Benchmarking Implementations of Term Rewriting and Pattern Matching in Algebraic, Functional, and Object-Oriented Languages – The 4th Rewrite Engines Competition

Hubert Garavel, Mohammad-Ali Tabikh, and Imad-Seddik Arrada

Univ. Grenoble Alpes, Inria, Cnrs, Lig, F-38000 Grenoble, France
E-mail: hubert.garavel@inria.fr

Abstract

Many specification and programming languages have adopted term rewriting and pattern matching as a key feature. However, implementation techniques and observed performance greatly vary across languages and tools. To provide for an objective comparison, we developed an open, experimental platform based upon the ideas of the three Rewrite Engines Competitions (2006, 2008, and 2010), which we significantly enhanced, extended, and automated. We used this platform to benchmark interpreters and compilers for a number of algebraic, functional, and object-oriented languages, and we report about the results obtained for CafeOBJ, Clean, Haskell, LNT, LOTOS, Maude, mCRL2, OCaml, Opal, Rascal, Scala, SML (MLton and SML-NJ), Stratego/XT, and Tom.

Keywords: Abstract data type, algebraic specification, compiler, functional programming, interpreter, object-oriented programming, programming language, specification language, term rewrite engine, term rewrite system

1 Introduction

There is a large corpus of scientific work on term rewriting. Beyond theoretical results, one main practical result is the introduction of algebraic terms and rewrite rules into many specification and programming languages, some of which are no longer available or maintained, some of which remain confidential, but a few others are widespread or increasingly popular.

In some of these languages, one directly finds the familiar concepts of abstract sorts, algebraic operations, and equations or rewrite rules to define the semantics of operations. In other languages based on functional, imperative, or object-oriented paradigms, one finds the term rewriting concepts under derived/restricted forms: types inductively defined using free constructors, and pattern matching.

In the presence of many different languages, which one(s) should be preferred for concrete applications? The present study addresses this question by assessing the performance of languages implementations on a set of rewriting-oriented benchmarks. A decade ago, this question was already dealt with by the three Rewrite Engines Competitions (REC, for short), organized in 2006 [6], 2008 [10], and 2010 [9]. Our work builds upon these three competitions.

A more personal motivation for undertaking the present study was related to CÆSAR.ADT [12] [18], a compiler developed at INRIA Grenoble since the late 80s. Based on Schnoebelen's pattern-matching compiling algorithm [37], CÆSAR.ADT translates LOTOS abstract data types (seen as many-sorted conditional term rewrite systems with free constructors and priorities between equations) into C code. In 1992, CÆSAR.ADT, initially written in C, was rewritten to a large extent in LOTOS abstract data types — it might have been the first rewrite engine to bootstrap itself, slightly before Opal [8] and long before ASF+SDF [43]. Since then, CÆSAR.ADT has been routinely used as part of the CADP toolbox [15] for model checking specifications of concurrent systems. For this purpose, it has been specifically optimized to reduce memory consumption by implementing data structures very compactly. It has also been used to build two large compilers: itself (using bootstrapping) and the XTL compiler [29].

In 2007, a comparative study [44] reported average performance results for CÆSAR.ADT, but only on a small number of benchmarks. This triggered our desire to assess CÆSAR.ADT against widespread implementations of term rewriting and pattern matching, to learn if this compiler is still state of the art.

Another personal motivation behind the present study concerns LNT [16], which has been designed as a user-friendly replacement language for LOTOS, more suitable for use in industry. These two languages are quite different: LOTOS relies upon ACT-ONE's [11] abstract data types, whereas LNT is a clean combination of imperative-programming constructs with first-order functional-programming discipline: mutable variables, assignments, `if-then-else` conditionals, pattern-matching `case`, `while` and `for` loops with `break`, functions having `return` statements and `in`, `out`, and `in-out` parameters, etc.

At the moment, definitions of LNT types and functions are implemented in two successive steps: first, by translation [16] to LOTOS abstract data types, using a generalization of the ideas proposed in [36], then to C code, by reusing the aforementioned CÆSAR.ADT compiler. This is a quite challenging approach, since a language with an imperative syntax (LNT) is first translated to term rewrite systems, then back to another imperative language (C). One may thus wonder whether such a two-step translation, dictated by software-reuse considerations, is efficient enough in practice, especially for model-checking verification [15], which is highly demanding in terms of memory and computing time.

Assessing the performance of implementations of term rewriting and pattern matching (including the case of LOTOS and LNT) raises a number of difficult questions. Is it possible to compare very different languages on an equal footing? Which are the right tools to be involved in the comparison? Where can one

find the term rewriting specifications to be used as benchmarks? What is the experimental setting suitable for a proper assessment?

The present article addresses these questions. It is organized as follows. Section 2 lists the various tools implementing term rewriting and pattern matching we considered for this study. Section 3 describes the common language REC-2017 in which benchmarks (i.e., conditional term rewrite systems) can be encoded. Section 4 presents the translators we developed for converting this common language to the input languages of the tools. Section 5 reports about the collection of 85 benchmarks prepared for our study. Taking advantage of this collection, Section 6 provides quantified insight about the verbosity of the input languages. Section 7 describes the execution platform used to run the tools on the benchmarks. Section 8 gives experimental results and draws the Top-5 podium of the most efficient tools. Section 9 discusses potential threats to validity. Finally, Section 10 provides concluding remarks.

2 Selected Tools

Table 1 lists all the languages assessed by the present study. For each language, column 2 gives bibliographic references; column 3 indicates whether the language is *alg*ebraic, *fun*ctional, or *obj*ect-oriented[1]; column 4 gives the name of the tool (interpreter or compiler) used to execute programs written in this language and the version number of this tool[2]; column 5 gives the URL of the reference web site.

The present study assesses numerous languages/tools that were neither considered in [44] nor in the three Rewrite Engines Competitions [6] [10] [9]: CafeOBJ, LNT, LOTOS, OCaml, Opal, Rascal, Scala, and SML. Conversely, a few languages/tools assessed during these former studies have not been retained for the present study: ASF+SDF (superseded by Rascal and Stratego/XT), Elan (superseded by Tom), μCRL (superseded by mCRL2), Termware (its last version 2.3.3 was issued in May 2009 and the tool did not participate in the 3rd Competition), and TXL (the developer informed us that the tool was not designed for the REC benchmarks and that there was no point in trying it). Concerning the languages/tools listed in Table 1, the following remarks can be made:

- Certain languages (namely, OCaml and Rascal) possess both an interpreter and a compiler; we evaluated each of them.
- The mCRL2 tool set provides two different rewrite engines: *jitty* (*just-in-time* [40] [41]) and *jittyc* (*just-in-time compiled*), which we both evaluated.

[1] This classification is subjective, since some languages belong to multiple paradigms: LNT claims to be functional and imperative, Maude algebraic and object-oriented, Opal algebraic and functional, OCaml and Scala functional and object-oriented, etc.; due to lack of space, we only indicate the paradigm that we consider to be the principal one, within the scope of this study.

[2] The tool name is only mentioned if it is different from the language name.

language	bib. ref.	kind	tool version	web site
CafeOBJ	[7]	alg.	1.5.5	http://cafeobj.org
Clean	[35]	fun.	2.4 (May 2017)	http://clean.cs.ru.nl
Haskell	[28]	fun.	GHC 8.0.1	http://www.haskell.org
LNT	[4][15]	fun.	CADP 2017-b	http://cadp.inria.fr
LOTOS	[22][15]	alg.	CADP 2017-b	http://cadp.inria.fr
Maude	[5]	alg.	2.7.1	http://maude.cs.illinois.edu
mCRL2	[20]	alg.	201409.0	http://www.mcrl2.org
OCaml	[26]	fun.	4.04.1	http://www.ocaml.org
Opal	[34]	alg.	OCS 2.4b	http://projects.uebb.tu-berlin.de/opal
Rascal	[42]	obj.	0.8.0	http://www.rascal-mpl.org
Scala	[33]	obj.	2.11.8	http://www.scala-lang.org
SML	[32]	fun.	MLton 20130715	http://www.mlton.org
SML	[32]	fun.	SML/NJ 110.80	http://www.smlnj.org
Stratego[a]	[3]	alg.	2.1.0	http://www.metaborg.org
Tom	[1]	obj.	2.10	http://tom.loria.fr

[a] The full name is "Stratego/XT", which we often abbreviate to "Stratego" so as to save space in tables.

Table 1: List of tools considered for the 4th Rewrite Engines Competition

The *innermost* rewriter mentioned in [44] is no longer available.

- SML (i.e., Standard ML) is the only language in Table 1 that does not allow conditional patterns. To overcome this limitation, we chose to use SML jointly with the Nowhere preprocessor [27] that adds support for Boolean guards in pattern matching.
- Among the many SML implementations available, we selected two compilers: SML/NJ and MLton.
- We evaluated the Clean compiler in two different ways: (i) by invoking the compiler `clm` in the standard way; (ii) by designing a custom script `clm-hack` that invokes `cocl` and `clm` with special options. The latter approach was suggested by the developers of Clean to address performance issues.

