Rapid Prototyping for Algebraic Specifications

R A P System User's Manual

Heinrich Hussmann
Universität Passau
Fakultät für Mathematik und Informatik
Postfach 2540
D-8390 Passau

Version 2.0

März 1985

This work was sponsored by the Sonderforschungsbereich 49 - Programmiertechnik at the Technical University of Munich and the ESPRIT project 432 - METEOR.

Preface to the second edition

This report is a second, completely revised edition of the RAP User's Manual printed as MIP-8504.

The documentation given here refers to RAP system version 2.0, it is of no use for RAP versions 1.X.

For information on the distribution of RAP, error reports, suggestions, etc. please send electronic mail to the EUNET adress rap@unipas.uucp (or ordinary mail to the author).

I have to to acknowledge a large amount of constructive criticism by my collegues Alfons Geser, Thomas Pinegger and Peter Padawitz.

CHAPTER 1

INTRODUCTION

1.1. Development of Algebraic Specifications

RAP is an experimental system which supports the development of modular algebraic specifications of abstract data types. For this purpose a small but powerful specification language has been defined based on the concept of hierarchical algebraic specifications ([WPPDB 83]). The RAP system supports the design process within this language on the syntactic, semantic, and pragmatic level. The support on the syntactic level consists in a rigorous syntax check of the text, a trivial but important component of assistance. On the semantic level, some checks for interesting properties are performed (not yet implemented in version 2.0). Both these levels of analysis are seen as prerequisites for the third stage which tries to handle with the real "meaning" of the specification. Of course, there is no way of verifying formally the correctness of a specification, since its purpose is to provide the basis for any verification. But by a pragmatic analysis the user may compare the specification against his/her intentions, using the central system component: **prototyping of algebraic specifications**.

The class of equational Horn specifications ("conditional equations") can be given an operational semantics by term rewriting and resolution. Thus, specifications can be "executed" immediately. Execution does not only mean evaluation of the specified functions with given arguments, but to **solve systems of equations** based on a given specification. Solving equations comprises simple proofs of properties of the specified functions, too. The inverse of a specified function can this way be simulated by the system, a property which programmers may use e.g. to generate test data sets.

The kernel of the system is an algorithm for solving equations in conditional-equational theories ("conditional narrowing"), which is very similar to resolution algorithms used in logic programming languages. In particular, it is easy to translate a given PROLOG program into an algebraic specification. For more details on the algorithm and theoretical results see [Hussmann 85]. In this paper the reader is introduced into the practical use of the system.

1.2. Overview

The RAP system has been implemented at the University of Passau by H. Hussmann with the help of U. Fraus, R. Hennicker and F. Nickl. The work on the system started in spring 1984, a first version was available in spring 1985, the current, completely revised version

2.0 has been released in summer 1986. The system is written in PASCAL (about 8000 lines of code) and is currently available for the operating systems VAX/VMS and UNIX 4.X BSD.

Main components of the RAP system are:

- classical user interface (RAP command language)
- syntax check for algebraic specifications and systems of equations (RAP specification language)
- optimized conditional narrowing procedure
- powerful interactive debugger on top of the narrowing procedure.

This manual is organized as follows: The rest of this chapter gives an introductory example for the use of RAP. We define the syntax of the RAP specification language formally in chapter 2 and explain the RAP command language and the user interface of the RAP system in chapter 3.

The material presented in chapters 1 to 3 is sufficient for a naive use of the system. The remaining chapters aim at a deeper understanding of the system. Chapter 4 gives a description of the algorithms used in RAP, chapter 5 informs about advanced use of RAP, in particular about interactive debugging and adjustment of system parameters.

1.3. A Simple Example

In this section, we give a small example for a RAP specification and its execution by RAP. We assume that a specification has been written into a file which is read by the system. Let a file contain the following RAP specification text which is to be understood as a simple specification of lists of natural numbers together with a function which computes the length of such lists.

```
type LIST

 basedon NAT

 sort    List

 cons    atom: (Nat)List,
         list: (List,List)List

 func    number: (List)Nat

axioms all (l1, l2: List, n: Nat)

 number(atom(n)) -> 1,
 number(list(l1,l2)) -> add(number(l1),number(l2))
```

```
endoftype


noetherian LIST
complete LIST


task EVAL_NUMBER

 basedon NAT, LIST
 unknown n: Nat

goals

 number(list(list(atom(1),atom(2)),list(atom(3),atom(4)))) = n

endoftask


task TAB_NUMBER

 basedon NAT, LIST
 unknown l: List, n: Nat

goals

 number(l) = n

endoftask
```

This text is composed from **type** declarations, additional statements about types, and **task** declarations. A type defines data domains and function symbols on them in an algebraic way, a tasks consists essentially of a system of equations to be solved by the system, and the additional statements serve only for increasing the efficiency of the system.

The type LIST defines lists similar to the basic data structure of LISP. It is hierarchically based on the standard type NAT (keyword **basedon**). Newly introduced by LIST are objects of sort List (keyword **sort**) which are constructed as follows:

either a list is an atom (labelled with a natural)
or it consists of two other lists.

This recursive definition is expressed by the signature of constructor functions (keyword **cons**).

There is only one function working with lists (besides these constructors) defined here. "number" is intended to count the number of atoms in a list. Its signature is given following the keyword **func**.

The **axioms**-part formalizes the meaning of "number" by giving two equations. The axioms of LIST form a confluent term rewriting system, as presupposed by RAP. To indicate this difference to equational logic, the axioms are written with an arrow instead of the equality sign. The term rewriting relation generated by LIST is, in addition, noetherian and complete, the lines "noetherian LIST" and "complete LIST" state this behaviour.

The two tasks formulate questions which are intended to verify whether "number" actually does count the atoms of a list. A task contains the following components

a list of the used types (keyword **basedon**)
a declaration of unknown variables
a system of equations constructed from the unknowns and the functions
defined within the used types.

The task EVAL_NUMBER asks for an evaluation of the function "number" with a given list as its argument, the task TAB_NUMBER asks for a table of all pairs (argument, value) for "number".

When the RAP system is invoked, given the input file described above as a parameter, it issues the following messages.

```
RAP System Version 2.0-3 SUN/UNIX
Reading standard type declarations.
Reading type LIST.
Reading task EVAL_NUMBER.
Reading task TAB_NUMBER.
Compiling axioms into internal form.
RAP>
```

This means that RAP has checked the syntax of the input text and is now ready for executing commands. Below a short dialogue with the system is shown. User input is printed boldface.

A simple command is "dir", displaying the directory of the types and tasks known within the current session:

RAP> **dir**
Types:
BOOL NAT ID
LIST

Tasks:
EVAL_NUMBER TAB_NUMBER
RAP>

Note that the directory may contain besides the types from the input text some standard types. The "list" command lists up a type or task definition:

RAP> **list eval_number**

```
 task EVAL_NUMBER

   basedon NAT,LIST

   unknown n: Nat

 goals

   number(list(list(atom(1),atom(2)),list(atom(3),atom(4)))) = n

 endoftask
```

With the command "run", the interpreter is started. It tries to enumerate solutions for the given task:

RAP> **run eval_number**
*** Solution found ***

[n = 4]

CPU time: 0.05 secs

More solutions? **yes**
No more solutions found.
Total CPU time: 0.08 secs
RAP>

That was a rather easy task. TAB_NUMBER is more difficult:

RAP> **list tab_number**

```
 task TAB_NUMBER

   basedon NAT,LIST

   unknown l: List, n: Nat

 goals

   number(l) = n
```

```
endoftask
```

There is an infinite number of correct solutions for this task. The system starts to enumerate them. After each solution the user is asked whether (s)he wishes the enumeration to be continued. The solutions contain free variables generated by the system (which are indicated by an asterisk followed by a natural number).

```
RAP> run tab_number
*** Solution found ***

[l = atom(*2),
 n = 1]

CPU time: 0.03 secs

More solutions?  yes
*** Solution found ***

[l = list(atom(*5),atom(*11)),
 n = 2]

CPU time: 0.15 secs

More solutions?  yes
*** Solution found ***

[l = list(atom(*5),list(atom(*22),atom(*44))),
 n = 3]

CPU time: 0.70 secs

More solutions?  yes
*** Solution found ***

[l = list(list(atom(*14),atom(*28)),atom(*55)),
 n = 3]

CPU time: 0.95 secs

More solutions?  yes
*** Solution found ***

[l = list(atom(*5),list(atom(*22),list(atom(*86),atom(*170)))),
 n = 4]
```

CPU time: 3.53 secs

More solutions? **yes**
*** Solution found ***

[l = list(atom(*5),list(list(atom(*47),atom(*92)),atom(*181))),
 n = 4]

CPU time: 3.90 secs

More solutions? **no**
Total CPU time: 3.92 secs
RAP>

To get an impression of the internal steps performed by the interpreter, the last command is given again using the /DEBUG option. The narrowing algorithm is now traced on the screen. The prompt "RAP/DEBUG>" appears repeatedly after a certain amount of steps, empty input to this prompt causes the system to proceed.

