Debugging in REDUCE

H. Melenk
Konrad–Zuse–Zentrum
für Informationstechnik Berlin
Takustrasse 7
D–14915 Berlin-Dahlem
Federal Republic of Germany
Email: melenk@zib.de

1 Introduction

The module **rdebug** supports the use of the trace and break debugging facilities of Portable Standard LISP (PSL) for REDUCE programming. These include

- Entry-Exit tracing of functions,
- Assignment tracing,
- Setting of break points,
- Conditional trace and break.
- Trace of rules when they fire.

In contrast to the bare LISP level tracing the values are printed in algebraic style whenever possible. The output has been specially tailored for the needs of algebraic mode programming. Most features can be applied without modifying the target program, and they can be turned on and off dynamically at run time.

To make the facilities available, load the module by a command

```
load rdebug.
```

Restriction: the investigated program should not be compiled, when the **trst** functions are to be applied.

2 Trace: TR, UNTR

The command **tr** puts one or several procedures to under trace. Every time such a function is executed, a message is printed during the procedure entry and another one is generated at the return time. The entry message records the actual procedure arguments equated to the dummy parameter names, and at the exit time the procedure value is printed. Recursive calls are marked by an indentation and a level number.

```
tr <proc1>,<proc2>,...,<procn>;
```

Here $< proc1 >, < proc2 >, \ldots, < procn >$ are names of procedures to be added to the set of traced procedures. Tracing is stopped for one or several functions by the command **untr**:

```
untr <proc1>,<proc2>,...,<procn>;
```

3 Assignment Trace: TRST, UNTRST

One often needs information about the inner behavior of a procedure, especially if it is a longer piece of code. For a procedure declared in a **trst** command

```
trst <proc1>,<proc2>,...,<procn>;
```

all executed explicit assignments and passed labels inside these procedures are printed during procedure execution. For removing the extended trace use a statement **untrst** or **untr**:

```
untrst <proc1>,<proc2>,...,<procn>;
```

Note: When your program contains a **for** loop, REDUCE translates this to a sequential piece of LISP instructions. When using **trst**, the printout is driven by the unfolded code. When the code contains a **for each in** command, the name of the control variable is internally used to keep the remainder of the list during the loop control, and you will see the corresponding assignments in the printout rather than the individual values in the loop steps. E.g.

```
procedure fold(u); for each x in u sum x;
trst fold;
fold {z,z*y,y};
```

produces the following output:

```
fold being entered
   u:   {z,y*z,y}$
x := (z,y*z,y)$
g0003 := 0$
g0003 := z$
x := (y*z,y)$
g0003 := y*z + z$
x := (y)$
g0003 := y*z + y + z$
x := ()$
fold = y*z + y + z$
```

In this example the printed assignments for x show the various stages of the loop control. The variable $g0003$ is an internally generated slot for the sum.

4 Conditional tracing: TRWHEN

The trace output can be tunrned on or off automatically by a boolean expression which is linked to a traced procedure by the command **trwhen**:

```
trwhen <name>,<boolean-expr>;
```

The boolean expression must follow standard REDUCE syntax. It may contain references to global values and to the actual parameters of the procedure. As long as the procedure is not compiled, the original names of the dummy arguments are used. For a compiled procedure the original names are not available; instead the names $a1$, $a2$, ... must be used. Example: the following procedure produces trace output only if the main variable of its argument is x:

```
procedure hugo(u); otto(u);
tr hugo;
trwhen hugo,mainvar(u)=x;
```

Note: for a symbolic procedure, the **trwhen** command must be given in symbolic mode or with prefix *symbolic*.

5 Breakpoints: BR, UNBR

A **break loop** is an interrupt of the program execution where control is given temporarily to the terminal for entering commands in a standard command – evaluate – print loop. When a break occurs, you can inspect the current environment or even alter it, and the interrupted computation may be terminated or continued [1]. A break can be caused

- by an internal error,
- by an explicit call of the function *break*,
- at entry and exit time of a procedure.

5.1 Break switch

When the switch **break** is set on, every evaluation error causes a break loop. Most of these breaks are non-continuable; however, you have the opportunity to read the actual values of local variables in the environment which caused the error.

