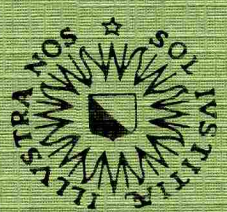
PARALLEL OBJECT-ORIENTED TERM REWRITING: THE BOOLEANS

P.H. Rodenburg
Department of Philosophy, Rijksuniversiteit te Utrecht

J.L.M. Vrancken
Programming Research Group, University of Amsterdam

Logic Group
Preprint Series
No. 38
July 1988



Department of Philosophy
University of Utrecht
Heidelberglaan 2
3584 CS Utrecht
The Netherlands

PARALLEL OBJECT-ORIENTED TERM REWRITING : THE BOOLEANS.

P.H. Rodenburg
Department of Philosophy, Rijksuniversiteit te Utrecht

J.L.M.Vrancken
Programming Research Group, University of Amsterdam,
Kruislaan 409, 1098 SJ Amsterdam.

Abstract: As a first case study in parallel object-oriented term rewriting, we give two implementations of term rewriting algorithms for boolean terms, using the parallel object-oriented features of the language Pool-T. The term rewriting systems are specified in the specification formalism ASF. Correctness proofs for the implementations are sketched.

1980 Mathematics Subject Classification (1985): 03B40, 68Q99.
1982 CR Categories: D.1.1, F.4.1.
Key Words and Phrases: Term Rewriting Systems, confluency, Object-oriented Programming, Parallelism.
Note: The investigation reported here was conducted as part of the PRISMA project, which is a joint effort with Philips' Research Laboratories, partially supported by the Dutch government through SPIN.

1. Rewriting Boolean Terms, version 1

1.1. Below we will specify a term rewriting system for boolean terms in the ASF-formalism. An introduction to ASF is given in [BHK]. For term rewriting systems we refer to [K]. Sizable examples of ASF-specifications that constitute complete term rewriting systems may be found in [RH].

```
module Booleans
begin
  exports
    begin
      sorts    B
      functions T : -> B
                F : -> B
                ~_ : B -> B      -- logical not
                _ & _ : B # B -> B -- logical and
                _ v _ : B # B -> B -- logical or
    end
  equations
  [1] ~T = F
```

```
    [2] ~F = T
    [3] T & T = T
    [4] T & F = F
    [5] F & T = F
    [6] F & F = F
    [7] T v T = T
    [8] T v F = T
    [9] F v T = T
    [10] F v F = F
end Booleans
```

Comments:

- We will deviate in two minor details from official ASF syntax. One is the omission of the dots around the v operator, which actually should have been .v. .This has the advantage that in the parsing process the lexical tokens are single characters, which prevents the need for a separate lexical scan. The other deviation concerns the context free grammar of terms and will be explained below.
- The specification is interpreted as a rewriting system $\top$ in the obvious way: the equations are read as rewrite rules from left to right.
- It is evident that $\top$ is strongly terminating; and confluency follows from regularity (the rewrite rules are left- linear and do not overlap). $\top$ is therefore a complete term rewriting system.
- This specification is an extension of the specification of the booleans in [RH]. The most efficient implementation of the module BOOL as it appears there would simply check whether the symbol F occurs: a rather atypical shortcut.

1.2. The rewriting algorithm

The main idea of the algorithm may be expressed as follows : a term is represented as a tree, the nodes of which are active objects that execute in parallel. This is easily realized in the language Pool-T, which has active objects executing in parallel as its main feature. For an introduction to the Pool-T see [A], [AU], and [BM]. The representation of terms that is used in the implementation below is rather straightforward : the parsing of a term yields a *termobject* that contains a main function symbol and two daughter termobjects. The main symbol is either 'T' or 'F' or an operator: '~', '&' or 'v'. Depending on the arity of the main function symbol (0, 1 or 2), 0, 1 or 2 daughter objects obtain the proper subterm, transformed into a termobject, as their value.

