FunBlocks Checker: Checking termination and confluence of FunBlocks programs

Marvin Fourastié

July 2021

Master thesis

Université de Genève

Abstract

The teaching of computer science has become essential in the majority of educational programs. Therefore, many programming languages were born with the will to propose a first programming experience to beginners, usually through sequences of instructions to execute. However, the programming language FunBlocks explores another approach focusing on state transformations based on term rewriting systems. Additionally, it is an educative programming language providing a visual interface inspired by block programming. The rewriting theory allows the user to create rules consisting in transform string of symbols (called terms) into other ones. Term rewriting has its own theory and contains some properties that are interesting for the learning of computer science. As the majority of these properties are undecidable in general, many verification tools exists, but there are not well suited for an educative purpose.

In an effort to provide an adapted response, we propose the FunBlocks checker that is a verification tool working on FunBlocks. Our tool aims to give the user the possibility to check two important properties, namely termination and confluence, using a simple command line interface. To do this, we use Maude, a language following the rewriting theory and containing a formal environment which can check if no infinite derivation exist (termination) or if each term has at most one normal form (confluence). The goal is to offer an extra educational value to FunBlocks by providing intuitive feedback emphasising on important properties in computer science, though the use of rewrite rules.

Contents

1 Introduction

Nowadays, a large majority of schools introduce computer science in their educative program, mainly through the programming of algorithms. Thereby, programming languages like Scratch [1] or Swift Playground [2] aim to propose a visual way to make a program. These types of languages provide predefined structures to build sequences of instructions that will be executed in a visual interface. Even though the majority of educative programming languages follows this pattern, some attempts have been made to bring another way to learn computer science, based on the declarative paradigm and term rewriting. Indeed, term rewriting focuses on state changes rather than instruction flow by the writing of rules, that are directed equations used to compute by replacing subterms of a given formula with equal terms. These rules allow to make an abstraction of state modifications of a program thanks to term substitutions.

Rewriting theory contains several properties that are important in computer science, among them, termination and confluence are the central ones. That is why some verification tools exist to check these two properties. Termination and confluence are both generally undecidable, the first one is verified when no infinite derivation exist and the second one is verified when each term has at most one normal form. The current verification tools for termination and confluence mainly focus on performance and are not well suited to be used as educative tools. Some attempts have been made to make the verification tools more user-friendly for beginners programmers (for example using a web interface), but no one has built a checker implemented over an educative programming language. Nevertheless the result of the verifications can provide valuable insights in order to give the user some information about his code.

On this purpose, we suggest the FunBlocks checker, a tool which takes a program and verifies properties to give the user an intuitive feedback about termination and confluence. Our tool works on FunBlocks, a language based on rewriting, where a program is expressed as a set of rules that can be used to transform some initial data into a satisfying solution [3]. To do this, we use Maude [4] which is a language

conceptually close to FunBlocks since it is a declarative language and based on rewrite systems. Furthermore, Maude provides a set of tools contained in the Maude Formal Environment (MFE) [5] which is specially built to verify rewrite system properties. The goal of this work is to create a checker for FunBlocks that is easy to use and adapted for an educational purpose.

All along this thesis, we will explore firstly the related works that have been made and afterward, we will develop the design and the use of the proposed FunBlocks checker tool. To end, we will discuss some perspectives and additional enhancements that could be done.

2 Background

In order to express clearly the different methods used to make the verifications, we will introduce some definitions we need from the term rewriting theory. Obviously we will not cover all the theory but we will take only the definitions we need. Moreover, we will also formulate our own ones to have clear meanings of the terms used.

To define rewrite systems, we will use the definition given by Nachum Dershowitz and Jean-Pierre Jouannaud [6]: "Rewrite systems are directed equations used to compute by repeatedly replacing subterms of a given formula with equal terms until the simplest form possible is obtained". Additionally, since rewrite systems have the full power of Turing machine [6], it can be used for programming languages like FunBlocks. A rewrite system is composed by a set of rules (namely directed equations) written as $l \to r$ with l and r that are terms. For instance, the application of the previous rule leads to the substitution of the term l by the term r.

Now we may wonder how to define a term. Once again we will build our definition using the existing theory. Starting from the definition given by the group of researchers "Terese" [7] we define the terms as follow: "Terms are strings of symbols from an alphabet, consisting of the signature and a countably infinite set [...] of variables". A signature is a non-empty set of function symbols each equipped with a fixed arity (the number of argument it is supposed to have) [7]. Therefore, we will classify the terms over a signature as:

- *Variables*, that are symbols constrained to range over a particular type. A variable can be substituted by any term of the same type.
- *Constants*, that are actually 0-ary functions.
- *Functions*, that are n-ary functions (with $n > 0$).

We have defined terms and rewriting systems. So now, we have to mention two restrictions on rewrite rules that will be imposed [7]. Let us consider a rewrite rule on the form $l \to r$:

- The left-hand side l is not a variable.

- Every variable occurring in the right-hand side r occurs in l as well.

Since rewrite systems are the cornerstone of the FunBlocks language, we have to introduce the main properties of them. There are five desirable properties involved in the verification of rewrite systems [8]:

1. *Termination*, meaning that no infinite derivations are possible.
2. *Confluence*, meaning that each term has at most one normal form.
3. *Soundness*, meaning that terms are only rewritten to equal terms.
4. *Completeness*, meaning that equal terms have the same normal form.
5. *Correctness*, meaning that all normal forms satisfy given requirement.

All five properties are in general undecidable [8]. As termination and confluence are the properties we will focus on, we will give more precise definitions later. A rewrite system is sound if each rule of the system represents a valid equation in an equational specification. For example, if $M = N$ is correct in an equational theory, so the corresponding sound rewriting system can contain a rule $M \to N$. Completeness is also related to equational theory since for a given axiomatization, if any two terms that are equal in the equational theory can be rewritten by a rewrite system to the same term [8]. Finally, the correctness is verified if normal forms are all in the set containing the right terms. In other words, all the irreducible normal forms of left terms in a rewrite system must be a right term of a rule of this system.

Termination

It is stated that a term rewriting system is terminating if and only if it admits a compatible reduction order [7]. This reduction order is said compatible if $l < r$ for every rewrite rule $l \to r$ in the term rewrite system [7]. Now we have to define what is a reduction order. This last has to follow three properties [7]:

- *Monotone*: $s_i > t \to f(s_1, \ldots, s_i, \ldots, s_n) > f(s_1, \ldots, t, \ldots s_n)$ for f a function of arity n.
- *Closed under substitution*: $s > t \to \sigma s > \sigma t$, for all substitution σ.
- *Well-founded*: no infinite descending chain. For example: $(\mathbb{N}, <)$ is well-founded (because of the least element 0), but $(\mathbb{Z}, <)$ is not well-founded.

