

The Mozart Debugger

Benjamin Lorenz

Version 1.2.3
December 1, 2001



Abstract

This manual describes Ozcar, a symbolic debugger which provides well known features like single stepping, breakpoints, and environment inspection. Moreover, it supports debugging of multiple threads; their concurrent behaviour can be observed and manipulated. Ozcar has been smoothly integrated into the Oz Programming Interface. It can be started and stopped from within Emacs, which itself serves both as a source view manager, showing the current position in a debugged program by highlighting the corresponding source line, and as a breakpoint manager, allowing to set breakpoints at arbitrary source lines. Finally, the application `ozd` is provided to debug standalone programs, see Chapter *The Oz Debugger: ozd*, (*Oz Shell Utilities*) for more information.

Credits

Mozart logo by Christian Lindig

License Agreement

This software and its documentation are copyrighted by the German Research Center for Artificial Intelligence (DFKI), the Swedish Institute of Computer Science (SICS), and other parties. The following terms apply to all files associated with the software unless explicitly disclaimed in individual files.

The authors hereby grant permission to use, copy, modify, distribute, and license this software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. No written agreement, license, or royalty fee is required for any of the authorized uses. Modifications to this software may be copyrighted by their authors and need not follow the licensing terms described here, provided that the new terms are clearly indicated on the first page of each file where they apply.

IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE AND ITS DOCUMENTATION ARE PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Contents

1	Getting Started	1
1.1	Invocation from the OPI	1
1.2	Invocation from the Shell	1
2	Ozcar's Main Components	3
2.1	Thread View	4
2.1.1	Thread States	4
2.1.2	Thread Selection	4
2.2	Stack View	4
2.2.1	Stackframe Selection	4
2.3	Variable View	5
2.4	Status Line	5
2.5	Source view	5
2.5.1	Color scheme	5
3	A First Interaction	7
4	Execution Control	11
4.1	Step Points	11
4.2	Action Step Into	12
4.3	Action Step Over	12
4.4	Action Unleash	12
4.5	Example	12
4.6	Breakpoints	13
4.6.1	Static Breakpoints	13
4.6.2	Dynamic Breakpoints	15
5	Environment Inspection	17
5.1	The Query Dialog	17
5.1.1	Evaluation of Expressions	17
5.1.2	Execution of Statements	18

6	Exceptions	21
7	Reference Section	23
7.1	The Main Menu	23
7.1.1	Ozcar	23
7.1.2	Action	23
7.1.3	Thread	24
7.1.4	Stack	25
7.1.5	Options	25
7.2	The SubThreads Menu	26
7.3	The Queries Menu	27

Getting Started

There are two ways in which Ozcar can be invoked. First, from the OPI, second, from the Shell.

1.1 Invocation from the OPI

To start Ozcar from the OPI, you have to apply the elisp function `oz-debugger`, which is bound to the key sequence `C- . C- . d` by default. Executed with prefix argument (`C-u C- . C- . d`), the Ozcar window is closed again. Note, however, that the debugger will only be suspended; after re-starting it, you can continue debugging exactly where you stopped before.