In the implementation below, only the term rewriting is done in parallel. Parsing is done sequentially by an object called *driver*. Another possibility would be to let each termobject do its own bit of parsing, for instance as follows : a newly created termobject is initialized with a

string for the term it is to represent. It determines the main function symbol and 0, 1 or 2 substrings from this string, according as the main symbol is T, F, ~ , v or &. The main function symbol is retained, 0, 1 or 2 new termobjects are created and these are initialized with the substrings. This algorithm has to compete with the traditional sequential parsing algorithm. Now the latter takes a look at each character in the string exactly once, whereas the parallel algorithm goes through the string over and over again; for this reason we preferred sequential parsing.

We use the following contextfree grammar:

```
<term>   ::= <factor> |
             <factor>  "&"  <factor> |
             <factor>  "v"  <factor>

<factor> ::= "(" <term> ")" |
             "~"  <factor>   |
             "T"  |
             "F"
```

This grammar contains the second deviation from ASF. The first operand of a binary operator symbol may be a term in ASF (thereby allowing terms like T v T v T which equals (T v T) v T) whereas it is only allowed to be a factor in our grammar. The advantage of this is that the parsing can now be done by a simple recursive descent parser instead of an LR-parser.

The following *rewriting algorithm* is executed by the termobjects: as soon as a termobject comes into existence, it starts to rewrite. It checks if it has any daughters. If it has, these daughters are requested to send their normal forms. Then the object determines the normal form of the term it represents and answers the request from its mother to send this normal form.

1.3. The implementation and its correctness proof

Concerning the program below, only the behavior of the driver, the automatically created first object, remains to be explained. It asks the user to type in a boolean term. Thus boolean terms enter the system as character strings. The string is parsed in a traditional way, only the result of the parsing is a somewhat untraditional termobject. This termobject is asked for its normal form, the normal form is printed and the driver asks for the next boolean term from the user.

We now sketch a proof that the two main parts of the program, the rewriting and the parsing, function properly. First the parsing part, the methods parseterm and parsefactor. One can prove the following statements (by *errors* we mean syntax errors in the input string):

— Parseterm and parsefactor will terminate if the input is a finite string (induction on *l–i*).

— If the string index *i* points to the first character of a correct term (according to the grammar given above; the term is just somewhere in the string and may be followed by other characters) and if no error has occurred yet, then parseterm and parsefactor will yield the correct termobject for the longest correct term starting at position *i*, will give the string index the value 1 + "the index of the last character of the term just parsed" and will not generate an error message (simultaneous induction on the lengths of terms and factors).

— At most one error message is generated. After the detection of an error, nothing further is done with the erroneous term and the user is immediately prompted for the next term.

— If a call to parseterm [parsefactor] does not cause an error message, then the part of the string from the index value before the call up to (but excluding) the value after the call, obeys the syntax of terms [factors] (induction on the length of strings).

The proofs are straightforward. We have not taken the trouble to check formally if the error messages always make sense. Such a check is possible, but tedious. In test runs we did not detect any nonsensical error messages. In any case, we *have* established the equivalence that really matters: a term is correct iff it is processed without generating an error message.

For the rewriting part, one proves the following statement :

— answering the message "sendnormalform", a termobject will send the normal form of the term it represents.

This is done by a case distinction on the basis of the value of the instance variable "mainsymbol" and by realizing that the recursive descent, needed in the proof, does indeed terminate.

