

Recent developments in proving termination of rewriting automatically

Hans Zantema

Department of Computer Science, TU Eindhoven
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands
email: h.zantema@tue.nl

Prose, September 21, 2006

Term rewriting

Term rewriting is a natural and basic framework to describe computations

Example:

Natural numbers are terms composed from the constant 0 and the unary symbol s

Term rewriting

Binary operations $+$ and $*$ are defined on these natural numbers by the following rules

$$\begin{aligned}0 + x &\rightarrow x \\s(x) + y &\rightarrow s(x + y) \\0 * x &\rightarrow 0 \\s(x) * y &\rightarrow y + (x * y)\end{aligned}$$

Term rewriting

Binary operations $+$ and $*$ are defined on these natural numbers by the following rules

$$\begin{aligned}0 + x &\rightarrow x \\s(x) + y &\rightarrow s(x + y) \\0 * x &\rightarrow 0 \\s(x) * y &\rightarrow y + (x * y)\end{aligned}$$

Applying these rules as long as possible (*rewriting to normal form*) always yields the desired result:

$$\begin{aligned}s(\underbrace{s(0)}_x) * \underbrace{s(s(s(0)))}_y &\rightarrow s(s(s(0))) + (s(0) * s(s(s(0)))) \rightarrow \dots \\&\dots \rightarrow s(s(s(s(s(s(0))))))\end{aligned}$$

Why?

- All rules are *sound*: the meaning of the left hand side is equal to the meaning of the right hand side
- so by rewriting the meaning does not change

- All rules are *sound*: the meaning of the left hand side is equal to the meaning of the right hand side

so by rewriting the meaning does not change

- For every ground term containing $*$ or $+$ a rule is applicable

so if no rule is applicable then the term only consists of 0 and s

- All rules are *sound*: the meaning of the left hand side is equal to the meaning of the right hand side

so by rewriting the meaning does not change

- For every ground term containing $*$ or $+$ a rule is applicable

so if no rule is applicable then the term only consists of 0 and s

- No *infinite* computations are possible

This latter is called *termination*, and that's what we want to prove automatically

Proving termination

Several techniques have been developed for proving termination of rewriting, roughly divided into

Proving termination

Several techniques have been developed for proving termination of rewriting, roughly divided into

- *Semantical methods*: interpret in some well-founded domain (like natural numbers) in which value decreases at every rewrite step

Proving termination

Several techniques have been developed for proving termination of rewriting, roughly divided into

- *Semantical methods*: interpret in some well-founded domain (like natural numbers) in which value decreases at every rewrite step
- *Syntactical methods* like recursive path order RPO: if $\ell \succ_{RPO} r$ for all rules $\ell \rightarrow r$, then the system is terminating

Proving termination

Several techniques have been developed for proving termination of rewriting, roughly divided into

- *Semantical methods*: interpret in some well-founded domain (like natural numbers) in which value decreases at every rewrite step
- *Syntactical methods* like recursive path order RPO: if $\ell \succ_{RPO} r$ for all rules $\ell \rightarrow r$, then the system is terminating
- *Transformational methods*: transform rewrite system such that termination of transformed system implies termination of original system

Proving termination

Several techniques have been developed for proving termination of rewriting, roughly divided into

- *Semantical methods*: interpret in some well-founded domain (like natural numbers) in which value decreases at every rewrite step
- *Syntactical methods* like recursive path order RPO: if $\ell \succ_{RPO} r$ for all rules $\ell \rightarrow r$, then the system is terminating
- *Transformational methods*: transform rewrite system such that termination of transformed system implies termination of original system

Most powerful transformational method: *dependency pairs*
(Arts, Giesl, 2000)

The last years several tools have been developed for proving termination fully automatically

The last years several tools have been developed for proving termination fully automatically

Since 2003, every year there is a *competition* comparing the power of these tools by applying them on a database of rewrite systems fully automatically

The last years several tools have been developed for proving termination fully automatically

Since 2003, every year there is a *competition* comparing the power of these tools by applying them on a database of rewrite systems fully automatically

Characteristics of 2006 competition:

