this document. And despite a huge gap for upgrades, it still performs process control and scheduling of computations both on chained and in parallel flows of information. Furthermore, constructing such script manually is still a boring, error-prone task even for small certifications with a small number of processes. A huge certification (let's say, with more than 20 sub-processes) would imply an incredibly

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

amount of time to be implemented even by our simpler standards. And it would take even longer to be debugged, etc.

What's more, due to our AG-based mechanism, we argue that such improvements are not only easy to perform, they're actually a one-of job that once done becomes 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 only to change the respective attribute on the respective tree nodes.

Generating scripts automatically presents several difficulties, orthogonal to any generation mechanism, including our AG-based setting. Code translating presents the usual concerns of assuring that the result is both syntactically and semantically perfect, and that all constructors/primitives/declarations, etc 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 for example, the variable that stores the information that left 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 executional contexts). What is more, such mechanism is even harder to implement in parallel processing, where all the outputs are aggregated into one single process (remember Listing 2, 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's 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 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 variables' names 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 those meaning is the number of sub-processes per tree node. So if the first processing tree of a sequence has 4 sub-processes, than the variables on that processing tree are named from 1 to 4, and on the second processing tree the variables are 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 it's sub-processes. To do so, we ensure that that any information is first assign to a variable from which all the sub processes read their input.

Also for parallel computations, the results from all sub-processes are channeled to a variable that aggregates them. It's important to notice though that the outputs are not combined: they are simply channeled to a variable that feeds the aggregator component of the parallel computation, and 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 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 variables scope rules, 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, etc are written and in the their 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 [4] is introduced as a 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 [6]. 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.

Zippers were originally conceived by [3] 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 [5] 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

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

4 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.

5 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.

6 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.