RAP>**run/debug tab_number**
* NON-CONSTRUCTOR SOLUTION FOUND - REJECTED

* NEW GOAL NO. 1 GENERATED
number(*0) = *1
 WITH
[l = *0,
 n = *1]

RAP/DEBUG>
* CURRENT GOAL: NO. 1
number(*0) = *1
 WITH
[l = *0,
 n = *1]

* NARROWING STEP USING RULE LIST.1
1 = *1
 WITH
[l = atom(*2),
 n = *1]

RAP/DEBUG>
* EXPANSION OF VARIABLE *1
* SUCCESS
*** Solution found ***

```
[l = atom(*2),
 n = 1]

CPU time: 0.10 secs

More solutions?  yes
RAP/DEBUG>
* CURRENT GOAL: NO. 1
number(*0) = *1
 WITH
[l = *0,
 n = *1]

* NARROWING STEP USING RULE LIST.2
add(number(*3),number(*4)) = *1
 WITH
[l = list(*3,*4),
 n = *1]

RAP/DEBUG>
* NON-CONSTRUCTOR SOLUTION FOUND - REJECTED


* NEW GOAL NO. 2 GENERATED
add(number(*3),number(*4)) = *1
 WITH
[l = list(*3,*4),
 n = *1]

RAP/DEBUG>
* CURRENT GOAL: NO. 1
number(*0) = *1
 WITH
[l = *0,
 n = *1]


* GOAL EXHAUSTED

RAP/DEBUG>
* CURRENT GOAL: NO. 2
add(number(*3),number(*4)) = *1
 WITH
[l = list(*3,*4),
```

```
 n = *1]

* NARROWING STEP USING RULE LIST.1
add(1,number(*4)) = *1
 WITH
[l = list(atom(*5),*4),
 n = *1]

RAP/DEBUG>
* GOAL REWRITTEN USING RULE NAT.6
* GOAL REWRITTEN USING RULE NAT.5
* NON-CONSTRUCTOR SOLUTION FOUND - REJECTED


* NEW GOAL NO. 3 GENERATED
succ(number(*4)) = *1
 WITH
[l = list(atom(*5),*4),
 n = *1]

RAP/DEBUG>
* CURRENT GOAL: NO. 2
add(number(*3),number(*4)) = *1
 WITH
[l = list(*3,*4),
 n = *1]

* NARROWING STEP USING RULE LIST.2
add(add(number(*9),number(*10)),number(*4)) = *1
 WITH
[l = list(list(*9,*10),*4),
 n = *1]

RAP/DEBUG>
* NON-CONSTRUCTOR SOLUTION FOUND - REJECTED


* NEW GOAL NO. 4 GENERATED
add(add(number(*9),number(*10)),number(*4)) = *1
 WITH
[l = list(list(*9,*10),*4),
 n = *1]

RAP/DEBUG>
```

```
* CURRENT GOAL: NO. 2
add(number(*3),number(*4)) = *1
 WITH
[l = list(*3,*4),
 n = *1]


* GOAL EXHAUSTED

RAP/DEBUG>
* CURRENT GOAL: NO. 3
succ(number(*4)) = *1
 WITH
[l = list(atom(*5),*4),
 n = *1]

* NARROWING STEP USING RULE LIST.1
succ(1) = *1
 WITH
[l = list(atom(*5),atom(*11)),
 n = *1]

RAP/DEBUG>
* EXPANSION OF VARIABLE *1
* SUCCESS
*** Solution found ***

[l = list(atom(*5),atom(*11)),
 n = 2]

CPU time: 0.57 secs

More solutions?  no
Total CPU time: 0.58 secs
RAP> exit
```

1.4. Where to Find More Examples

A list of examples for RAP can be found in [Geser, Hussmann 85]. Larger experiments with RAP are described in [Geser 86] (the specification of a microprocessor) and in [Padawitz 87] (the specification of a programming language for concurrent systems). A short summary of the experiences with the first version of RAP is [Geser, Hussmann 86].

CHAPTER 2

RAP Specification Language

2.1. General Structure

An algebraic specification in RAP is a system composed of a hierarchy of (algebraic, abstract) **types** together with a number of so-called **tasks**. A type defines a signature (i.e. a set of sort-identifiers and a set of function-identifiers with domain and codomain) together with a set of positive conditional axioms; a task consists of a system of equations together with a declaration of "unknown"-variables. Types are built up in a modular way: a type may make use of (i.e. be "based on") another type. Similarly, tasks are defined based on a specific type system.
In the following the context-free syntax of RAP specifications is defined using an extended Backus-Naur notation. The context conditions and the semantics of a specification are given in an informal style.

2.2. Conventions for the Syntax Description

We use an extended Backus-Naur form for the description. Nonterminals are enclosed in angle brackets (<>). Concatenation of strings is denoted by juxtaposition. Further meta-symbols of the syntax description are
{ } }* |
The string A | B denotes the choice between the strings A and B, { A } the choice between A and the empty string, { A }* the repetition of A an arbitrary number of times.

We distinguish between the lexical and the context-free part of the syntax in order to define the role of delimiter symbols.

2.3. Lexical Elements

Lexical elements are the basic parts of a specification text. There are 6 classes of lexical elements: identifiers, keywords, numbers, identifier-literals, special symbols and display-formats. A display-format is a special kind of string used for the definition of mixfix operations. The syntax is as follows:

```
<id>        ::= <letter> { <letter> | _ | <digit> }*
<keyword>::= all | axioms | basedon | cons |
              complete | display | endoftask | endoftype |
              except | exist | func | goals | hidden |
```

noetherian | sort | task | type | unknown
<number>::= <digit> { <digit> }*
<Id-literal>::= ' <id>
<special symbol>::= = | (|) | , | : | => | -> | &
<display-format>::= " { <display-substring> { _ <display-substring> }* } "
<letter> ::= A | B | C | ... | Z |
a | b | c | ... | z
<digit> ::= 0 | 1 | 2 | ... | 9

The following additional rules hold for the lexical elements:

- A keyword is not allowed as an <id>.
- Keywords have to be written lowercase.
- Numbers have to consist of less than 10 digits.
- Identifiers are considered to be equal if they coincide in the first 24 characters. Upper- and lowercase characters are distinguished, for example the identifier "Nat" is not equal to the identifier "NAT".
- As a display-substring every sequence of printable ASCII characters is allowed which does not contain the characters """ (double-quote) and "_" (underscore). A display-substring must not exceed 24 characters. See section 2.9 of this chapter for details on the use of display-formats.

Subsequent lexical elements are delimited by strings consisting of one or more of the following parts:

- blank, line feed, horizontal tabulator
- comments.

Delimiters which stand before or after a special symbol may be omitted. The delimiter rules do not apply to display-substrings. The delimiter characters above may appear within a display-substring as ordinary characters.

Any text not containing the character "}" is admitted as a **comment**. Comments have to be enclosed in braces "{" and "}". Comments may be inserted between lexical elements without changing the meaning of the text.

2.4. Context-free Syntax

A RAP specification text has the following form:
<system> ::= <unit> { <unit> }*
<unit> ::= <type> | <task> | <prop>

The specification text has to be composed from units which may be

- **type**s defining function symbols and their behaviour in an algebraic way,
- **task**s defining questions about the behaviour of the specified functions, and

- **Properties** of the specified functions which can be used to control execution implicitly and to define a mixfix representation for function symbols.

2.4.1. type Syntax

```
<type>     ::= type <id>
                { <primitives> }
                { <signature> }
                { <axioms> }
               endoftype
<primitives>::= basedon <id-list>
<id-list>   ::= <id> { , <id> }*
<signature>::= { <sort-decl> | <funct-decl> | <visib.-restr> }*
<sort-decl>::= sort <id-list>
<funct-decl>::= func <functity> { , <functity> }* |
               cons <functity> { , <functity> }*
<functity>::= <id-list> : <args> <id>
<args>     ::= { ( <id-list> ) }
<visib.-restr>::= hidden <id-list>
<axioms> ::= axioms { all ( <vars> ) } <axiom> { , <axiom> }*
<vars>     ::= <var-decl> { , <var-decl> }*
<var-decl>::= <id-list> : <id>
<axiom>  ::= { ( <id> ) } { <premises> } <rewrite>
<premises>::= <equation> { & <equation> }* =>
<equation>::= <term> { = <term> }
<rewrite> ::= <term> { -> <term> }
<term>     ::= <number> | <Id-literal> | <id> | <application>
<application>::= <id> { ( <term> { , <term> }* ) }
```

2.4.2. task Syntax

```
<task>     ::= task <task-id>
                { <primitives>  }
                { <unknown-decl> }
                { <goals> }
               endoftask
<unknown-decl>::= unknown <vars>
<goals>    ::= goals { exist ( <vars> ) } <equation> { , <equation> }*
```

2.4.3. Properties Syntax

<prop> ::= <noetherian> | <complete> | <display>
<noetherian>::= noetherian <id> { except <id-list> }
<complete>::= complete <id> { except <id-list> }
<display> ::= display <id> <funct-display> { , <funct-display> }*
<funct-display>::= <id> = <display-format>

2.5. Context Conditions

2.5.1. General Remarks

For the sake of readability, context conditions are defined by English text here. Note that an attributed grammar may be extracted from the text in a straightforward way. The position of a syntactic unit of interest within a context is indicated by schemes of specification text containing meta-variables for identifiers and dots (...).

