Benchmarking Implementations of Term Rewriting and Pattern Matching in Algebraic, Functional, and Object-Oriented Languages

— The 4th Rewrite Engines Competition —

Hubert Garavel

joint work with

Imad Arrada and **Mohammad-Ali Tabikh**

Inria Grenoble – LIG
Université Grenoble Alpes
http://convecs.inria.fr

April 2018

1. Motivation

The CAESAR.ADT compiler (1/2)

- CAESAR.ADT: a compiler for LOTOS data types
  - designed for model checking purpose
  - implements data structures very compactly
  - compiles pattern matching [Schnoebelen-88]
  - boostrapped (written itself in LOTOS data types)

- A heavily used compiler:
  - designed in 1989-1992
  - used every day since then
  - only 2 publications [Garavel-89-c] [Garavel-Turlier-93]

The CAESAR.ADT compiler (2/2)

- 2007: performance study [van-Weerdenburg-07]
  - reports average performance results for CAESAR.ADT
  - but measured on few experiments only

- Questions:
  - how does CAESAR.ADT compare with other tools?
  - do we maintain it? do we replace it?

The LNT2LOTOS translator

- Most LOTOS users complained about data types
- LNT: a more "user-friendly" language
  - imperative syntax: assignments, **return**, **if-then-else**, **case** with pattern-matching, **while** and **for** loops with **break**, exceptions, etc.
  - functional-language semantics (first-order only)
- LNT2LOTOS [Garavel-Lang-Serwe-17]
  - translator from LNT to LOTOS data types (+ some C)
  - LNT (imperative) -> LOTOS (algebraic) -> C (imperative)
  - is this "crazy" translation scheme efficient enough?

More generally...

- There are many tools for term rewriting:
  - Maude, Elan, Tom, etc.
- Many languages implement pattern-matching on algebraic terms:
  - functional languages: SML, OCaml, Haskell, etc.
  - object-oriented languages: Scala, Rust, etc.
- Are these implementations efficient?
  - how to compare them? (CPU time, memory)
  - which are the best algorithms?

Initial questions

- 2015: we undertook a systematic comparison
- Which are the right tools against which CAESAR.ADT and LNT2LOTOS should be compared?
- Where are the term-rewrite specifications to be used as benchmarks?

2. Former competitions for rewrite engines

Former Rewrite Engines Competitions

- 2006: 1st REC competition [Roşu-06]
  tools: ASF+SDF, Elan, Maude
- *2007: [van-Weerdenburg-07]*
  tools: ASF+SDF, Clean, Haskell, LOTOS (CADP), Maude, μCRL, mCRL2 (innermost and jitty rewriters)
- 2008: 2nd REC competition [Durán-et-al-09]
  tools: ASF+SDF, Maude, Stratego, Termware, Tom
- 2010: 3rd REC competition [Durán-et-al-10]
  tools: ASF+SDF, Maude, Stratego/XT, Tom, TXL

Tool selection

- Retained: Haskell, LOTOS, Maude, mCRL2, Tom
- Excluded: Termware (performed poorly), TXL (discouraged by its author)
- Upgraded:
  - mCRL2 replaces μCRL
  - Rascal replaces ASF+SDF
  - Stratego/XT 2.0 replaces Stratego/XT 1.0
  - Tom replaces ELAN
- Included: CafeOBJ, Clean, LNT, OCaml, Opal, Scala, SML/NJ, SML/MLTON

15-18 tools under assessment

- CafeOBJ
  JAIST (JP)
- Clean
  Raboud Univ. (Nijmegen, NL)
- Haskell (GHG compiler)
  Univ. Glasgow (UK)
- LNT (CADP tools)
  INRIA Grenoble (FR)
- LOTOS (CADP tools)
  INRIA Grenoble (FR)
- Maude
  SRI (California, US)
- mCRL2 (jitty and jittyc rewriters)
  Tech. Univ. Eindhoven (NL)
- OCaml (interpreted or compiled)
  INRIA Rocquencourt (FR)
- Opal
  Tech. Univ. Berlin (DE)
- Rascal (interpreted or compiled)
  CWI Amsterdam (NL)
- Scala
  EPFL Lausanne (CH)
- SML/NJ (+ Nowhere preprocessor)
  Univ. Princeton (New Jersey, US)
- SML/MLTON (+ Nowhere preprocessor)
  NEC Res. Labs (New Jersey, US)
- Stratego/XT
  Univ. Delft (NL)
- Tom
  LORIA / INRIA Nancy (FR)