- 11 participating tools
- over 1000 rewrite systems
- subdivided in 8 categories
- time limit: 1 minute per tool per system (second round: 5 minutes)
- total running time: two weeks
- strongest tools: AProVE (Giesl et al, Aachen) and Jambox (Endrullis, VU Amsterdam), both written in JAVA

Strong improvements compared to 2005: for several rewrite systems termination proofs were given in 2006 where all tools in 2005 failed

Strong improvements compared to 2005: for several rewrite systems termination proofs were given in 2006 where all tools in 2005 failed

Main reasons for these improvements:

Strong improvements compared to 2005: for several rewrite systems termination proofs were given in 2006 where all tools in 2005 failed

Main reasons for these improvements:

- Encode search in large search spaces into propositional *SAT* satisfiability problem, and call state-of-the-art SAT solver to solve it

Strong improvements compared to 2005: for several rewrite systems termination proofs were given in 2006 where all tools in 2005 failed

Main reasons for these improvements:

- Encode search in large search spaces into propositional SATisfiability problem, and call state-of-the-art SAT solver to solve it
This approach was introduced independently in several tools for several search problems, while in 2005 no tool had it

Strong improvements compared to 2005: for several rewrite systems termination proofs were given in 2006 where all tools in 2005 failed

Main reasons for these improvements:

- Encode search in large search spaces into propositional SATisfiability problem, and call state-of-the-art SAT solver to solve it
This approach was introduced independently in several tools for several search problems, while in 2005 no tool had it
- The *matrix method* (Endrullis, Hofbauer, Waldmann, Zantema, 2006)

Termination of a rewrite system can be proved by *recursive path order* (RPO)

Termination of a rewrite system can be proved by *recursive path order* (RPO)

An order on operation symbols is called a *precedence*

Termination of a rewrite system can be proved by *recursive path order* (RPO)

An order on operation symbols is called a *precedence*

Definition (RPO)

Given a precedence \succ then $s \succ_{RPO} t$ iff

$s = f(s_1, \dots, s_n)$ and

(1) $s_i = t$ or $s_i \succ_{RPO} t$ for some $1 \leq i \leq n$, or

(2) $t = g(t_1, \dots, t_m)$, $s \succ_{RPO} t_i$ for all $1 \leq i \leq m$, and either

(a) $f \succ g$, or

(b) $f = g$ and $\langle s_1, \dots, s_n \rangle \succ_{RPO}^{\tau(f)} \langle t_1, \dots, t_m \rangle$

Termination of a rewrite system can be proved by *recursive path order* (RPO)

An order on operation symbols is called a *precedence*

Definition (RPO)

Given a precedence \succ then $s \succ_{RPO} t$ iff

$s = f(s_1, \dots, s_n)$ and

(1) $s_i = t$ or $s_i \succ_{RPO} t$ for some $1 \leq i \leq n$, or

(2) $t = g(t_1, \dots, t_m)$, $s \succ_{RPO} t_i$ for all $1 \leq i \leq m$, and either

(a) $f \succ g$, or

(b) $f = g$ and $\langle s_1, \dots, s_n \rangle \succ_{RPO}^{\tau(f)} \langle t_1, \dots, t_m \rangle$

Theorem (RPO)

If \succ is well-founded and $\ell \succ_{RPO} r$ for every rule $\ell \rightarrow r$ of a TRS R , then R is terminating

So for proving termination by RPO we have to choose

So for proving termination by RPO we have to choose

- an order \succ on the operation symbols, and

So for proving termination by RPO we have to choose

- an order \succ on the operation symbols, and
- a way $\tau(f)$ to compare sequences of terms, for every symbol f

So for proving termination by RPO we have to choose

- an order \succ on the operation symbols, and
- a way $\tau(f)$ to compare sequences of terms, for every symbol f

such that $\ell \succ_{RPO} r$ for every rule $\ell \rightarrow r$

So for proving termination by RPO we have to choose

- an order \succ on the operation symbols, and
- a way $\tau(f)$ to compare sequences of terms, for every symbol f

such that $\ell \succ_{RPO} r$ for every rule $\ell \rightarrow r$

Until recently this was done by direct search / backtracking

So for proving termination by RPO we have to choose

- an order \succ on the operation symbols, and
- a way $\tau(f)$ to compare sequences of terms, for every symbol f

such that $\ell \succ_{RPO} r$ for every rule $\ell \rightarrow r$

Until recently this was done by direct search / backtracking

Sometimes faster: introduce a boolean variable p_{fg} for every pair of symbols f, g , representing whether $f \succ g$, and express the requirements in a SAT problem on these variables