3 The REC-2017 Language

For benchmarking implementations of term rewriting, a crucial difficulty is that almost all the tools listed in Section 2 have a different input language — the only exception being SML/NJ and MLton, which both operate on SML programs. To address this issue, the organizers of the 2nd and 3rd Rewrite Engines Competition designed a common language [10, Sect. 3] [9, Sect. 3.1], which they named REC and to which we will refer as REC-2008. It is a human-readable, tool-independent format for describing many-sorted conditional term rewrite systems, as well as property queries (namely, confluence checks and computation

⟨*rec-spec*⟩ ::= `REC-SPEC` ⟨*spec-id*⟩ `\n`
`SORTS \n`
⟨*sort-id*⟩$^{*[\ ]}$ `\n`
`CONS \n`
⟨*cons-decl*⟩$^{*[\n]}$ `\n`
`OPNS \n`
⟨*opn-decl*⟩$^{*[\n]}$ `\n`
`VARS \n`
⟨*var-decl*⟩$^{*[\n]}$ `\n`
`RULES \n`
⟨*rule*⟩$^{*[\n]}$ `\n`
`EVAL \n`
⟨*term*⟩$^{*[\n]}$ `\n`
`END-SPEC \n`
	⟨*rule*⟩ ::= ⟨*left*⟩ `->` ⟨*term*⟩
	| ⟨*left*⟩ `->` ⟨*term*⟩ `if` ⟨*cond*⟩
	⟨*left*⟩ ::= ⟨*opn-id*⟩
	| ⟨*opn-id*⟩ `(`⟨*pattern*⟩$^{+[,]}$`)`
	⟨*pattern*⟩ ::= ⟨*var-id*⟩
	| ⟨*cons-id*⟩
	| ⟨*cons-id*⟩ `(`⟨*pattern*⟩$^{+[,]}$`)`
	⟨*term*⟩ ::= ⟨*var-id*⟩
	| ⟨*cons-id*⟩
	| ⟨*cons-id*⟩ `(`⟨*term*⟩$^{+[,]}$`)`
	| ⟨*opn-id*⟩
	| ⟨*opn-id*⟩ `(`⟨*term*⟩$^{+[,]}$`)`
	⟨*cond*⟩ ::= ⟨*term*⟩ `-><-` ⟨*term*⟩
	| ⟨*term*⟩ `->/<-` ⟨*term*⟩
	| ⟨*cond*⟩ `and-if` ⟨*cond*⟩

⟨*cons-decl*⟩ ::= ⟨*cons-id*⟩ `:` ⟨*sort-id*⟩$^{*[,]}$ `->` ⟨*sort-id*⟩	
⟨*opn-decl*⟩ ::= ⟨*opn-id*⟩ `:` ⟨*sort-id*⟩$^{*[,]}$ `->` ⟨*sort-id*⟩	
⟨*var-decl*⟩ ::= ⟨*var-id*⟩$^{*[\ ]}$ `:` ⟨*sort-id*⟩	

Table 2: EBNF grammar defining the syntax of the REC-2017 language

of normal forms) on such systems. The main limitation of REC-2008 was its exclusive focus on algebraic specification languages.

We adopted REC-2008 as a starting point, but gradually evolved it to make it compatible with the characteristics of functional and object-oriented languages. Such evolution led to a new language REC-2017[3], which is defined by the EBNF grammar of Table 2, in which ⟨*rec-spec*⟩ is the axiom and where ⟨e⟩$^{*[c]}$ (respectively, ⟨e⟩$^{+[c]}$) denotes the concatenation of $n \geq 0$ (resp. $n > 0$) non-terminals ⟨e⟩ separated by the character c, which can be a space, a comma, or a newline (noted "`\n`"). Semicolons can be used wherever commas can be used, which is sometimes convenient for structuring long lists of parameters (see, e.g., [17]). A simple example of a REC-2017 specification is given in Table 3.

Contrary to REC-2008, REC-2017 is strictly line-based, meaning that newline characters may only occur where they are specified in the grammar. Comments start with `%` and extend to the end of the line. Inclusion of external files is possible using `#include` directives similar to those that exist in the C language.

Identifiers (i.e., non-terminals ⟨*spec-id*⟩, ⟨*sort-id*⟩, ⟨*cons-id*⟩, ⟨*opn-id*⟩, and ⟨*var-id*⟩) always start with a letter, followed by any number of letters, digits, `_` (underscores), `'` (primes), or `"` (seconds); thus, fancy notations for numbers and infix functions (e.g., `123`, `+`, `&`, `==`, etc.) are not supported. Identifiers must be different from the grammar keywords (i.e., `SORTS`, `CONS`, `if`, etc.) and must

[3] See also `http://gforge.inria.fr/scm/viewvc.php/rec/2015-CONVECS/doc`

```
REC-SPEC simple
SORTS    % abstract data domains
  Bool Nat
CONS     % primitive operations
  true :  -> Bool
  false :  -> Bool
  zero :  -> Nat
  succ :  Nat -> Nat
OPNS     % defined functions
  and :  Bool Bool -> Bool
  plus :  Nat Nat -> Nat
VARS     % free variables
  A B : Bool
  M N : Nat
RULES    % function definitions
  and (A, B) -> B  if  A ->< true
  and (A, B) -> false  if  A ->< false
  plus (zero, N) -> N
  plus (succ (M), N) -> succ (plus (M, N))
EVAL     % terms to be evaluated
  and (true, false)
  plus (succ (zero), succ (zero))
END-SPEC
```

Table 3: Simple REC-2017 specification defining Booleans and natural numbers

not be prefixed with `REC_`, `rec_`, `Rec_`, or any case-sensitive variation of these. There is a unique name space for all identifiers, meaning, for instance, that it is not possible to have a sort and a variable sharing the same identifier. As in REC-2008, operation overloading is not permitted. Also, two identifiers must not differ only by their case (e.g., defining `x0` and `X0` simultaneously is invalid).

A REC-2017 specification is divided into six sections. The `SORTS` section declares a set of sorts. The `CONS` and `OPNS` sections respectively declare two sets of constructor and non-constructor operations; each operation is given by its identifier, the (possibly empty) list of sorts of its arguments and the sort of its result. The `VARS` section declares a set of free variables. The `RULES` section contains a list of rewrite rules; the left-hand side of each rule defines a non-constructor on arguments specified by a pattern (i.e., a term containing only constructors and free variables), whereas the right-hand side may be an arbitrary term; rewrite rules may be conditional or not; conditions, if present, are the logical conjunction (noted `and-if`) of one or many elementary conditions of the form "t_1`->><-`t_2" (meaning that both terms t_1 and t_2, which have the same sort, can be rewritten to some common term t) or "t_1`->/<-`t_2" (which is the negation of "t_1`->><-`t_2"). The `EVAL` section gives a list of closed terms (i.e., terms that do not contain free variables), the ground normal forms of which have to be computed.

There are additional static semantic constraints: in the `CONS` section, construc-

tors having the same result sort must be gathered and declared in sequence; in the `OPNS` section, non-constructors with arity zero (i.e., constants) must be declared before being used[4]; each pattern and each term present in a REC-2017 specification must be well-typed; implicit type conversions between different sorts are not allowed; in the `RULES` section, all variables used in rewrite rules must have been declared in the `VARS` section; each variable occurring on the right-hand side of a rewrite rule (including in conditions) must be present in the left-hand side pattern of this rule, i.e., the REC-2017 specification is a 1-CTRS according to the classification proposed in [31, Def. 6.1]; each left-hand side of a rule must be linear, i.e., must not contain multiple occurrences of the same variable[5]; the set of rules defining the same non-constructor (i.e., all rules whose left-hand sides have the same ⟨*opn-id*⟩) must be gathered and appear in sequence; these rules must be deterministic, meaning that either their left-hand side patterns are disjoint[6] or their conditions (if any) are mutually exclusive, thus implying confluence; these rules do not have to be complete, meaning that partial functions are allowed, provided they are invoked only on arguments for which they are defined; finally, termination is also required, even if it is not always decidable. Hence, because the specification is convergent, all tools, whichever rewrite strategy they implement, should terminate and produce the same results when evaluating the terms listed in the `EVAL` section.

Compared to REC-2008, the main change brought by REC-2017 is the distinction between constructors and non-constructors: in REC-2008, all operations were declared in one single section `OPS`, whereas REC-2017 introduces two separate sections `CONS` and `OPNS`, as well as the free-constructor discipline, i.e., the prohibition of equations between constructors implied by the syntactic definitions of ⟨*left*⟩ and ⟨*pattern*⟩ in Table 2. The "`get normal form for`" directives of REC-2008 have been replaced by the `EVAL` section of REC-2017. Also, some features of REC-2008 that were only used in some algebraic languages and have no counterpart in functional and object-oriented languages have also been removed, namely, the "`check the confluence of`" query and the OBJ/Maude-like `assoc` (associativity), `comm` (commutativity), `id` (identity, i.e., neutral element), and `strat` (strategy) attributes.