The program

```
## Boolean reduce program with sequential parsing and parallel
rewriting.


ROOT UNIT

CLASS Termobject

VAR mainsymbol : Character,
    obj1, obj2 : Termobject

ROUTINE new(ob1:Termobject, mnsymbol : Character, ob2 :
Termobject)
    Termobject :
RETURN NEW ! init(ob1, mnsymbol, ob2)
END new
```

```
METHOD init(ob1:Termobject, mnsymbol : Character, ob2 :
Termobject)
     Termobject :
   mainsymbol <- mnsymbol;
   obj1 <- ob1;
   obj2 <- ob2
RETURN SELF
END init

METHOD sendnormalform() Character :
RETURN mainsymbol
END sendnormalform

BODY
ANSWER(init);
  IF mainsymbol = '~' THEN
     IF (obj2 ! sendnormalform() ) = 'T' THEN
           mainsymbol <- 'F'
     ELSE mainsymbol <- 'T'
     FI
  ELSIF mainsymbol = '&' THEN
     IF ((obj1 ! sendnormalform()) = 'T') &
        ((obj2 ! sendnormalform()) = 'T')  THEN
           mainsymbol <- 'T'
     ELSE mainsymbol <- 'F'
     FI
  ELSIF mainsymbol = 'v' THEN
     IF ((obj1 ! sendnormalform()) = 'F') &
        ((obj2 ! sendnormalform()) = 'F')  THEN
           mainsymbol <- 'F'
     ELSE mainsymbol <- 'T'
     FI
  FI;  ## no action necessary if mainsymbol is already T or F.

  ANSWER(sendnormalform)
END Termobject

##-----------------------------------------------

CLASS Driver

VAR termstring : String,
    termobject : Termobject,
    i, l : Integer,
    screen : Write_File,
    keyboard : Read_File,
    enough, erroroccurred : Boolean

METHOD error(s : String) Driver :
    erroroccurred <- TRUE;
    screen ! new_line() ! write_string(" error found by ") !
             write_string(s) ! new_line()
RETURN SELF
END error

METHOD parseterm() Termobject :
LOCAL obj, obj1, obj2 : Termobject, operator : Character IN
    IF i > l THEN error(" parseterm : unexp.end of string.")
```

```
    ELSE
     obj1 <- parsefactor();
     IF ~ erroroccurred THEN
       IF i > l THEN
          obj <- obj1
       ELSIF (termstring @ i = 'v') |
             (termstring @ i = '&')   THEN
          operator <- termstring @ i;
          i <- i+1;
          obj2 <- parsefactor();
          IF ~ erroroccurred THEN
              obj <- Termobject.new(obj1, operator, obj2)
          FI
       ELSE obj <- obj1
       FI
     FI
    FI
RETURN obj
END parseterm

METHOD parsefactor() Termobject :
LOCAL obj, obj1 : Termobject IN
    IF i > l THEN error(" parsefactor : unexp.end of string.")
    ELSIF termstring@i = '(' THEN
      i <- i + 1;
        obj <- parseterm();
          IF ~ erroroccurred THEN
            IF i <= l THEN
              IF termstring@i ~= ')' THEN
                error("parsefactor : unexpected char.")
              ELSE     i <- i+1
              FI
            ELSE error("parsefactor : right paren. missing")
            FI
          FI
    ELSIF termstring@i = '~' THEN
      i <- i+1;
      obj1 <- parsefactor();
      IF ~ erroroccurred THEN
        obj <- Termobject.new(NIL, '~', obj1)
      FI
    ELSIF (termstring@i = 'T') | (termstring@i = 'F')THEN
      obj <- Termobject.new(NIL, termstring@i, NIL);
      i <- i+1
    ELSE error("parsefactor : unexpected character")
    FI
RETURN obj
END parsefactor

METHOD squeezespaces(s : String) String :
LOCAL s2 : String, i, l : Integer IN
   s2 <- "";
   l <- s ! size();
   i <- 0;
   DO i < l THEN
     i <- i + 1;
     IF s @ i ~= ' ' THEN
        s2 <- s2 ! concat(s ! substring(i,i))
```

```
      FI
    OD
RETURN s2
END squeezespaces

BODY
  screen <- Write_File.standard_out();
  keyboard <- Read_File.standard_in();
  enough <- FALSE;
  screen ! write_string(" you can quit with q.") ! new_line() !
                                                   new_line();

  DO ~ enough THEN
    screen ! write_string(" give a boolean term > ");
    termstring <- keyboard ! read_string();

    IF termstring = "q" THEN enough <- TRUE
    ELSE
      termstring <- squeezespaces(termstring);
      erroroccurred <- FALSE;
      i <- 1;
      l <- termstring ! size();
      termobject <- parseterm();
      IF ~ erroroccurred THEN
        IF i <= l THEN
          error("driver : remaining characters at end of term")
        FI;

        IF ~ erroroccurred THEN
          screen ! write_char(termobject ! sendnormalform())
        FI
      FI;
      screen ! new_line()!new_line()
    FI
  OD
END Driver
```