5.2 Break call

A call

```
    lisp break();
```

initiates an "programmed" break loop. In contrast to explicitly introduced write statements, a break loop allows you to read actual values dynamically, e.g. if you don't know the critical variables in advance.

5.3 Breakpoint declaration

When the command **br** is given for a set of procedures, the program execution is interrupted every time such a procedure is entered and for a second time when it is left.

```
   br <proc1>,<proc2>,...,<procn>;
```

[1] Not all cases allow a continuation.

The break property can be removed by calling the command **unbreak**

```
unbr <proc1>,<proc2>,...,<procn>;
```

5.4 Break loop control

In a break situation the evaluation is stopped temporarily and the control returns to the terminal with a special prompt:

```
break[1]1:
```

The number in square brackets counts the break level - it is increased when a break occurs inside a break; the normal REDUCE statement counter follows. Each break loop supports its own statement numbers and input and output buffers. After terminating of a break loop the previous statement counters and buffers are restored.

In a break loop all REDUCE commands can be entered. Additionally, there is a set of single character commands which allow you to control the break environment. All these begin with an underscore character:

_a;	terminate break and return to the top REDUCE level
_c;	continue execution of interrupted procedure
_i;	print a backtrace (list of procedures in the call hierarchy)
_l `<var>`;	read the content of the local variable `<var>`
_m;	print the last (LISP-) error message
_q;	terminate the break loop and return to the next higher level

Global variables can be accessed as usual in the REDUCE language. They can also be set to different values in the break loop. The inspect values assigned to dummy arguments and scalar variables of procedures in the actual call hierarchy, you need a special command *_l*. These values cannot be altered in the break loop. Example:

```
procedure p1(x);
  begin scalar y1; y1:=x^2; return p2(y1); end;
procedure p2(q); q^2;
br p2;
x:=22;
p1(alpha);
```

In the corresponding break loop caused by calling $p2$ indirectly via $p1$, you can access the global x, the local x and $y1$ of $p1$ and the q of $p2$:

```
p2 being entered

   q:   alpha**2$
Break before entering ‘p2’

break[1]1: x;

22

break[1]2: _l x;

alpha

break[1]3: _l y1;

     2
alpha

break[1]4: _l q;

     2
alpha


break[1]6: _c;
Break after call ‘p2’, value ‘(expt (expt alpha 2) 2)’

break[1]1: _c;

     4
alpha
```

6 Conditional break: BRWHEN

A break depending on a condition can be assigned to a procedure using the command **brwhen**

```
brwhen <name>,<boolean-expr>;
```

The conventions correspond to those of **trwhen**.

7 Trace for rules and rule sets: TRRL, UNTRRL

The command **trrl** allows you to trace individual rules or rule sets when they fire.

```
trrl <rs1>,<rs2>,...,<rsn>;
```

where each of the $< rs_i >$ is

- a rule or a rule set,
- a name of a rule or rule set (that is a non–indexed variable which is bound to a rule or rule list),
- an operator name, representing the rules assigned to this operator.

The specified rules are (re–) activated in REDUCE in a style that each of them prints a report every time if fires. The report is composed of the name or the rule or the name of the rule set plus the number of the rule in the set, the form matching the left hand side (“input) and the resulting right hand side (“output). For an explicilty given rule, **trrl** assigns a generated name.

With **untrrl** you can remove the tracing from rules

```
untrrl <rs1>,<rs2>,...,<rsn>;
```

The rules are reactivated in their original form. Alternatively you can use the command **clearrules** to remove the rules totally from the system. Please do not modify the rules between **trrl** and **untrrl** – the result may be unpredictable.

8 Output control: TROUT, TRLIMIT

The trace output can be redirected to a separate file by using the command **trout**, followed by a file name in string quotes. A second call of **trout** closes the actual output file and assigns a new one. The file name NIL (without string quotes) causes the trace output to be redirected to the standard output device.

Remark: under Windows a file name starting with “win:” causes a new window to be opened which receives the complete output of the debugging services.

The integer valued share variable **trlimit** defines an upper limit for the number of items printed in formula collections, e.g. when tracing a **for each** loop. The initial value is 5. A different value can be assigned to increase or lower the output size.

If you want to select LISP style printing during trace, set

```
    lisp(trprinter!* := 'printx);
```

after loading **rdebug**.