As an example, let us consider the following term rewriting system:

$$\begin{aligned}\rho_1 &: f(a,b,x) \to f(x,x,x)\\ \rho_2 &: g(x,y) \quad \to x\\ \rho_3 &: g(x,y) \quad \to y\end{aligned}$$

The system is composed by three rules (namely ρ_1, ρ_2 and ρ_3), where a and b are constants, f and g are functions and x and y are variables. Thus, the following sequence of rewriting is infinite:

$$f(g(a,b),g(a,b),g(a,b)) \to_{\rho_2} f(a,g(a,b),g(a,b)) \to_{\rho_3} f(a,b,g(a,b)) \to_{\rho_1} f(g(a,b),g(a,b),g(a,b))\ldots$$

Thereby, the previous system is non-terminating and, by extension, no reduction order exists.

Confluence

The confluence of rewrite system is a direct consequence of the Church-Rosser properties [6]. Hence, we will consider that a system is Church-Rosser (or have the Church-Rosser property) if it is confluent. Confluence means that whatever the reduction path, the canonical form starting from a term is unique. To define this property more formally, we have to define the terms *overlap* and *critical pair*.

- *Overlap*: Two rules $l_0 \to r_0$ and $l_1 \to r_1$ overlap if a non-variable subterm of l_0 can be matched with the reductible expression l_1 (and vice-versa).

- *Critical pair*: It is a pair of terms $\langle t_0, t_1 \rangle$ obtained from a term t where t_0 is obtained by applying a rule on t and t_1 is obtained by applying another rule on a subterm of t (and vice-versa). Critical pairs is a consequence of overlap (even if an overlap not necessarily always leads to critical pairs).

let us illustrate these properties with an example. Let the following system composed by two rules:

$$\begin{aligned}\rho_1 &: f(g(x,y),z) \to g(x,z)\\ \rho_2 &: g(x,y) \quad\quad \to x\end{aligned}$$

where f and g are functions and x, y and z are variables. Now let us consider the two reductions:

$$f(g(x,y),z) \to_{\rho_1} g(x,z) \qquad\qquad f(g(x,y),z) \to_{\rho_2} f(x,z)$$

We notice that the subterm $g(x,y)$ of $f(g(x,y),z)$ from ρ_1 can be matched with the reductible expression $g(x,y)$ which is the left term of the rule ρ_2. Thus, by definition, we can say that ρ_1 and ρ_2 overlap. Furthermore, $\langle g(x,z), f(x,z)\rangle$ is a critical pair. If we add a rule $\rho_3 : f(x,z) \to x$ in the previous rewrite system, it becomes confluent since ρ_2 can be applied on $g(x,z)$ and ρ_3 can be applied on $f(x,z)$ leading to the same irreducible form x.

From these definitions leads an important property saying that a terminating rewriting system is confluent if and only if all critical pairs are convergent.

Now let us explain the main features of FunBlocks [3], as it is the language we accept to verify the termination and the confluence. FunBlocks is a declarative language based on term rewriting. A program on FunBlocks is basically composed by two elements: a program state and a set of rules. This last is represented by "cases" of functions that can be typed or not. FunBlocks allows to define types, composed by a name and some constructors, rules, that are the functions signatures, and cases, that are the rewrite rules. In this language, the execution is made by the user applying rewrite rules on the program state until reaching a canonical form. In FunBlocks, it is possible to work with a textual interface (see 5.1.1) or a visual interface [1] based on block programming.

[1] `https://blissful-bhaskara-4b1032.netlify.app/`

3 Related Works

Through this chapter, we want to emphasize the main domains explored to build the FunBlocks checker. Indeed, the goal was not to develop an application from scratch but to find useful tools, implementations and techniques that can be exploited to build an effective tool. At first, we will introduce a bit of theory by exploring the existing verification methods for term rewriting systems. Then, as our tool has to be above all an educative material, we will see what are the actual tools to learn term rewriting concepts. Finally, we will describe why Maude is a good solution to handle verification operations.

3.1 Verification methods for term rewriting systems

Termination and confluence are the central properties of term rewriting system. This can explain why we notice active researches about these properties on rewrite systems. As our work focus on these two properties, we want to present the techniques and methods currently used by the termination and the confluence checkers.

3.1.1 Termination

To begin, let us explore the termination problem that is a fundamental problem in computer science, as well as a termination competition [1] is organized every year allowing researchers to submit their termination checker tools. Several methods exist to prove termination, indeed, since the termination problem is generally undecidable, we cannot see directly if a rewrite system is terminating or not.

The goal of the proof is generally to find a compatible reduction order. To do this, we define the polynomial interpretation which can be use to prove termination of term rewriting systems [9]. The principle of the polynomial interpretation is to give for each term a weight in the polynomial form. These weights describe a reduction order [7] and consequently can be used to prove termination. To build this polynomial interpretation of term, each n-ary function is associated to a multivariate

[1] `http://termination-portal.org/wiki/Termination_Portal`

integer polynomial of degree n. As a result, for each rule $l \to r$, if $\tau(t)$ stands for the polynomial interpretation of t, we should have $\tau(l) - \tau(r)$ positive [6]. The polynomial interpretation is the most typical interpretation of a compatible reduction order.

Another way to prove termination is the methods based on path ordering. The basic idea is that two terms are compared regarding the paths through them [10]. We can denote three types of path ordering:

- *Multiset path ordering* [11]: The reduction order is based on multiset defined from the terms.
- *Lexicographic path ordering* [12]: The subterms of the same function symbol are compared lexicographically from left-to-right.
- *Recursive path ordering* [6]: Result of a combination of multiset path ordering and lexicographic path ordering.

The Knuth-Bendix ordering suggests a reduction order where the weights are not in the polynomial form. For instance, the weight of any variable is w and the weight of a term is just the sum of the weights of all the symbols appearing in it [13]. All the methods presented until now are used for direct proofs by analyzing the terms. However, these methods have some limitations when they have to be automated. Indeed, in some cases, it is impossible to prove a rewrite system terminating using only polynomial ordering, path ordering and Knuth-Bendix ordering [14].