3. The REC format

The REC-2008 format

- Introduced during the 2nd REC competition (2008)
  - description of conditional term rewrite systems
  - tool-independent format
  - human-readable, text-based format

- Lack of dedicated tools for supporting REC
  - no parser, no type checker
  - 3 tools could read REC files: Maude, Stratego, Tom

The REC-2017 format

- Derived from the REC-2008 format
- Main changes:
  - line-based format (to be handled by Unix scripts)
  - distinction between (free) constructors and non-constructors (separate "CONS" and "OPNS" sections)
  - introduction of an "EVAL" section that replaces the directives "**get normal form of**"
  - introduction of C-like "**#include**" directives
  - elicitation of static semantics constraints
  - elicitation of dynamic semantics constraints

Example of a REC-2017 specification

```
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
```

Syntax of the REC-2017 format

```
⟨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
```

```
⟨cons-decl⟩ ::= ⟨cons-id⟩ : ⟨sort-id⟩*[,] -> ⟨sort-id⟩
 ⟨opn-decl⟩ ::= ⟨opn-id⟩ : ⟨sort-id⟩*[,] -> ⟨sort-id⟩
 ⟨var-decl⟩ ::= ⟨var-id⟩*[ ] : ⟨sort-id⟩
     ⟨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⟩
```

- no notations for numbers
- no infix operators (+, *, mod)

Static semantics

- Strong typing with basic features only:
  - no overloading of functions
  - no implicit type conversions
- Free-constructor discipline:
  - no equations between constructors
- Simplifying constraints:
  - constructors of the same type must be grouped
  - rewrite rules defining the same non-constructor must be grouped

https://gforge.inria.fr/scm/viewvc.php/rec/2015-CONVECS/doc/rec-2017-language.txt?view=log

Dynamic semantics (1/2)

- The target tools have different rewrite strategies:
  - OCaml: strict evaluation
  - Haskell: lazy evaluation
  - Maude: associative/commutative rewriting
  - mCRL2: just-in-time rewriting
  - CAESAR.ADT: decreasing priority between equations
  - Stratego/XT: user-defined rewrite strategies
- On the same REC benchmark:
  - different tools may give different results
  - some tools may terminate or not

Dynamic semantics (2/2)

- This issue was already there in earlier REC competitions ($\Rightarrow$ different categories of benchmarks)

- Chosen approach:
  - require all REC benchmarks to be confluent and terminating
  - thus, all rewrite strategies produce the same result
  - perform rewriting on closed terms only (EVAL section)
  - partially-defined functions are tolerated, but should only be invoked where they are defined

4. The REC translators

Manual vs automatic translation

- In the three original REC competitions:
  - a few tools could read the REC-2008 format natively
  - for the other tools, the REC-2008 benchmarks were manually translated to the input language of each tool
  - this was tedious, error-prone, and possibly biased
- For our study, manual translation would not scale
  - more than 1500 files to maintain
  - numerous and frequent modifications

$\Rightarrow$ automatic translation was the only feasible option

The REC-2017 translators (1/2)

- Development of a "serious" compiler for REC-2017
  - lack of time / lack of resources / lack of interest

- A lightweight approach was preferred:
  - exploiting the REC-2017 syntax (sections, lines)
  - translators = collection of Unix scripts
  - acrobatic combination of shell, cpp, grep, sed, awk all connected by data flows using Unix pipes
  - only 250 lines of code per translator!
  - a bit slow for large REC files (e.g., MAA)