4 The REC-2017 Translators

Following the introduction of the REC-2008 language, various approaches were adopted during the 2nd and 3rd Rewrite Engines Competitions, to process specifications written in this language: (i) Maude was enriched with an environment to parse REC-2008 specifications; (ii) translators were developed to convert REC-2008 into the input languages accepted by ASF+SDF, Stratego/XT,

[4] This constraint arises from OCaml and SML; it eases the translation from REC-2017 to these languages.
[5] Non-linear patterns can be replaced by linear patterns by adding extra conditions.
[6] Consequently, there is no notion of priority between rewrite rules.

and Tom; unfortunately, these translators are no longer available today or their target languages have evolved; (iii) in other cases, the translation from REC-2008 to the input languages of the remaining tools (e.g., TXL) had to be done manually.

Our study is significantly broader in scope, as we are considering many more input languages, and we also plan to have more benchmarks written in REC-2017. Manually translating these benchmarks into each input language, detecting and correcting the unavoidable mistakes introduced during translation, and maintaining consistency between hundreds or thousands of files on the long run would not have been feasible.

It would not have been realistic either to ask tool developers to modify their tools to directly parse REC-2017 specifications. We therefore undertook the development of automated translators from REC-2017 to the 13 input languages. Actually, we developed 17 translators in total, since we experimented two different translations for both CafeOBJ (noted "CafeOBJ-A" and "CafeOBJ-B")[7] and Tom (noted "Tom-A" and "Tom-B")[8], and we also built a translator that produces files in the TRS input format of the AProVE termination checker[9] [19].

To keep these translators as simple as possible, we made a few radical decisions concerning their design and implementation.

The REC-2017 language has many static semantics constraints (listed in Section 3), which need to be checked automatically, since term rewrite systems are an error-prone formalism. Ideally, each benchmark should be checked at the REC-2017 source-code level, before being translated to the various target languages. Instead, we took the reverse approach: no checks are done *before* translation, all checks being done *after* translation. Concretely, each benchmark is translated first and the results of the translators are checked afterwards, thus deferring the verification of static semantics constraints to the target compilers and interpreters. In this approach, a REC-2017 benchmark is deemed to be correct if all its translations are accepted by the corresponding tools. The confluence property is checked by the Opal compiler, which enforces a sufficient condition of determinism, and the termination property is checked by the AProVE tool, which often succeeds in proving quasi-decreasingness, but sometimes loops forever on certain benchmarks.

We chose to perform translation at a mostly syntactic level, excluding all sophisticated semantic optimizations that could advantage or disadvantage certain target tools. This way, all the tools receive semantically-equivalent input files that only differ by syntax. In particular, we decided that the translators should not try to exploit the predefined types (Booleans, integers, polymorphic lists, etc.) and predefined functions (arithmetic operators, etc.) that may exist in

[7] CafeOBJ-A uses equations (`eq`, `ceq`, and `red` clauses), whereas CafeOBJ-B uses rewrite rules (`trans`, `ctrans`, and `exec` clauses).

[8] Tom-A makes no difference between constructors and non-constructors, whereas Tom-B defines non-constructors by pattern-matching (`%match` clause).

[9] `http://aprove.informatik.rwth-aachen.de`

the target languages. Concretely, all the target tools receive the constructors attached to each sort and the rewrite rules attached to each non-constructor, such information being unchanged from the source REC-2017 benchmarks. This was dictated by three reasons: (i) keeping translations simple; (ii) avoiding difficult choices between bounded-precision and arbitrary-precision integers; (iii) focusing assessment on the implementation of algebraic terms and rewrite rules.

Rather than using sophisticated compiler-construction tools (such as Rascal, Stratego/XT, or even LNT [14]) to build our translators, we adopted an agile, lightweight approach based on scripting. Taking advantage of the simple syntax of REC-2017 (based upon lines and code sections), we wrote our translators using a combination of `awk` (3070 lines, excluding blank lines and comments) and Bourne shell (1080 lines), with intensive use of Unix commands such as `cpp`, `grep`, and `sed`, all interconnected by pipes in data-flow programming style. Although this approach is not the most commendable nor the most optimal (for instance, the translator to Clean requires four passes), it has the merits of flexibility and conciseness (244 lines per translator on average).

Although all our translators are different, many of them share common traits, which can be summarized by the following list of questions concerning the target languages and the corresponding answers gathered in Table 4:

- Column (a): Are constructors and non-constructors handled differently (noted "D") or identically (noted "I")?
- Column (b): Are constructors declared together with their result type (noted "T") or separately (noted "S")?
- Column (c): Does the target language provide predefined equality/inequality functions (noted "E") or must these functions be defined explicitly (noted "–")?
- Column (d): Does the target language provide a predefined printing function for algebraic terms (noted "P") or must such a function be defined explicitly, e.g., as a monad (noted "–")?
- Column (e): Are rewrite rules encapsulated inside the non-constructor function they define (noted "F") or can they occur separately (noted "S")?
- Columns (f), (g), (h), and (i): Should type (resp. constructor, non-constructor, free-variable) identifiers start with a lower-case letter (noted "L"), an upper-case letter (noted "U"), or any of these (noted "–")?
- Columns (j) and (k): Should a constructor (resp. non-constructor) f with arity zero be invoked as "$f\,()$" (noted "()") or simply as "f" (noted "–")?
- Columns (l) and (m): Should a constructor (resp. non-constructor) f with arity $n > 0$ be invoked as "$f\,(e_1, ..., e_n)$" (noted "A" for application) or "$f\ e_1\ ...\ e_n$" (noted "J" for juxtaposition)?

language	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)	(k)	(l)	(m)
CafeOBJ-(A,B)	D[a]	S	E	P	S	–	–	–	–	–	–	A	A
Clean	D	T	–[b]	–	S	U	U	L	L	–	–	J	J
Haskell	D	T	E	P	S	U	U	L	L	–	–	J	J
LNT	D	T	E	P	F	–	–	–	–	–[c]	–[c]	A	A
LOTOS	D[d]	S	–	P	S	–	–	–	–	–	–	A	A
Maude	I[e]	S	E	P	S	–	–	–	–	–	–	A	A
mCRL2	D	T	E	P	S	–	–	–	–	–	–	A	A
OCaml	D	T	E	–	F	L	U	L	L	–	–	J	A
Opal	D	T	E	–	S	L	U	L	L	–	–	A	A
Rascal	D	T	E	P	S	U	U	L	L	()	()	A	A
Scala	D	T	E	P	F	U	U	L	L	()	()	A	A
SML	D	T	E	–	F	L	U	L	L	–	()	A	A
Stratego	I	S	–	P	S	–	–	–	–	()	()	A	A
Tom-A	I	T[f]	E	P	S	–	–	–	–	()	()	A	A
Tom-B	D	T	E	P	F	–	–	–	–	()	()	A	A

[a] CafeOBJ has annotations "`{constr}`" for constructors.

[b] In Clean, the library module `GenEq` provides generic comparison functions `===` and `=!=`, but we decided not to use them, as we were informed that they could be less efficient than user-defined comparison functions.

[c] In LNT, empty parentheses are optional after constructors and non-constructors with arity zero.

[d] Standard LOTOS does not have the notion of constructor, but the CÆSAR.ADT compiler introduces a distinction between constructors and non-constructors by means of comments "`(*! constructor *)`".

[e] Maude has annotations "`[ctor]`" that play no role when interpreting rewrite rules, but are understood and used by complementary tools (such as the Maude sufficient completeness checker).

[f] In abstract-syntax definitions for Tom-A, no distinction is made between constructors and non-constructors; they are all declared together with their result type.

Table 4: Overview of the idiosyncrasies of all target languages

Our translators do not build abstract syntax trees when parsing REC-2017 specifications. Instead, they exploit the line-based structure of REC-2017, so that most of the translation is done using substitutions specified by regular expressions. Although some parts of REC-2017 are not regular (namely, the sublanguages described by the ⟨*pattern*⟩ and ⟨*term*⟩ non-terminals, which are obviously context-free), regular expressions are sufficient to translate these parts, as the target languages are syntactically close to REC-2017.

