Experiences with the RAP system

- a specification interpreter combining term rewriting and resolution

Alfons Geser
Heinrich Hussmann
Universität Passau
Postfach 2540
D-8390 Passau

Abstract

This paper summarizes our experience gained using the RAP system, an interpreter for algebraic specifications which combines term rewriting and resolution techniques. A short description of the system is given and some points are worked out which may be of general interest for future development of similar systems: termination criteria, efficiency and search strategies.

1 INTRODUCTION

Using a formal approach to software development some tools for development and validation of specifications are necessary. Since the whole process of program development relies on a formal specification, it is important to ensure the quality and "correctness" of a specification. "Correctness" of a specification means that it meets the informal requirements which have been produced during a requirement analysis. A usual way of checking this informal kind of correctness is to build a prototype out of the specification and to test whether it behaves as intended ("rapid prototyping", cf. [Budde et al. 84]). The RAP system is a tool for automatic construction of a prototype out of an algebraic specification.

The specification language used is that of hierarchical algebraic specifications as defined e.g. in [Wirsing et al. 83]. A specification consists of a list of primitive types, a signature (declaring sort and function symbols together with their functionalities) and a finite set of positive conditional equations. The RAP system offers a complete syntax check for such specifications and an evaluation algorithm which works similar to a logic programming system. The user "asks" a query by supplying a system of equations and the system looks for solutions of these equations in the specified theory.

This work has been sponsored by the ESPRIT project 432 METEOR and the SFB 49 Programmieretechnik, Technische Universität München

This paper summarizes the experience gained during one year of experimentation with different versions of this system. The paper is organized as follows: In section 2 the algorithm used by the system is described and an example of a specification is given. RAP is compared to similar systems. Section 3 contains some theses of general interest resulting from our experiments, each of them illustrated by simple examples.

2 THE RAP SYSTEM

2.1 General Description Of The System

The RAP system has been constructed at the University of Passau as a part of the CIP project at the Technical University of Munich. It is written in PASCAL and is available for VAX computers under VMS and for IBM VM/SP CMS. Versions for other operating systems (including Berkeley-UNIX) are in preparation. For detailed information on the RAP system have a look at [Hussmann 85b] and [Geser, Hussmann 85].

A simple example will illustrate how the system works. The following text is a specification of labelled binary trees. Comments are enclosed in braces. The specified functions have the following meaning:

```

nil      construct an empty tree
cons     construct a tree node out of a label and two sons
isnil    test whether a tree contains a node
left     select left son of a node
right    select right son of a node
label    select label of a node

```

For the labels a primitive type ATOM is presupposed which defines constants a, b, c, d.

```

type TREE = Tree, nil, cons, isnil, left, right, label:
                                     {declaration of specification
                                     name and list of exported
                                     objects}
basedon ATOM, BOOL                  {list of primitive types}
sorts Tree                          {declaration of a signature}
cfuncts nil : Tree,                 {cfuncts means "constructor"}
      cons : (Tree,Atom,Tree)Tree { functions"}

functs isnil : (Tree)Bool,
      left, right : (Tree)Tree,
      label : (Tree)Atom

laws all (t1,t2 : Tree, m : Atom) {list of (oriented, generally}
  isnil(nil) -> true,             { conditional) axioms}
  isnil(cons(t1,m,t2)) -> false,  { containing universally}
  left(cons(t1,m,t2)) -> t1,     { quantified variables}
  right(cons(t1,m,t2)) -> t2,
  label(cons(t1,m,t2)) -> m

endoftype

```

The following type enriches TREE by a function computing the height of a tree. The maximum function max on natural numbers is supposed to be defined in the type MAX.

```

type HEIGHT = height:

  basedon TREE, NAT, MAX, ATOM
  functs height: (Tree)Nat

  laws all (t1, t2: Tree, m: Atom)
    height(nil) -> 0,
    height(cons(t1,m,t2)) -> succ(max(height(t1),height(t2))),
  endoftype