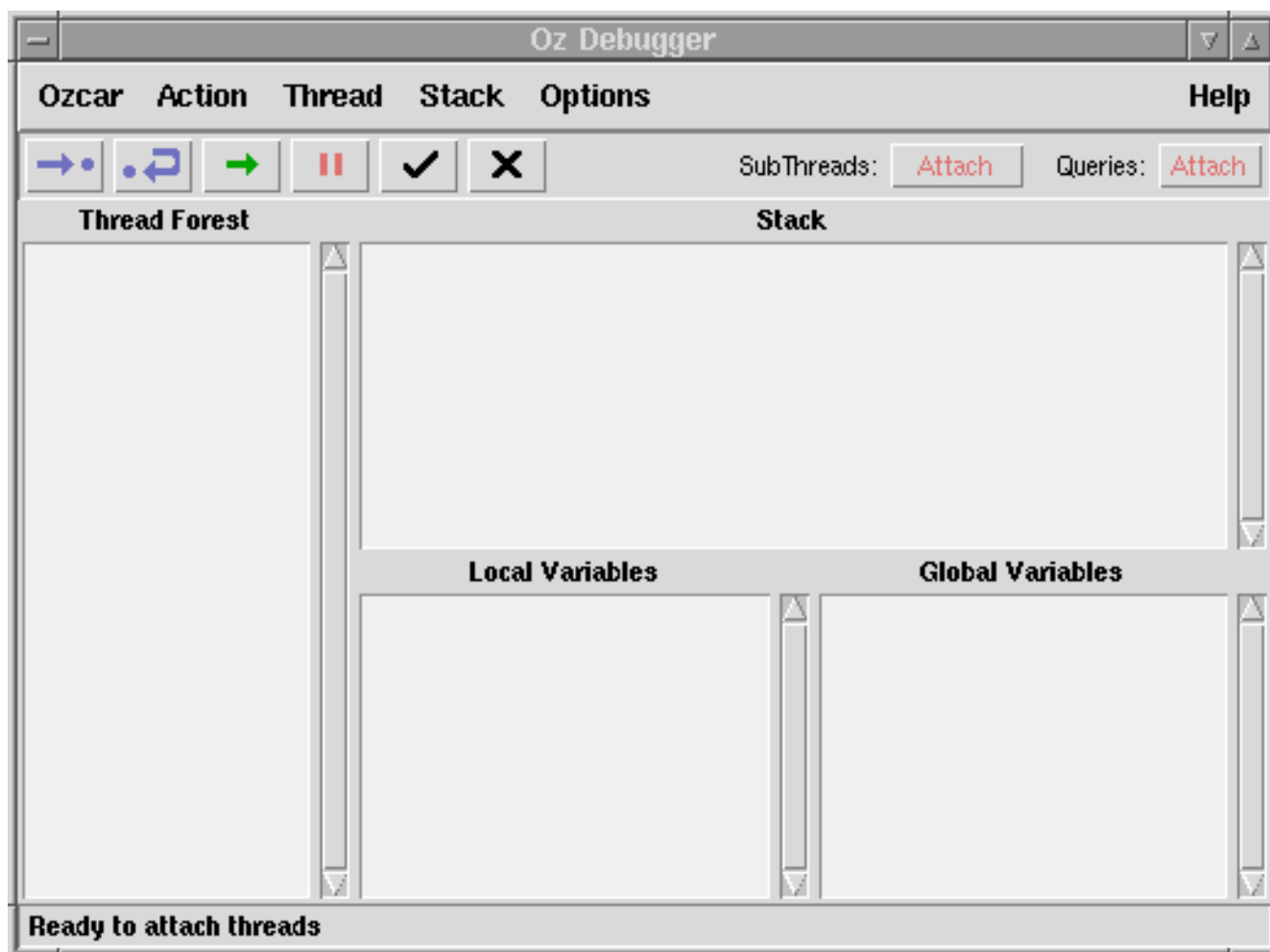
1.2 Invocation from the Shell

Another way of using Ozcar is to debug Oz applications (see Chapter *Getting Started, (Application Programming)*). For this purpose, the program `ozd` is provided (which is an oz application itself, btw.). If you want to debug the application `foo`, you just type from the shell `'ozd -E foo'`. This starts an Oz engine, which in turn starts Ozcar and Emacs (the latter because of the `-E` option). See also Chapter *The Oz Debugger: ozd, (Oz Shell Utilities)*.

Ozcar's Main Components

Figure 2.1 shows the main window after Ozcar has been invoked for the first time.

Figure 2.1: The main window



2.1 Thread View

On the left, there is a window labeled Thread Forest. Here you can see all the threads which are currently attached. Their hierarchical dependencies are illustrated by printing them as nodes of a tree: Children are always inserted below their parent, indented to the right.

