Tracing in REDUCE

Herbert Melenk

Konrad–Zuse–Zentrum
für Informationstechnik Berlin
Takustraße 7
D–14195 Berlin–Dahlem
Federal Republic of Germany

E-mail: melenk@zib.de

Francis J. Wright

School of Mathematical Sciences
Queen Mary and Westfield College, University of London
Mile End Road, London E1 4NS, UK.

E-mail: F.J.Wright@QMW.ac.uk
http://www.maths.qmw.ac.uk/~fjw/

14 March 1999

1 Introduction

The package `rtrace` provides portable tracing facilities for REDUCE programming. These include

- entry-exit tracing of procedures,
- assignment tracing of procedures,
- tracing of rules when they fire.

In contrast to conventional Lisp-level tracing, values are printed in algebraic style whenever possible if the switch `rtrace` is on, which it is by default. The output has been specially tailored for the needs of algebraic-mode programming. Most features can be applied without explicitly modifying the target program, and they can be turned on and off dynamically at run time.

If the switch `rtrace` is turned off then values are printed in conventional Lisp style, and the result should be similar to the tracing provided by the underlying Lisp system.

To make the facilities available, load the package using the command

```
load_package rtrace;
```

Alternatively, the package can be set up to auto load by putting appropriate code in your REDUCE initialisation file. An example is provided in the file `reduce.rc` in the `rtrace` source directory.

2 RTrace versus RDebug

The `rtrace` package is a modification (by FJW) of the `rdebug` package (written by HM, and included in the `rtrace` source directory). The modifications are as follows. The procedure-tracing facilities in `rdebug` rely upon the low-level tracing facilities in PSL; in `rtrace` these low-level facilities have been (partly) re-implemented portably. The names of the tracing commands that have been re-implemented portably have been changed to avoid conflicting with those provided by the underlying Lisp system by preceding them with the letter "r", and they provide a generalized interface that supports algebraic mode better. An additional set of rule tracing facilities for inactive rules has been provided. Beware that the `rtrace` package is still experimental!

This package is intended to be portable, and has been tested with both CSL- and PSL-based REDUCE. However, it is intended not as a replacement for `rdebug` but as a partial re-implementation of `rdebug` that works with CSL-REDUCE, and it is assumed that PSL users will continue to use `rdebug`. It should, in principle, be possible to use both. Any `rtrace` functions with the same names as `rdebug` functions should either be identical or compatible; `rtrace` should be loaded after `rdebug` in order to retain any enhancements provided by `rtrace`. Perhaps at some future time the two packages should be merged. However, note that `rtrace` currently provides *only tracing* (hence the name) and does not support break points. (The current version also does not support conditional tracing.)

3 Procedure tracing: RTR, UNRTR

Tracing of one or more procedures is initiated by the command `rtr`:

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

and cancelled by the command `unrtr`:

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

Every time a traced procedure is executed, a message is printed when the procedure is entered or exited. The entry message displays the actual procedure arguments equated to the dummy parameter names, and the exit message displays the value returned by the procedure. Recursive calls are marked by a level number. Here is a (simplistic) example, using first the default algebraic display and second conventional Lisp display:

```
algebraic procedure power(x, n);
   if n = 0 then 1 else x*power(x, n-1)$

rtr power;

(power)

power(x+1, 2);

Enter (1) power
   x:  x + 1$
   n:  2$
Enter (2) power
   x:  x + 1$
   n:  1$
Enter (3) power
   x:  x + 1$
   n:  0$
Leave (3) power = 1$
Leave (2) power = x + 1$
Leave (1) power = x**2 + 2*x + 1$

 2
x  + 2*x + 1
```

```
off rtrace;

power(x+1, 2);

Enter (1) power
   x:  (plus x 1)
   n:  2
Enter (2) power
   x:  (plus x 1)
   n:  1
Enter (3) power
   x:  (plus x 1)
   n:  0
Leave (3) power = 1
Leave (2) power = (!*sq ((((x . 1) . 1) . 1) . 1) t)
Leave (1) power = (!*sq ((((x . 2) . 1) ((x . 1) . 2) . 1) . 1) t)

 2
x  + 2*x + 1

on rtrace;

unrtr power;

(power)
```

Many algebraic-mode operators are implemented as internal procedures with different names. If an internal procedure with the specified name does not exist then `rtrace` tracing automatically applies to the appropriate internal procedure and returns a list of the names of the internal procedures, e.g.

```
rtr int;

(simpint)
```

This facility is an extension of the `rdebug` package.

Tracing of *compiled* procedures by the `rtrace` package is not completely reliable, in that recursive calls may not be traced. This is essentially because tracing works only when the procedure is called by name and not when it is

called directly via an internal compiled pointer. It may not be possible to avoid this restriction in a portable way. Also, arguments of compiled procedures are not displayed using the names given to them in the source code, because these names are no longer available. Instead, they are displayed using the names `Arg1`, `Arg2`, etc.