```

RAP uses the axioms as term rewrite rules oriented from left to right. This way it allows to "execute" the specification. For this purpose a query must be formulated as a system of equations containing free variables ("unknowns"). Such queries are called tasks in RAP. A first example of a task is:

```

task DESCR1 = t:           {declaration of task name and list of
                             unknowns of interest}
basedon TREE, ATOM         {list of types used}

unknowns t : Tree          {declaration of unknown variables}

goals                      {list of (non-oriented, non-conditional)}
  label(t) = a,            { equations}
  label(left(t)) = b,
  label(right(t)) = c

endoftask

```

A substitution of terms for the unknown variables which fulfills the goals is called a solution. We are only interested in most general solutions, so solutions may contain free variables produced by the system (which are denoted by " $*\langle \text{number} \rangle$ "). When executing DESCR1, the RAP system finds the solution

```
[t = cons(cons(*10,b,*12),a,cons(*13,c,*15))]
```

and detects that there is no other solution (CPU time: 6.41 secs). Note that the solution above covers an infinite set of more special solutions which are obtained by substitution of arbitrary tree-terms for the $*$ -variables.

2.2 Theoretical Background: The Narrowing Algorithm

The RAP system uses an algorithm which combines resolution and term rewriting techniques. This "conditional narrowing" algorithm is a generalisation of the unification procedures defined by Fay and Hullot ([Hullot 80]) and Kaplan ([Kaplan 83]). Below the algorithm is sketched. We assume the reader's familiarity with basic notions of term rewriting (see [Huet, Oppen 80]). $\equiv$ denotes the syntactical equality of terms.

Let t_i, t_i', l and r be wellformed terms. The specification defines a set of axioms of the general form

```
[ t1 = t1' & ... & tn = tn' => l -> r ]
```

for some n . For $n=0$, the syntax is simply $[l -> r]$.

We call a system of equations together with a substitution a goal. A goal is denoted by

```
[ t1 = t1', ... , tn = tn' WITH sigma ]
```

where the t_i and t_i' are terms, σ is a substitution.

There are two ways of deriving a new goal from a given goal
 $G \equiv [t_1 = t_1', \dots, t_n = t_n' \text{ WITH } \sigma]$

Unification Step:

If there is a (most general) substitution τ such that
 $\tau(t_i) \equiv \tau(t_i')$ for $i = 1, \dots, n$,
 the new goal
 $G' \equiv [\] \text{ WITH } \tau.\sigma$
 can be derived.

Narrowing Step:

If there are a non-variable subterm ti/u of a ti (i between 1 and n), a rule
 $[p_1 = p_1' \ \& \ \dots \ \& \ p_m = p_m' \Rightarrow l \rightarrow r]$
 and a (most general) substitution τ such that
 $\tau(l) = \tau(ti/u)$,
 the new goal
 $G' \equiv [\tau(p_1) = \tau(p_1'), \dots, \tau(p_m) = \tau(p_m'),$
 $\tau(t_1) = \tau(t_1'), \dots, \tau(ti[u \leftarrow r]) = \tau(ti'),$
 $\dots, \tau(t_n) = \tau(t_n') \text{ WITH } \tau.\sigma]$
 can be derived. There is an analogous step for subterms of ti' . Assume that the chosen rule is renamed using appropriate "new" identifiers at every application of this step. Note that the axioms are applied only from left to right.

Let a task T introduce the system S of equations. A substitution σ is called a solution of the task T , if from the goal $[S \text{ WITH } \emptyset]$ the goal $[\] \text{ WITH } \sigma$ can be derived. ($\emptyset$ denotes the empty substitution, $\emptyset$ denotes the empty system of equations.) A semi-algorithm for finding solutions is gained by fair enumeration of these derivations, i.e. by a fair search in a proof tree consisting of nodes labelled by goals and arrows labelled by narrowing steps.

It can be shown that this algorithm is correct w.r.t. conditional equational logic. It is complete, too, provided that the axiom set is confluent. In this case all equalities which are deducible in the equational calculus are also deducible by oriented term reduction (cf. [Huet, Oppen 80]). Note that RAP does not require the rewriting relation to be noetherian. For completeness we adopt the notion defined in [Hullot 80]: If there exists a substitution for which the equalities of the task can be proven within the conditional equational calculus, then there exists a derivation of finite length for this solution (or for an equivalent one, in the specified theory, or for a more general one). For a more detailed treatment of these issues cf. [Hussmann 85a].