To overcome these limitations, the dependency pairs method offers another way to prove termination. So, where the direct proofs will compare left and right-hand sides of rules, the central idea of the dependency pairs approach is to compare left-hand sides of rules only with those subterms of the right-hand sides that may possibly start a new reduction [14]. Let us define the dependency pairs as follow [14]:

Let a term rewriting system with f, g function symbols, $s_1 \dots s_n$ and $t_1 \dots t_n$ arguments of functions and $C[\ ]$ some context. A context can be defined by an incomplete term containing n empty places. Thus, $C[t_1, \dots, t_n]$ denotes the result of replacing the holes of C from left to right by the terms $t_1, \dots, t_n$ [7]. So, we consider the rewrite rule:

$$f(s_1 \dots s_n) \to C[g(t_1 \dots t_n)]$$

then $\langle f(s_1 \dots s_n), g(t_1 \dots t_n) \rangle$ is a dependency pair. To prove termination, we first need to define a chain [14]:

A sequence of dependency pairs $\langle s_1, t_1\rangle\langle s_2, t_2\rangle$ is an chain if there exists a substitution σ such that $t_j\sigma \rightarrow^\star s_{j+1}\sigma$ holds for every two consecutive pairs $\langle s_j, t_j\rangle$ and $\langle s_{j+1}, t_{j+1}\rangle$ in the sequence. These definitions leads to the theorem stating that a term rewriting system is terminating if and only if no infinite chain exists. Let us give an example with the following rewrite system:

$$\begin{aligned}
\rho_1 &: f(x, a) && \rightarrow x \\
\rho_2 &: f(g(x), g(y)) && \rightarrow f(x, y) \\
\rho_3 &: h(a, g(y)) && \rightarrow a \\
\rho_4 &: h(g(x), g(y)) && \rightarrow g(h(f(x, y), g(y)))
\end{aligned}$$

Here, f, g and h are functions, x and y are variables and a is a constant. Regarding to the definition given above, we have the dependency pairs:

$$\begin{aligned}
&\langle f(g(x), g(y)), f(x, y)\rangle \\
&\langle h(g(x), g(y)), f(x, y)\rangle \\
&\langle h(g(x), g(y)), h(f(x, y), g(y))\rangle
\end{aligned}$$

Hence, in our example, we have the chain:

$$\begin{aligned}
&\langle h(g(x_1), g(y_1)), h(f(x_1, y_1), g(y_1))\rangle \\
&\langle h(g(x_2), g(y_2)), h(f(x_2, y_2), g(y_2))\rangle
\end{aligned}$$

because $h(f(x_1, y_1), g(y_1))\sigma \rightarrow^\star h(g(x_2), g(y_2))\sigma$ holds for the substitution that replaces x_1 by $g(a)$, x_2 by a , and both y_1 and y_2 by a. However we can note that the chain above is not infinite.

The methods presented earlier are the ones used for the automated proofs of termination including the ones we use in our verification tool. Now let us have a look on the methods used to prove confluence.

3.1.2 Confluence

Even if confluence is, with the termination, one of the central property in rewrite systems, the research on this topic is less rich than it is for termination. However we can find some methods to automate confluence checking of a term rewriting system.

The tools existing to verify confluence mainly base on critical pairs analysis. As we remind the fact that a terminating rewriting system is confluent if and only if

all critical pairs are convergent. Thus, the criteria of confluence and non-confluence seem quite clear, and so, these direct criteria can be defined as follow [15]:
Let R a terminating rewrite system:

- *Knuth-Bendix criterion (confluence)*: R is confluent if and only if, for all $\langle s, t\rangle$ being a critical pair, $s = t$. In other words, all critical pairs must be joinable.

- *Simple non-confluence criterion*: If there exist some terms s' and t' such that $s \to^\star s'$ and $t \to^\star t'$ for some $\langle s, t\rangle$ being a critical pair and where s' and t' are normal forms and $s' \neq t'$, so R is non-confluent.

In the context of our work, we check only terminating rewrite systems. However, if termination cannot be proved, there exist other criteria [15] called Gramlich–Ohlebusch's criterion, Huet–Toyama–van Oostrom's criterion and Huet's strong-closedness criterion which verify other sufficient conditions for confluence. Then, to make the proof we sometimes need to apply divide-and-conquer methods [15] based on the fact that it is possible to decompose a rewrite system R to a composition of several rewrite system $R_1, R_2, \ldots, R_n$ [16].

For example, let the following rewrite system:

$$\begin{aligned}
\rho_1 &: f(x) &&\to g(x)\\
\rho_2 &: h(f(x)) &&\to h(g(x))\\
\rho_3 &: f(x) &&\to h(f(x))\\
\rho_4 &: g(x) &&\to h(g(x))
\end{aligned}$$

f, g and h are function and x is a variable. Additionally we note $R = \{\rho_1, \rho_2, \rho_3, \rho_4\}$ the rewrite system. In this case, we cannot prove R confluent with the Knuth-Bendix criterion. So we need to apply the divide-and-conquer method consisting in checking confluence for $R_1 = \{\rho_1, \rho_2\}$ and $R_2 = \{\rho_3, \rho_4\}$. We can now conclude that R_1 is confluent by Knuth-Bendix criterion and R_2 is confluent by Huet–Toyama–van Oostrom's criterion. Thus, R is confluent.

We have presented until now the techniques used to check properties of rewrite systems. They are the ones used in automated verifications of termination and confluence that will interest us to build our tool.

3.2 Tools for term rewriting

Starting from the term rewriting theory, a lot of tools have been implemented for different purposes. Among them, we are interested in two particular aspects of these tools. The first one is represented by the tools for verifications that are build for performance and aim to provide fast and efficient proofs on rewrite systems. The second one are the tools having an educational purpose. Indeed, as the FunBlocks checker has the main goal to be an educational material, we will also present the tools designed to make the term rewriting theory more understandable for potential students.

3.2.1 Tools for verification

Termination and confluence are undecidable properties and, consequently, a lot of projects are born with the will to build the most efficient tool possible. First of all, let us talk about termination tools. Here, we will explore three checkers: NaTT [17], MU-TERM [18] and AProVE [19]. The common point of these tools is that they use several techniques to prove termination (or non-termination). To ensure an efficient result, they are all three implementing direct methods (namely polynomial ordering, path ordering and Knuth-Bendix ordering) associated with some heuristics and dependency pairs based methods.

NaTT (Nagoya Termination Tool) aims to be powerful due to the use of weighted path order (WPO) and efficient due to the strong cooperation with external SMT solvers and dependency pairs methods based. Furthermore, after a participation to the Termination Competition, NaTT claims to be among the best tools for termination proofs of term rewriting systems [17]. The tool offers a command line interface to interact easily with the checker, but it is build mainly for performance. It allows only the standard file formats [2] for term rewriting systems used for the termination problems. This last point may lead to some limitations in order to use this tool as a backend for another implementation.