The REC-2017 translators (2/2)

- Syntactic and static semantics checks:
  - no checks before translation (i.e., on REC-2017 source files)
  - all checks after translation: a REC-2017 file is reputed to be correct if its 17 translations are accepted by all the target tools
- Confluence:
  - checked by the Opal compiler
  - sufficient conditions ("deterministic" rules)
- Termination:
  - design of a translator from REC-2017 to AProVE
  - AProVE often proves quasi-decreasingness, but may also loop forever (e.g., integer division with premises)

Differences between translators (1/2)

- Translators differ in 13 points:
  - (a) Are constructors and non-constructors handled identically (noted "I") or not (noted "D")?
  - (b) Are constructors declared together with their result type ("T") or separately ("S")?
  - (c) Are equality/inequality functions defined automatically ("E")?
  - (d) Are printing functions defined automatically ("P")?
  - (e) Are rewrite rules encapsulated within the non-constructor they define ("F") or separately ("S")?

Differences between translators (2/2)

- (f) Should a type identifier always start with an upper-case (noted "U") or a lower-case letter (noted "L")?
- (g), (h), (i) Same question for constructors, non-constructors, and free variables
- (j) Should a constructor F with arity 0 be invoked as "F" or "F ()"?
- (k) Same question for a non-constructor
- (l) Should a constructor F with arity > 0 be invoked as "F x y ... " (noted "J") or "F (x, y, ...)" (noted "A")?
- (m) Same question for a non-constructor

Overview of the 13 differences

language	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)	(k)	(l)	(m)
CafeOBJ	D	S	E	P	S	–	–	–	–	–	–	A	A
Clean	D	T	–	–	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	–	–	–	–	–	–	A	A
LOTOS	D	S	–	P	S	–	–	–	–	–	–	A	A
Maude	I	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	E	P	S	–	–	–	–	()	()	A	A
Tom–B	D	T	E	P	F	–	–	–	–	()	()	A	A

Translation of REC-2017 terms

- For all languages but one: line-based translations using regular expressions are enough
- For OCaml only: an ad-hoc C program counting commas and nested parentheses was written
  - this is due to OCaml's "irregular" syntax

constructor	arity	REC-2017 term	OCaml expression
no	0	f	f
yes	0	C	C
no	1	f (e)	(f e)
yes	1	C (e)	(C e)
no	>1	f (e1, e2)	(f e1 e2)
yes	>1	C (e1, e2)	(C (e1, e2))

Multiple translations

- Some languages/tools call for multiple translations

- 2 translators for CafeOBJ
  - CafeOBJ-A: uses eq, ceq, red
  - CafeOBJ-B: uses trans, ctrans, exec

- 2 translators for TOM
  - TOM-A: no distinction between constructors and non-constructors
  - TOM-B: distinction between constructors and non-constructors; uses %match

Example: source REC-2017 code

```
REC-SPEC Factorial5
SORTS
  Nat
CONS
  d0 : -> Nat                              % zero
  s : Nat -> Nat                           % successor
OPNS
  plus : Nat Nat -> Nat                    % addition
  times : Nat Nat -> Nat                   % product
  fact : Nat -> Nat                        % factorial
VARS
  N M : Nat
RULES
  plus (d0, N) -> N
  plus (s (N), M) -> s (plus (N, M))
  times (d0, N) -> d0
  times (s (N), M) -> plus (M, times (N, M))
  fact (d0) -> s (d0)
  fact (s (N)) -> times (s (N), fact (N))
EVAL
   fact (s (s (s (s (s (d0))))))       % fact (5)
END-SPEC
```

