

# Aldor Debugger v0.61 Manual

by

Jinlong Cai, Marc Moreno Maza

Ontario Research Center for Computer Algebra

Department of Computer Science

The University of Western Ontario

London, Ontario

June 2004

© Jinlong Cai, Marc Moreno Maza 2004

# TABLE OF CONTENTS

<b>TABLE OF CONTENTS</b>	<b>1</b>
<b>Chapter 1 Aldor Debugger v0.61 Manual</b>	<b>2</b>
1.1 Preparing a Program for Debugging . . . . .	2
1.2 Starting the Debugger . . . . .	3
1.3 Entering Debugger Commands . . . . .	3
1.4 Continuing Execution of the Process . . . . .	5
1.5 Setting breakpoints . . . . .	8
1.6 Examining the Paused Process . . . . .	10
1.6.1 Looking at the Source Files . . . . .	10
1.6.2 Looking at the Call Stack . . . . .	11
1.6.3 Looking at the Data . . . . .	12
1.6.4 Updating the Data . . . . .	13

## Chapter 1

### Aldor Debugger v0.61 Manual

#### 1.1 Preparing a Program for Debugging

- Include debuglib in the Aldor source program
- Insert start!()\$NewDebugPackage into where the Aldor debugger will be invoked
- Limitations on the Aldor source program:
  - Take every "import from" expression to top level(outside any functions).
  - Every other single expression should be included in a function.

The following Aldor source program is well prepared for debugging:

```
#include "aldor"
#include "debuglib"
start!()$NewDebugPackage;
import from MachineInteger;

main(): () == {
    x: MachineInteger := 1;
    stdout << x << newline;
}
main();
```

## 1.2 Starting the Debugger

Compile the program and run the Aldor debugger with debugger option.

Note that the debugger cannot run in interpreter mode (-Gloop and -GInterp).

```
$ aldor -wdebugger -ldebuglib -laldor -grun deb40.as
```

```
-----  
Aldor Runtime Debugger
```

```
v0.60 (22-May-2000), (c) NAG Ltd 1999-2000.
```

```
v0.61 (05-Dec-2003), ORCCA @ UW0.
```

```
Type "help" for more information.  
-----
```

```
main(p:MachineInteger == {2}) ["deb40.as" at line 12]
```

```
12      import from String, Character;
```

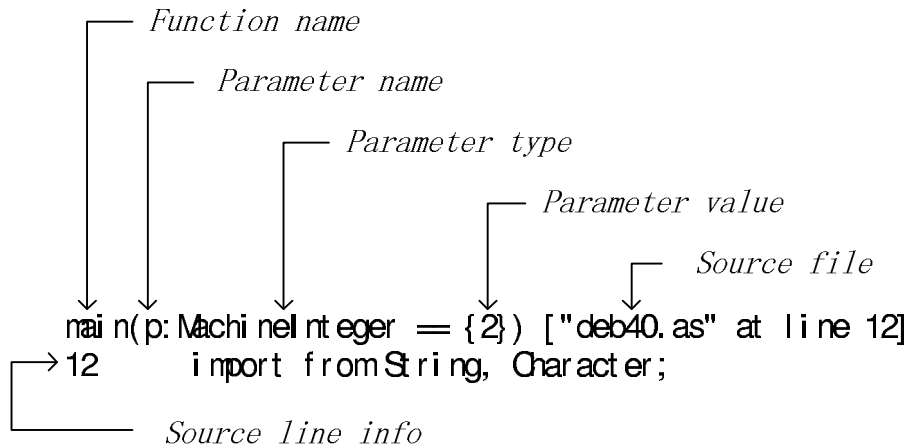
```
(debug)
```

## 1.3 Entering Debugger Commands

The debugger shows a prompt when it is ready for the user inputs from the terminal:

```
(debug) you type here
```

When you enter commands, you use the Backspace key to delete the wrong characters. When you finish entering a command, press the Enter key to submit the



completed line to the debugger for processing. On a blank line, press the Enter key to re-execute the default command. The system default command is step. If you used next command after step command, then the default command is next. Help command is a very useful command:

(debug) **help**

Available commands:

- break: display or insert breakpoints.
- cont: continue execution of program.
- delete: delete breakpoints.
- disable: disable breakpoints.
- enable: enable breakpoints.
- halt: abort the program.
- help: access this help system.
- hints: hints and tips for using the debugger.
- int: interactive or non-interactive debugging.
- next: move to next statement (stepping over calls).

off: turn off the debugging system and continue.  
quit: abort the program.  
step: move to next statement (stepping into calls).  
tips: hints and tips for using the debugger.  
verbose: display of event details.  
where: display a stack trace.  
print: print out the value of a variable.  
update: update the value of a variable.  
shell: execute a shell command.

Use "help <command>" for more help on <command>.

(debug) help print

Syntax: print x

where x is a variable or operation whose return value is printable.

## 1.4 Continuing Execution of the Process

After you are satisfied that you understand what is going on, you can move the process forward and see what happens. The following commands you can use to do this.

***next***: Single step by instruction, over calls

***step***: Single step by instruction, into calls

The following example demonstrates stepping through lines of source code.

(debug)

```

main(p:MachineInteger == {2}) ["deb40.as" at line 12]
12      import from String, Character;

(debug)
main(p:MachineInteger == {2}) ["deb40.as" at line 13]
13      import from Array SI, TextWriter;

(debug)
main(p:MachineInteger == {2}) ["deb40.as" at line 15]
15      local x: SI := p*p;

(debug)
main(p:MachineInteger == {2}) ["deb40.as" at line 16]
16      y: SI := foo(x);

(debug) next
main(p:MachineInteger == {2}) ["deb40.as" at line 17]
17      z: STR := "ORCCA @ UWO";

(debug)
main(p:MachineInteger == {2}) ["deb40.as" at line 18]
18      w: Array SI := new(2, x);

(debug)
main(p:MachineInteger == {2}) ["deb40.as" at line 20]
20      hin(p1:SI): SI == {

```

```
(debug) step
hin(p1:MachineInteger == {8}) ["deb40.as" at line 21]
21          stdout << z << newline;
```

```
(debug)
ORCCA @ UW0
hin(p1:MachineInteger == {8}) ["deb40.as" at line 22]
22          stdout << w << newline;
```

```
(debug)
[4,4]
foo(f:MachineInteger == {8}) ["deb40.as" at line 29]
29 foo(f:SI): SI == {
```

```
(debug)
hin(p1:MachineInteger == {8}) ["deb40.as" at line 23]
23          foo(p1);
```

```
(debug)
main(p:MachineInteger == {2}) ["deb40.as" at line 26]
26      hin(y);
```

```
(debug)
```



## 1.5 Setting breakpoints

The following commands are to set breakpoints, delete breakpoints, disable breakpoints and enable breakpoints:

```
(debug)break [<file>] [<line>]
```

With no arguments, this command lists the current set of breakpoints. If one argument is specified then it must be a valid line number in the current file. If two arguments are given then the first is a filename and the second a line number within that file. For example:

```
(debug) break
```

```
No breakpoints defined.
```

```
(debug) break deb40.as 23
```

```
Breakpoint set: [*] "deb40.as" at line 23 (hit 0 times).
```

```
(debug) cont
```

```
Hit breakpoint #1 hin(p1:SingleInteger == 8) ["deb40.as" at line 23]
```

```
hin(p1:SingleInteger == 8) ["deb40.as" at line 23]
```

```
23                               print << i << newline;
```

```
(debug) print i
```

```
4
```

```
(debug) break
```

```
1) [*] "deb40.as" at line 23 (hit 1 times).
```

```
(debug)delete [<breakpoint list>]
```

This command can be used to remove a breakpoint from the set of active and inactive breakpoints. If no arguments are given, ALL breakpoints are deleted. For example:

```
(debug) break
1) [*] "deb40.as" at line 10 (hit 0 times).
2) [*] "deb40.as" at line 23 (hit 0 times).
(debug) delete 1
Deleted 1 breakpoint.
(debug) break
2) [*] "deb40.as" at line 23 (hit 0 times).
```

```
(debug)disable [<breakpoint list>]
```