Among the termination tools cited, MU-TERM and AProVE are arguably the most popular ones and the most powerful ones. The first one claims to be the fastest to prove termination [18]. In other hand, MU-TERM (like the other tools presented) uses direct proofs and dependency pairs methods for the verifications. One of the specificity of this tool is that it can handle OBJ/Maude syntax additionally to the standard syntax for termination. MU-TERM gives also the possibility to be used

[2] `https://www.lri.fr/~marche/tpdb/format.html`

directly on the browser thanks to its web interface which allow the user to explore the different features of the tool.

On the other side, AProVE has similar features and performances as MU-TERM but it has some additional features that make this application easier to integrate into other software. This tool support a lot of format besides the standard ones for termination: XML (`.xml`), Prolog (`.pl`), Haskell (`.hs`), Java (`.jar`), LLVM (`.llvm`) and C (`.c`). Furthermore, AProVE can be use as a backend for Maude (like MU-TERM) and provides a web interface and a desktop interface.

The next type of tools we will explore are the ones to prove confluence. We will more specially talk about two tools: ACP [20] and Saigawa [21]. Both tools have participated to the Confluence Competition (CoCo) [3] and use the confluence criteria and the divide-and-conquer methods for the proofs.

The first one, ACP (Automated Confluence Prover) has some restrictions of utilisation: Standard ML of New Jersey, a Linux/UNIX platform and external SAT and SMT provers are needed to use it. Therefore, this tool is not well suited to be integrated in an other implementation. Saigawa brings as an extra that it is usable through the command line. ACP and Saigawa are very similar in their verification process, but moreover the last one provides a more usable interface even if it is a tool made firstly for performance.

3.2.2 Tools with educational purpose

Term rewriting systems are a good basis to learn computer science as the execution of a program can be seen as a rewrite sequence on program states [22]. On this purpose, tools have been developed to help students to handle rewriting computations and properties. Most of the tools we will present are essentially user-friendly interfaces of termination or confluence checkers.

TTT2 (Tyrolean Termination Tool 2) [23] provides a web interface where the user can enter a piece of code or upload a file in `.trs` format and then check termination of the system. It has its own backend and gives a clear and readable output. Indeed, this tool provides the proof of the termination (or non-termination) done by the system. The goal of TTT2 is mainly to be an efficient tool for termination and then offers a clear output to understand better the result.

CSI [24] is based on TTT2 but for confluence. The web interface is almost identical as the tool presented before and the input and output are in the same format. As

[3] `http://project-coco.uibk.ac.at/`

well for TTT2, we can choose between several methods to check termination and we have the possibility to test among some predefined files.

Contrary to the tools presented earlier, TRS.Tool [25] is different as it aims to be more complete than a confluence and termination checker. Indeed, it gives also the basic properties of a term rewriting system (including the termination and the confluence). Furthermore, this tool has a web interface with readable feedback and test files. The outputs allow the user to see in tables, not only the properties of the system, but also of each rule. We can add that the input format is the same as the last two tools, but TRS.Tool is a way less efficient to check termination and confluence.

The last tool we wanted to present is not a tool which just checks properties from an input file, but which helps the user to build a correct rewrite system using Knuth-Bendix completion [26]. This tool is the KBCV (Knuth-Bendix Completion Visualizer) [27] and has been developed by the same team as the creators of TTT2 and CSI. Obviously, this tool is less pertinent regarding to our work at first glance, but the interface is very interesting for an educative tool. The tool aims to be simple to use thanks to the buttons permitting to have a direct interaction with the interface. The outputs are once again very clear and all the computations made to verify the correctness of the set of rules are done internally. To use this tool, the user has to enter a set of equation and the game is to build the associated rewrite system from the set of equation entered.

To conclude, we can notice that a lot of tools have been made for term rewriting system. Most of them are build for performance but we have seen that, as rewrite system can be a good support to the learning of computer science, some tools have been thought to be easy to use and to give clear outputs to make the user understand better the computations made by the checkers. However these tools are not well suited to be integrated in an application as a plug-in or a library can do.

3.3 Verification of rewrite rules using Maude

To create our checker tool, we had the will to use existing implementation as much as we can to build something efficient, usable and user-friendly. In this section, we will present Maude [28] and more specially how it can be used to perform verification of a term rewriting system. Additionally, we will also compare Maude with other tools that could be used for this purpose.

3.3.1 The Maude System and its applications

Maude is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming for a wide range of applications. The goal claimed by Maude is that the language fulfills requirements on three dimensions [28]: simplicity, the program has to be as simple as possible, expressiveness, a wide range of applications should be naturally expressible, and performance, the computations have to be fast and efficient. These three goals fit perfectly in order to build applications on Maude. This language follows a rewrite semantic and its expressiveness allows the user to build his own language using Maude. That is why it can also be seen as a metalanguage.

We can find in the literature some applications built on Maude, like one which integrates CafeOBJ into Maude [29]. The goal is to enhance CafeOBJ functionality with Maude features. To do this, one has to use Full Maude and the metalanguage modules. Full Maude is, according to the documentation [28], "an extension of Maude, written in Maude itself, that endows the language with an even more powerful and extensible module algebra than that available in Core Maude". More precisely, Full Maude is mainly use to develop further extensions of Maude and is a basis to build our own application. Maude provides also metalanguages modules to construct user interfaces and metalanguage applications, in which Maude is used not only to define a domain-specific language or tool, but also to build an environment for a given language or tool.

Maude has also been used as a backend for high level Petri nets (HLPNs) analysis [30]. The idea of this work was to translate HLPNs to Maude rewriting system by translating the places to functional modules and transitions to conditional rules. These examples show that Maude is a well suited language to build applications that use rewrite logic.

3.3.2 The Maude Formal Environment

Many tools have been developed to enrich the Maude system. One of these tool is the Maude Formal Environment (MFE) [5] which aims to be used to verify properties of term rewriting logic from a Maude module. The MFE is composed by five tools but we will explore the two that will interest us: the Maude Termination Tool (MTT) [31] and the Church-Rosser Checker (CRC). As these tools have been developed in isolation, the goal of the MFE is to provide an executable formal specification in Maude within which a user can interact with tools to mechanically verify properties

of Maude specifications [5]. In the MFE, a user can easily use these tools in the same interface and directly through command lines in his shell.