Polynomials

Find an interpretation by strictly monotone polynomials such that by every rewrite rule the interpretation strictly decreases

Polynomials

Find an interpretation by strictly monotone polynomials such that by every rewrite rule the interpretation strictly decreases

Example:

$$\begin{aligned}0 + x &\rightarrow x \\s(x) + y &\rightarrow s(x + y) \\0 * x &\rightarrow 0 \\s(x) * y &\rightarrow y + (x * y)\end{aligned}$$

Polynomials

Find an interpretation by strictly monotone polynomials such that by every rewrite rule the interpretation strictly decreases

Example:

$$\begin{aligned}0 + x &\rightarrow x \\s(x) + y &\rightarrow s(x + y) \\0 * x &\rightarrow 0 \\s(x) * y &\rightarrow y + (x * y)\end{aligned}$$

Interpretation $[\cdot]$ in positive integers:

$$[0] = 1, [s]x = x + 1, x[+]y = 2x + y, x[*]y = 3xy$$

Interpretation $[\cdot]$ in positive integers:

$$[0] = 1, [s]x = x + 1, x[+]y = 2x + y, x[*]y = 3xy$$

$$\begin{aligned} [0][+]x &= 2 + x > x \\ ([s]x)[+]y &= 2(x + 1) + y > 2x + y + 1 = [s](x[+]y) \\ [*]x &= 3x > 1 = [0] \\ ([s]x)[*]y &= 3(x + 1) * y > 2y + 3xy = y[+](x[*]y) \end{aligned}$$

for all $x, y > 0$, proving termination

Old approach for finding such interpretations: check requirements for all (or great number of randomly chosen) interpretations, typically

Old approach for finding such interpretations: check requirements for all (or great number of randomly chosen) interpretations, typically

- every constant is either 1, 2 or 8

Old approach for finding such interpretations: check requirements for all (or great number of randomly chosen) interpretations, typically

- every constant is either 1, 2 or 8
- every unary symbol is either identity, successor, or multiply by 2

Old approach for finding such interpretations: check requirements for all (or great number of randomly chosen) interpretations, typically

- every constant is either 1, 2 or 8
- every unary symbol is either identity, successor, or multiply by 2
- every binary symbol is either addition, multiplication, or \dots

Old approach for finding such interpretations: check requirements for all (or great number of randomly chosen) interpretations, typically

- every constant is either 1, 2 or 8
- every unary symbol is either identity, successor, or multiply by 2
- every binary symbol is either addition, multiplication, or \dots

Important: per symbol only a few options, otherwise the search space is intractable

New approach for finding such interpretations:

New approach for finding such interpretations:

- choose for constants: A
- choose for unary symbols $Ax + B$
- choose for binary symbols $Ax + By + C$
- ...

where A, B, C are unknown numbers represented in binary notation in n bits

New approach for finding such interpretations:

- choose for constants: A
- choose for unary symbols $Ax + B$
- choose for binary symbols $Ax + By + C$
- ...

where A, B, C are unknown numbers represented in binary notation in n bits

Transform requirements on these n -tuples of bits to a satisfiability problem on these bits

New approach for finding such interpretations:

- choose for constants: A
- choose for unary symbols $Ax + B$
- choose for binary symbols $Ax + By + C$
- ...

where A, B, C are unknown numbers represented in binary notation in n bits

Transform requirements on these n -tuples of bits to a satisfiability problem on these bits

Apply SAT solver on the result

New approach for finding such interpretations:

- choose for constants: A
- choose for unary symbols $Ax + B$
- choose for binary symbols $Ax + By + C$
- ...

where A, B, C are unknown numbers represented in binary notation in n bits

Transform requirements on these n -tuples of bits to a satisfiability problem on these bits

Apply SAT solver on the result

If successful: transform the resulting satisfying assignment back to desired values A, B, C, \dots

For this we need to be able to represent basic arithmetic:
 $+$, $*$, $>$, \dots in propositional logic