The trickiest point is probably the translation of REC-2017 expressions (resp. patterns) written in application form, i.e., "$f\ (e_1, ..., e_n)$", into equivalent expressions (resp. patterns) written in juxtaposition form, i.e., "$f\ e_1\ ...\ e_n$". In most cases, this translation can be done using the Unix command `sed`, by applying two successive substitutions based on regular expressions: first, all commas are removed using the substitution [`,` → ε]; then, each function symbol f followed by an opening parenthesis is moved after this parenthesis, using the substitution [f`(` → `(`f]. For instance, the term "`f (a, g (b, c), h (d))`" is first

arity	kind	call in REC	declaration in OCaml	call in OCaml
$= 0$	non-constructor	f	**let** $f : t = ...$	f
$= 0$	constructor	C	**type** $t = C \mid ...$	C
$= 1$	non-constructor	$f\,(e_1)$	**let** $f\,(x_1{:}t_1) : t = ...$	$(f\ e_1)$ or $f\,(e_1)$
$= 1$	constructor	$C\,(e_1)$	**type** $t = C$ **of** $t_1 \mid ...$	$(C\ e_1)$ or $C\,(e_1)$
> 1	non-constructor	$f\,(e_1, ..., e_n)$	**let** $f\,(x_1{:}t_1) ... (x_n{:}t_n) : t = ...$	$(f\ e_1\ ...\ e_n)$
> 1	constructor	$C\,(e_1, ..., e_n)$	**type** $t = C$ **of** $t_1 * ... * t_n \mid ...$	$C\ (e_1, \ ...\ , e_n)$

Table 5: Regularity and irregularity in OCaml syntax

translated to "`f (a g (b c) h (d))`" and then to "`(f a (g b c) (h d))`".

The only exception is OCaml, which has different syntaxes for calling constructors and non-constructors, as shown in Table 5. To address this problem, we equipped our translator to OCaml with a small C program that distinguishes between constructors and non-constructors, and counts the nesting level of parentheses to decide whether commas between arguments must be preserved or removed. More generally, our most involved translators are those for ML-based languages (namely, OCaml and SML) because these languages have many particular syntactic cases (e.g., for constructors and non-constructors with arity zero or one), as well as static-semantics rules that forbid forward declarations (e.g., a constant function has to be declared before it is used) to promote readability.

5 Selected Benchmarks

To assess the tools listed in Section 2, a collection of conditional term rewrite systems is necessary. Despite the abundant literature on term rewriting, very few benchmarks are available on the Web. We therefore considered with great care the benchmarks developed during the 2006, 2008, and 2010 Rewrite Engines Competitions [6] [10] [9]. We completed these benchmarks with alternative versions kindly sent to us by F. Durán and P.-E. Moreau. We also added the benchmarks used in [44].[10]

Because these benchmarks have not been written by us, our initial intent was to keep them unchanged as much as possible when translating them into the REC-2017 language, so as to avoid introducing bias in subsequent experiments. However, we progressively realized that most benchmarks could not be reused directly, and that a number of changes were required to make them usable:

- Many benchmarks existed in several variants: we removed duplicated benchmarks, trying to follow differences between original and derived versions, to retain only the best variant.

[10] All these benchmarks are available from `http://gforge.inria.fr/scm/viewvc.php/rec/2006-REC1`, `http://gforge.inria.fr/scm/viewvc.php/rec/2008-REC2`, `http://gforge.inria.fr/scm/viewvc.php/rec/2010-REC3`, and `http://gforge.inria.fr/scm/viewvc.php/rec/2007-Weerdenburg`

- A few benchmarks were incorrect. For instance, the Fibonacci suite was improperly defined for $n = 1$. In some other benchmarks, a few operations had been lost during manual translations between different input languages. Specific actions were taken to repair such benchmarks.

- In many benchmarks, deep changes were made to enforce the separation (introduced in the REC-2017 language) between constructors and non-constructors. Constructors were identified and all equations between constructors were removed by splitting each problematic operation into a free constructor and a non-constructor [39]. Similar changes were done to also eliminate the implicit equations between constructors added by the three REC-2008 attributes `assoc` (associativity), `comm` (commutativity), and `id` (neutral element).

- A few benchmarks designed to perform rewriting on open terms (i.e., terms containing free variables) have been modified to rewrite only closed terms in their `EVAL` section.

- Some benchmarks had been specifically written for tools that assume different priority levels among rewrite rules (e.g., certain "default" rules are only applied if all other rules have failed). Such benchmarks were modified by adding extra conditions to these "default" rules, so as to avoid non-termination issues with tools not implementing priorities.

- Similarly, some benchmarks had been written for tools that apply particular rewrite strategies such as lazy evaluation, just-in-time rewriting, etc. Other benchmarks relied on the REC-2008 attribute `strat` that specifies in which order the arguments of an operation must be evaluated. A typical example was the $\texttt{ifthenelse}(c, x, y)$ operation defined by two rules: $\texttt{ifthenelse}(\texttt{true}, x, y) \rightarrow x$ and $\texttt{ifthenelse}(\texttt{false}, x, y) \rightarrow y$, which often caused performance or non-termination issues with rewrite engines based on functional application, as x and y are both evaluated whatever the value of c. Such benchmarks were modified, e.g., by replacing `ifthenelse` operations with conditional rules.

- Confluence was checked by translation to Opal, which requires deterministic rules. A few benchmarks were purposely non confluent, since they intended to compute all solutions to a problem. Two of them (`confluence` and `soundness-of-parallel-engines`) even used an undocumented "`get all normal forms of`" query. Because the REC-2017 language requires rewrite rules to be deterministic, thus confluent, such benchmarks had to be severely restricted; in some cases, their original intent was entirely lost, but this is a price to pay for having benchmarks that can be processed by a majority of tools.

- Termination was checked, whenever possible, by translation to AProVE, which produced proofs of quasi-decreasingness for many benchmarks.

- Certain benchmarks are intrinsically parametric: for instance, computing the factorial $n!$ depends on the value of n; solving the Hanoi tower problem depends on the number of disks; etc. Existing benchmarks would often test several parameter values in the same REC file, with the drawback that one could not distinguish between tools that quickly fail on small parameter values and tools that succeed on small values, but later fail on larger ones. To better measure the scalability of tools, we split each of these benchmarks into several new instances, each instance testing a single parameter value. Code duplication between various instances of the same benchmark was avoided by means of the `#include` directive introduced in the REC-2017 language.
- When introducing parameterized instances, we eliminated a few parameter values that were so large that no tool could feasibly handle them.
- Conversely, some parameter values were too small, and we tried to increase them, still making sure that one tool at least could handle them.
- Similarly, certain benchmarks (such as `langton`*) did non-trivial calculations, but were still too easy for most tools. We made these benchmarks more complex by iterating their calculations over a domain of values.

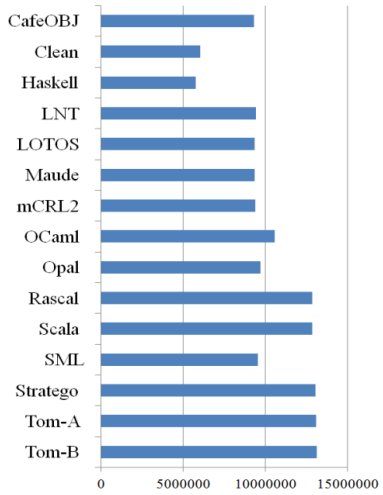
In addition to this work on existing benchmarks, we introduced new benchmarks that had never been used for the Rewrite Engines Competitions:

- We added two instances `tak18` and `tak38` of the Takeuchi function, which was one of the Kranz-Stansifer benchmarks [38] not already included in the set of REC benchmarks.
- We added a new benchmark `intnat` that defines signed integers and their related operations according to the axiomatization $\mathbf{F}_2^2$ proposed in [13]. This benchmark contains 1900+ tests to check that multiply, divide, and modulo operations are correctly defined.
- We added eight new benchmarks (`add`*, `mul`*, and `omul`*) that define binary adders and multipliers operating on 8-bit, 16-bit, and 32-bit machine words. Each of these benchmarks contains 4000+ tests to make sure that results are correctly computed.
- We added a large benchmark `maa`, which features a cryptographic algorithm formerly used to authenticate financial transactions [17]; this algorithm is described as a large term rewrite system (13 sorts, 18 constructors, 644 non-constructors, and 684 rewrite rules). The corresponding REC-2017 specification is 1575-line long; in comparison, the largest benchmarks inherited from the three former Rewrite Engines Competitions have less than 300 lines.

Doing so, we obtained a collection[11] of 85 benchmarks totalling more than 40,000 lines of code written in the REC-2017 language. We organized this collection

[11] `http://gforge.inria.fr/scm/viewvc.php/rec/2015-CONVECS`

language	# tokens
CafeOBJ[a]	9,294,297
Clean	6,037,299
Haskell	5,754,474
LNT	9,395,563
LOTOS	9,334,063
Maude	9,313,456
mCRL2	9,354,987
OCaml	10,565,289
Opal	9,701,085
Rascal	12,845,312
REC-2017	9,173,664
Scala	12,825,710
SML	9,517,116
Stratego	13,020,691
Tom-A	13,065,023
Tom-B	13,106,915



[a] CafeOBJ-A and CafeOBJ-B have the same number of tokens.