Now let us have a look at the checkers of the MFE that will interest us the most namely the MTT and the CRC that perform respectively verification of termination and confluence. As Maude follows the rewrite theory, MTT does several transformations and then use an existing tool (like AProVE or MU-TERM) as a backend to check termination. Through the MFE, we can check the termination using a simple command line and get the output in the current shell.

The CRC checks the confluence of an assumed terminating Maude module. So, the MTT can be use prior to use the CRC to check if the module is terminating. To verify confluence, the CRC will look at all the critical pairs and will try to join them. Each time the CRC performs a verification, it will give as an output what are the critical pairs and if they are joinable.

The MFE is a very valuable tool as it allows a user to make verifications with efficient tools and quite directly thanks to a command line interface. The combination of Maude and the MFE offers a hybrid and extensible tool to build application based on rewrite system.

3.3.3 Another similar application

Now we will have a look at the most similar tool that can be use like Maude in order to verify properties of rewrite rules. CiME3 [32] is a toolkit which provide verification for termination and confluence on rewriting system. CiME3 makes the verification or uses a alternate prover (for termination) and then produce a proof trace that will be send to a certification engine like CeTa [33] or COQ/COCCINELLE [34]. The proof certification is a plus compared to the other tools, but it stays less extensible and well suited to build an application. Indeed, even if CiME3 provides a complete verification process as it is able to check termination and confluence, this tool is more thought to be complete and exhaustive as a prover than extensible like Maude and the MFE can be.

3.4 Synthesis

Term rewriting is a topic where a lot of implementations are done by researchers around the world. Papers about it are usually about the verification of rewrite systems which is a very challenging aspect as the properties are generally undecidable. However, we saw also that rewriting can be an interesting basis to learn computer

science as well as verification tools provide intuitive interface to allow the user to familiarize with rewrite rules. Furthermore, Maude brings valuable features as it can be use as a metalanguage to an implementation using rewriting logic.

Therefore, what it is missing the most is a tool that mix efficient verification, educational aspect and expressiveness in the sense it would be easy to use and would accept a large range of input. For instance a toolkit like CiME3 provide complete verification proof but no interface for the user, AProVE can handle a lot of input format but is hard to use as an educational tool by itself...

That is why, using Maude we aim to develop a tool that provide verification of a set of rewrite rules of a language designed for people who want to learn computer science. By considering the research done on these subjects, we want to bring something new by building an application for a large scale public. It has to be an educational tool based on a language following rewrite logic (namely FunBlocks) that will perform verifications and give valuable insights to the user.

4 Design of the tool

Here, we want to detail the proposed solution of our FunBlocks checker. We will discuss how the tool was thought, what are the required features and how the whole system works. To explain clearly our approach we will split our design in three parts: the translation from FunBlocks to Maude, the integration of the Maude Formal Environment and the command line based user interface. For now, we will not detail the implementation but we will rather have a high level view of the whole design.

4.1 Design of the system

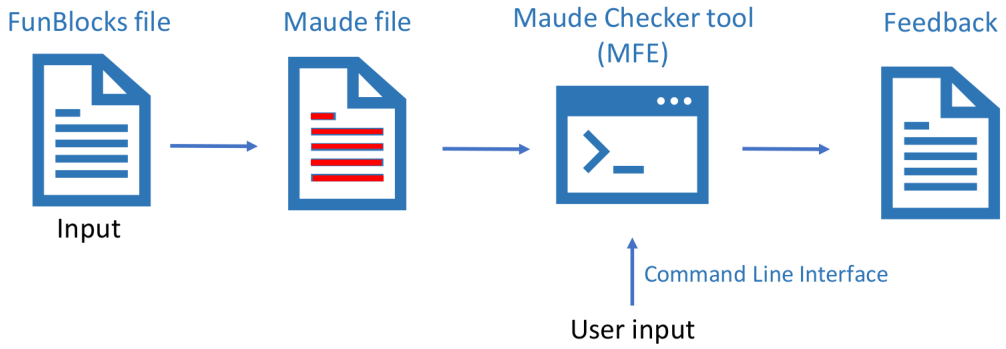

Figure 4.1: Simplified system design diagram

The figure 4.1 shows the simplified view of the system. As we can see, there are four main elements: the FunBlocks file, the Maude file, the user input (coupled with the MFE) and the feedback. Note that the last two elements are produced directly in the user shell. The execution flow is as follow:

1. The user launches the tool and gives as input his FunBlocks file (successfully compiled).

2. The system produces the corresponding Maude module from the FunBlocks file given in input.

3. The user enters in the shell the command which correspond to the verification he wants to do.

4. The system gives a feedback in the shell as a result of the verification.

We can make some comments about the different steps. First of all, in the step 1, the FunBlocks code has to be syntactically and semantically correct otherwise it can lead to unexpected error as in the step 2, the checker will parse the code to produce the corresponding Maude module. Furthermore, we can only make verifications on one FunBlocks file at the time. If a user wants to check another file, he has to redo the step 1 in order to have the correct Maude module in the system. Finally, the step 3 and 4 can be repeated as long as the user wants to perform verifications. The goal of the FunBlocks checker is to make the verifications easily through the command line interface and have a clear and readable response. As now we saw what the system takes and produces, we will have a look on the different computations made by the tool.

A program in FunBlocks is basically composed of two elements: a program state and a set of rules. For our verification we will only need the set of rules where each rule should be typed by the user. FunBlocks accepts untyped rules but in Maude, every terms need to be typed, that is why our checker accepts only typed rules. We will explain in the next chapter how we implemented the translation from FunBlocks to Maude, here we just want to explain the choices we have made about the design.

In order to have an intuitive tool, we wanted to hide as much as possible the computations to show only valuable information to the user. When the user will enter the command to produce the Maude module, the system will create a file called `funrules.maude` that will be the direct translation of the FunBlocks program. When `funrules.maude` is produced we use the MFE to make the verifications. The tool checks the termination with the MTT and the confluence with the CRC (see 3.3.2). Again, the goal of the project is not to create a whole system from scratch but to use appropriate tools to perform the verifications. Furthermore we want our tool to be easy to use thanks to the command line interface which allows the user to call the command of the MFE. We want our tool to be use through the command line, so we provide a list of commands (see A) to do the actions available of the MFE. The feedback given are based on the output of the MFE and are formatted to be the most readable possible.

4.2 FunBlocks checker usage

The tool works with a command line interface (CLI), the user can communicate exclusively through his shell and will have the responses in it too. The installation process is available in the README of the GitHub repository [1] and the list of command is in A. Now let us explore the use cases associated to all the actions available.