2.5.2. Declarations of Identifiers

The occurrences of identifiers in specifications are divided into occurrences for **declaration** and for use. At a declaration occurrence, the **kind** of an identifier is associated with it (e.g. function-identifier, sort-identifier). The following rules cover the possibilities for declaring identifiers.

    type T ... endoftype
    task T ... endoftask

declares T as a type resp. task identifier. Note that the declaration of T does not end before the "endoftype" resp. "endoftask".

    sort ... s ...

declares s as a sort identifier.

    func ... f ...: ...
    cons ... f ...: ...

declare f as a function identifier.

    all (... v ...: ...)
    exist (... v ...: ...)
    unknown ... v ...: ...

declare v as a variable identifier.

axioms ... (A) ...

declares A as an axiom identifier.

2.5.3. Use of Identifiers

The following rules cover the possibilities for using identifiers. All identifiers have to be declared before they are used. The kind of an identifier in its declaration and its use have to be the same.

func ...: (s1, ..., sn)s
cons ...: (s1, ..., sn)s

use s1, ..., sn, s as sort identifiers.

hidden ... h ...

uses h as a sort or function identifier.

all (...: s)
exist (...: s)
unknown ...: s

use s as a sort identifier.

Identifiers occurring within a term are used as function or variable identifiers (or unknown-identifiers, within a task).

basedon ... T ...
display T ...
noetherian T ...
complete T

use T as a type identifier.

noetherian ... except ... A ...

uses A as an axiom identifier.

complete ... except ... f ...
display ... f = ...

use f as a function identifier.

2.5.4. Export and Import of Identifiers; Visibility

- An identifier is called **declared in** (type, task, resp.) T if its declaration takes place within the declaration of T (i.e. between "type" and "endoftype" or "task" and "endoftask")

- Each sort and function i declared in T is called **exported** by T if it is not used in a "hidden ... i" within T.

- All sort and function identifiers which are exported by T are imported within another type or task by "basedon ... T".

- Type and task identifiers are global, i.e. they are visible everywhere after their declaration.

- All other identifiers are local, i.e. they are visible between their introduction by a declaration or an import and the end of the unit where they were introduced.

- An identifier may be used only when it is visible.

- As soon as an identifier is visible, it must not be declared once more.

2.5.5. Miscellaneous Constraints

- Type-, task-, and axiom identifiers must be uppercase (as they may be used within the command language).

- An identifier i in

  cons ...: (...)i
  hidden ... i ...

  must be declared within the surrounding unit. Note that the restriction for "cons" means that all constructor functions of a sort have to be declared in the same type where the sort is declared.

- The same kind of <prop> may use a type id at most once.

- In a display definition

  display ... f = "s1 _ ... _ sn" ...

  the number of substrings n has to match the declaration of f, i.e. f has to be declared having n - 1 arguments.

2.5.6. Strong Use of Sorts

- If v is declared by

  all (... v ...: s)
  exist (... v ...: s)
  unknown ... v ...: s

  then the sort of the term v is s.

- If f is declared by

  cons ... f ...: (s1, ..., sn)s
  func ... f ...: (s1, ..., sn)s

  then the sort of the term f(t1,...,tn) is s, and t1, ..., tn have to be terms of sort s1, ..., sn, resp.

- A <number> has the sort "Nat", an <id-literal> has the sort "Id".

- Within an equation t = u or a rewrite t -> u, t and u have to be of the same sort.

2.6. Standard types

The context conditions and the semantics of a given specification are evaluated after prefixing the specification with the declarations of some standard types. Important consequences are:

- The sorts and functions declared in the standard types are available in every user-defined type or task just by a basedon-statement.
- The system provides efficient standard implementations for the functions of the standard types. E.g. the arithmetical operations are executed on closed terms by machine instructions rather than symbolic computation.

There are three standard types:

BOOL (truth values),
NAT (natural numbers)
ID (identifiers).

Their defining text is listed in appendix 1.

2.7. Semantics of a type system

A type together with all the types referenced directly or indirectly by "basedon" forms a typs system. The composed type of a type system is a large type defined by putting together the signatures and axioms of all the types within the system. The denotational semantics of a type system is defined as the model class of the composed type, i.e. the class of all heterogenous algebras where

- with every sort a nonempty carrier set is associated
- with every function symbol a function on the corresponding carrier sets is associated
- every axiom holds for arbitrary instantiations of the variable-identifiers with elements of the carrier sets. The rewrite-arrow (->) is interpreted as equality in the models.

For more details on the semantics see [WPPDB 83].

The conditional narrowing algorithm is correct with respect to this semantics. It is complete, too, if the axioms form a confluent term rewriting system ([Hussmann 85]). A **solution** of a task is a substitution of terms for the unknown-identifiers such that the instantiated goals hold in the model class of the type system.

The differences between classical total algebra semantics and the RAP semantics are:

- The designation of constructor functions in the "cons"-part of a type. This constructor signature is used by the conditional narrowing algorithm: it looks only for solutions built of constructor functions.
- RAP considers the axioms to be oriented from left to right (as expressed by the arrow of the axiom-syntax). This does not induce differences in the model-theoretic semantics, but in the deduction calculus used. RAP relies on term-rewriting techniques. This means that validity and derivability coincide only completely when the axiom set forms a confluent ("Church-Rosser") and noetherian term rewriting system. More details about this restriction can be found in [Hussmann 85].
- RAP does not allow only hierarchical models as [WPPDB 83] does. The language constructs for modular specifications ("basedon", "hidden") are used only for additional context checks, but not for the semantics. Nevertheless, RAP semantics is correct w.r.t. hierarchical semantics.

2.8. Semantics of a task

The semantics assigned to a task is a set of substitutions (of terms to variables),called the **solutions** of the task. A given task is characterized by

the type system it refers to
a set of variables, divided into unknown- and exist-variables
a set of equations (goals).

A substitution σ for the unknown-variables is called a solution of the task iff there is a substitution τ for the exist-variables such that the goals, substituted by $\sigma \cup \tau$, are valid in all models of the type system.

Note that the only difference between unknown- and exist-variables is that the substitutions found for the unknown-variables are regarded as the solution and therefore are shown to the user.

2.9. User-ensured Properties

A given type system may have a number of properties which can be used for several purposes, for instance to increase the efficiency of the interpreter. Besides that, they are interesting for a semantic analysis of specifications. Among them the following are of particular importance:

- the axiom set considered as a rewrite system is terminating (noetherian)
- the axiom set is sufficiently complete.

A later version of RAP is planned to provide sufficient criteria for checking these properties. In the current implementation the user is required to state them by input text.

A statement of the form

    noetherian T except A1, ..., An

assumes that the axioms of T without the axioms named A1, ..., An may be taken as part of a global rewriting system such that this rewriting system does not allow infinite rewriting paths.

A statement of the form

    complete T except f1, ..., fn

assumes that all non-constructor function symbols declared in T except f1, ..., fn are defined in a sufficiently complete way, i. e. they can be removed from any ground term by rewriting with the axioms.

> *Please check carefully whether your specifications satisfy such properties. If you are in doubt, try the behaviour of the system with such assumptions, anyway. They may lead to a big gain in efficiency. But note that solutions may be lost if such statements are given incorrectly.*

2.10. Display definitions

Display definitions are a mechanism to define a mixfix syntax for terms, i.e. a more readable representation for terms. For instance, it is convenient to read "if B then S1 else S2 fi" instead of "if(B,S1,S2)" or "3 + 4" instead of "add(3,4)".

To circumvent the difficulties of parsing such expressions, the current version of RAP uses mixfix notation only for system output, not for input. Moreover, the use of display definitions can be switched off and on (via the SET PARAM DISPLAYUSED command).

Display definitons are grouped together into units defining a representation for the functions of one specific type ("display T ..."). A display definiton for a function symbol f with n arguments has the form

f = "s_1 _ ··· _ s_{n+1}"

where the s_i are arbitrary (possibly empty) strings of printable ASCII characters. Obviously, double-quotes and underscore must not occur within such a string. Typical display definitions are:

f = "" - a constant which will not be shown at all
g = "_" - a unary function symbol which will not be shown at all (typically an injection)
if = "if _ then _ else _ fi" - the usual representation of an "if"-function
add = "_ + _" - the usual representation for addition.

For the standard types, also standard displays are defined. Display definitions can be inspected during a RAP session by a SHOW DISPLAY command.

For some special characters there is defined a special interpretation if they occur within a display-substring:

horizontal tab: equivalent to a space
line feed: ignored
backslash: interpreted as a line feed during output.

For instance, the display definition

if = "if _ \ then _ \ else _ \ fi"

leads to a printout of the term "if(B,S1,S2)" as

```
if B
  then S1
  else S2
fi
```

CHAPTER 3

RAP COMMAND LANGUAGE

3.1. General Structure of the System

The RAP system reads an input file containing text in the RAP specification language, performs syntax checks on this text, and subsequently enters an interactive interpreter for tasks contained in the input text. If the system detects syntactical errors during the reading phase, the second phase is skipped.

The internal status of the system consists of

a representation of the input
a bookkeeping for summaries of task executions.

There are basically two operating modes of the system. In the standard ("top-level") mode there are commands available for

inspection of the system status
invoking the interpreter for a task
adjusting system parameters.

The so-called "DEBUG"-mode can be entered only during the execution of a task. In this mode a number of additional commands allows

watching internal steps of the interpreter
inspection of the interpreter status
interactive control on the interpreter.