2. Rewriting Boolean Terms, version 2

2.1 In this section we introduce another specification of the booleans, one that allows so-called lazy rewriting. In *Booleans2*, the right-hand side operand of a term `Tv (...)` has no influence on the normal form. The most efficient procedure in such cases is to stop processing terms as soon as they are seen to be irrelevant. This is most easily implemented in languages with an interrupt mechanism. As far as we know, the languages of the Pool family do not have such a mechanism. Below, we try to make the best of it without interrupts, checking whether a stop message has arrived as often as possible.

```
module Booleans2
begin
  exports
```

```
    begin
      sorts     B
      functions T: -> B
                F: -> B
                ~_: B -> B        -- logical not
                _ & _: B # B -> B  -- logical and
                _ v _: B # B -> B  -- logical or
    end
  variables p : -> B
  equations
  [1] ~T = F
  [2] ~F = T
  [3] T & p = p
  [4] F & p = F
  [5] T v p = T
  [6] F v p = p
end Booleans2
```

2.2. The implementation of Booleans2 rewriting

We only need to change the class Termobject. The syntax of terms and the way we represent terms by objects is still the same. Termobjects should now issue stop messages; and when they receive such a message, they must send stop messages to their non-NIL daughters that are still running.

The proof of the correct behavior of termobjects is now somewhat more involved than in section 1. One may prove the following statements, in the order in which they are given:

- a termobject sends one message to its non-NIL daughters, either "stop" or "sendnormalform";
- a termobject receives exactly one message (after initialization), either "stop" or "sendnormalform" (the top object receives a "sendnormalform" message from the driver);
- if a termobject receives a "stop" message, it sends a "stop" message to its non-NIL daughters and exits.
- if a termobject receives a "sendnormalform" message, it sends the correct normal form of the term it represents.

The definition of CLASS Termobjects for Booleans2

```
CLASS Termobject

VAR mainsymbol : Character,
```

```
    obj1, obj2 : Termobject

ROUTINE new(ob1:Termobject, mnsymbol : Character, ob2 :
Termobject)
    Termobject :
RETURN NEW ! init(ob1, mnsymbol, ob2)
END new

METHOD stop() Termobject :
RETURN SELF
END stop

METHOD init(ob1:Termobject, mnsymbol : Character, ob2 :
Termobject)
     Termobject :
   mainsymbol <- mnsymbol;
   obj1 <- ob1;
   obj2 <- ob2
RETURN SELF
END init

METHOD sendnormalform() Character :
RETURN mainsymbol
END sendnormalform

BODY
ANSWER(init);
  IF mainsymbol = '~' THEN
     SEL ANSWER(stop)
         THEN obj2 ! stop()
     OR  THEN
           IF obj2 ! sendnormalform()  = 'T' THEN
               mainsymbol <- 'F'
           ELSE mainsymbol <- 'T'
           FI;
           ANSWER(stop,sendnormalform)
     LES
  ELSIF mainsymbol = '&' THEN
     SEL ANSWER(stop)
         THEN obj1 ! stop(); obj2 ! stop()
     OR  THEN
           IF obj1 ! sendnormalform() = 'T' THEN
               mainsymbol <- obj2 ! sendnormalform()
           ELSE obj2 ! stop();
               mainsymbol <- 'F'
           FI;
           ANSWER(stop,sendnormalform)
     LES
  ELSIF mainsymbol = 'v' THEN
     SEL ANSWER(stop)
         THEN obj1 ! stop(); obj2 ! stop()
     OR  THEN
           IF obj1 ! sendnormalform() = 'F' THEN
              mainsymbol <- obj2 ! sendnormalform()
           ELSE obj2 ! stop();
              mainsymbol <- 'T'
           FI;
           ANSWER(stop, sendnormalform)
```

```
    LES
  ELSE ANSWER(stop, sendnormalform)
  FI
END Termobject
```