Transform FunBlocks file to Maude

The user has to specify the file he want to load and the system will generate a `funrules.maude` file that will be the file where the verifications will be made. Once it is done, the CLI will display a message indicating whether the generation has been done without any error or the generation is impossible due to errors.

Verify termination

When the command associated to the termination check is entered, the system will call the MFE to make the verification of the `funrules.maude` file. The user will have a message which will inform him if the termination has been successful or not. Furthermore, he will have the possibility to visualize details about the proof. In this case, the user will be able to open an additional file in his browser to see more details about the verification process.

Verify confluence

This action is similar to the previous one, but there are some notable differences we have to discuss about. When the confluence process is finished, the totality of the information is located in the feedback given in the shell. As it provide sufficient information, we did not add log visualization because we wanted our application to provide information simple to read. Moreover, the CRC considers the module terminating, thus, when we want to verify confluence, the MTT is called first to verify termination.

Change the FunBlocks file

The verifications are always performed on the `funrules.maude` file, that is why the user needs to recall the command to transform the FunBlocks file to Maude in order to generate another `funrules.maude` file.

[1] `https://github.com/fourastiemarvin/funblocks-checker`

5 Implementation

In the previous chapter, we have presented the system in its whole but we did not detail its implementation. The goal here is to present not only the tools, language and structures involved in the FunBlocks checker but also how each part of the system have been implemented. We will see especially the computations that are performed and how we thought it to build a responsive application.

5.1 Technologies involved

5.1.1 FunBlocks

As our tool make verifications on FunBlocks, we have to present more precisely the specifications of this language involved in the verifications done in the tool. It follows the term rewriting theory and so here are how the terms are represented:

- Variables : `$x, $y, $z,...`
- Constants: `zero, one, empty,...`
- Functions: `sort(empty), if(true, $x, $y), cons(1, cons(2, empty)),...`

To write a FunBlocks program, the user can create types, rules and cases. They are constructed as follow:

- *Types* can be created as `type Nat :: zero | succ Nat` for the natural number example. Here the type is called `Nat` and has two constructors: the constant `zero` and the function `succ` that takes a `Nat` as argument. It is also possible to create generic types, for example a type `Tree` which can be defined as follow: `type Tree $t :: empty | leaf $t | node (Tree $t) (Tree $t)`. Here, the `Tree` contains elements of generic type `$t`.
- *Rules* are the functions signatures, for example:
  `rule depth $t :: Tree $t => Nat` is a rule that defines the function `depth` with `Tree $T` as domain and `Nat` as co-domain.

- *Cases* are similar to rewrite rules in term rewriting systems. In FunBlocks, they are cases of rules that have been defined and are declared as follow: `case add(succ($r), zero) => succ($r)`. This case is based on the rule `add` that could have been defined as: `add :: Nat -> Nat => Nat`.

Additionally, we can note that FunBlocks is written in Swift. We will see later that to translate the FunBlocks code in Maude, we will have to use the AST to parse properly the code.

5.1.2 Maude

Now we want to have a look at the Maude specification. We won't go into details about the language but we will just explain the important concepts we used for our tool. The main element of a Maude file is the module which is composed by [28]:

- *sorts*, giving names for the types of data.
- *subsorts*, organizing the data types in a hierarchy.
- *kinds*, that are implicit and intuitively correspond to "error supertypes" that, in addition to normal data, can contain "error expressions".
- *operators*, providing names for the operations that will act upon the data and allowing us to build expressions (or terms) referring to such data.

These element are common for all types of module, however there are two types of modules: functional modules and system modules:

- *Functional modules* admit equations that represent theory and should be terminating and confluent
- *System modules* admit rules that represent states changes and have no constraints on termination and confluence

Nevertheless the important thing to remember is that both type of modules follow rewrite logic and all the reduction are made regarding to the rewrite theory. To make our translation between FunBlocks and Maude, we will use functional module, because the requirements match with the FunBlocks specifications, but also because the MFE is able to perform the verification only for functional modules. Furthermore we have to introduce the concept of "views" that has been used to make the translations. The definition of "view" given in the Maude manual [28] stands that they "specify how a particular target module or theory is claimed to satisfy a source theory". We will use them to deal with generic types.

5.1.3 The Maude Formal Environment

We have already explained in 3.3.2 the elements of the MFE, here, we want to explain the way to run and use it in our tool. To launch the MFE, we have to run first the Maude 2.7 environment and more especially Maude++ which is an extension that allows calls to termination backend. When we are in the Maude shell, we can call the MFE using the command `load`. This will launch the MFE and we can now use the corresponding commands [1] that are allowed to use the checkers (MTT, CRC, ChC, SCC, ITP).

As we said, we will use for now the MTT (for termination) and the CRC (for confluence). The first one gives two types of output: `"The module FUNRULES is terminating"` or `"The module FUNRULES is non-terminating"` as we always make the verifications on the file `funblocks.maude` which is the translation of the FunBlocks code. We precise that the MTT uses as backend AProVE. As the feedback produced by the result of the MTT is very limited, we chose to use the AProVE log to give to the user the most valuable insights about the termination proof process. Thanks to that, we can provide the method used, the rules and the paths tested and obviously the result of the proof.

For the CRC, we judged that the output produced is enough to give to the user the source of his potential mistakes. The confluence verification process is based on the critical pairs analysis: if they are joinable, it will give a positive response, in contrary, it will give the critical pair that cannot be proved joinable.

5.2 Computations

5.2.1 FunBlocks to Maude translation

To make the verifications on the FunBlocks code, we want to use the MFE and so we need to transform the FunBlocks code into Maude file. The first attempt was to write a parser in Maude which will read the file and interpret the code directly. The advantage of this method is that after this, we could create a user interface in pure Maude and handle the command directly. The initial idea was that the whole system would work in a Maude environment. However, we had some difficulties to solve some cases while parsing the FunBlocks code, like inference of variables or generic types handling.

[1] `https://github.com/maude-team/MFE/wiki/List-of-commands`

Therefore, we changed the idea and decided to use a more accommodate solution consisting in use the Swift parser of FunBlocks. Thus, we reuse this code to add a function in the `AST.swift` file (code which produce the FunBlocks AST) to produce the corresponding Maude code for each element of the AST. We chose to translate as follow:

Type declarations:

FunBlocks

```
type Nat :: zero | succ Nat
```

Maude

```
(fmod Nat is
    sort Nat .
    op zero : -> Nat .
    op succ : Nat -> Nat .
endfm)
```

The corresponding specification of types in FunBlocks is the declaration of a functional module which will allow us to provide functions (namely constructors) to these types.