This command can be used to disable breakpoints. They will remain in place but will not trigger the debugger if hit. If no arguments are given, ALL breakpoints are disabled. For example:

```
(debug) break
1) [*] "deb40.as" at line 23 (hit 0 times).
2) [*] "deb40.as" at line 10 (hit 0 times).
(debug) disable 1
Disabled 1 breakpoint.
(debug) break
1) [ ] "deb40.as" at line 23 (hit 0 times).
2) [*] "deb40.as" at line 10 (hit 0 times).
```

```
(debug)enable [<breakpoint list>]
```

This command can be used to enable breakpoints that were previously disabled. If no arguments are given, All breakpoints are enabled. For example:

```
(debug) break
1) [ ] "deb40.as" at line 23 (hit 0 times).
2) [*] "deb40.as" at line 10 (hit 0 times).
(debug) enable 1
Enabled 1 breakpoint.
(debug) break
1) [*] "deb40.as" at line 23 (hit 0 times).
2) [*] "deb40.as" at line 10 (hit 0 times).
```

## 1.6 Examining the Paused Process

This section describes how to examine runtime stack of the paused process.

### 1.6.1 Looking at the Source Files

You can perform the following shell operations to look at source files:

```
(debug)shell more deb40.as
```

```
#include "aldor"
```

```
#include "debuglib"
```

```
start!()$NewDebugPackage;
```

```
main(): () == {
```

```
    import from SingleInteger;
```

```

        x: SingleInteger := 1;

        print << x << newline;
    }

```

```

main();
(debug)

```

### 1.6.2 Looking at the Call Stack

You can examine the call stack of the process. For example:

```

(debug) where

```

```

main(p:MachineInteger == {2}) ["deb40.as" at line 12]
12      import from String, Character;

```

```

(debug) where

```

```

1: main(p:MachineInteger == {2}) ["deb40.as" at line 12]

```

```

(debug)

```

```

main(p:MachineInteger == {2}) ["deb40.as" at line 13]
13      import from Array SI, TextWriter;

```

```

(debug)

```

```

main(p:MachineInteger == {2}) ["deb40.as" at line 15]
15      local x: SI := p*p;

```

```

(debug)

```

```

main(p:MachineInteger == {2}) ["deb40.as" at line 16]
16      y: SI := foo(x);

```

```
(debug)
foo(f:MachineInteger == {4}) ["deb40.as" at line 29]
29 foo(f:SI): SI == {
```

```
(debug) where
2: main(p:MachineInteger == {2}) ["deb40.as" at line 16]
1: foo(f:MachineInteger == {4}) ["deb40.as" at line 29]
```

### 1.6.3 Looking at the Data

You can look at variables and evaluate expressions involving them. For example:

```
(debug)
main(p:MachineInteger == {2}) ["deb40.as" at line 16]
16      y: SI := foo(x);
```

```
(debug) print x
```

```
4
```

```
(debug)
foo(f:MachineInteger == {4}) ["deb40.as" at line 29]
29 foo(f:SI): SI == {
```

```
(debug)
main(p:MachineInteger == {2}) ["deb40.as" at line 17]
17      z: STR := "ORCCA @ UWO";
```

```
(debug) print y
```

8

(debug)

main(p:MachineInteger == {2}) ["deb40.as" at line 18]

18       w: Array SI := new(2, x);

(debug)

main(p:MachineInteger == {2}) ["deb40.as" at line 20]

20       hin(p1:SI): SI == {

(debug) print z

ORCCA @ UW0

(debug) print w

[4,4]

(debug)print gcd(x,y)

4

(debug)print foo(1)

2

(debug)

#### 1.6.4 Updating the Data

You can update the variables. For example:

(debug) print x

4

(debug) *update x:=100*

(debug) print x

100

```
(debug)
main(p:MachineInteger == {2}) ["deb40.as" at line 20]
20      hin(p1:SI): SI == {

(debug) print z
ORCCA @ UW0

(debug) update z:="ISSAC"

(debug) print z
ISSAC
(debug)
```