Of course, the surest way to avoid objects working on terms none is interested in is to have them start rewriting only after they have received a sendnormalform message (putting the code for rewriting within the method sendnormalform). Unfortunately, this would make our entire program sequential.

3. Plans for further work

A program for rewriting boolean terms is of small value by itself; automated rewriting is useful only for complicated term rewriting systems. Even there, it is useful only if it is based on a theory with some degree of generality. Ideally, particular rewriting programs would be generated automatically. In the near future we plan to implement more complicated systems from [RH] and to combine the idea of parallel object-oriented rewriting with the theory of graph rewriting as treated in [BEGKPS].

REFERENCES.

[A] AMERICA, P. *Definition of the programming language Pool-T*, ESPRIT project 415, Doc. Nr. 0091, Philips Research Laboratories, Eindhoven (1985).

Rationale for the design of Pool, ESPRIT project 415, Doc. Nr. 0053, Philips Research Laboratories, Eindhoven (1986).

[AU] AUGUSTEIJN, L. *Pool-T User Manual*, ESPRIT project 415, Doc. Nr. 0104, Philips Research Laboratories, Eindhoven (1986).

Pool-T Standard Environment, ESPRIT project 415, Doc. Nr. 0138, Philips Research Laboratories, Eindhoven.

[BEGKPS] BARENDREGT, H.P., M.C.J.D. VAN EEKELEN, J.R.W. GLAUERT, J.R. KENNAWAY, M.J.PLASMEIJER & M.R.SLEEP.*Term Graph Rewriting*, University of Nijmegen, internal report nr. 87 (1986).

[BHK] BERGSTRA,J.A., P.KLINT & J.HEERING. *ASF - An algebraic specification formalism*, Centre for Mathematics and Computer Science, Report CS-R8705 (1987).

[BM] BEEMSTER, M. & H. MULLER, *User manual of pl, a Pool-T interpreter*, ESPRIT project 415, Doc. Nr. P0038, Philips Research Laboratories, Eindhoven (1987).

[K] KLOP, J.W. *Term rewriting systems : a tutorial*, Bulletin of the European Association for Theoretical Computer Science, June 1987.

[RH] RODENBURG, P.H. & D.J.HOEKZEMA, *Specification of the Fast Fourier Transform Algorithm as a Term Rewriting System*, University of Utrecht, Logic Group Preprint Series no. 27 (1987).

Gauss elimination as a Term Rewriting System, University of Utrecht, Logic Group Preprint Series no. 30 (1988).

Logic Group Preprint Series
Department of Philosophy
University of Utrecht
Heidelberglaan 2
3584 CS Utrecht
The Netherlands