FunBlocks

```
type List $T :: empty | cons (List $T) $T
```

Furthermore, we have also to treat the parameterized types and handle generic types. As in FunBlocks the terms are typed regarding to the context, we cannot see explicitly what is the type of a term, especially for generic types. Consequently, we chose to generate several views which will match to all the types defined:

Maude

```
(view $T1 from TRIV to Nat is sort Elt to Nat . endv)
(view $T2 from TRIV to Bool is sort Elt to Bool . endv) ...
(fmod List{T :: TRIV} is
    sort List{T} .
    op empty : -> List{T} .
    op cons : List{T} T$Elt -> List{T} .
```

```
endfm)
```

Rule declarations:

FunBlocks	**Maude**
`rule add :: Nat -> Nat => Nat`	`op add : Nat Nat -> Nat .`

In Maude, every equations (corresponding to FunBlocks rules in our case) must be typed. That is why we chose to only accept cases of typed rule when we parse the FunBlocks file. We can see above that rules declarations are function declarations in Maude.

Case declarations:

FunBlocks	**Maude**
`case add($x, zero) => $x`	`var $x : Nat .` `eq add($x, zero) = $x .`

The FunBlocks cases are directly translated in Maude equations, but the cases represent a challenging part of the translation as the variables in FunBlocks are contextually typed. As we said earlier, every terms have to be typed in Maude, thus, we must infer the types of the variables from the rules declarations. Another limitation stands for the generic types, similarly to the views, we defined several variables typed with all the types previously defined.

5.2.2 Structure of the files

Now, we will describe how the FunBlocks file and the generated Maude file are written. First of all the FunBlocks file has to be correct syntactically and semantically and has to fulfill the requirements of typing described earlier. An example of a correct FunBlocks file input can be:

```
type Bool :: true | false
type Nat :: zero | succ Nat
type List $T :: empty | cons (List $T) $T
rule size :: List $T => Nat
```

```
rule isEmpty :: List $T => Bool
case size(empty) => zero
case size(cons(empty, $x)) => succ(zero)
case isEmpty(empty) => true
```

In the program below, we declared two types `Bool` with two constants `true` and `false` and `List $T` which represent a parameterized type with two constructors. We declared also two rules: `size` and `isEmpty`. With these declarations, we added three cases. As we discuss earlier, the type of the lists are not explicitly given in FunBlocks, it depends on the context.
The corresponding Maude file will be generated as follow:

```
(fmod Bool is
    sort Bool .
    op true : -> Bool .
    op false : -> Bool .
endfm)

(fmod Nat is
    sort Nat .
    op zero : -> Nat .
    op succ : Nat -> Rel .
endfm)

(view $T1 from TRIV to Bool is sort Elt to Bool . endv)
(view $T2 form TRIV to Nat is sort Elt to Nat . endv)

(fmod List{T :: TRIV} is
    sort List{T} .
    op empty : -> List{T} .
    op cons : List{T} T$Elt -> List{T} .
endfm)

(fmod FUNRULES is
    including Bool .
    including Nat .
    including List{$T1} .
```

```
    including List{$T2}
    op size : List{$T1} -> Nat .
    op size : List{$T2} -> Nat .
    op isEmpty : List{$T1} -> Bool .
    op isEmpty : List{$T2} -> Bool .
    eq size(empty) = zero .
    var $x1 : Bool .
    var $x2 : Nat .
    eq size(cons(empty, $x1)) = succ(zero) .
    eq size(cons(empty, $x2)) = succ(zero) .
    eq succ($x1) = succ(succ($x1)) .
    eq succ($x2) = succ(succ($x2)) .
    eq isEmpty(empty) = true .
endfm)
```

All the verifications will be made on the functional module `FUNRULES`. The modules before `FUNRULES` represent the type declarations with the constructors. However, we can notice several specificities, like the parameterized type `List`. We need to declare additionally two "views" corresponding to the types declared. Then, in the main module `FUNRULES`, we have to include the types declared and `List{$T1}` and `List{$T2}` which represent the list of `Bool` and the list of `Nat`. Similarly, we need to do this with the variables and the equations to handle every types of list.

5.2.3 User interface

In order to have a simple and usable interface, we made a command line interface to have all the interaction through a shell. The goal is to have a quick and readable answer and also to be adapted to a wide range of user. The commands available are listed in A. To implement the CLI, we used Docopt [2]. Each command entered correspond to a command of the MFE and the responses are the outputs of the MFE, but we have also additional commands that will allow to translate the FunBlocks code into Maude code and to get some help.

[2] `https://github.com/docopt/docopt`

6 Further works

The FunBlocks checker is a new project that aims to be used and maintained in the future. In order to enrich the functionalities of this tool, we provide some further works that could be explored to future projects on this implementation. The tool as it actually exists is a first version and, therefore, there are a lot of features that can be added to make the checker as complete as possible.

6.1 Evaluations

The FunBlocks checker, being a tool for the FunBlocks language, has to be an educative tool. On this purpose, it seems essential to evaluate the tool with a panel users. The tool has been built to be easy to use, but we cannot really know how suitable it is for users to work with the FunBlocks checker. That is why user tests would be the first work to do. Furthermore, additional tests can be done directly on the tool in order to potentially detect some issues with the verifications. To do this, we suggest to test the tool with a set of FunBlocks codes. The FunBlocks checker has been tested but not with a big set of examples.

6.2 Improvements

Several improvement can be made, for example, we could improve the feedback given, that are for now the responses of the MFE. It could be valuable to format even more the outputs to give the exact rules where we detect a problem or the paths explored by the checker. A lot of work can be done on the verification process such as select the rules we want to check, select the reduction method we want to apply, ... Obviously, we can improve the CLI by adding commands which correspond to the improvement bring. We can also enhance the aesthetic of the CLI.

6.3 Additional features

Our tool has been developed with the will to handle additional features. The one that seems pretty clear is the variety of verifications, as the MFE provides other checkings that have not been used for now. We can also allow verifications on programs with untyped rules or embed reduction of terms directly in the tool. A fully visual interface could also be thought to have like a complete development environment online or offline.

These further works represent a non-exhaustive list of the works that could be done on the tool. The goal is to have a solid basis for additional features and improvements to enrich the FunBlocks checker and allow a wide range of users to play with it.