2.1.1 Thread States

Different node shapes correspond to different thread states: The state of a thread can be seen as a pair of two values. The first one determines if the thread is currently stopped by the debugger. Nodes of such threads are printed with a normal font, running threads are printed in bold face. The second component of the pair can have one of the following four values:

runnable	The thread is runnable. This means it can be scheduled by the virtual machine. Such a thread is printed in green.
blocked	The thread waits for a synchronization condition. It cannot be scheduled by the virtual machine. Such a thread is printed in yellow.
crashed	The thread got an unhandled exception. Such a thread is printed in red.
dead	The thread is dead. Such a thread is printed in grey.

2.1.2 Thread Selection

One thread (if there is any one attached) is the selected thread; its node is marked with an asterix. Some actions, like single stepping or selecting stack frames, are always relative to this thread. You can select a thread by clicking on it with the left mouse button or by using the left and right cursor keys.

2.2 Stack View

Right beside the thread window, there is another window labeled Stack or Stack of Thread <id>. It prints the stack of the currently selected thread (if there is any). Beware: the topmost frame is displayed at the bottom. The procedure arguments are printed in bold face and can be further investigated (using the Inspector) by clicking on them.

2.2.1 Stackframe Selection

One frame is the selected frame. It is displayed white on blue. Initially, the topmost frame is selected implicitly, without being marked in any way. You can navigate through the stack by clicking on a frame (search for a position within the line where no arguments are displayed!) or by using the up and down cursor keys.

When the current thread is running for a longer time, its stack is printed in grey to visualize that the display is out of date.

2.3 Variable View

Below the stack window, there are two windows, labeled Local Variables and Global Variables, to display information about local and global variables of the currently selected stack frame. The local environment is sorted by introduction order of the variables in the source code, the global environment is sorted alphabetically. As with the arguments in the stack window, you can click on the (bold faced) values of the variables to inspect them.

2.4 Status Line

At the bottom of the main window, there is a Status Line which is used to display miscellaneous useful information.

2.5 Source view

This view is the only one which is not located inside Ozcar's main window. Instead, Emacs serves for this purpose by highlighting source code lines appropriately while single stepping.

2.5.1 Color scheme

Emacs gives the highlighted lines different colors, depending on how the line was reached by the corresponding thread. The idea is quite simple: If nothing really special has happened (just some booring step into or unleash), the color is blue, otherwise, that is, if a breakpoint has been reached or an unhandled exception was raised, the color is red.

A First Interaction

So, how can Ozcar actually be used? Consider the following small program:

```
local
  S = 'hello'
in
  {Show S # ' world!'}
end
```

Attaching the Thread Let's check how this code is executed by the virtual machine of mozart—step by step. The default behaviour of Ozcar is to ‘catch’ all threads which are created by Emacs queries, so all you have to do in order to debug the code is to feed it in the normal way, for example using the Emacs function `oz-feed-paragraph` (`C-c C-p`). You will see Ozcar printing the following: First, you are informed by the status line that there has been attached a new thread with thread number 53, and this thread was selected automatically. In the thread forest window, you see a node, labeled with the thread number, and tagged with an asterix.

Checking the Environment The thread has been stopped directly after its creation, so the stack window displays a single stack frame, containing the first application of the program, `S = 'hello'`, which is, translated into core syntax, `{Value.'=' _ hello}`. The variable name seems to be lost, and in fact, it is, at least in the stack window. But, there is no need to despair, we have our variable windows, and indeed, the variable `S` can be found in the local environment, still unbound (visualized by the underscore char). The right arrow at the beginning of the line in the stack window means: You are about to apply the function, but you haven't yet! The number beside the arrow means: This is stack frame number one.