2.3 Comparison To Other Systems

There are three classes of known tools which can be compared with RAP:

- Interpreters for algebraic specifications
- Term rewriting laboratories
- Logic programming systems.

The RAP system can be used as a classical term rewriting machine interpreting algebraic specifications, like, among others, ASSPÉGIQUE [Bidoit, Choppy 85]). But the narrowing algorithm for solving equations is much more powerful, it can be used e.g. for the generation of test data (cf. [Bouge et al. 85]).

A major difference between RAP and usual term rewriting laboratories is the use of a sorted and hierarchical specification language instead of pure equations. Most of these systems are based on the Knuth-Bendix completion algorithm, e.g. REVE ([Lescanne 83]), REVEUR4 ([Remy, Zhang 84]) and ERIL ([Dick 85]). Such tools could turn out to be useful as preprocessors to RAP-like systems, where out of an arbitrary specification the completion algorithm produces a confluent term rewriting system which can then be processed by RAP. RAP itself does not provide any check for confluence of the axiom set. The combination of narrowing and the Knuth-Bendix algorithm looks particularly attractive, since both algorithms use the same primitive operations on terms (e.g. unification, critical pairs).

There is a new term rewriting laboratory built onto REVE which uses a narrowing algorithm similar to RAP: NARROWER ([Rety et al. 85]). In contrast to RAP, it only handles unconditional axiom sets.

RAP differs from most logic programming systems in its general concept. It aims at a complete evaluation method independent of user-given control flow information (like sequential order of goals). In RAP, no control flow information is contained in the specification. So RAP specifications are rather "declarative", even if a loss of efficiency has to be paid for that.

Differences to PROLOG are a modular and strongly typed language and the use of functions and equations instead of predicates. The use of functions and equations is also a feature of the new logic programming language SLOG ([Fribourg 84a]). It uses almost the same algorithm as RAP, but in a more conventional style, dependent of the order of clauses. PROLOG and SLOG provide only a depth-first search of the proof tree, RAP offers a breadth-first search mode, alternatively. The language EQLOG ([Goguen, Meseguer 84]), the implementation of which is under development, will also be based on a narrowing algorithm.

3 RESULTS OF EXPERIMENTATION

3.1 Termination Needs Support

The fact that resolution does not guarantee termination does heavily affect its practice. The greater part of the tasks entered into the system actually do not terminate. This shows that termination criteria like those proposed by [Hullot 80] are not helpful, since they try to ensure the termination of the "pure" narrowing algorithm by strong syntactical restrictions on the axioms - which are not fulfilled in many interesting cases. We claim that termination is rather a matter of a pair of a specification and a task than of a specification alone.

It seems promising to increase the domain of termination by cancelling unnecessary infinite paths in the proof tree. Such a path is often generated by a divergent goal, i.e. a nonterminating goal with no solution. We have to capture divergent goals above all, since by cancelling other goals, i.e. goals with a nonempty set of solutions, we could loose completeness.

A trivial terminating task is given by

```

task NC = t1, m, t2:
  basedon TREE, ATOM
  unknowns t1, t2: Tree, m: Atom

goals
  nil = cons(t1,m,t2).
endoftask.

```

No narrowing step is possible and unification terminates showing that the equation has no solution.

Let us construct out of this task a slightly smarter one by adding a nonterminating equation. The naive narrowing algorithm performs an infinite sequence of narrowing steps for this second equation, although it cannot find any solution, due to the failure of the first equation. There are several possibilities to decide this fact in the algorithm. The goal above terminates in a PROLOG interpreter. If the equations are exchanged, however, termination is lost. This does not hold in RAP. Even if $t1$ is replaced by $\text{left}(t1)$, such that further narrowing is possible, RAP can refuse the goal.

The system detects that nil and cons are irreducible symbols, i.e. there is no axiom with nil or cons as its leftmost symbol. So, they cannot be removed by rewriting steps (even after substitutions have been applied) and the goal has no solution in the resp. theory. The so-called "termination with help of boolean rewrite systems" in [Fribourg 84b] is a functional translation of this technique.