For this we need to be able to represent basic arithmetic:
 $+$, $*$, $>$, \dots in propositional logic

This can be done straightforwardly, e.g. $a + b = d$:

$$\begin{array}{rcccccc} c \rightarrow & 0 & 0 & 1 & 1 & 1 & 0 \\ a = 7 \rightarrow & & 0 & 0 & 1 & 1 & 1 \\ b = 21 \rightarrow & & 1 & 0 & 1 & 0 & 1 \\ \hline d = 28 \rightarrow & & 1 & 1 & 1 & 0 & 0 \end{array}$$

For this we need to be able to represent basic arithmetic:
 $+$, $*$, $>$, \dots in propositional logic

This can be done straightforwardly, e.g. $a + b = d$:

$$\begin{array}{rcccccc} c \rightarrow & 0 & 0 & 1 & 1 & 1 & 0 \\ a = 7 \rightarrow & & 0 & 0 & 1 & 1 & 1 \\ b = 21 \rightarrow & & 1 & 0 & 1 & 0 & 1 \\ \hline d = 28 \rightarrow & & 1 & 1 & 1 & 0 & 0 \end{array}$$

this is correct if and only if $a_i \leftrightarrow b_i \leftrightarrow c_i \leftrightarrow d_i$ and
 $c_{i-1} \leftrightarrow ((a_i \wedge b_i) \vee (a_i \wedge c_i) \vee (b_i \wedge c_i))$
for $i = 1, \dots, n$, and $\neg c_0 \wedge \neg c_n$

Multiplication can be expressed by repeated addition and duplication:

Multiplication can be expressed by repeated addition and duplication:

```
 $r := 0;$   
for  $i := 1$  to  $n$  do  
  begin  
     $s := 2 * r;$   
    if  $b_i$  then  $r := s + a$  else  $r := s$   
  end
```

Executing this program yields $r = a * b$

Multiplication can be expressed by repeated addition and duplication:

```
 $r := 0;$   
for  $i := 1$  to  $n$  do  
  begin  
     $s := 2 * r;$   
    if  $b_i$  then  $r := s + a$  else  $r := s$   
  end
```

Executing this program yields $r = a * b$

All ingredients are easily expressed in propositional logic, so by introducing several fresh variables for representing intermediate values for r, s multiplication can be expressed in propositional logic

Checking $>$ on binary numbers = lexicographic comparison:
easily expressed in propositional logic

Checking $>$ on binary numbers = lexicographic comparison:
easily expressed in propositional logic

So all ingredients of these polynomial interpretations transform
to propositional logic

Checking $>$ on binary numbers = lexicographic comparison:
easily expressed in propositional logic

So all ingredients of these polynomial interpretations transform
to propositional logic

Final formula is conjunction of several small formulas

Checking $>$ on binary numbers = lexicographic comparison:
easily expressed in propositional logic

So all ingredients of these polynomial interpretations transform
to propositional logic

Final formula is conjunction of several small formulas

Writing these small formulas in CNF yields big CNF, on which
SAT tools are directly applicable

The matrix method

Same idea as polynomials, except that now the interpretations are in *vectors* over natural numbers rather than in natural numbers

The matrix method

Same idea as polynomials, except that now the interpretations are in *vectors* over natural numbers rather than in natural numbers

Well-founded order:

$$(v_1, \dots, v_d) > (u_1, \dots, u_d) \iff v_1 > u_1 \wedge v_i \geq u_i \text{ for } i = 2, 3, \dots, d$$

The matrix method

Same idea as polynomials, except that now the interpretations are in *vectors* over natural numbers rather than in natural numbers

Well-founded order:

$$(v_1, \dots, v_d) > (u_1, \dots, u_d) \iff v_1 > u_1 \wedge v_i \geq u_i \text{ for } i = 2, 3, \dots, d$$

Interpretation for symbol of arity k :

$$[f](\vec{x}_1, \dots, \vec{x}_k) = A_1 \vec{x}_1 + \dots + A_k \vec{x}_k + \vec{v}$$

where A_1, \dots, A_k are $d \times d$ matrices and \vec{v} is a vector, all over n -bits integers