Example: (1) generated Maude code

```
fmod Factorial5 is
  sorts Nat .
  op d0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op plus : Nat Nat -> Nat .
  op times : Nat Nat -> Nat .
  op fact : Nat -> Nat .
  eq plus (d0, N:Nat) = N:Nat .
  eq plus (s (N:Nat), M:Nat) = s (plus (N:Nat, M:Nat)) .
  eq times (d0, N:Nat) = d0 .
  eq times (s (N:Nat), M:Nat) = plus (M:Nat, times (N:Nat, M:Nat)) .
  eq fact (d0) = s (d0) .
  eq fact (s (N:Nat)) = times (s (N:Nat), fact (N:Nat)) .
endfm
reduce fact (s (s (s (s (s (d0)))))) .
quit
```

Example: (2) generated Haskell code

```haskell
data Nat = D0 | S Nat
  deriving (Show, Eq, Ord)

plus :: Nat -> Nat -> Nat
times :: Nat -> Nat -> Nat
fact :: Nat -> Nat

plus D0 n = n
plus (S n) m = (S (plus n m))

times D0 n = D0
times (S n) m = (plus m (times n m))

fact D0 = (S D0)
fact (S n) = (times (S n) (fact n))

main = do
 print (fact (S (S (S (S (S D0))))))
```

Example: (3) generated LOTOS code

```
specification FACTORIAL5 [PRINT] : noexit
library BOOLEAN endlib
type LOTOS_TYPE is BOOLEAN
sorts
  Nat
opns
  _==_, _/=_ : Nat, Nat -> BOOL
  d0 (*! constructor *) : -> Nat
  s (*! constructor *) : Nat -> Nat
  plus : Nat, Nat -> Nat
  times : Nat, Nat -> Nat
  fact : Nat -> Nat
eqns
forall
  N, M : Nat ,
  REC_Nat_X, REC_Nat_Y : Nat
ofsort BOOL
  REC_Nat_X == REC_Nat_X = TRUE;
  (* otherwise *) REC_Nat_X == REC_Nat_Y = FALSE;
ofsort BOOL
  REC_Nat_X /= REC_Nat_X = FALSE;
  (* otherwise *) REC_Nat_X /= REC_Nat_Y = TRUE;
ofsort Nat
  plus (d0, N) = N;
  plus (s (N), M) = s (plus (N, M));
  times (d0, N) = d0;
  times (s (N), M) = plus (M, times (N, M));
  fact (d0) = s (d0);
  fact (s (N)) = times (s (N), fact (N));
endtype
behaviour
  PRINT !fact (s (s (s (s (s (d0))))));
  stop
endspec
```

Example: (4) generated LNT code

```
module Factorial5 is

type Xnat is
   d0,
   s (rec_x1_1:Xnat)
with "==", "!="
end type

function plus (rec_x1:Xnat, rec_x2:Xnat) : Xnat is
   case rec_x1, rec_x2 in
   var M:Xnat, N:Xnat in
     d0, N -> return N
   | s (N), M -> return s (plus (N, M))
   end case
end function

function times (rec_x1:Xnat, rec_x2:Xnat) : Xnat is
   case rec_x1, rec_x2 in
   var M:Xnat, N:Xnat in
     d0, N -> return d0
   | s (N), M -> return plus (M, times (N, M))
   end case
end function

function fact (rec_x1:Xnat) : Xnat is
   case rec_x1 in
   var N:Xnat in
     d0 -> return s (d0)
   | s (N) -> return times (s (N), fact (N))
   end case
end function

process MAIN [PRINT:any] is
   PRINT (fact (s (s (s (s (s (d0)))))));
   stop
end process

end module
```

Example: (5) generated Scala code

```
sealed abstract class Nat
  case class D0() extends Nat
  case class S (rec_x1:Nat) extends Nat

object factorial5 {

def plus (rec_x1:Nat, rec_x2:Nat):Nat =
  (rec_x1, rec_x2) match {
    case (D0(), n) => n
    case (S (n), m) => S (plus (n, m))
  }

def times (rec_x1:Nat, rec_x2:Nat):Nat =
  (rec_x1, rec_x2) match {
    case (D0(), n) => D0()
    case (S (n), m) => plus (m, times (n, m))
  }

def fact (rec_x1:Nat):Nat =
  rec_x1 match {
    case D0() => S (D0())
    case S (n) => times (S (n), fact (n))
  }

def main (args:Array[String]):Unit = {
  println (fact (S (S (S (S (S (D0())))))))
  }

}
```