Section 3.4 contains a list of the standard RAP commands available together with a description of their effect. The DEBUG commands are explained in chapter 5.

3.2. Interactive Help about RAP Commands

The RAP system offers two features for on-line assistance about RAP commands: The HELP command and a menu-like technique for command browsing and command selection.

The HELP command gives interactive access to RAP command documentation. It is designed for information purposes only, no action of the system is invoked. The manual text is organized in a tree-like structure, the user is guided to select the actual text by entering keywords.

The second assistance feature serves for completing incomplete commands interactively. It is invoked automatically if an incomplete command is entered, but it can also be activated explicitly by typing a question mark ("?") at any position of a command. This tool gives the menu of all possible continuations of the incomplete command. The user can select from this menu (or quit by entering an "end-of-file"-character).

3.3. Prompts and Control Keys

The system uses different prompts to indicate its current status.

RAP> Top-level prompt for standard commands.

RAP/DEBUG>
Prompt during DEBUG mode for standard and DEBUG commands.

RAP/HELP> Prompt during the interactive HELP facility for keywords leading to additional information.

The RETURN key is in general interpreted by the system as something like "continue normally". It has the following effects:

no effect - in standard mode
equivalent to the DEBUG command "step" - in DEBUG mode
leave the HELP facility - during HELP facility

The only control key interpreted by RAP is the "logical end-of-file" (CTRL/D in UNIX, CTRL/Z in VMS). It is taken as a request for terminating the current activity, depending on the current mode:

leave the RAP system - in standard mode
leave the DEBUG mode (and the current execution) - in DEBUG mode
leave the HELP facility - during HELP facility
stop the completion of an incomplete command - during command completion.

3.4. RAP Commands

3.4.1. RAP Command Syntax

The general syntax of RAP commands is defined by the following extended BNF grammar:

<RAP-command>::= <command> { <subcommand> }* { <option> }* { <parameter> }
<subcommand>::= <id>
<command>::= <id>
<option> ::= / <id> { = <value> }

<parameter>::= <value> { . <value> }
<value> ::= <id> | <number>

The parts of a command are separated by blanks (space key). The "/"- and "="-characters are also taken as separators. <id> is defined as in the RAP specification language, but upper- and lowercase characters are not distinguished. RAP commands, subcommands and options can be abbreviated. The abbreviation has to be unique within the whole command set (including the DEBUG commands). The interactive command completion facility indicates allowed abbreviations by displaying the relevant part of the keywords uppercase.

3.5. Standard RAP Commands

HELP

Display information about RAP commands.

This command invokes an interactive manual about the RAP command language. Empty input (RETURN key) returns from help mode. A HELP command without a parameter lists the RAP commands and asks for entering a keyword (command name) to obtain more information.

Parameter: To obtain information about a specific command, the command (including subcommands) can be given to HELP as a parameter.

DIR

Directory of known types and tasks.

A list of all known type- and task-names is displayed, i.e. a list of all identifiers for which a type- or task-declaration has been read from the input file.
The /TASKS option gives additional information about previous runs of the known tasks.

Options: /TASKS /TYPES /ALL (Default: /ALL)

/TYPES Give a listing of the type names only.

/TASKS Give a listing of the task names only. The task names are displayed as a table which contains additional information about

- number of runs
- whether a solution has been found.

/ALL Give a listing of all type and task names.

LIST

Display the defining text of a type or task.

Given a known type- or task-name as parameter, this command displays the textual representation of the type or task in a pretty-printed manner.

Parameter: The parameter of a LIST command has to be a type- or task-name known in the current session. This means that a type- or task-declaration with this name is contained in the file read initially by the system. A list of all known type- and task-names can be obtained by the DIR command.

RUN

Conditional Narrowing Algorithm.

The RUN command invokes the Conditional Narrowing procedure as an interpreter for a task. By default, it searches (in a breadth-first manner) for solutions. When a solution is found, it is displayed and the user is asked "More solutions?". The following responses are allowed to this prompt:

- "y" (or the empty input) : continue and search for more solutions
- "n" : stop the run, do not look for more solutions
- "d" : continue and search for more solutions, but in DEBUG mode.

The behaviour of the interpreter is controlled by the system parameters (see SHOW PARAM and chapter 5) and the options given on the command line.

Parameter: The parameter of a RUN command has to be a name of a known task, i.e. of a task which is contained in the input file initially read by the system. A list of all known type- and task-names can be obtained by the DIR command.

Options: /DEBUG /NODEBUG (Default: /NODEBUG)
/TRACE /NOTRACE (Default: /NOTRACE)
/INQUIRE /NOINQUIRE (Default: /INQUIRE)
/TIMELIMIT=<value> (Default: /TIMELIMIT=INFINITE)
/DEPTHLIMIT=<value> (Default: /DEPTHLIMIT=INFINITE)
/SOLNLIMIT=<value> (Default: /SOLNLIMIT=INFINITE)
/SEARCH=<value> (Default: /SEARCH=BF)

/DEBUG Run the interpreter in DEBUG mode. Each internal step (narrowing and optimizations) is indicated by messages on the screen. The amount of information given here is dependent on the debug level which can be adjusted by the SET PARAM DEBUGLEVEL command. When a fixed amount of computation (dependent on the debug level) is done, the algorithm is stopped and the user is prompted for commands by "RAP/DEBUG>". At this stage all RAP commands execpt RUN are admitted and a number of additional commands become available. See chapter 5 for details.

/NODEBUG Run the interpreter in standard mode without debugging.

/TRACE Produce a trace file. Each internal step (narrowing and optimizations) is reported onto a trace file the name of which is installation dependent. The amount of information reported depends on the trace level which can be adjusted by the SET PARAM TRACELEVEL command.
The user interaction of the RUN remains unchanged.

/NOTRACE Do not produce a trace file.

/INQUIRE Ask for "More solutions?" if a solution has been found.

/NOINQUIRE Continue automatically after a solution has been found; do not ask.

/TIMELIMIT=<value>
Define a time limit for the current run. The time limit is given in CPU seconds or by the keyword INFINITE (default value).
If the time limit is exceeded, the interpreter is stopped and RAP returns to standard mode.

/DEPTHLIMIT=<value>
Define a depth limit for the current run. The depth limit is given as a natural number or by the keyword INFINITE (default value).
If a goal is generated which has a deeper position within the proof tree than the current depthlimit, this goal is discarded. This way, the DEPTHLIMIT options allows to restrict the search to a finite part of the proof tree.

/SOLNLIMIT=<value>
Define a solution limit for the current run. The solution limit is given as a natural number or by the keyword INFINITE (default value).
This option allows to stop the interpreter after a fixed number of solutions has been found.

/SEARCH=<value>

Define the search strategy for the current run. There are three possible values for this option:

- BF : breadth-first search (default)
- DF : depth-first search
- SF : smallest-first search.

See chapter 4 (section 4.2.5.1) for more details on search strategies.

SHOW

Inspection of the system status.

In any mode, the SHOW command can be used to have a look at the current values of system parameters (by SHOW PARAM) and at the internal bookkeeping about runs (by SHOW SUMMARY). During DEBUG mode, SHOW also serves for inspection of the current status of the narrowing algorithm (proof tree, goals etc.). For more information on debugging and system parameters see chapter 5.

Subcommands:

PARAM SUMMARY DISPLAY RULE

SHOW PARAM

Show system parameters.

This command displays the current settings of various parameters which can be used to influence the behaviour of the Conditional Narrowing algorithm. Information about the setting of a single parameter can be obtained by a SHOW PARAM <parameter-name> command.
The system parameters can be set up by SET PARAM <parameter-name> commands. For more information about the meaning of the parameters see chapter 5.

Subcommands:

DEBUGLEVEL TRACELEVEL OPTIMIZATIONS REDEXSELECTION CONSTRSOLNS GARCOL DISPLAYUSED

SHOW RULE

Show rewrite rules (axioms).

This command looks for the defining text of rewrite rules. It needs as a parameter

- a name of a type (displaying all axioms of this type) or
- a name of an axiom.

Parameter: A parameter for SHOW RULE may be

- a type identifier known in the current RAP session
- a pair of a known type identifier and an axiom name.

Such pairs have to be written as <typename>.<axiomname>, i.e. separated by a dot and without blank characters.

SHOW DISPLAY

Show display definitions for mixfix output.

This command can be used for inspection of "display" statements issued in the input file. If for a function symbol there is no display defined this way, the standard (prefix) representation is shown. The display definitions are used by the system for pretty-printing of terms. Mixfix mode can be entered by a SET PARAM DISPLAYUSED ON command. See also chapter 5 and section 2.10.

Parameter: The parameter of a SHOW DISPLAY command has to be the name of a type known in the current RAP session.

SHOW SUMMARY

Show summary of previous RUNs of a task.

This command remembers solutions of a given task which have been found in earlier RUNs. It is particularly useful for documentation of the results of RAP sessions. The combination of options /LONG/ALL/FILE e.g. writes a survey of the results achieved for a given task onto a file.

Parameter: The parameter of a SHOW SUMMARY command has to be a task identifier known in the current RAP session.

Options: /FILE /NOFILE (Default: /NOFILE)
/ALL /LAST (Default: /LAST)
/SHORT /LONG (Default: /SHORT)

/FILE Write the summary onto a file instead of the terminal. The name of the file is installation dependent.