4 Assignment tracing: RTRST, UNRTRST

One often needs information about the internal behaviour of a procedure, especially if it is a longer piece of code. For an interpreted procedure declared in an `rtrst` command:

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

all explicit assignments executed (as either the symbolic-mode `setq` or the algebraic-mode `setk`) inside these procedures are displayed during procedure execution. All procedure tracing (assignment and entry-exit) is removed by the command `unrtrst` (or `unrtr`, for which it is just a synonym):

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

Assignment tracing is not possible if a procedure is compiled, either because it was loaded from a "fasl" file or image, or because it was compiled as it was read in as source code. This is because assignment tracing works by modifying the interpreted code of the procedure, which must therefore be available.

Applying `rtr` to a procedure that has been declared in an `rtrst` command, or vice versa, toggles the type of tracing applied (and displays an explanatory message).

Note that when a program contains a `for` loop, REDUCE translates this to a sequence of Lisp instructions. When using `rtrst`, the printout is driven by the "unfolded" code. When the code contains a `for each ... in` statement, the name of the control variable is internally used to keep the remainder of the list during the loop, and you will see the corresponding assignments in the trace rather than the individual values in the loop steps, e.g.

```
procedure fold u;
   for each x in u sum x$
```

```
rtrst fold;

(fold)

fold {z, z*y, y};
```

produces the following output (using CSL-REDUCE):

```
Enter (1) fold
   u:  {z,y*z,y}$
x := [z,y*z,y]$
G0 := 0$
G0 := z$
x := [y*z,y]$
G0 := z*(y + 1)$
x := [y]$
G0 := y*z + y + z$
x := []$
Leave (1) fold = y*z + y + z$

y*z + y + z

unrtrst fold;

(fold)
```

In this example, the printed assignments for `x` show the various stages of the loop. The variable `G0` is an internally generated place-holder for the sum, and may have a slightly different name depending on the underlying Lisp systems.

5 Tracing active rules: TRRL, UNTRRL

The command `trrl` initiates tracing when they fire of individual rules or rule lists that have been activated using `let`.

```
trrl <rl1>, <rl2>, ..., <rln>;
```

where each of the $< rl_i >$ is:

- a rule or rule list;
- the name of a rule or rule list (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 such that each of them prints a report every time it fires. The report is composed of the name of the rule or the name of the rule list together with the number of the rule in the list, the form matching the left side ("input") and the resulting right side ("output"). For an explicitly given rule or rule list, `trrl` assigns a unique generated name.

Note, however, that `trrl` does not trace rules with constant expressions on the left, on the assumption that they are not particularly interesting. [This behaviour may be made user-controllable in a future version.]

The command `untrrl` removes the tracing from rules:

`untrrl <rl1>, <rl2>, ..., <rln>;`

where each of the $< rl_i >$ is:

- a rule or rule list;
- the name of a rule or rule list (that is, a non-indexed variable which is bound to a rule or rule list or a unique name generated by `trrl`);
- an operator name, representing the rules assigned to this operator.

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.

Here are two simple examples that show tracing via the rule name and via the operator name:

```
trigrules := {sin(~x)^2 => 1 - cos(x)^2};

                     2             2
trigrules := {sin(~x)  => 1 - cos(x) }

let trigrules;
trrl trigrules;
```

```
1 - sin(x)^2;

Rule trigrules.1: sin(x)**2 => 1 - cos(x)**2$

      2
cos(x)

untrrl trigrules;
trrl sin;

1 - sin(x)^2;

Rule sin.23: sin(x)**2 => 1 - cos(x)**2$

      2
cos(x)

untrrl sin;
clearrules trigrules;
```

6 Tracing inactive rules: TRRLID, UNTRRLID

The command `trrlid` initiates tracing of individual rule lists that have been assigned to variables, but have not been activated using `let`:

```
trrlid <rlid1>, <rlid2>, ..., <rlidn>;
```

where each of the $< rlid_i >$ is an identifier of a rule list (that is, a non-indexed variable which is bound to a rule list). It is assumed that they will be activated later, either via a `let` command or by using the `where` operator. When they are activated and fire, tracing output will be as if they had been traced using `trrl`. The command `untrrlid` clears the tracing. This facility is an extension of the `rdebug` package.

Here is a simple example that continues the example above:

```
trrlid trigrules;

1 - sin(x)^2 where trigrules;
```

```
Rule trigrules.1: sin(x)**2 => 1 - cos(x)**2$

      2
cos(x)

untrrlid trigrules;
```

7 Output control: RTROUT

The trace output (only) can be redirected to a separate file by using the command `rtrout`, followed by a file name in string quotes. A second call of `rtrout` closes any current output file and opens a new one. The file name `NIL` (without string quotes) closes any current output file and causes the trace output to be redirected to the standard output device.

The `rdebug` variables `trlimit` and `trprinter!*` are not implemented in `rtrace`. If you want to select Lisp-style tracing then turn off the switch `rtrace`:

```
off rtrace;
```

after loading the `rtrace` package. Note that the `rtrace` switch controls the display format of both procedure and rule tracing.