5. The REC benchmarks

The 3rd REC benchmarks (2010)

- Group 1: TRS (unconditional term rewrite systems)
  - 5 models, 25 instances
- Group 2: CTRS (conditional term rewrite systems)
  - 5 models, 17 instances
- Group 3: MODULO (associativity/commutativity)
  - 4 models, 6 instances
  - only Maude supports rewriting modulo AC
- Group 4: CS (context sensitive)
  - 1 model, 3 instances
  - non-functional evaluation: rewrite on open terms

Collecting benchmarks (1/2)

- Gather former REC benchmarks:
  - REC 2008 and 2010 benchmarks
  - merge TRS and CTRS into a single class
- Look for other models available:
  - personal archives of Pierre-Etienne Moreau
  - examples from Muck van Weerdenburg
- Identify multiple/derived versions of the same model
- Turn all models into the REC-2017 format:
  - identification of constructors
  - separation between constructors and non-constructors
  - modification of models dealing with open terms

Collecting benchmarks (2/2)

- Handle parametric models
  - introduce shared code libraries for parametric models (REC-LIB package and C-like "**#include**" directives)
- Ensure correctness
  - check correctness by translation to target languages
  - check confluence using Opal
  - check termination using AProVE (when possible)
  - correction of mistakes
  - save those incorrect models that could not be repaired in a special package named REC-BAD

Adding new benchmarks

- Complexify models that were too simple:
  - langton*
- Introduction of new benchmarks:
  - tak*: Takeuchi function
  - intnat: signed integers
  - add*: binary adders on 8, 16, 32 bits
  - mul*, omul*: binary multipliers on 8, 16, 32 bits
  - maa: Message Authenticator Algorithm
    (13 sorts, 18 construct., 646 non-construct., 684 rules)