7 Conclusion

The FunBlocks checker is a tool that fulfill two main aspects: verification and educational purpose. Indeed this tool is made to provide verification on a FunBlocks program using the rewrite system theory and the methods existing for verification. In order to provide valuable feedback to the user, we focused on termination and confluence that are arguably the most important properties in term rewriting. Thanks to state of the art checkers and the Maude language, we built a tool that is efficient and easy to use.

Moreover, we have to mention that FunBlocks aims to be an educative language to learn programming using term rewriting, thus, our tool must fulfill this condition as well. By filtering the output of the checkers used and constructing a simple command line interface, we wanted to create a verification tool that can be use as well by beginners than confirmed programmers. Finally, our tool has been developed to be maintainable in order to add features and to serve as a basis for a new project.

Bibliography

[1] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The Scratch Programming Language and Environment," *ACM Transactions on Computing Education*, vol. 10, pp. 1–15, Nov. 2010.

[2] W. Malik, "The Swift Playground in Xcode," in *Learn Swift 2 on the Mac*, pp. 15–27, 2015.

[3] D. Racordon, E. Stachtiari, D. Morard, and D. Buchs, "Functional Block Programming and Debugging," p. 6, 2017.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada, "Maude: specification and programming in rewriting logic," *Theoretical Computer Science*, vol. 285, pp. 187–243, Aug. 2002.

[5] F. Durán, C. Rocha, and J. M. álvarez, "Towards a Maude Formal Environment," in *Formal Modeling: Actors, Open Systems, Biological Systems*, vol. 7000, pp. 329–351, 2011.

[6] N. Dershowitz and J.-P. Jouannaud, "CHAPTER 6 - Rewrite Systems," in *Formal Models and Semantics*, Handbook of Theoretical Computer Science, pp. 243–320, Jan. 1990.

[7] Terese, *Term Rewriting Systems*. Cambridge University Press, Mar. 2003.

[8] N. Dershowitz, "Computing with rewrite systems," *Information and Control*, vol. 65, pp. 122–157, May 1985.

[9] A. B. Cherifa and P. Lescanne, *Termination of rewriting systems by polynomial interpretations and its implementation.* report, INRIA, 1987.

[10] D. Kapur, P. Narendran, and G. Sivakumar, "A path ordering for proving termination of term rewriting systems," in *Mathematical Foundations of Software Development*, vol. 185, pp. 173–187, 1985.

[11] N. Dershowitz, "Orderings for term-rewriting systems," in *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*, pp. 123–131, Oct. 1979. ISSN: 0272-5428.

[12] S. Kamin and J.-J. Lévy, "Attempts for generalising the recursive path orderings," 1980.

[13] J. Dick, J. Kalmus, and U. Martin, "Automating the Knuth Bendix ordering," *Acta Informatica*, vol. 28, pp. 95–119, Feb. 1990.

[14] T. Arts and J. Giesl, "Termination of term rewriting using dependency pairs," *Theoretical Computer Science*, vol. 236, pp. 133–178, Apr. 2000.

[15] T. Aoto, J. Yoshida, and Y. Toyama, "Proving Confluence of Term Rewriting Systems Automatically," in *Rewriting Techniques and Applications*, vol. 5595, pp. 93–102, 2009.

[16] Y. Toyama, "On the Church-Rosser property for the direct sum of term rewriting systems," *Journal of the ACM*, vol. 34, pp. 128–143, Jan. 1987.

[17] A. Yamada, K. Kusakari, and T. Sakabe, "Nagoya Termination Tool," *arXiv:1404.6626 [cs]*, Apr. 2014.

[18] B. Alarcón, R. Gutiérrez, S. Lucas, and R. Navarro-Marset, "Proving Termination Properties with mu-term," in *Algebraic Methodology and Software Technology*, vol. 6486, pp. 201–208, 2011.

[19] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke, "Automated Termination Proofs with AProVE," in *Rewriting Techniques and Applications*, vol. 3091, pp. 210–220, 2004.

[20] T. Aoto, *ACP: System Description for CoCo 2020*.

[21] N. Hirokawa and D. Klein, *Saigawa: A Confluence Tool*.

[22] S. Winkler and A. Middeldorp, "Tools in Term Rewriting for Education," *Electronic Proceedings in Theoretical Computer Science*, vol. 313, pp. 54–72, Feb. 2020.

[23] M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp, "Tyrolean Termination Tool 2," in *Rewriting Techniques and Applications*, vol. 5595, pp. 295–304, 2009.

[24] H. Zankl, B. Felgenhauer, and A. Middeldorp, "CSI – A Confluence Tool," in *Automated Deduction – CADE-23*, (Berlin, Heidelberg), pp. 499–505, 2011.

[25] A. Julián, "Term Rewriting Systems .Net Framework," Master's thesis.

[26] D. E. Knuth and P. B. Bendix, "Simple Word Problems in Universal Algebras," in *Automation of Reasoning*, pp. 342–376, Berlin, Heidelberg: Springer Berlin Heidelberg, 1983.

[27] T. Sternagel and H. Zankl, "KBCV – Knuth-Bendix Completion Visualizer," in *Automated Reasoning*, vol. 7364, pp. 530–536, 2012.

[28] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott, *All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*. July 2007.

[29] A. Riesco, "An Integration of CafeOBJ into Full Maude," in *Rewriting Logic and Its Applications*, vol. 8663, pp. 230–246, 2014.

[30] X. He, R. Zeng, S. Liu, Z. Sun, and K. Bae, "A Term Rewriting Approach to Analyze High Level Petri Nets," in *2016 10th International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pp. 109–112, July 2016.

[31] F. Durán, S. Lucas, and J. Meseguer, "MTT: The Maude Termination Tool (System Description)," in *Automated Reasoning*, vol. 5195, pp. 313–319, 2008.

[32] E. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain, "Automated Certified Proofs with CiME3," p. 11.

[33] C. Sternagel, R. Thiemann, S. Winkler, and H. Zankl, "CeTA - A Tool for Certified Termination Analysis," *arXiv:1208.1591 [cs]*, Aug. 2012.

[34] R. Berlier, "CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verication of termination certicates," p. 30.

A List of commands

`funblocks_checker init FILENAME`
Generate the Maude file from a FunBlocks program. `FILENAME` is the full path of the corresponding FunBlocks file

`funblocks_checker [--log] (ct | check_termination)`
Check termination of the current FunBlocks file (generated with the `init` command). We can use `ct` or `check_termination`. The flag `--log` is optional, use it to display in the browser a description of the proof

`funblocks_checker (cf | check_confluence)`
Check confluence of the current FunBlocks file (generated with the `init` command). We can use `cf` or `check_confluence`.