Table 6: Verbosity estimation of all languages, measured on the 85 REC benchmarks

into two parts: 15 "simple" benchmarks, which (almost) all the tools listed in Section 2 can handle successfully, and 70 "non-trivial" benchmarks, some of which are likely to be significantly more challenging for the tools.

6 Language Conciseness

As a by-product of our study, we can compute, for each language L, a *verbosity estimation* metric. This metric is not based on the number of code lines, a traditional measure that is much too subjective, as it depends on how often the translator targeting language L inserts line breaks.

Instead, our metric is defined as follows: we concatenate the source files of all the 85 benchmarks encoded in language L, and count the number of lexical tokens (i.e., keywords, identifiers, punctuations, mathematical symbols, etc.) present in these files. Compound symbols, such as "`->`", "`==`", or "`()`", count only for one.

This metric based on the number of tokens is quite objective, as it quantifies how many symbols a human should insert to get a working program. Moreover, counting tokens avoids the subjective debate of keywords, e.g., "**begin**" and "**end**" versus "{" and "}". Such quantitative feedback can be useful to language developers: if a language L' requires twice as many tokens as another language L to encode the same problem, then language L' is likely to be more difficult to learn and use by humans.

Table 6 summarizes our measurements. One can thus classify the languages in four categories: the "concise" ones (Clean and Haskell), the "normal" ones (CafeOBJ, LNT, LOTOS, Maude, mCRL2, Opal, and SML), a "slightly verbose" one (OCaml), and the "verbose" ones (Rascal, Scala, Stratego, and Tom).

Such results deserve a few comments: (i) The well-known conciseness of Clean and Haskell is confirmed; (ii) OCaml and SML suffer from the lack of predefined printers for values of constructor types, so that a printing function must be explicitly defined for each type; (iii) Opal lacks predefined printers too, but also lacks predefined structural equality to compare values of constructor types; (iv) Rascal, Scala, Stratego/XT, and Tom have large numbers of tokens partly because every call to a operation with arity zero must, according to Java conventions, be followed by a "`()`" token; notice that such large numbers probably arise from constructors with arity zero, since SML, which requires "`()`" only after non-constructors with arity zero, has a much lower number of tokens.

7 Execution Platform

To assess the tools listed in Section 2 in a reproducible manner, we installed them on two separate machines, each working with a single user, local file systems only (no NAS, NFS, Samba, etc.), and in stand-alone mode (no remote administration by computer staff, no automatic download of patches, etc.). Such constraints forced us to reuse retired servers.

As for the processors, we selected the widespread x86 (32-bit) and x64 (64-bit) architectures. We used a Sun Ultra 20 M2 server (2007) powered by one AMD Opteron 1210 (x86, dual core, 1.8 GHz) with 2 GB RAM, and a Transtec 2500L server (2004) powered by two AMD Opteron 246 (x64, single core, 2.0 GHz) with 16 GB RAM.

As for the operating system, we selected Linux because it is commonplace in software competitions and because several of the tools listed on Section 2 are not available on Windows. We chose the stable Debian release (Debian Linux 8 "Jessie"). For the Java-based tools (Rascal, Scala, and Tom) we installed OpenJDK (version 1.8.0-91).

Because some REC-2017 benchmarks manipulate large algebraic terms and certain tools make heavy use of recursion, the maximal stack size had to be increased; we set it to 32 MB on x86 and 512 MB on x64. For the Java-based tools, we set the JVM stack size to the same value and increased the overall JVM memory size to the maximum.

Because some tools seem to run forever or, at least, take much longer than others (we sometimes observed differences in two orders of magnitude), we used the Linux `timeout` command to allocate each tool a maximum amount of time. We set the timeout value to 360 seconds (i.e., six minutes) at most per benchmark — the choice of this value is justified hereafter. For the few tools that protect

themselves against interrupts by catching signals, the uncatchable POSIX signal SIGKILL is used to force termination.

To collect execution statistics, we use the `memtime` utility originally developed by the Uppaal team in 2002 and later enhanced at INRIA Grenoble[12]. Among the six values returned by `memtime` after each execution, we only retain the exit status (zero if ok, non-zero otherwise) and wall-clock time. The four remaining values (CPU time, memory usage, etc.) are not relevant in our context. For instance, memory usage is not meaningful, as it only concerns the main process: if a tool is provided as a shell script that launches child processes in sequence or in parallel, then `memtime` will only report the memory consumption of that shell script itself, ignoring all its child processes.

Thus, tools are launched as follows: "`memtime timeout 360` *tool options...*" and their execution can terminate in four different ways: *success* (normal completion, exit code is zero), *failure* (failed execution, exit code is non-zero, meaning that the tool properly halted, declaring it cannot tackle the problem), *crash* (abnormal interrupt by a signal, usually SIGSEV or SIGBUS, meaning that the tool terminated abruptly, often due to a programming error, such as dereferencing a null pointer, or to a stack overflow not handled properly), or *timeout* (interrupt when the wall-clock time exceeds the specified duration).

Because tools can fail, crash, or time out on different benchmarks, determining a performance ranking between tools is a non-trivial problem, since dilemmas arise from conflicting criteria. For instance, how to compare a tool that computes for a long time until it is halted by timeout with another tool that quickly stops and declares that it cannot solve the problem? Both tools have failed, and the former has even taken more time than the latter! Such issues have been studied, e.g., in [21] in the particular case of planning algorithms for robotics and artificial intelligence. "*Any comparison, competitions especially, has the unenviable task of determining how to trade-off or combine the three metrics (number solved, time, and number of steps).*" Based on this remark, we adopted two complementary metrics:

- Our first metric is the *score*, which counts how many benchmarks have been successfully tackled by a given tool within the specified timeout duration. This is the standard solution advocated in [21]: "*Because no planner has been shown to solve all possible problems, the basic metric for performance is the number or percentage of problems actually solved within the allowed time. This metric is commonly reported in the competitions.*"

- Our second metric is the *user time*, defined as the total wall-clock time spent by a given tool on a given benchmark, from the moment a tool is invoked until the tool terminates or is halted. The user time is always less or equal to the specified timeout duration. It is a somewhat heterogeneous metric that covers all steps used in problem solving, and not only the time spent in

[12] `http://cadp.inria.fr/resources/#memtime` (version 1.4)

"pure" rewriting. For an interpreter, the user time measures the time spent in processing the benchmark. For a compiler, the user time is the sum of the time spent in compiling the benchmark source file to binary code and the time spent in executing this binary code. The user time also includes the time needed to parse input files, the time taken by the C compiler for tools that generate C code, the time taken by the Lisp interpreter for tools that generate Lisp code, the time taken to launch, warm up, and halt the JVM for tools generating Java code, etc.

To determine a performance ranking between tools, we combine these two metrics (score, user time) using a lexicographic order, considering score first, and then user time only to distinguish between tools having the same score.

We developed a highly automated execution platform, with two families of shell scripts: the `run*` scripts to launch tools on benchmarks and collect execution results, and the `tab*` scripts to analyze these data and build spreadsheet files containing global statistics in CSV format.

8 Experimental Results

Table 7 summarizes the results obtained when running all the tools on the 70 "non-trivial" REC-2017 benchmarks, with a timeout set to 360 seconds. In each table cell of the form "X / Y", both values X and Y are sums computed over all the 70 benchmarks; X refers to the x86 platform (2 GB RAM)[13] and Y to the x64 platform (16 GB RAM)[14]. The last two columns of Table 7 give our two chosen metrics: score and user time.

Many useful findings can be drawn from Table 7. First, the score values should not be understood as absolute numbers; especially, the 50% score value should not be seen as a threshold separating "good" tools from "bad" ones. Indeed, if the 15 "simple" benchmarks mentioned in Sect. 5 were also taken into account in Table 7, then all scores would be above 52%, as each tool can tackle each simple benchmark in two minutes at most.

Also, it appears that most of the 70 benchmarks are "difficult". Precisely, only 21% (on 32 bits) and 27% (on 64 bits) of these benchmarks can be successfully tackled by all the tools within six minutes, meaning that all the other benchmarks make at least one tool fail, crash, or time out.