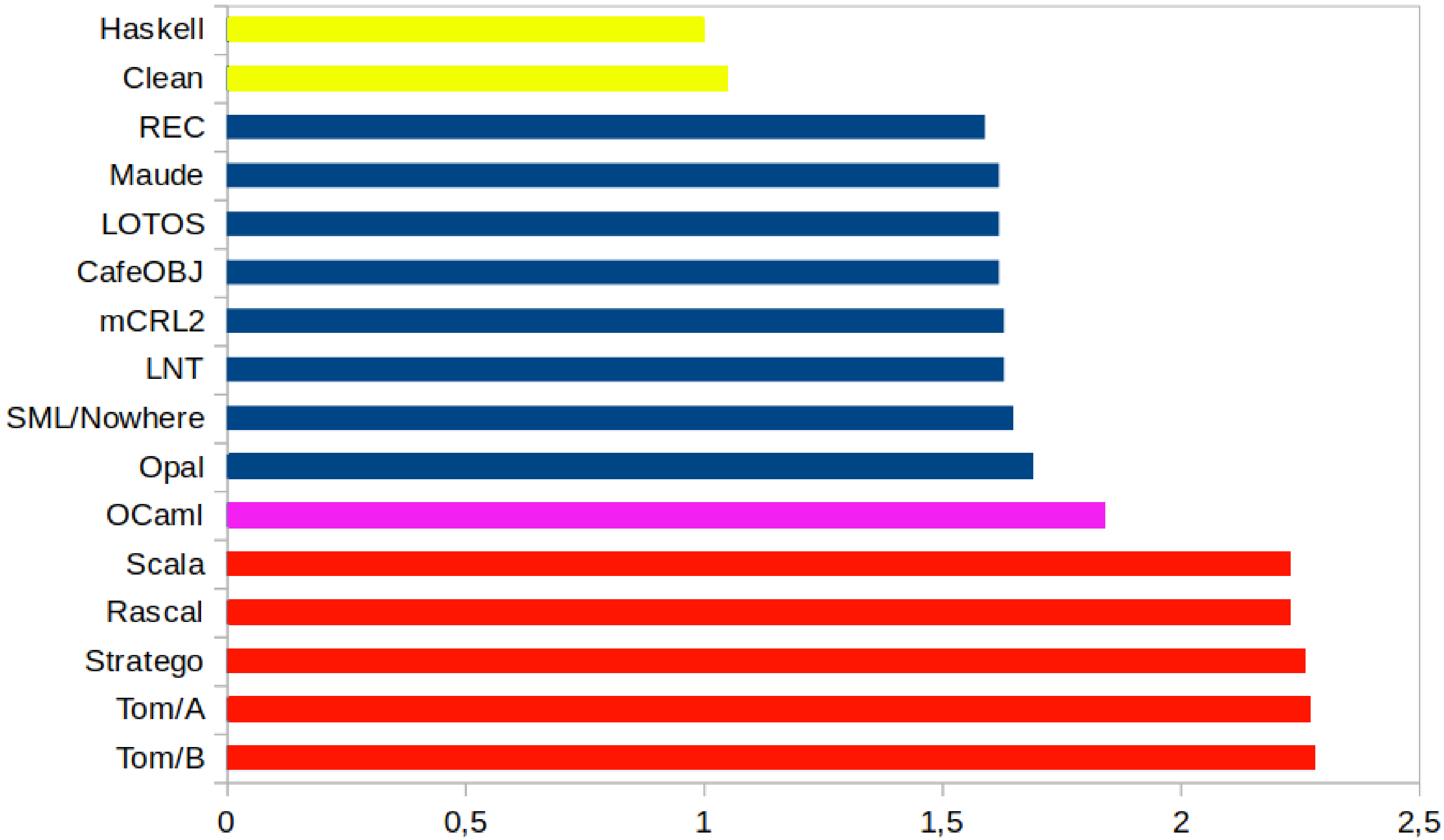
The resulting collection

- 85 benchmarks in REC language
  - 48,000+ lines of REC code

- Divided into two packages:
  - REC-SIMPLE (15 benchmarks) :
    "easy" examples
    all tools can process them in 2 minutes at most
  - REC (70 benchmarks):
    "difficult" examples
    all tools have been assessed on these benchmarks

Measuring language conciseness

- counted in lexical tokens (keywords, identifiers, symbols)
- the base is Haskell: 1.0 means 5,754,474 lexical tokens



6. The benchmark execution platform

Hardware/software platform

- Requirements for reproducibility:
  - single-user mode
  - local file system (no NAS, NFS, SAMBA, etc.)
  - standalone (no remote admin. by computer staff)
- Reuse of old workstations
  - 32 bits: Sun Ultra 20 M2 (2007)
    AMD Opteron 1210 dual core 1.8 GHz, 2 GB RAM
  - 64 bits: Transtec 2500L (2004)
    2 x AMD Opteron 246 2.0 GHz, 16 GB RAM
- Common operating system: Debian Linux 8

Collecting tool execution statistics

- Use of the **memtime 1.4** utility
  - originally developed for Uppaal (in 2002)
  - later enhanced at INRIA Grenoble
  - see http://cadp.inria.fr/resources
- Usage: **memtime** *COMMAND* ...

  Exit [0]

  0.68 user, 2.07 system, 110.51 elapsed -- Max VSize = 15572KB, Max RSS = 1916KB
- Limitation: only time results have been used
  - memory results are not meaningful, as **memtime** only measures the memory consumption of the main process, ignoring all the sub-processes launched by this process

Imposing timeouts on tool execution

- Termination issues:
  - only a few tools terminate properly: Haskell, LNT, LOTOS, Opal (because they have exhausted all available memory)
  - most other tools seem to compute forever
  - upper time limits and interrupts are needed
- Use of the Linux **timeout** utility
- Usage: **timeout** 360 *COMMAND* ...

the execution of COMMAND will be halted after 360 seconds

- Problem: some tools manipulate POSIX signals
  - they protect themselves from timeout interrupts
  - mCRL2 (bug fixed), Scala (bug reported)

Combining memtime and timeout

- Wrong combination:

  timeout 360 **memtime** *COMMAND* ...