/NOFILE Show the listing on the terminal only.

/SHORT Give only a listing of the substitutions found.

/LONG Include more information:
- current setting of parameters and options
- if additionally the /FILE option is in effect: defining text of the task.

/LAST Give a summary of the last RUN only.

/ALL Give a summary of all previous RUNs within the current RAP session.

Warning: *For technical reasons, the current implementation uses a fixed file name for the output file. Thus, a second SHOW SUMMARY within a session will overwrite the result of the first one.*

SET

Set up system status.

The SET command can be used in any mode for setting up system parameters (see SET PARAM). In DEBUG mode, additional applications of SET become available.

SET PARAM

Set up system parameters.

By SET PARAM <parameter-name> commands global parameters can be set up which change the behaviour of the interpreter. In the following, only the most important system parameters are explained. For a full explanation of system parameters, see chapter 5.

Subcommands:
DEBUGLEVEL TRACELEVEL OPTIMIZATIONS REDEXSELECTION CONSTRSOLNS GARCOL DISPLAYUSED

SET PARAM DEBUGLEVEL

Set up DEBUG level.

The DEBUG level controls the amount of output displayed in DEBUG mode. There are three possible levels: BRIEF, STANDARD (default), DETAILED. See chapter 5 for more details.

Parameter: One of the keywords BRIEF, STANDARD, DETAILED.

SET PARAM TRACELEVEL

Set up TRACE level.

The TRACE level controls the amount of output which will be written onto a trace file, if the /TRACE option is used. There are three possible levels: BRIEF, STANDARD (default), DETAILED, which are interpreted similar to the DEBUGLEVEL. See chapter 5 for details.

Parameter: One of the keywords BRIEF, STANDARD, DETAILED.

SET PARAM DISPLAYUSED

Enable/disable usage of mixfix notation.

If the usage of mixfix notation is activated using this command, RAP interprets the "display" statements contained in the input file and pretty-prints terms in mixfix notation according to this statements. See also the SHOW DISPLAY command.
For mixfix notation see also section 2.10.

Parameter: One of the keywords ON, OFF.

EXIT

Leave the RAP system.

The EXIT command terminates the RAP system and returns control to the calling operating system. A logical "end-of-file" (installation dependent) is equivalent to EXIT.

3.6. Error Messages

The error messages issued by the RAP system contain a word indicating the severity of the error (WARNING, ERROR, FATAL ERROR or SYSTEM ERROR) and a text which tries to fix the reason and the location of the error.

3.6.1. RAP specification language errors

These error messages are composed of several lines. The line of the source text where the error was detected is displayed, together with its line number. In a second line below, the exact position is marked the error has been detected at. As a mark providing additional information a lexical element is used, which is in most cases equal to the source text above it, e.g.:

```
   120     func f : s
                   s
------ERROR: UNDECLARED IDENTIFIER.
```

In some cases the lexical element used for marking the error position is not equal to the source text above it. This means that the position the error has been detected at differs from the position the error results from, e.g.:

```
   240      f(x,y) = 3,
                  f
------ERROR: WRONG NUMBER OF ARGUMENTS.
```

Here the error was detected after reading the arguments of f, but the reason for the error is that f is used with a number of arguments different from its declaration. (The reader familiar with compiler constrcution may notice here that RAP uses a rigorous one-pass compiler architecture.)

Sometimes the system gives additional information about the error. If an error occurs in the context-free specification syntax, the system prints a list of lexical elements allowed at the error position (after the word "EXPECTED:"), if a sort-conflict occurs, the conflicting sorts are named, etc.

3.6.2. RAP Command Language Errors

Error messages issued during the syntax check for commands have a simpler structure. In a blank line printed below the user's input the position of the error is marked by the sign "^". (This line is sometimes omitted.) Then the line containing the error message follows. E.g.:

```
RAP> run/file t2
         ^
------ERROR: UNKNOWN OPTION .
```

CHAPTER 4

THE ALGORITHMS USED IN RAP

4.1. Preliminaries

In this chapter, we give a precise definition of the conditional narrowing algorithm used by RAP. No effort is made here to justify the correctness and completeness of the algorithm, for theoretical results see [Hussmann 85] and [Padawitz 86]. The descriptions below explain in detail the behaviour of RAP, nevertheless the actual implementation differs in some technical points from the presentation given here. We use the notions of [Huet, Oppen 80] (terms, substitutions, term rewriting, etc).

4.2. Conditional Narrowing

4.2.1. The Given Input

Throughout this chapter we assume a fixed type system and a fixed task system to be given. For the purposes of solving equations, there is no need to refer to signatures, type hierarchies, etc. From the given type system a finite set R is extracted containing only the rewrite rules. Elements of R are of the form:
[t1 = t1' & ... & tn = tn' => l -> r]
where ti, ti', l, r are terms.

Similarly, the given task T is seen as a sequence of equations:
T = [t1 = t1', ..., tn = tn'].
In a more abstract view T is a set of equations, but for an explanation of the behaviour of RAP it may be more illustrating to respect the order of the equations (as the system does).

The basic objects the interpreter deals with are so-called **goals**. A goal is a sequence equations together with a substitution, it has the form
[t1 = t1', ..., tn = tn' **with** σ]
where ti, ti' are terms, σ a substitution.

From the given task T the **start goal** is derived by adding the identity substitution *id* as a **with**-part (*id*(x) = x for all variables x). To avoid name clashes, the variables of the start goal are renamed using "new" variables occurring nowhere else.

From the user-ensured properties contained in the input the following information is extracted:

a subset NoethRules of R (rewriting within NoethRules always terminates)
a set ComplFuncts of function symbols (declared to be defined completely).

4.2.2. Basic Operations: Unification, Narrowing

The Conditional Narrowing algorithm uses two elementary operations for deriving solutions from T in the theory defined by R.

4.2.2.1. Unification

A unification step detects solutions in the empty theory, i.e. solutions which can be found without consideration of the rule set R.

If for a goal
 G = [t1 = t1', ..., tn = tn' **with** σ]
there is a substitution τ such that τti = τti' for all i, then a unification step is applicable to G and delivers as a result the solution
 Unif(G) = $\mu\sigma$
where μ is the most general unifier (in Robinsons's sense) corresponding to τ.

4.2.2.2. Narrowing

A narrowing step derives from a goal G another goal by application of a rule taken from R. To make a rule applicable, the goal G may be specialized ("narrowed") by application of a substitution. In the framework of conditional rewrite rules, narrowing moreover is combined with a resolution-like technique.

If for a goal
 G = [S **with** σ]
where S = [t1 = t1', ..., tn = tn']
there is a subterm S/u of one of the equations of G and a rule r∈R
 r = [P => t -> t']
where P = [p1 = p1' & ... & pm = pm']
such that

S/u is not a variable
S/u and l are unifiable, i.e. there is a substitution τ such that τ(S/u) = τt,

then a narrowing step is applicable to G and delivers as a result a new goal
 Narr(G,u,r) = [μ(P & S[u←t']) **with** $\mu\sigma$].
where μ is the most general unifier corresponding to τ.

Note that the definition above uses a (straightforward) extension of the concepts of subterm replacement and substitution to systems of equations.

4.2.3. Conditional Narrowing Algorithm: First Version

The conditional narrowing algorithm in its purest form consists in a systematic search for a sequence of goals
 G_1, G_2, ..., G_n
such that G_1 is the start goal, G_{i+1} is derivable from G_i by a narrowing step, and a solution

is derivable from G_n by a unification step.

Th recursive procedure below performs such a search by exhaustion of the whole proof tree constituted by the possible narrowing steps. Given a task T, the computation is started by
CNA_0(StartGoal(T)).

procedure CNA_0 (**Goal** G):

```
begin
        if Unif(G) is defined
                then DisplaySolution(Unif(G))
        fi;
        for all u∈Occ(G), r∈R
                do
                        if Narr(G,u,r) is defined
                                then CNA0(Narr(G,u,r))
                        fi
                od
end;
```

4.2.4. Conditional Narrowing Algorithm: Second Version

One difficulty of the narrowing algorithm is that it has to cope with nonterminating paths in the proof tree. RAP uses an implementation open for several search strategies. For this purpose, the algorithm manages a sequence Q of active goals from which the next goal to consider is chosen depending on a search strategy. To ease the description of the algorithm, Q is assumed to contain unique **goal numbers** instead of the goals. The goals themselves are stored in an array G indexed by these numbers. In a second table NS the so-called **narrowing status** is associated with the goal numbers, i.e. information about the narrowing steps which can yet be performed with the resp. goal. Below a second version of CNA is given using this refinement. We asume that for a given goal G = [S **with** σ] the function NarrRed computes the set of possible narrowing **redices** within G. A redex is a pair of an occurrence and a rule:
NarrRed(G) = { $\langle u,r \rangle \in Occ(S) \times R$: Narr(G,u,r) is defined }.
When a goal enters Q, it is associated with the narrowing status NarrRed(G). During the algorithm, redices are removed from the narrowing status step by step.

procedure CNA_1 (**Task** T):

```
begin
        Nat goalno := 0;
        Seq Nat Q := [];
        Nat Array Goal G := init;
```

```
    Nat Array Set Redex NS := init;

    procedure EnterNewGoal (Goal g):
    begin
        if Unif(g) is defined
            then DisplaySolution(Unif(g))
        fi;
        goalno := goalno + 1;
        G[goalno] := g;
        NS[goalno] := NarrRed(g);
        Q := Insert(goalno,Q)
    end;

    EnterNewGoal(StartGoal(T));
    while Q ≠ []
        do
            let Nat curgoal = first(Q);
            co Current goal number: curgoal oc
            if NS[curgoal] = []
                then Q := Q - [curgoal] co Goal exhausted oc
                else
                    let <u,r> = some elem of NS[curgoal];
                    EnterNewGoal(Narr(G[curgoal],u,r));
                    co New goal number goalno generated oc
                    NS[curgoal] := NS[curgoal] - [<u,r>];
            fi
        od
end;
```

4.2.5. Strategies

RAP offers several variants of CNA via adjustable system parameters. Two of them can be easily explained on this level. See also chapter 5 on system parameters.