Typically $d \approx 3$ and $n \approx 3$

Typically $d \approx 3$ and $n \approx 3$

Search space is huge, but everything can be expressed in SAT
similarly: only arithmetic needed is $+$, $*$, $>$

Typically $d \approx 3$ and $n \approx 3$

Search space is huge, but everything can be expressed in SAT
similarly: only arithmetic needed is $+$, $*$, $>$

For small rewrite systems this easily yields formulas in 10,000
variables and over 100,000 clauses

Typically $d \approx 3$ and $n \approx 3$

Search space is huge, but everything can be expressed in SAT
similarly: only arithmetic needed is $+$, $*$, $>$

For small rewrite systems this easily yields formulas in 10,000
variables and over 100,000 clauses

For SAT solvers like ZChaff, Minisat, SatElite this is no problem

Example:

Example:

$$h(g(s(x), y), g(z, u)) \rightarrow h(g(u, s(z)), g(s(y), x))$$

Example:

$$h(g(s(x), y), g(z, u)) \rightarrow h(g(u, s(z)), g(s(y), x))$$

$$[h](\vec{x}_0, \vec{x}_1) = \begin{pmatrix} 3 & 1 \\ 1 & 0 \end{pmatrix} \cdot \vec{x}_0 + \begin{pmatrix} 1 & 3 \\ 0 & 1 \end{pmatrix} \cdot \vec{x}_1 + \begin{pmatrix} 0 \\ 2 \end{pmatrix}$$

$$[g](\vec{x}_0, \vec{x}_1) = \begin{pmatrix} 2 & 1 \\ 1 & 0 \end{pmatrix} \cdot \vec{x}_0 + \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix} \cdot \vec{x}_1$$

$$[s](\vec{x}_0) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \vec{x}_0 + \begin{pmatrix} 0 \\ 2 \end{pmatrix}$$

Conclusions

Conclusions

- SAT solving turns out to be helpful for proving termination of rewriting automatically, e.g. for
 - recursive path order
 - polynomial interpretations
 - matrix interpretations

Conclusions

- SAT solving turns out to be helpful for proving termination of rewriting automatically, e.g. for
 - recursive path order
 - polynomial interpretations
 - matrix interpretations

in particular where search space is intractable for exhaustive search

Conclusions

- SAT solving turns out to be helpful for proving termination of rewriting automatically, e.g. for
 - recursive path order
 - polynomial interpretations
 - matrix interpretations

in particular where search space is intractable for exhaustive search

- We gave a few examples of this, ignoring several crucial details like monotonicity requirements

Conclusions

- SAT solving turns out to be helpful for proving termination of rewriting automatically, e.g. for
 - recursive path order
 - polynomial interpretations
 - matrix interpretations

in particular where search space is intractable for exhaustive search

- We gave a few examples of this, ignoring several crucial details like monotonicity requirements
- For the full power of these techniques combination with earlier techniques like dependency pairs and relative termination is essential

Are we done in this area?

Are we done in this area?

$$\begin{aligned} f(t, x, y) &\rightarrow f(g(x, y), x, s(y)) \\ g(s(x), 0) &\rightarrow t \\ g(s(x), s(y)) &\rightarrow g(x, y) \end{aligned}$$

Are we done in this area?

$$\begin{aligned}f(t, x, y) &\rightarrow f(g(x, y), x, s(y)) \\g(s(x), 0) &\rightarrow t \\g(s(x), s(y)) &\rightarrow g(x, y)\end{aligned}$$

Here x, y are variables, g stands for *greater than* and t stands for *true*

- second and third rule are the standard rules for $>$ over the naturals composed from 0 and s (successor)
- The first rule describes the obviously terminating loop
while $x > y$ do $y := y + 1$

Are we done in this area?

$$\begin{aligned}f(t, x, y) &\rightarrow f(g(x, y), x, s(y)) \\g(s(x), 0) &\rightarrow t \\g(s(x), s(y)) &\rightarrow g(x, y)\end{aligned}$$

Here x, y are variables, g stands for *greater than* and t stands for *true*

- second and third rule are the standard rules for $>$ over the naturals composed from 0 and s (successor)
- The first rule describes the obviously terminating loop
while $x > y$ do $y := y + 1$

However, no tool can prove termination of this system