In Emacs, the source line which contains `{Value.'=' _ hello}` is highlighted, like this:


Single Stepping Now, how can we apply the function `Value.'='`? One way is to use the mouse, clicking on the  button in the top left corner of the main window. This activates the function ‘step into’, which would enter the body of the current application. But since ‘Value.’=’ is a primitive procedure with an implementation in C++, there is no visible body and thus, execution stops again directly after leaving the application, detectable by the left arrow at the beginning of the stack frame line.

Figure 3.1: The main window after feeding the program

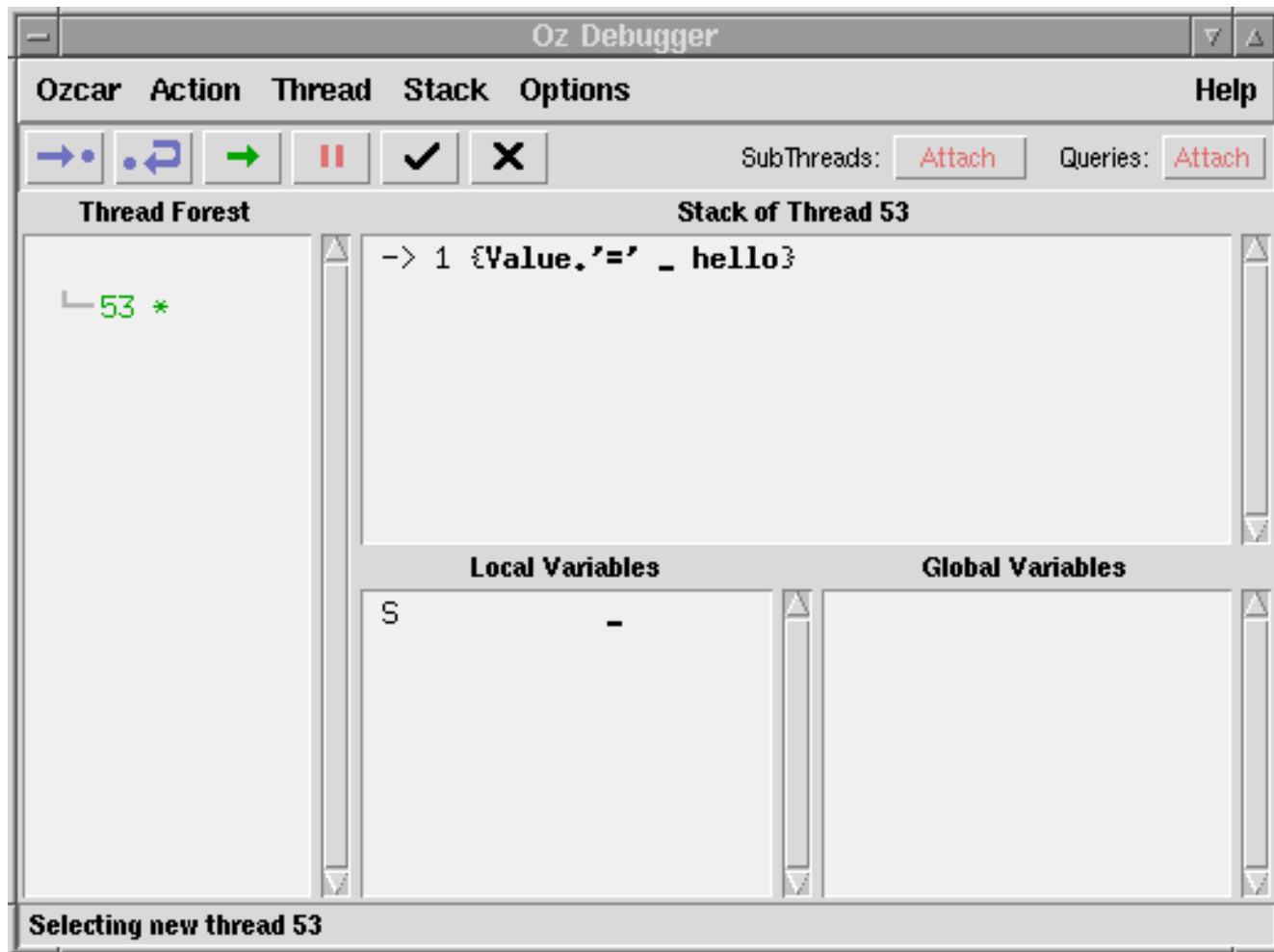




Figure 3.2: Emacs shows the exact position of the stopped thread



Detaching the Thread You can click another two times on the  button to apply also the 'Show' application. Finally, the stack is empty, and the thread terminates. You see the thread's node in the thread view changing its color to grey. To detach the thread, simply click on the  button.

Execution Control

This chapter describes the various possibilities to execute only well defined parts of your program. First, the positions must be defined where the execution of a thread can be stopped.

4.1 Step Points

These positions are called step points. What follows is a list of all currently defined step points. Note that all such points are dual: Both entry and exit point of the corresponding syntactic construct actually constitute a step point. Emacs always highlights the whole line where the thread is currently stopped, together with a second mark which shows the column within this line where the syntactic construct starts or ends.

Definition A thread can be stopped just before and just after the definition of a procedure, function or class. Emacs highlights the `proc`, `fun`, `class` or `end` keyword, respectively. The stack view displays a stack frame containing the single word `definition`.

Application You can stop a thread right before and after a procedure, function or method application. Emacs highlights the opening and closing brace, respectively. The stack view displays a stack frame which contains the procedure, function or method name (if there exists any, `$` otherwise), together with the argument list. All arguments can be inspected by clicking on them with the left mouse button.