During experimentation with the data type SET, it turned out that further improvements of an unsatisfiability tester are desirable in the case of equations between constructors. The tester fails also in the goal

```

[cons(t1,m,t2) = t1 WITH sigma]

```

because of the variable $t1$, which is recursively specified.

An improved tester, currently under design, is intended to give on any sort a fine partition of the set of constructor terms with variables, thereby coping with recursive variable occurrences.

3.2 Safety First

Trying to gain efficiency can lead to loss of termination. An instance for this phenomenon is sketched in the following example due to [Wos et al. 84]:

```

type P = p:
  basedon NAT
  functs p : (Nat,Nat)Nat
  laws all (x:Nat)
    p(x,x) -> true
endofetype

```

The task

```

p(succ(x),x) = true

```

has no solution, as can easily be seen. The presence of multiple copies of the same variable in axioms and equations resp. must be

handled with care in order to avoid nontermination within unification. A simple test which is called occur-check asserts that no variable is unified with a term containing it. In contrast to e.g. RAP, most PROLOG systems do not use occur-checks in order to "increase efficiency", and thus do not discard the goal above, but enter an infinite loop.

3.3 Keep The Proof Tree Small By Pruning

The main reason why some goals do not work effectively is the exponential growth of the proof tree. It is a difficult but important task to keep the tree fairly small. To do this can even lead to termination - when the tree becomes finite.

There are basically two ways to achieve this:

- to use normalisation in order to generate only few nodes (as described in 3.5)
- to scratch nodes in the proof tree without losing completeness.

The latter technique is called pruning ([Barr, Feigenbaum 81]). A node can be scratched if its goal is known

- to have no solution (this is described in 3.1),
- to be a special case of some other goal already in the tree,
- to have a substitution which is a special case of some solution already found.

The second property is reflected formally by the subsumption theorem ([Rety et al. 85]). This theorem is used in a subsumption tester built into RAP. Note that, due to the confluence property which is required for every specification, there exist many tree nodes containing equal goals.

It turned out that subsumption by goals works best when the whole proof tree information is kept stored. Then the execution time for a terminating goal is nearly independent of the search strategy. If exhausted goals are deleted, like in PROLOG systems, this does not hold (cf. 3.4.).

At a first sight, subsumption by solutions looks also rather attractive. The best thing that actually can happen is that, after having found a solution, the whole computation is able to come to an early end. Unfortunately this kind of behaviour is quite rare, and so is any application of subsumption by solution. Normally, solutions are more special than the substitutions in goals, probably since solutions contain additional unification steps. Those steps usually make the solution more special than the substitutions of intermediate goals.

3.4 Breadth-first Is Superior To Depth-first

Basically, there are two search strategies: depth-first (df) and breadth-first (bf). RAP offers both strategies optionally for test purposes. So df and bf can be compared easily.

In PROLOG, the user wants to keep control over the operational behaviour of the interpreter. Efficiency is estimated more important than completeness. This is a reason for a search strategy that finds the first solution very early, namely df. The task

```
task DESCR3 = t:
  basedon TREE,ATOM,BOOL
  unknowns t:Tree
```

```
goals
  isnil(left(left(left(t)))) = true,
  isnil(right(left(left(t)))) = false,
  isnil(right(left(t))) = false,
  isnil(right(t)) = true
endoftask
```

has a unique solution. Its execution times are shown in the following table (CPU times in seconds on a VAX 11/750).

Task DESCR3	df	bf	(exhausted goals deleted)
first solution found	5.31	39.91	
termination achieved	3039.78	41.03	

As the first solution usually is found at the left hand side of the proof tree, df appears better. To achieve termination, the whole tree is to be exhausted. The times above are valid for an implementation where exhausted goals are deleted in order to save memory space. Df performs considerably poorer than bf does, due to the fact that df does not keep enough information about the tree history to apply subsumption effectively.

The following table shows the times in a slightly changed implementation, where exhausted goals are kept. Then bf and df are almost equal in speed.