nr. 1 C.P.J. Koymans, J.L.M. Vrancken, *Extending Process Algebra with the empty process*, September 1985.
nr. 2 J.A. Bergstra, *A process creation mechanism in Process Algebra*, September 1985.
nr. 3 J.A. Bergstra, *Put and get, primitives for synchronous unreliable message passing*, October 1985.
nr. 4 A.Visser, *Evaluation, provably deductive equivalence in Heyting's arithmetic of substitution instances of propositional formulas*, November 1985.
nr. 5 G.R. Renardel de Lavalette, *Interpolation in a fragment of intuitionistic propositional logic*, January 1986.
nr. 6 C.P.J. Koymans, J.C. Mulder, *A modular approach to protocol verification using Process Algebra* , April 1986.
nr. 7 D. van Dalen, F.J. de Vries, *Intuitionistic free abelian groups*, April 1986.
nr.8 F. Voorbraak, *A simplification of the completeness proofs for Guaspari and Solovay's R*, May 1986.
nr.9 H.B.M. Jonkers, C.P.J. Koymans & G.R. Renardel de Lavalette, *A semantic framework for the COLD-family of languages*, May 1986.
nr.10 G.R. Renardel de Lavalette, *Strictheidsanalyse*, May 1986.
nr.11 A. Visser, *Kunnen wij elke machine verslaan? Beschouwingen rondom Lucas' argument*, July 1986.
nr.12 E.C.W. Krabbe, *Naess's dichotomy of tenability and relevance*, June 1986.
nr.13 Hans van Ditmarsch, *Abstractie in wiskunde, expertsystemen en argumentatie*, Augustus 1986
nr.14 A.Visser, *Peano's Smart Children, a provability logical study of systems with built-in consistency* , October 1986.
nr.15 G.R.Renardel de Lavalette, *Interpolation in natural fragments of intuitionistic propositional logic*, October 1986.
nr.16 J.A. Bergstra, *Module Algebra for relational specifications*, November 1986.
nr.17 F.P.J.M. Voorbraak, *Tensed Intuitionistic Logic*, January 1987.
nr.18 J.A. Bergstra, J. Tiuryn, *Process Algebra semantics for queues*, January 1987.
nr.19 F.J. de Vries, *A functional program for the fast Fourier transform*, March 1987.
nr.20 A. Visser, *A course in bimodal provability logic*, May 1987.
nr.21 F.P.J.M. Voorbraak, *The logic of actual obligation, an alternative approach to deontic logic*, May 1987.
nr.22 E.C.W. Krabbe, *Creative reasoning in formal discussion*, June 1987.
nr.23 F.J. de Vries, *A functional program for Gaussian elimination*, September 1987.
nr.24 G.R. Renardel de Lavalette, *Interpolation in fragments of intuitionistic propositional logic*, October 1987.(revised version of no. 15)
nr.25 F.J. de Vries, *Applications of constructive logic to sheaf constructions in toposes*, October 1987.
nr.26 F.P.J.M. Voorbraak, *Redeneren met onzekerheid in expertsystemen*, November 1987.
nr.27 P.H. Rodenburg, D.J. Hoekzema, *Specification of the fast Fourier transform algorithm as a term rewriting system*, December 1987.

nr.28 D. van Dalen, *The war of the frogs and the mice, or the crisis of the Mathematische Annalen,* December 1987.

nr.29 A. Visser, *Preliminary Notes on Interpretability Logic,* January 1988.

nr.30 D.J. Hoekzema, P.H. Rodenburg, *Gauß elimination as a term rewriting system,* January 1988.

nr. 31 C. Smorynski, *Hilbert's Programme,* January 1988.

nr. 32 G.R. Renardel de Lavalette, *Modularisation, Parameterisation, Interpolation,* January 1988.

nr. 33 G.R. Renardel de Lavalette, *Strictness analysis for POLYREC, a language with polymorphic and recursive types,* March 1988.

nr. 34 A. Visser, *A Descending Hierarchy of Reflection Principles,* April 1988.

nr. 35 F.P.J.M. Voorbraak, *A computationally efficient approximation of Dempster-Shafer theory,* April 1988.

nr. 36 C. Smorynski, *Arithmetic Analogues of McAloon's Unique Rosser Sentences,* April 1988.

nr. 37 P.H. Rodenburg, F.J. van der Linden, *Manufacturing a cartesian closed category with exactly two objects,* May 1988.

nr. 38 P.H. Rodenburg, J. L.M.Vrancken, *Parallel object-oriented term rewriting : The Booleans,* July 1988.