[13] The detailed 32-bit results are available from https://gforge.inria.fr/scm/viewvc.php/rec/2015-CONVECS/results-rec/2018-04-07-overview-360-32.csv?view=log and https://gforge.inria.fr/scm/viewvc.php/rec/2015-CONVECS/results-rec/raw-v2/2018-04-07-rec360-32.csv?view=log

[14] The detailed 64-bit results are available from https://gforge.inria.fr/scm/viewvc.php/rec/2015-CONVECS/results-rec/2018-04-05-overview-360-64.csv?view=log and https://gforge.inria.fr/scm/viewvc.php/rec/2015-CONVECS/results-rec/raw-v2/2018-04-05-rec360-64.csv?view=log

tool	successes	failures	crashes	timeouts	score	time (seconds)
CafeOBJ-A	31 / 31	8 / 4	0 / 4	31 / 31	44.3% / 44.3%	12490 / 13946
CafeOBJ-B	38 / 38	15 / 8	0 / 3	17 / 21	54.3% / 54.3%	8561 / 10170
Clean	30 / 40	30 / 22	8 / 1	2 / 7	42.9% / 57.1%	805 / 2632
Clean hack	54 / 54	10 / 10	0 / 0	6 / 6	77.1% / 77.1%	2623 / 2697
Haskell	70 / 68	0 / 0	0 / 0	0 / 2	100% / 97.1%	1867 / 2091
LNT	62 / 63	8 / 7	0 / 0	0 / 0	88.6% / 90.0%	1135 / 3028
LOTOS	62 / 63	8 / 7	0 / 0	0 / 0	88.6% / 90.0%	990 / 2884
Maude	64 / 67	2 / 0	1 / 0	3 / 3	91.4% / 95.7%	2095 / 2122
mCRL2 jitty	44 / 46	0 / 0	6 / 2	20 / 22	62.9% / 65.7%	9586 / 9914
mCRL2 jittyc	48 / 52	0 / 0	4 / 0	18 / 18	68.6% / 74.3%	8148 / 8101
OCaml compiler	64 / 64	0 / 0	0 / 0	6 / 6	91.4% / 91.4%	2772 / 2718
OCaml interpreter	60 / 61	2 / 2	0 / 0	8 / 7	85.7% / 87.1%	3937 / 3855
Opal	57 / 59	1 / 0	2 / 1	10 / 10	81.4% / 84.3%	4759 / 4817
Rascal compiler	34 / 37	4 / 2	0 / 0	32 / 31	48.6% / 52.9%	14486 / 14286
Rascal interpreter	37 / 40	2 / 0	0 / 0	31 / 30	52.9% / 57.1%	12274 / 12322
Scala	49 / 49	10 / 7	0 / 0	11 / 14	70.0% / 70.0%	6092 / 7147
SML: MLton	58 / 58	5 / 5	0 / 0	7 / 7	82.9% / 82.9%	4622 / 4997
SML: SML/NJ	52 / 52	5 / 5	0 / 0	13 / 13	74.3% / 74.3%	5672 / 5628
Stratego	48 / 49	3 / 0	0 / 0	19 / 21	68.6% / 70.0%	8260 / 9062
Tom-A	51 / 52	3 / 1	0 / 0	16 / 17	72.9% / 74.3%	7578 / 8064
Tom-B	60 / 60	3 / 1	0 / 0	7 / 9	85.7% / 85.7%	5438 / 5887

Table 7: Execution results for the 70 REC benchmarks on 32-bit / 64-bit platforms

It is expected from a software competition to rank tools according to their performance and merits. As discussed above, we rank the tools according to a lexicographic ordering that first compares the scores and, if the scores are identical, compares the total time spent. For conciseness, we only discuss the tools present on the Top-5 podium, i.e., those 5 (or 7) tools (out of 21) that can solve at least 85% of the 70 REC benchmarks, using at most 360 seconds per benchmark:

1. **Haskell ranks first** and, in particular, performs clearly better than any other functional language. This result seems in contradiction with prior studies, e.g., [23], which concluded that Clean was faster than Haskell, and the Computer Language Benchmarks Game[15], which reports that OCaml is often faster than Haskell. This could be explained by improvements brought to the GHC compiler over the last ten years (i.e., since the publication of [23]) and by the fact that our study strictly focuses on constructor types and pattern matching, whereas many examples of the Computer Language Benchmarks Game deal with numbers or arrays and rely on programmer's skills, which may strongly affect performance results for a particular tool.

[15] http://benchmarksgame.alioth.debian.org/u64q/ocaml.html

2. **Maude ranks second**, very close to Haskell on the 64-bit platform. This good score is remarkable for at least three reasons: (i) Maude works as an interpreter, not a compiler; (ii) its interpreter does not seem to exploit the "`[ctor]`" annotation used to distinguish constructors from non-constructors; (iii) although Maude has many sophisticated features (subsorts, associativity/commutativity/identity properties, reflection, strategies, modules, object-orientation, etc.), it does not neglect to do basic things (i.e., plain rewriting) efficiently.

3. **OCaml ranks third**. More precisely, the compiled version of OCaml (based on the native-code generator "`ocamlopt.opt`") takes the 3rd position, while the interpreted version (based on the bytecode generator "`ocamlc.opt`" and the bytecode interpreter "`ocamlrun`") takes the 6th position.

4. **CADP ranks fourth**. More precisely, its LOTOS compiler CÆSAR.ADT takes the 4th position and its LNT translator LNT2LOTOS takes the 5th position, reflecting the fact that LNT is first translated to LOTOS before being compiled using CÆSAR.ADT. The implementations of these languages are fast, but sometimes fail due to memory exhaustion, which might be solved by enabling the garbage collector and/or the hash-consing optimization, none of which is activated by default. Also, the present study contradicts the results of [44] claiming that mCRL2 performs better than LOTOS — notice that another study on the benchmarking of model checking tools [30] further disconfirms the claims of [44].

5. **Tom ranks fifth**, as the Tom-B translation variant takes the 7th position, with a score of 85.7%.

Interestingly, the Top 5 podium is the same on 32-bit and 64-bit platforms, although the numerical score of each tool might slightly differ on both platforms.

One may also question the influence of timeouts — see the related discussion (*Are Time Cut-offs Unfair?*) in [21, pp. 18 and 28]. Our timeout value was set to 360 seconds, so that the full cycle of experiments on all benchmarks takes roughly 24 hours, thus allowing daily progress on the study. Retrospectively, this value is appropriate, as it does not prevent the best tool (i.e., Haskell) from reaching a score of 100%, meaning that each of our benchmarks can feasibly be solved within 360 seconds.

Short timeout values give a significant advantage to tools that compute fast; larger timeout values might deeply modify the scores displayed in the Table 7 if the failure numbers of the "timeouts" column are transferred to the "successes" column. To investigate this possibility, we increased our timeout value to 1800 seconds (i.e., at most 30 minutes per benchmark) and redid our experiments, which took many days this time. Such a larger timeout value especially benefits Rascal, CafeOBJ, Scala, and Stratego/XT, whose scores increase by (up to) +18.6%, +11.6%, +7.1%, and +5.7% respectively. However, the Top 5 podium remains unchanged. It is worth noticing that on the 64-bit platform,

Maude now reaches a 100% score, like Haskell, but still remains in second position, as it appears to be 30% slower than Haskell on average.

9 Threats to Validity

Our results might be affected by various limitations, which we discuss hereafter:

- *Missing languages/tools.* Even though our study takes into account a much larger number of languages and tools than the former Rewrite Engines Competitions, we might have, due to lack of time and knowledge, omitted some prominent languages and tools. This does not affect the validity of our results, but calls for new, broader studies in the future.

- *Missing benchmarks.* We took great care to have a collection of benchmarks as large and as diverse as possible, both by reusing classical benchmarks from prior competitions and by developing new original benchmarks. Similar collections of term rewrite systems perhaps exist, although we did not find them by searching the Web nor by asking senior researchers in the field. As they are, the results of our study provide nevertheless valuable information that tool developers can exploit without the need for additional benchmarks.

- *Platform specifics.* The execution platform used for this study perhaps lacks diversity: the 32-bit and 64-bit machines both use old AMD Opteron processors and the same operating system. Conducting experiments with other processors and operating systems would be worthwhile; yet, x86/x64 and Linux are massively present in software competitions and research labs. Also, the Java-based tools have only been assessed using OpenJDK; their performance might be slightly improved using Oracle JDK, the proprietary implementation of Java.

- *Undetected errors.* Our execution platform does not check whether tool results (i.e., the ground terms obtained after evaluating the terms listed in each `EVAL` section) are correct, because each tool displays these results in a custom format. We checked most results visually, but not automatically.

- *Imprecise measurements.* In our study, each tool is considered as a "black box", from the point of view of an ordinary user more interested in tool performance than in internal algorithmics. Obtaining finer information, such as the time specifically spent in "pure" rewriting, and properly estimating the resources consumed by third-party software (C compiler, Lisp interpreter, Java runtime, etc.) would require the active participation of all tool developers to make such data available.

  Also, enforcing limitations on resource usage and accurately measuring the time and memory consumed by software tools is a difficult task [2]. In this respect, the `memtime` and `timeout` commands only deliver approximate

results; in particular, wall-clock time does not distinguish between sequential and multithreaded executions (although our outdated machines partly avoid this issue, as both can run at most two processes or two threads in parallel). More precise measurements of memory footprint and multi-core usage could be obtained using the BenchKit [25] or BenchExec [2] platforms intended for software competitions.