Task DESCR3	df	bf	(exhausted goals kept)
first solution found	5.44	50.01	
termination achieved	49.72	50.79	

Theorem proving and test data generation need a fair enumeration of solutions, i.e. every solution should be presented eventually. This is no problem, if the proof tree is finite. But df does not find any solution to the right of an infinite path. If we want to evaluate a task like e.g.

```
task HEIGHT2 = t:
  basedon TREE,NAT,HEIGHT
  unknowns t:Tree
```

```
goals
```

```
  height(t) = 2
```

```
endoftask
```

we have to apply the rules in a fair way to get the three solutions. With df, only the first solution is found. In bf, however, number and location of infinite paths do not matter particularly. Obviously, df has a great disadvantage here. Nevertheless there are attempts to get fairness within a df-like strategy, namely by means of depth bounds ([Bouge et al. 85]). We prefer bf in this case. One reason is, for instance, that empirical aids like bounds are not

needed here. The experiments show clearly, that the possibility of a choice between different strategies is a useful feature.

3.5 Use Confluence To Improve Efficiency

Normalisation is a powerful tool together with resolution. When a set of rules is known to be noetherian, it may be used to rewrite goals in a df way without losing completeness. As the whole rule set is not required to be noetherian, it is left to the user to indicate a correct subset of normalising rules.

Instead of following each path and thus getting an enormous tree complexity the rewriting steps keep track with a single path and deliver just one final node. This node contains the normal form w.r.t. the normalising rule set. As normalisation is based upon noetherian and confluent rules, the normal form is unique (cf. [Huet, Oppen 80]) and therefore completeness is preserved.

Rewriting can be speeded up by introduction of shared subterms like e.g. the terms l, ll in the following example:

```
task DESCR3S = t:
  basedon TREE,ATOM,BOOL
  unknowns t,l,ll:Tree

goals
  l = left(t),
  ll = left(l),
  isnil(left(ll)) = true,
  isnil(right(ll)) = false,
  isnil(right(l)) = false,
  isnil(right(t)) = true
endoftask
```

The task is essentially the same as DESCR3 above but with sharing. Its execution time is summarized in the following table:

Task DESCR3S	df	bf	
=====			(exhausted goals kept)
first solution found	4.78	43.55	
termination achieved	45.53	47.67	
=====			(exhausted goals deleted)
first solution found	4.79	35.43	
termination achieved	1043.14	36.39	
=====			

In DESCR3, the terms left(t) and left(left(t)) have to be rewritten for each occurrence separately, in DESCR3S however, each term is rewritten only once. Again, the confluence property asserts that nothing is lost. Although the times do not look very exciting, it turns out that nothing is paid for this improvement. On the contrary, the syntactical representation of the goal becomes more concise.

4 CONCLUSION

In this paper, the experience with a specification interpreter has been sketched using small, uniform examples. We have got good results also with large type and task specifications. Among others, we studied the programming language DSPL (cf. [Broy, Pair, Wirsing

81]), a microprocessor (cf. [Geser 86]) and more sneaky specifications, e.g. the lambda calculus and finite sets. So RAP does not only apply to "toy examples", but problems arise when an enormous search space has to be visited. This is a matter not of the syntactical size of goals and types but of the intrinsic complexity of the goal. A task similar to DESCR3, containing, say, 10 rather large equations behaves not much worse than DESCR3 (for finding the first solution using df), whereas the fair enumeration of trees with height 5 is in fact impossible. The same problem we met in the microprocessor specification of [Geser 86], where RAP is e.g. forced to investigate all (65536) possible instances of an 16-bit word, a hopeless situation. Here interference by the user is urgently needed, and this will not change in the near future. In these cases, the possibility of interactive observation and control of the narrowing algorithm built into RAP was very helpful.

Industrial applicability means, above all, prototyping. So the processing of large data is not in the scope of a specification interpreter, but to show the basic structure of new specifications. So, after minor improvements, like better structure sharing and unsatisfiability testers, the RAP technology will be usable in practice.

Acknowledgements

We would like to thank C. Lengauer for his inspiring lectures in Passau, and him and M. Wirsing for many fruitful discussions.