4.2.5.1. Search Strategy

The way of realizing Insert(goalno,Q) determines the order in which the proof tree is visited. As the next goal to consider is always taken from the beginning of Q, we can achieve

depth-first serach by adding goalno to the end of Q
breadth-first search by adding goalno to the beginning of Q.

Depth-first search treats Q as a stack of goals, breadth-first search uses Q as a queue. Depth-first search is a more "programmed" search able to find complex solutions rather early at the cost of being unfair, breadth-first search is a slow and safe search with a more "logical" taste. A third variant is available for test purposes: "Smallest-first" search carries out Insert(goalno,Q) by sorting goalno into Q according to the size of the corresponding goal. Moreover, it is sometimes ineterestting to restrict the search space to a finite initial part of the proof tree. This can be done using the /DEPTHLIMIT-option of the RUN command.

4.2.5.2. Redex Selection for Narrowing

The function NarrRed determines the set of redices (pairs <occurrence,rule>) where narrowing can be applied. We decided to represent such a set by a sequence lexicographically ordered by

the leftmost-innermost ordering of the occurrences
the user-given order of the rules.

The algorithm above uses the classical "full" narrowing strategy, i.e. all redices found within a goal are tried. The more efficient variant of leftmost-innermost narrowing [Fribourg 85] can be obtained by restricting the result of NarrRed to the first part belonging to one single occurrence. Leftmost-innermost narrowing considers only those narrowing steps which concern the first occurrence in the leftmost-innermost ordering.

4.2.6. Optimizations

The efficiency of the narrowing algorithm can be improved by a number of additional steps. Some of them rely on the following notion:
A function symbol f is called **irreducible** iff there is no rule r ∈ R such that
r = [P => f(t1, ..., tn) -> t]
(with arbitrary terms t1, ..., tn, t, sequence of equations P).
One optimization uses this notion to detect unsolvable equations, a number of optimizations try to simplify new goals before they are entered into the goal queue, a last group of optimizations compares new goals with stored information.

4.2.6.1. Unsatisfiable Goals

We call a goal **unsatisfiable** if it contains an equation
f(t1, ..., tn) = g(s1, ..., sm)
where f and g are different irreducible function symbols (ti, sj arbitrary terms). As a special case, equations between different Nat- and Id-literals lead to unsatisfiability.

4.2.6.2. Decomposition

A decomposition of a goal G is defined if G = [e_1, ..., e_m **with** σ] and
e_j = [f(t1, ..., tn) = f(s1, ..., sn)]
where f is an irreducible symbol and j ∈ {1, ..., m} (ti, sj arbitrary terms). The result of

the decomposition step is
Decompose(G,j) = [e_1, ..., e_{j-1}, t1 = s1, ..., tn = sn, e_{j+1}, ..., e_m **with** σ].

4.2.6.3. Expansion

An expansion of an auxiliary variable is defined for a goal G if
G = [e_1, ..., e_j, ..., e_m **with** σ]
where e_j = [t = x] or e_j = [x = t], x a variable and:

x does not occur in t
t contains only variables and irreducible function symbols.

The result of the expansion step is
Expand(G,j) = [τe_1, ..., τe_{j-1}, τe_{j+1}, ..., τe_m **with** $\tau\sigma$]
where τ = [t/x].

4.2.6.4. Rewriting

A goal G = [S **with** σ] can be rewritten at occurrence u using an unconditional rule r = [t -> t'] iff

there is a substitution τ such that S/u = τt
r $\in$ NoethRules (termination ensured by the user).

The result of the rewrite step is
Rewrite(G,r,u) = [S[u$\leftarrow\tau$t'] **with** σ].

4.2.6.5. Conditional Rewriting

The rewriting steps can be generalized to conditional rules. A goal G = [S **with** σ] can be rewritten using a rule r = [P => t -> t'] if G can be rewritten using [t -> t'], and the following holds:
Simplify(τP) = [t1 = t1 & ... & tn = tn]
with arbitrary terms t1, ..., tn. Simplify is defined below in 2.6.7 (for goals, but premises can be seen as a special case of a goal).

4.2.6.6. Evaluation of Arithmetic

If a goal contains a ground subterm built from the arithmetical standard functions in NAT, the subterm is replaced by the corresponding Nat-literal.

4.2.6.7. Simplification

The steps 2.6.2 to 2.6.5 together form a set of simplification rules for goals. We can define a compound procedure out of them:

```
procedure Simplify (var Goal G; var Bool unsatisfiable):

begin
    unsatisfiable := false;
    var Bool modified := false;
    repeat
        if Unsatisfiable(G)
            then unsatisfiable := true
            else
                if Decompose(G,j) is defined for a j
                    then
                        G := Decompose(G,j);
                        modified := true
                fi;
                if Expand(G,j) is defined for a j
                    then
                        G := Expand(G,j);
                        modified := true
                fi;
                if Rewrite(G,r,u) is defined for some r, u
                    then
                        G := Rewrite(G,j);
                        modified := true
                fi;
        fi
    until unsatisfiable or not modified
end;
```

Note the mutual recursion between the procedures Simplify and Rewrite caused by conditional rewriting. The simplification steps are integrated into CNA by applying Simplify to the intermediate goals passed to EnterNewGoal, i.e. the auxiliary procedure EnterNewGoal becomes:

```
procedure EnterNewGoal (Goal g'):
begin
    Goal g := g';
    Simplify(g);
    if Unif(g) is defined
    ...
end;
```

4.2.6.8. Redex Selection for Rewriting

Again there is a system parameter controlling two variants of the algorithm. The redex selection strategy of Rewrite can be defined to be

leftmost-innermost or
leftmost-outermost.

This means that the redex <u,r> used by Rewrite is guaranteed to be the leftmost-innermost or leftmost-outermost one, resp. Note that rewriting always selects one single redex, i.e. one occurrence and one rule applying there. Rewriting thus consists of deterministic ("trap door") steps. Narrowing, in contrast, usually acts on more than one redex, even in the innermost case. So, narrowing is nondeterministic and builds up a proof tree rather than a sequence.

4.2.6.9. Subsumption

A goal G = [S **with** σ] is called **subsumed** by another goal G' = [S' **with** σ'] iff there is a substitution τ such that S = τS' and $\sigma = \tau\sigma'$. G = [S **with** σ] is called subsumed by a solution μ iff there is a substitution τ such that $\sigma = \tau\mu$. The algorithm CNA is further extended by the following tests:

- EnterNewGoal enters a goal only into Q if it is not subsumed by a goal created earlier. This means that the subsumption of the new goal can be caused by all goals in Q as well as by other goals no longer present in Q (removed by the "exhausted" step, by subsumption, or manually by the user).
- A new goal is only entered into Q if it is not subsumed by a solution found already.
- If Unif in EnterNewGoal delivers a solution, all goals in Q are removed which are subsumed by this solution.

Subsumption tests for narrowing have been introduced by [Rety et al. 85].

4.2.6.10. Duplicate Solutions

All solutions found by the algorithm are compared with the solutions which have been found earlier. A new solution is only announced to the user if it differs from all previous solutions. Solutions are compared here up to a renaming.

4.2.6.11. Constructor Solutions

If a solution is found by the algorithm, it is tested whether it consists of a constructor substitution (i.e., the substituted terms are built of variables and constructors only). Non-constructor solutions are rejected, i.e., they are not announced to the user.

The restriction to constructor solutions replaces the classical restriction to normalized substitutions in unconditional narrowing. In the case of conditional equations it is in general undecidable whether a term is normalized. (Nevertheless, the completeness theorem for conditional narrowing is the same as in the unconditional case, it holds for normalized solutions only.)

CHAPTER 5

ADVANCED USE OF RAP

5.1. General Remarks

This chapter contains material about additional features built into RAP which may help to control the behaviour of the system. The first two sections give information about interactive debugging in RAP. The DEBUG mode described there is helpful for all users to find bugs in specifications and to tune specifications for acceptable performance. To understand the whole information offered in DEBUG mode, reading of chapter 4 is recommended. The last two sections of chapter 5 treat material which is of interest only in rather particular situations: for instance, if you want to disable optimizations, or to define a startup file.

5.2. The DEBUG Mode

The DEBUG mode of the RAP system is a tool for observing and controlling the conditional narrowing algorithm interactively. If the algorithm (a RUN command) is performed in the DEBUG mode, it will stop after a certain amount of deduction steps and display the intermediate goal on the screen. Some special commands (valid only in the DEBUG mode) are available to obtain further information and to control the search.