  - if timeout occurs, no statistics are displayed
- Correct combination:

  memtime timeout 360 *COMMAND* ...
- Execution can terminate in 4 different ways:
  - SUCCESS: normal completion (exit code is zero)
  - FAILURE: failed execution (exit code is non zero)
  - CRASH: abnormal termination by a signal (SIGSEGV, SIGBUS...)
  - TIMEOUT: interruption after timeout expired

Compilers vs interpreters

- For compiled languages, we distinguish between:
  - COMP: compilation phase of source file to binary code
  - EXEC: execution run of binary code
- For interpreted languages:
  - TOTAL: total time for processing the source files
- In both cases, we measure full-problem solving
  - for compiled languages: COMP + EXEC
  - for interpreted languages: TOTAL

Benchmark execution

- Chosen timeout values:
  - REC-SIMPLE package: 120 seconds (all tools succeed)
  - REC package: 360 seconds
    this value was chosen so that executing all tools on all benchmarks takes approximately one day

- A fully automated execution platform:
  - scripts for running tools on benchmarks
  - production of spreadsheet files (in CSV format)
  - scripts for producing execution statistics

7. Defining a meaningful score metric

Dilemmas

- How to compare:
  1. a tool that could solve the problem after a long time but is halted by timeout
  2. another tool that immediately stops, declaring that it cannot solve the problem?
  - both tools have failed
  - but the former has taken more time than the latter!
- More generally, how to combine:
  - success or failure to solve the problem
  - CPU time taken
  - presence or absence of timeout?

A non-trivial problem

"Any comparison, competitions especially, has the unenviable task of determining how to trade-off or combine the three metrics (number [of problems] solved, time, and number of steps)."

Adele E. Howe et Eric Dahlman. *A Critical Assessment of Benchmark Comparison in Planning*. Journal of Artificial Intelligence Research 17 (2002), 1-33.

Chosen metric

- We adopted the standard solution mentioned by [Howe-Dahlman-02]:

  "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."

8. Experimental results

Top-5 podium (April 2018)

5 tools (out of 21) solve >85% of the benchmarks:

- GHC / Haskell (1st)
- Maude (2nd)
- OCaml (3rd [compil.] and 6th [interp.])
- CADP / LOTOS+LNT (4th [LOTOS] and 5th [LNT])
- Tom (7th)

This ranking is identical on 32- and 64-bit platforms

32-bit results

tool	score	successes	failures	crashes	timeouts	time
haskell	100%	70	0	0	0	1867.17
maude	91.4%	64	2	1	3	2095.07
ocaml-compil	91.4%	64	0	0	6	2771.8
lotos	88.6%	62	8	0	0	989.5
lnt	88.6%	62	8	0	0	1134.71
ocaml-interp	85.7%	60	2	0	8	3937.49
tom-b	85.7%	60	3	0	7	5437.96
sml-mlton	82.9%	58	5	0	7	4621.88
opal	81.4%	57	1	2	10	4759.27
clean-hack	77.1%	54	10	0	6	2622.99
sml-smlnj	74.3%	52	5	0	13	5672.29
tom-a	72.9%	51	3	0	16	7578.02
scala	70.0%	49	10	0	11	6091.82
mcrl2-jittyc	68.6%	48	0	4	18	8148.28
stratego	68.6%	48	3	0	19	8260.1
mcrl2-jitty	62.9%	44	0	6	20	9585.8
cafeobj-b	54.3%	38	15	0	17	8561.05
rascal-interp	52.9%	37	2	0	31	12274.4
rascal-compil	48.6%	34	4	0	32	14486.2
cafeobj-a	44.3%	31	8	0	31	12489.7
clean	42.9%	30	30	8	2	805.48

https://gforge.inria.fr/scm/viewvc.php/rec/2015-CONVECS/results-rec/2018-04-07-overview-360-32.csv?view=log
https://gforge.inria.fr/scm/viewvc.php/rec/2015-CONVECS/results-rec/raw-v2/2018-04-07-rec360-32.csv?view=log