References

- [Barr, Feigenbaum 81]
A. Barr, E. A. Feigenbaum, The handbook of artificial intelligence, Vol. I. Heuris Tech Press, Stanford 1981
- [Bidoit, Choppy 85]
M. Bidoit, C. Choppy, ASSPEGIQUE: An integrated environment for algebraic specifications, Proc. TAPSOFT 85, LNCS 186, pp.246-260
- [Bouge et al. 85]
L. Bouge, N. Choquet, L. Fribourg, M. C. Gaudel, Application of PROLOG to test sets generation from algebraic specifications. Proc. TAPSOFT 85 Conf., LNCS 186, pp.262-275
- [Broy, Pair, Wirsing 81]
M. Broy, C. Pair, M. Wirsing, A systematic study of models of abstract data types. Report 81-R-042, Université de Nancy, Nancy 1981. To appear also in TCS.
- [Budde et al. 84]
R. Budde, K. Kuhlenkamp, L. Mathiassen, H. Züllinghoven (Eds.): Approaches to prototyping. Berlin/Heidelberg/New York/Tokyo 1984.
- [CIP 85]
A. Horsch, B. Möller, H. Partsch, O. Paukner, P. Pepper, The Munich project CIP, Vol. II, Part I: Formal specification. Report TUM-I8509 Technische Universität München, München 1985

[Dick 85]

A. J. Dick, ERIL - Equational reasoning: an interactive laboratory. Internal report Rutherford Appleton Laboratory, Chilton, Didcot 1985

[Fribourg 84a]

L. Fribourg, Oriented equational clauses as a programming language. Report 84002 Laboratoires de Marcoussis, Marcoussis 1984. Short version in: Proc. 11th ICALP Conf., LNCS 172, pp. 162-173

[Fribourg 84b]

L. Fribourg, Handling function definitions through innermost superposition and rewriting. Report 84-69 LIIP, Université Paris 7, 1984. Short version in: Proc. 1st RTA Conf., to appear in LNCS

[Geser 85]

A. Geser, An algebraic specification of the INTEL 8085 microprocessor: A case study. Report Universität Passau, to appear.

[Geser, Hussmann 85]

A. Geser, H. Hussmann, Rapid prototyping for algebraic specifications - examples for the use of the RAP system, Report MIP-8517 Universität Passau, Passau 1985.

[Goguen, Meseguer 84]

J. A. Goguen, J. Meseguer, Equality, types, modules and generics for logic programming. Report CSLI-84-5 Leland Stanford Junior University, Stanford 1985

[Huet, Oppen 80]

G. Huet, D. C. Oppen, Equations and rewrite rules - a survey. In: R. V. Book (ed.), Formal language theory - perspectives and open problems. Academic Press

[Hullot 80]

J. M. Hullot, Canonical forms and unification. Proc. 5th CADE Conf., LNCS 87, pp. 318-334 1980

[Hussmann 85a]

H. Hussmann, Unification in conditional-equational theories, Report MIP-8502 Universität Passau, Passau 1985. Short version also in: Proc. EUROCAL 85, to appear in LNCS.

[Hussmann 85b]

H. Hussmann, Rapid prototyping for algebraic specifications - RAP system user's manual. Report MIP-8504 Universität Passau, Passau 1985

[Kaplan 83]

S. Kaplan, Fair conditional term rewriting systems: unification, termination and confluence. Report 194 LRI Université de Paris-Sud, Orsay 1984

[Lescanne 83]

P. Lescanne, Computer experiments with the REVE term rewriting system generator, Proc. 10th ACM Conf. on Principles of Programming Languages, Austin, Texas 1983

[Remy, Zhang 84]

J. L. Remy, H. Zhang, REVEUR4: A system for validating conditional algebraic specifications of abstract data types. In: T. O'Shea (ed.), Proc. ECAI 84: Advances in artificial intelligence, pp. 563-572, North-Holland 1984

[Rety et al. 85]

P. Rety, C. Kirchner, H. Kirchner, P. Lescanne, NARROWER: a new algorithm for unification and its application to logic programming. In: Proc. RTA 85 Conf., to appear in LNCS.

[Wirsing et al. 83] M. Wirsing, P. Pepper, H. Partsch, W. Dosch, M. Broy: On hierarchies of abstract data types. Acta Informatica 20(1983), 1-33.

[Wos et al. 84]

L. Wos, R. Overbeek, E. Lusk, J. Boyle: Automated Reasoning, introduction and applications, Prentice Hall Inc., 1984