There are two ways of entering the DEBUG mode:

> the /DEBUG option of the EXEC command
> the answer "d", if the system asks "More solutions?".

(A question for "More solutions?" takes place after a solution has been found unless the /NOINQUIRE option is active).

Another possibility for observing the behaviour of the algorithm is the /TRACE option of the RUN command which causes the system to produce a file reporting all steps it has performed in a style similar to the DEBUG information. The /TRACE option does not change the interactive behaviour of RAP (but the response time).

The amount of information given in DEBUG mode depends on the DEBUGLEVEL system parameter. The following values are allowed (Default: STANDARD):

STANDARD

The system stops

before choosing a "current goal" from the goal queue after a narrowing step, before starting the simplification process.

It displays

the current goal (G_{cur})
the new goal (G_{new}) immediately after a narrowing step
(still without simplification)
the new goal (G_{new}) before it is entered into the goal queue.

All other steps are only announced by short messages.

BRIEF
All steps are announced by short messages. No automatic display of goals. The system stops only before choosing the current goal.

DETAILED
In addition to the actions of STANDARD, the system stops after each step (including simplification steps) and displays the current state of the goal it is working on.

In the following, the commands are described which are available additionally within DEBUG mode. The RAP standard commands (see chapter 3) are also available in DEBUG mode, except of the RUN command. Note that the meaning of empty input (RETURN key) changes in DEBUG mode to a STEP command (perform the next step).

5.3. RAP DEBUG Commands

The commands described in this section are available only in DEBUG mode (prompt "RAP/DEBUG>"). All RAP commands described in chapter 3 are valid in DEBUG mode, too, except the RUN command.

SHOW OPTION

Show RUN command options.

This command displays the values assigned to the options of the current RUN.
Initially these options are set up by the RUN command line, but they can be changed by a SET OPTION /<optionname> command (within DEBUG mode). For more information on the effect of the options see the documentation for the RUN command.

SHOW SOLUTIONS

Show known solutions for the current RUN.

This command remembers solutions which have been found already within the current RUN.

SHOW GOAL

Display goals.

This command serves for inspection of goals during the execution of Conditional Narrowing.
The goal can be selected by a parameter. If no parameter is given, the "new goal" generated within the current narrowing step is shown.

Parameter: The (optional) parameter of a SHOW GOAL command has to be a number of a goal which has been created within the current RUN.

SHOW PROOFTREE

Display the proof tree structure of goals.

A SHOW PROOFTREE command results in a display of the goals generated within the current RUN (represented by the goal numbers). The tree structure of these goals is shown by indentation: If a goal g2 has been produced by a narrowing step from goal g1, then g2 is written below g1, indented by one space. The goal numbers are labelled with the current status of a goal, i.e.

- ACTIVE (contained in goal queue) or
- INACTIVE (e.g.: exhausted, deleted by user, subsumed)

The actual text of the goals can be inspected by a SHOW GOAL command.

SHOW QUEUE

Display the goal queue.

A SHOW QUEUE command displays the queue of goals which will be treated by the narrowing algorithm. The goals are represented by goal numbers.
The actual text of the goals can be inspected by a SHOW GOAL command, the tree structure of the goals is obtained by a SHOW PROOFTREE command.

SHOW RULE

Show rewrite rules (axioms).

In DEBUG mode, there is a special variant of the SHOW RULE command available: If a SHOW RULE command is issued without a parameter, the rule applied for the last step is shown automatically.

SET CURRENT

Define continuation point.

A SET CURRENT command rearranges the goal queue by putting the goal given to it as a parameter on top of the queue. This way it redefines the continuation point of the algorithm, since the next "current goal" will be taken from the top of the queue, so the algorithm will take the goal given to SET CURRENT as the next one.

Parameter: The parameter of a SET CURRENT command has to be the number of a goal contained in the current goal queue (see SHOW QUEUE).

SET OPTION

Set up RUN options.

This command serves for the readjustment of options which have been given to the current RUN command by options on the command line. Note that the new options are passed **as options** to a SET OPTION command!
See the documentation of the RUN command for the meaning of the options.

Options: /DEBUG /NODEBUG /TRACE /NOTRACE /INQUIRE /NOINQUIRE
/OLD_OPTIONS (Default: /OLD_OPTIONS)
/TIMELIMIT=<value> (Default: /TIMELIMIT=OLD_VALUE)
/DEPTHLIMIT=<value> (Default: /DEPTHLIMIT=OLD_VALUE)
/SOLNLIMIT=<value> (Default: /SOLNLIMIT=OLD_VALUE)
/SEARCH=<value> (Default: /SEARCH=OLD_VALUE)

/DEBUG Switch on DEBUG mode.

/NODEBUG Switch off DEBUG mode.

/TRACE Switch on generation of trace file.

/NOTRACE Switch off generation of trace file.

/INQUIRE Switch on inquire for "More solutions?".

/NOINQUIRE Switch off inquire for "More solutions?".

/OLD_OPTIONS
Do not change the current /TRACE, /DEBUG, /INQUIRE options.

/TIMELIMIT Define a new timelimit. The value has to be the new timelimit in CPU seconds or one of the keywords OLD_VALUE, INFINITE. The keyword OLD_VALUE means: Do not change the current /TIMELIMIT option.

/DEPTHLIMIT
Define a new depthlimit. The value has to be a natural number or one of the keywords OLD_VALUE, INFINITE. The keyword OLD_VALUE means: Do not change the current /DEPTHLIMIT option.

/SOLNLIMIT Define a new solution limit, i.e. number of solutions asked for. The value has to be a natural number or one of the keywords OLD_VALUE or INFINITE. The keyword OLD_VALUE means: Do not change the current /SOLNLIMIT option.

/SEARCH Define a new search strategy. The value has to be one of the keywords DF, BF, SF, OLD_VALUE. The keyword OLD_VALUE means the current strategy (default). The keyword OLD_VALUE means: Do not change the current /search option.

STEP

Procede one or more steps.

A STEP command without a parameter allows the Conditional Narrowing algorithm to perform the next step. It depends on the current debug level what is done before the next DEBUG prompt. (See SET PARAM DEBUGLEVEL for adjusting this system parameter.)
The STEP command is the default command in DEBUG mode, i.e. it is performed for empty input (RETURN key).

Parameter: If a number n is supplied to STEP as a parameter, the algorithm performs n steps without DEBUG mode and then returns to DEBUG mode.

GO

Continue without debugging.

The GO command terminates the DEBUG mode and continues the interpreter immediately without debugging.

DELETE GOAL

Remove a goal from the goal queue.

The DELETE GOAL command serves for removing goals from the goal queue which are decided by the user not to be treated further. If a DELETE GOAL command without a parameter is issued, it tries to delete the "new goal" generated within the current narrowing step. Note that there is no way for undoing a DELETE GOAL command, so the safer way for influencing the search is SET CURRENT.

Parameter: The parameter of a DELETE GOAL command has to be the number of a goal contained in the current goal queue (see SHOW QUEUE). If no parameter is supplied, the "new goal" generated within the last narrowing step is deleted.

EXIT

Leave the current RUN.

If an EXIT command (or, equivalently, an "end-of-file") is entered during DEBUG mode, the current RUN is stopped, RAP returns to standard mode.

5.4. System Parameters

There are a number of global parameters of the system which can be inspected by a SHOW PARAM command and changed by a SET PARAM command. The actual behaviour of an execution of the narrowing algorithm depends on the settings of these (global) **parameters** and the (local) RUN command **options**. In the following, the effect of the various system parameters is described. For a description of options, see the documentation of the RUN command.

DEBUGLEVEL, TRACELEVEL

These parameters control the amount of information shown in DEBUG mode on on a trace file, resp. See section 5.2 for more details.

Values:
BRIEF, STANDARD, DETAILED

Default:
STANDARD

DISPLAYUSED

controls whether terms are displayed using the standard functional (prefix) notation or whether a mixfix output according to user-defined "display"-definitions is produced. The "display"-definitions can be inspected by a SHOW DISPLAY <typename> command. See also section 2.10.

Values:

ON: use display definitions for mixfix output
OFF: generate standard output

Default:

OFF

CONSTRSOLNS

controls whether the system restricts solutions to constructor solutions (see section 4.2.6.11).

Values:

ON: only constructor solutions are shown
OFF: constructor and non-constructor solutions are shown.

Default:

OFF

GARCOL

There is a simple garbage collector (using the PASCAL dispose) built into RAP. It works simply by "disposing" unneeded parts of the heap as soon as possible. Switching on this garbage collector slows down the system significantly (about 50 percent), but it is recommended for very large RUNs. In particular, garbage collection is necessary if a RUN has been interrupted by a message which looks like "insufficient virtual memory".

Values:

ON: use the garbage collector
OFF: do not use the garbage collector.

Default:

OFF

OPTIMIZATIONS

The optimizations of conditional narrowing mentioned in chapter 4 can be disabled separately for experimental purposes. The SHOW PARAM and SHOW PARAM OPTIMIZATIONS command display the set of optimizations currently enabled, each of them represented by a keyword. A certain optimization is enabled or disabled by the commands:
SET PARAM OPTIMIZATION <keyword> ON
SET PARAM OPTIMIZATION <keyword> OFF, resp.