64-bit results

tool	score	successes	failures	crashes	timeouts	time
haskell	97.1%	68	0	0	2	2091.29
maude	95.7%	67	0	0	3	2122.24
ocaml-compil	91.4%	64	0	0	6	2718.39
lnt	90.0%	63	7	0	0	3028.2
lotos	90.0%	63	7	0	0	2883.85
ocaml-interp	87.1%	61	2	0	7	3855.15
tom-b	85.7%	60	1	0	9	5887.2
opal	84.3%	59	0	1	10	4816.77
sml-mlton	82.9%	58	5	0	7	4996.88
clean-hack	77.1%	54	10	0	6	2697.1
mcrl2-jittyc	74.3%	52	0	0	18	8100.86
sml-smlnj	74.3%	52	5	0	13	5627.93
tom-a	74.3%	52	1	0	17	8064.43
scala	70.0%	49	7	0	14	7147.14
stratego	70.0%	49	0	0	21	9061.54
mcrl2-jitty	65.7%	46	0	2	22	9913.92
clean	57.1%	40	22	1	7	2631.56
rascal-interp	57.1%	40	0	0	30	12322.4
cafeobj-b	54.3%	38	8	3	21	10170.1
rascal-compil	52.9%	37	2	0	31	14286.1
cafeobj-a	44.3%	31	4	4	31	13945.5

https://gforge.inria.fr/scm/viewvc.php/rec/2015-CONVECS/results-rec/2018-04-05-overview-360-64.csv?view=log
https://gforge.inria.fr/scm/viewvc.php/rec/2015-CONVECS/results-rec/raw-v2/2018-04-05-rec360-64.csv?view=log

9. Conclusion

Lessons learnt

- Focus on the most widely used part of term rewriting:
  - conditional term rewrite systems
  - free constructors
  - confluence
  - termination
- A clear vision of the common features between:
  - term rewrite systems
  - algebraic (abstract data types) languages
  - functional languages
  - (modern) object-oriented languages

Contributions

- A software platform for term rewriting:
  - REC-2017 format
  - 85 benchmarks in this format
  - translators for 16-18 languages
  - scripts for assessing the tools on these benchmarks

  http://gforge.inria.fr/scm/viewvc.php/rec/2015-CONVECS

- Already used in two case studies:
  - elegant definition of signed integers [Garavel-17]
  - specification of the MAA cryptographic function

    [Garavel-Marsso-17]

Future work (personal)

- CAESAR.ADT exhibits honourable performance
  - [van-Weerdenburg-07] results are not confirmed
- Study why CAESAR.ADT seems slower on 64 bit
- Benchmark CAESAR.ADT with its garbage collector
- Benchmark TRAIAN 3.0 (forthcoming compiler LNT$\rightarrow$ C) when it is available
- Understand what GHC (Haskell) is doing

Future work (collective)

- Improve the individual tools by cross-examination
- Restart the Rewrite Engine Competition?
  - include new languages/tools (e.g., Clojure, Erlang, Prolog, Racket, Rust, Scheme)
  - collect more REC benchmarks
  - try different machines (e.g., with Intel processors)
  - better check tool outputs (so far, we trust their results)
  - better distinguish between COMP and EXEC phases
  - measure memory consumption (ad hoc infrastructure)
  - finely tune optimizations (e.g., Java VM options)

Acknowledgements

Co-workers:

- Imad Larrada (summer project)
- Mohammad-Ali Tabikh (master1 project)

Tool experts:

- 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)
- Eelco Visser (Delft/Stratego)

Discussions:

- Bertrand Jeannet (Grenoble)
- Fabrice Kordon (Paris)