- *Misbehaving tools.* In our experiments, all tools are run in sequence on all benchmarks. There is always a possibility that a tool misbehaves and perturbs, either inadvertently or on purpose, the subsequent runs of other tools on the benchmarks, e.g., by leaving huge files that decrease available disk size or swap space, by forking processes that survive after the tool's termination and steal computing resources, or even by modifying the user's `crontab` file to launch periodic tasks. In our study, such issues are quite unlikely because experiments were conducted with already existing tools, but software competitions in general should address security issues and properly isolate tools using, e.g., virtual machines, as in BenchKit [25], or the `cgroups` feature of the Linux kernel, as in BenchExec [2].

- *Translation biases.* We observed that the main factor influencing the score of certain tools was (together with the amount of memory allocated to Java virtual machines) the quality of the translation from REC-2017 to the input languages of these tools. Depending on the translation, results can indeed greatly vary, as can be seen with Tom-A and Tom-B. We believe that our REC translators behave fairly: (i) they do not implement ad hoc optimizations to favour certain tools; (ii) they have not been designed to work against particular tools either: when performance issues were observed, translation outputs were sent to tool developers, whose feedback often helped to improve the translators; (iii) the generated files are available online for cross inspection by all stakeholders. In this respect, our fully-automated approach significantly improves over the former Rewrite Engines Competitions, in which some translations were done manually, thus allowing custom optimizations to be introduced for specific tools.

  Actually, we made two minor exceptions to this neutrality principle: (i) because the Opal compiler does not tolerate very long lines, our translator to Opal inserts a newline every twenty commas; this solved problems for three benchmarks; (ii) because the Java Virtual Machine has a limitation that methods cannot be larger than 64 kbytes, our translators to Rascal, Scala, and Tom (all based on Java) split each large `main` method into several smaller methods containing no more than 90 or 100 instructions each.

- *Misinterpretation of results.* Our study would not have been possible without the impressive work done by language designers and tool developers. Keeping this in mind, the results of Section 8 should not be construed as an absolute ranking of languages and tools, for at least two reasons: (i) our study focuses on certain specific language features, and (ii) our two metrics, score and user

time, encompass diverse algorithmic activities, many of which are performed by third-party software (C compiler, Lisp interpreter, Java runtime, etc.).

10 Conclusion

The present study builds upon the Rewrite Engines Competitions (2006, 2008, and 2010), but significantly extends them by bringing more languages/tools, more benchmarks, and full automation. These three former competitions led to somewhat inconclusive results, namely that tools were difficult to compare because they had different application domains, different strengths, and different weaknesses. Our study avoids this drawback by focusing on the most widely used fragment of term rewriting, keeping only those language features understood by all the tools (i.e., types defined by free constructors, conditional term rewrite systems that are deterministic and terminating, and evaluation of closed terms), excluding advanced features (e.g., predefined types, array types, higher-order functions, rewriting modulo associativity/commutativity/identity, context-sensitive rewriting, strategies, etc.) that are not implemented or not identically implemented across all the tools. Within these restrictions, our study highlights a large body of common traits shared by algebraic, functional, and object-oriented languages.

There have been recent efforts to define, for a large class of conditional term rewrite systems, a notion of complexity that approximates the maximal number of rewrite steps performed when reducing a term to its normal form [24]. Our compared evaluation of actual implementations can be seen as a practical complement to these theoretical results, possibly with a triple impact:

- Our experimental results refute the common beliefs that "speed performance is not an issue" and that "all tools are more or less linear in performance". The global success score measured on all 70 benchmarks varies between 50% and 100% across tools. The time required to process a large benchmark such as the MAA [17] varies by more than two orders of magnitude between the fastest tool and the slowest tool. Clearly, not all implementations are alike: performance and scalability definitely matter. In some cases (see Section 8), our results contradict conclusions from earlier studies or, at least, bring complementary information.

- Numerous scientific articles have been published about term rewrite engines and pattern-matching compiling algorithms, but it is unclear which ones are the most efficient in practice. Our software platform provides a basis for undertaking such a study. Should a systematic comparison be too demanding, it might be sufficient to learn from the tools (e.g., Haskell or Maude) that score high on the REC-2017 benchmarks.

- Our results may provide an incentive for tool developers to reconsider, and possibly improve, the way in which constructor types, term rewriting, and/or

pattern-matching are implemented in their tools. Radical decisions could even be taken, for instance, generating code in a higher-level language (e.g., Haskell) than C or Java, in order to defer the implementation of constructor types and pattern matching to a compiler shown to handle them efficiently.

Besides our evaluation results, our main contribution is a software platform[16] for benchmarking implementations of term rewriting and pattern matching. This platform has five components: (i) the REC-2017 language, which is a revised version of REC-2008 [10, Sect. 3] [9, Sect. 3.1]; (ii) a set of translators that automatically convert REC-2017 into 17 different languages; (iii) a collection of 85 benchmarks, consisting of (deterministic, free-constructor, left-linear, many-sorted, and terminating) conditional term rewrite systems expressed in the REC-2017 language, including closed terms to be evaluated; (iv) the output files generated by applying the translators to these benchmarks, thus providing test cases of possible interest to tool developers; and (v) a set of scripts for automatically running all the tools on all the benchmarks, recording execution results, and computing statistics.

This software platform already served in two recent case studies: a specification of the MAA cryptographic function [17] and an elegant definition of signed integers using term rewrite systems [13]. It could also be the starting point for re-launching the Rewrite Engines Competitions, which would progress the state of the art and address some of the limitations mentioned in Section 9.

Acknowledgements

We are grateful to Marc Brockschmidt (Cambridge/AProVE), Francisco Durán (Malaga/Maude), Steven Eker (Stanford/Maude), Florian Frohn (Aachen/AProVE), Carsten Fuhs (London/AProVE), John van Groningen (Nijmegen/Clean), Jan Friso Groote (Eindhoven/mCRL2), Paul Klint (Amsterdam/Rascal), Pieter Koopman (Nijmegen/Clean), Davy Langman (Amsterdam/Rascal), Xavier Leroy (Paris/OCaml), Florian Lorenzen (Berlin/Opal), Pierre-Etienne Moreau (Nancy/Tom), Jeff Smits (Delft/Stratego), Jurriën Stutterheim (Nijmegen/Clean), and Eelco Visser (Delft/Stratego) for their patient explanations and help concerning their respective tools. The present paper also benefited from discussions with Bertrand Jeannet (Grenoble), Fabrice Kordon (Paris), and Jose Meseguer (Urbana-Champaign).

[16] It is available on the SVN server of INRIA and can be obtained using the command "`svn checkout svn://scm.gforge.inria.fr/svnroot/rec/2015-CONVECS`", or "`svn checkout svn://scm.gforge.inria.fr/svnroot/rec`" to obtain also all the benchmarks used in the three former REC competitions.

References

[1] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking Rewriting on Java. In Franz Baader, editor, *Proceedings of the 18th International Conference on Term Rewriting and Applications (RTA'07), Paris, France*, volume 4533 of *Lecture Notes in Computer Science*, pages 36–47. Springer, June 2007.

[2] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Benchmarking and Resource Measurement – Application to Automatic Verification. In Bernd Fischer and Jaco Geldenhuys, editors, *Proceedings of the 22nd International Symposium on Model Checking Software (SPIN'15), Stellenbosch, South Africa*, volume 9232 of *Lecture Notes in Computer Science*, pages 160–178. Springer, August 2015.

[3] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17 – A Language and Toolset for Program Transformation. *Science of Computer Programming*, 72(1–2):52–70, 2008.

[4] David Champelovier, Xavier Clerc, Hubert Garavel, Yves Guerte, Christine McKinty, Vincent Powazny, Frédéric Lang, Wendelin Serwe, and Gideon Smeding. Reference Manual of the LNT to LOTOS Translator (Version 6.6). INRIA, Grenoble, France, February 2017.

[5] Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. *Maude Manual (Version 2.7.1)*, July 2016.

[6] Grit Denker, Carolyn L. Talcott, Grigore Rosu, Mark van den Brand, Steven Eker, and Traian-Florin Serbanuta. Rewriting Logic Systems. *Electronic Notes in Theoretical Computer Science*, 176(4):233–247, 2007.

[7] Razvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report – The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.

[8] Klaus Didrich, Andreas Fett, Carola Gerke, Wolfgang Grieskamp, and Peter Pepper. OPAL: Design and Implementation of an Algebraic Programming Language. In Jürg Gutknecht, editor, *Proceedings of the International Conference on Programming Languages and System Architectures, Zurich, Switzerland*, volume 782 of *Lecture Notes in Computer Science*, pages 228–244. Springer, March 1994.