Keywords:
REWRITE: Normalization by rewriting steps
CREWRITE: Conditional rewriting
EVALUATE: Evaluation of arithmetical functions
EXPAND: Expansion of variables
DECOMPOSE: Decomposition of irreducible symbols
GSUBSUME: Subsumption of goals by goals
SSUBSUME: Subsumption of goals by solutions.

REDEXSELECTION

The narrowing and rewriting steps of the algorithm have to choose from a set of redices (see sections 4.2.5.2 and 4.2.6.8). Different strategies of choice are adjustable by two parameters.

REDEXSELECTION REWRITE:

Values:
LI: leftmost-innermost redex selection
LO: leftmost-outermost redex selection

Default:
LI

The redex selection for the rewrite steps has influence only to the performance of the system. Generally, LI is faster. LO is offerred for experimental purposes.

REDEXSELECTION NARROW:

Values:
LI: leftmost-innermost redex selection
FULL: classical "full" redex selection
AUTOMATIC: experimental combination of LI and FULL

Default:
AUTOMATIC

The redex selection for narrowing steps has to select a number of redices from a given goal (not only one, as in rewriting steps). The FULL strategy simply takes all of them (in leftmost-innermost order). The LI strategy restricts the selection to a subset of redices which is concerned with the same subterm of the goal (the leftmost-innermost position where narrowing is possible). LI-narrowing has been shown to be complete if the axioms have the form of complete function definitions by structural recursion ([Fribourg 85]). The AUTOMATIC strategy tries to combine LI and FULL: If there is a redex within the goal which is labelled by a function symbol out of ComplFuncts (i.e. declared to be "complete" by the user), then all the redices less or equal than this redex in leftmost-innermodt order are taken, othewise all the redices. In both cases, the occurrences are combined with all the rules to obtain the actual set of redices. If all function symbols are declared to be complete, then AUTOMATIC meets LI, if not function symbol is complete, then AUTOMATIC meets FULL.

Example:

Concludingly, an example may illustrate the use of system parameters.
We give below run times for the same task under various combinations of the REDEXSELECTION NARROW and the OPTIMIZATION GSUBSUME parameter:

REDEXSELECTION NARROW	OPTIMIZATION GSUBSUME	CPU time (secs)
LI	OFF	0.75
FULL	OFF	33.32
LI	ON	0.88
FULL	ON	1.72

It is a general observation that the subsumption optimization is very useful in combination with the slow FULL redex selection strategy but it does not improve the performance of the fast LI strategy.

5.5. Startup Files

The RAP system looks for a startup file when it is called and reads commands from there. The name of this file is installation dependent (~/.raprc in UNIX, translation of the logical name RAP$INIT in VAX/VMS). Such a startup file is rather convenient if a special arrangement of system parameters is preferred for some reason.

REFERENCES

[Fribourg 85] L. Fribourg, Handling function defnitions through innermost superposition and rewriting. Proceedings RTA 85 Conference, LNCS 202, pp. 325-344

[Geser 86] A. Geser, A specification of the INTEL 8085 microprocessor: A case study. Report MIP-8608 Universität Passau, 1986.

[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, 1985.

[Geser, Hussmann 86]
A. Geser, H. Hussmann, Experiences with the RAP system - a specification interpreter combining term rewriting and resolution. Proceedings ESOP 86 Conference, LNCS 213, pp. 339-350.

[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 1980.

[Hussmann 85]
H. Hussmann, Unification in conditional-equational theories. Report MIP-8502 Universität Passau, 1985. Short verson in: Proceedings EUROCAL 85 Conference, LNCS 204, pp. 543-553

[Padawitz 87] P. Padawitz, ECDS - A rewrite rule based interpreter for a programming language with abstraction and communication, Report MIP-8703 Universität Passau, 1987.

[Rety et al. 85] P. Rety, C. Krichner, H. Kirchner, P. Lescanne, NARROWER: a new algorithm for unification and its application to logic programming. Proceedings RTA 85 Conference, LNCS 202, pp. 141-155

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

Standard type definitions

```
type BOOL
  sort Bool

  cons true,false : Bool
  func not : (Bool)Bool,
    and : (Bool,Bool)Bool,
    or  : (Bool,Bool)Bool,
```

```
          impl: (Bool,Bool)Bool,
          equiv : (Bool,Bool)Bool

     axioms all (x : Bool)

     not(true) -> false,
     not(false) -> true,

     and(true,x) -> x,
     and(x,true) -> x,
     and(false,x) -> false,
     and(x,false) -> false,

     or(false,x) -> x,
     or(x,false) -> x,
     or(true,x) -> true,
     or(x,true) -> true,

     impl(x,x) -> true,
     impl(true,false) -> false,
     impl(false,x) -> true,
     impl(x,true) -> true,

     equiv(x,x) -> true,
     equiv(true,x) -> x,
     equiv(x,true) -> x

endoftype

noetherian BOOL
complete BOOL
display BOOL
 not = "~ _",
 and = "_ ^ _",
 or = "_ V _",
 impl = "_=)_",
 equiv = "_(=)_"

type NAT

    basedon BOOL
    sort   Nat
    cons   zero : Nat,
           succ : (Nat)Nat
```

```
func   pred : (Nat)Nat,
       add : (Nat,Nat)Nat,
       sub : (Nat,Nat)Nat,
       mult : (Nat,Nat)Nat,
       div  : (Nat,Nat)Nat,
       mod  : (Nat,Nat)Nat,
       equal_Nat : (Nat,Nat)Bool,
       lt : (Nat,Nat)Bool,
       gt : (Nat,Nat)Bool,
       le : (Nat,Nat)Bool,
       ge : (Nat,Nat)Bool

axioms all (x,y,z,r : Nat)

  pred(zero) -> zero,
  pred(succ(x)) -> x,
  add(x,zero) -> x,
  add(x,succ(y)) -> succ(add(x,y)),
  add(zero,x) -> x,
  add(succ(x),y) -> succ(add(x,y)),
    sub(zero,x) -> zero,
  sub(succ(x),zero) -> succ(x),
  sub(succ(x),succ(y)) -> sub(x,y),
  mult(x,zero) -> zero,
  mult(x,succ(y)) -> add(mult(x,y),x),
  mult(zero,x) -> zero,
  mult(succ(x),y) -> add(mult(x,y),y),
  lt(x,y) = true  =>  div(x,y) -> zero,
  lt(x,y) = false =>  div(x,y) -> succ(div(sub(x,y),y)),
  lt(x,y) = true =>  mod(x,y) -> x,
  lt(x,y) = false =>  mod(x,y) -> mod(sub(x,y),y),

  le(x,x) -> true,
  le(0,x) -> true,
  le(succ(x),0) -> false,
  le(succ(x),succ(y)) -> le(x,y),

  ge(x,y) -> le(y,x),
  gt(x,y) -> not(le(x,y)),
  lt(x,y) -> not(ge(x,y)),

  equal_Nat(x,x) -> true,
  equal_Nat(0,succ(x)) -> false,
  equal_Nat(succ(x),0) -> false,
```

```
   equal_Nat(succ(x),succ(y)) -> equal_Nat(x,y)

endoftype

noetherian NAT
complete NAT
display NAT
 add = "_+_",
 mult = "_*_",
 equal_Nat = "_==_",
 le = "_<=_",
 lt = "_<_",
 ge = "_>=_",
 gt = "_>_"

type ID

  basedon BOOL
  sort   Id
  func   equal_Id : (Id,Id)Bool

  axioms all ( x, y : Id )

   equal_Id(x,x) -> true

   {x, y different standard denotations of sort Id =>
   equal_Id(x,y) -> false}

endoftype

noetherian ID
complete ID
display ID
 equal_Id = "_==_"


type CHAR

basedon BOOL

sort Char

func equal_Char: (Char,Char)Bool,
  le_Char: (Char,Char)Bool
```

```
axioms all (c: Char)

 equal_Char(c,c) -> true,
 le_Char(c,c) -> true

endoftype

noetherian CHAR
complete CHAR
display CHAR
 equal_Char = "_==_",
 le_Char = "_<=_"


type STRING

basedon CHAR, BOOL, NAT

sort String

cons  empty: String,
      append: (Char,String)String

func first: (String)Char,
     rest: (String)String,
     isempty: (String)Bool,
     make: (Char)String,
     conc: (String,String)String,
     length: (String)Nat,
     equal_String: (String,String)Bool

axioms all (c,x,y: Char, s,t: String)

 isempty(empty) -> true,
 isempty(append(c,s)) -> false,
 equal_String(empty,empty) -> true,
 equal_String(empty,append(x,t)) -> false,
 equal_String(append(x,t),empty) -> false,
 equal_String(append(x,s),append(y,t)) -> and(equal_Char(x,y),equal_String(s,
  rest(append(x,s)) -> s,
  first(append(x,s)) -> x,
  length(empty) -> zero,
  length(append(x,s)) -> succ(length(s)),
  make(c) -> append(c, empty),
```

```
  conc(empty, s) -> s,
  conc(append(x,s),t) -> append( x, conc(s,t))
endoftype


noetherian STRING
complete STRING  except first,rest
display STRING
 equal_String = "_ == _",
 make = "<_>",
 conc = "_ & _",
 length = "|_|"
```