[9] Francisco Durán, Manuel Roldán, Jean-Christophe Bach, Emilie Balland, Mark van den Brand, James R. Cordy, Steven Eker, Luc Engelen, Maartje de Jonge, Karl Trygve Kalleberg, Lennart C. L. Kats, Pierre-Etienne Moreau, and Eelco Visser. The Third Rewrite Engines Competition. In Peter Csaba Ölveczky, editor, *Proceedings of the 8th International Workshop*

on Rewriting Logic and Its Applications (WRLA'10), Paphos, Cyprus, volume 6381 of *Lecture Notes in Computer Science*, pages 243–261. Springer, 2010.

[10] Francisco Durán, Manuel Roldán, Emilie Balland, Mark van den Brand, Steven Eker, Karl Trygve Kalleberg, Lennart C. L. Kats, Pierre-Etienne Moreau, Ruslan Schevchenko, and Eelco Visser. The Second Rewrite Engines Competition. *Electronic Notes in Theoretical Computer Science*, 238(3):281–291, 2009.

[11] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1 – Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[12] Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, pages 147–162. North-Holland, December 1989.

[13] Hubert Garavel. On the Most Suitable Axiomatization of Signed Integers. In Phillip James and Markus Roggenbach, editors, *Post-proceedings of the 23rd International Workshop on Algebraic Development Techniques (WADT'16), Gregynog, Wales, UK*, volume 10644 of *Lecture Notes in Computer Science*, pages 120–134. Springer, December 2017.

[14] Hubert Garavel, Frédéric Lang, and Radu Mateescu. Compiler Construction using LOTOS NT. In R. Nigel Horspool, editor, *Proceedings of the 11th International Conference on Compiler Construction (CC'02), Grenoble, France*, volume 2304 of *Lecture Notes in Computer Science*, pages 9–13. Springer, April 2002.

[15] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 15(2):89–107, April 2013.

[16] Hubert Garavel, Frédéric Lang, and Wendelin Serwe. From LOTOS to LNT. In Joost-Pieter Katoen, Rom Langerak, and Arend Rensink, editors, *ModelEd, TestEd, TrustEd – Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, volume 10500 of *Lecture Notes in Computer Science*, pages 3–26. Springer, October 2017.

[17] Hubert Garavel and Lina Marsso. A Large Term Rewrite System Modelling a Pioneering Cryptographic Algorithm. In Holger Hermanns and Peter Höfner, editors, *Proceedings of the 2nd Workshop on Models for Formal Analysis of Real Systems (MARS'17), Uppsala, Sweden*, volume 244 of *Electronic Proceedings in Theoretical Computer Science*, pages 129–183, April 2017.

[18] Hubert Garavel and Philippe Turlier. CÆSAR.ADT : un compilateur pour les types abstraits algébriques du langage LOTOS. In Rachida Dssouli and Gregor v. Bochmann, editors, *Actes du Colloque Francophone pour l'Ingénierie des Protocoles (CFIP'93), Montréal, Canada*, pages 325–339, Paris, September 1993. Hermès.

[19] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Proving Termination of Programs Automatically with AProVE. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Proceedings of the 7th International Joint Conference on Automated Reasoning (IJCAR'14), Vienna, Austria*, volume 8562 of *Lecture Notes in Computer Science*, pages 184–191. Springer, July 2014.

[20] Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems*. The MIT Press, 2014.

[21] Adele E. Howe and Eric Dahlman. A Critical Assessment of Benchmark Comparison in Planning. *Journal of Artificial Intelligence Research*, 17:1–33, 2002.

[22] ISO/IEC. LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization – Information Processing Systems – Open Systems Interconnection, Geneva, September 1989.

[23] Jan Martin Jansen, Pieter W. M. Koopman, and Rinus Plasmeijer. From Interpretation to Compilation. In Zoltán Horváth, Rinus Plasmeijer, Anna Soós, and Viktória Zsók, editors, *Revised Selected Lectures of the 2nd Central European Functional Programming School, Cluj-Napoca, Romania*, volume 5161 of *Lecture Notes in Computer Science*, pages 286–301. Springer, June 2007.

[24] Cynthia Kop, Aart Middeldorp, and Thomas Sternagel. Complexity of Conditional Term Rewriting. *Logical Methods in Computer Science*, 13(1), 2017.

[25] Fabrice Kordon and Francis Hulin-Hubard. BenchKit, a Tool for Massive Concurrent Benchmarking. In *Proceedings of the 4th International Conference on Application of Concurrency to System Design (ACSD'14), Tunis La Marsa, Tunisia*, pages 159–165. IEEE Computer Society, June 2014.

[26] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml System Release 4.04 – Documentation and User's Manual*. INRIA, Paris, France, March 2016.

[27] Allen Leung. *Nowhere: A Pattern Matching Tool for Standard ML – Version 1.1 Manual*. New York University, NY, USA, December 2000.

[28] Simon Marlow, editor. *Haskell 2010 Language Report*, April 2010.

[29] Radu Mateescu and Hubert Garavel. XTL: A Meta-Language and Tool for Temporal Logic Model-Checking. In Tiziana Margaria, editor, *Proceedings of the International Workshop on Software Tools for Technology Transfer (STTT'98), Aalborg, Denmark*, pages 33–42. BRICS, July 1998.

[30] Franco Mazzanti and Alessio Ferrari. Ten Diverse Formal Models for a CBTC Automatic Train Supervision System. In John P. Gallagher, Rob van Glabbeek, and Wendelin Serwe, editors, *Proceedings of the 3rd Workshop on Models for Formal Analysis of Real Systems and the 6th International Workshop on Verification and Program Transformation (MARS/VPT'18), Thessaloniki, Greece*, volume 268 of *Electronic Proceedings in Theoretical Computer Science*, pages 104–149, April 2018.

[31] Aart Middeldorp and Erik Hamoen. Completeness Results for Basic Narrowing. *Applicable Algebra in Engineering, Communication and Computing*, 5:213–253, 1994.

[32] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *Definition of Standard ML (Revised)*. MIT Press, May 1997.

[33] Martin Odersky, Philippe Altherr, Vincent Cremet, Gilles Dubochet, Burak Emir, Philipp Haller, Stéphane Micheloud, Nikolay Mihaylov, Adriaan Moors, Lukas Rytz, Michel Schinz, Erik Stenman, and Matthias Zenger. *The Scala Language Specification – Version 2.11*. Programming Methods Laboratory, EPFL, Switzerland, March 2016.

[34] Peter Pepper and Florian Lorenzen, editors. *The Programming Language Opal – 6th Corrected Edition*. Department of Software Engineering and Theoretical Computer Science, Technische Universität Berlin, Germany, October 2012.

[35] Rinus Plasmeijer, Marko van Eekelen, and John van Groningen. *Clean Version 2.2 Language Report*. Department of Software Technology, University of Nijmegen, The Netherlands, December 2011.

[36] Olivier Ponsini, Caroline Fédèle, and Emmanuel Kounalis. Rewriting of Imperative Programs into Logical Equations. *Science of Computer Programming*, 56(3):363–401, May–June 2005.

[37] Philippe Schnoebelen. Refined Compilation of Pattern-Matching for Functional Languages. *Science of Computer Programming*, 11:133–159, 1988.

[38] Ryan Stansifer. Imperative versus Functional. *SIGPLAN Notices*, 25(4):69–72, 1990.

[39] Simon J. Thompson. Laws in Miranda. In William L. Scherlis, John H. Williams, and Richard P. Gabriel, editors, *Proceedings of the ACM Conference on LISP and Functional Programming (LFP'86), Cambridge, Massachusetts, USA*, pages 1–12, 1986.

[40] Jaco van de Pol. Just-in-time: On Strategy Annotations. *Electronic Notes in Theoretical Computer Science*, 57:41–63, 2001.

[41] Jaco van de Pol. JITty: A Rewriter with Strategy Annotations. In Sophie Tison, editor, *Proceedings of the 13th International Conference on Rewriting Techniques and Applications (RTA'02), Copenhagen, Denmark*, volume 2378 of *Lecture Notes in Computer Science*, pages 367–370. Springer, July 2002.

[42] Jeroen van den Bos, Mark Hills, Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. Rascal: From Algebraic Specification to Meta-Programming. In Francisco Durán and Vlad Rusu, editors, *Proceedings of the 2nd International Workshop on Algebraic Methods in Model-based Software Engineering (AMMSE'11), Zurich, Switzerland*, volume 56 of *Electronic Proceedings in Theoretical Computer Science*, pages 15–32, June 2011.

[43] Mark van den Brand, Paul Klint, and Pieter A. Olivier. Compilation and Memory Management for ASF+SDF. In Stefan Jähnichen, editor, *Proceedings of the 8th International Conference on Compiler Construction (CC'99), Amsterdam, The Netherlands*, volume 1575 of *Lecture Notes in Computer Science*, pages 198–213. Springer, 1999.

[44] Muck van Weerdenburg. An Account of Implementing Applicative Term Rewriting. *Electronic Notes in Theoretical Computer Science*, 174(10):139–155, 2007.