Conditional You can stop a thread before entering and after leaving a conditional. This makes it possible to investigate the arbiter. Emacs highlights the `case` and `end` keyword, respectively. The stack view displays a stack frame containing the word `conditional`, followed by the clickable value of the arbiter.

Thread Creation A thread can be stopped whenever a new thread is created explicitly. Emacs highlights the `thread` and `end` keyword, respectively. The stack view displays a stack frame containing the single word `thread`.


Installation of Exception Handlers A thread can be stopped when an exception handler gets installed. Emacs highlights the `try` and `end` keyword, respectively. The stack view displays a stack frame containing `exception handler`. Moreover, when an exception is caught, the thread can stop on the corresponding `catch` clause, and in the stack window you see the single word `catch`, followed by the clickable pattern that matched.

Entering Locks Finally, you can stop a thread when a locked code block is entered or leaved. Emacs highlights the `lock` and `end` keyword, respectively. The stack view displays a stack frame containing the single word `lock`, followed by the clickable value of the lock.


4.2 Action Step Into

The simplest way how to control the execution of a thread is to single step from step point to step point. This is exactly what step into () is doing. You get a very detailed view of how your program is executing. Often, it will be too detailed; this is where step over and unleash enter the scene.

4.3 Action Step Over

Whenever execution has stopped on a step point, you can decide if you want to enter the corresponding inner block (i.e. the procedure body, the case clause, etc). If yes, you do a step into (see above). If not, you do a step over (). This continues execution until the corresponding block has been left again. Typically, Emacs will show you standing on an `end` keyword or a closing brace then.

4.4 Action Unleash

Actually, step over is a special case of a more generic action, which is called unleash (). It continues the execution of the thread until the currently marked stack frame is just to be removed from the stack, or until the thread has finished executing the whole stack if no stack frame is marked. Remember you can mark a stack frame by clicking on it or by walking to it using the up and down cursor keys.

4.5 Example

To illustrate the functionality, consider the following program which calculates the faculty:

```
local
  fun {Fac N}
    if N < 2 then 1 else
      N * {Fac N-1}
    end
  end
in
  {Show {Fac 5}}
end
```


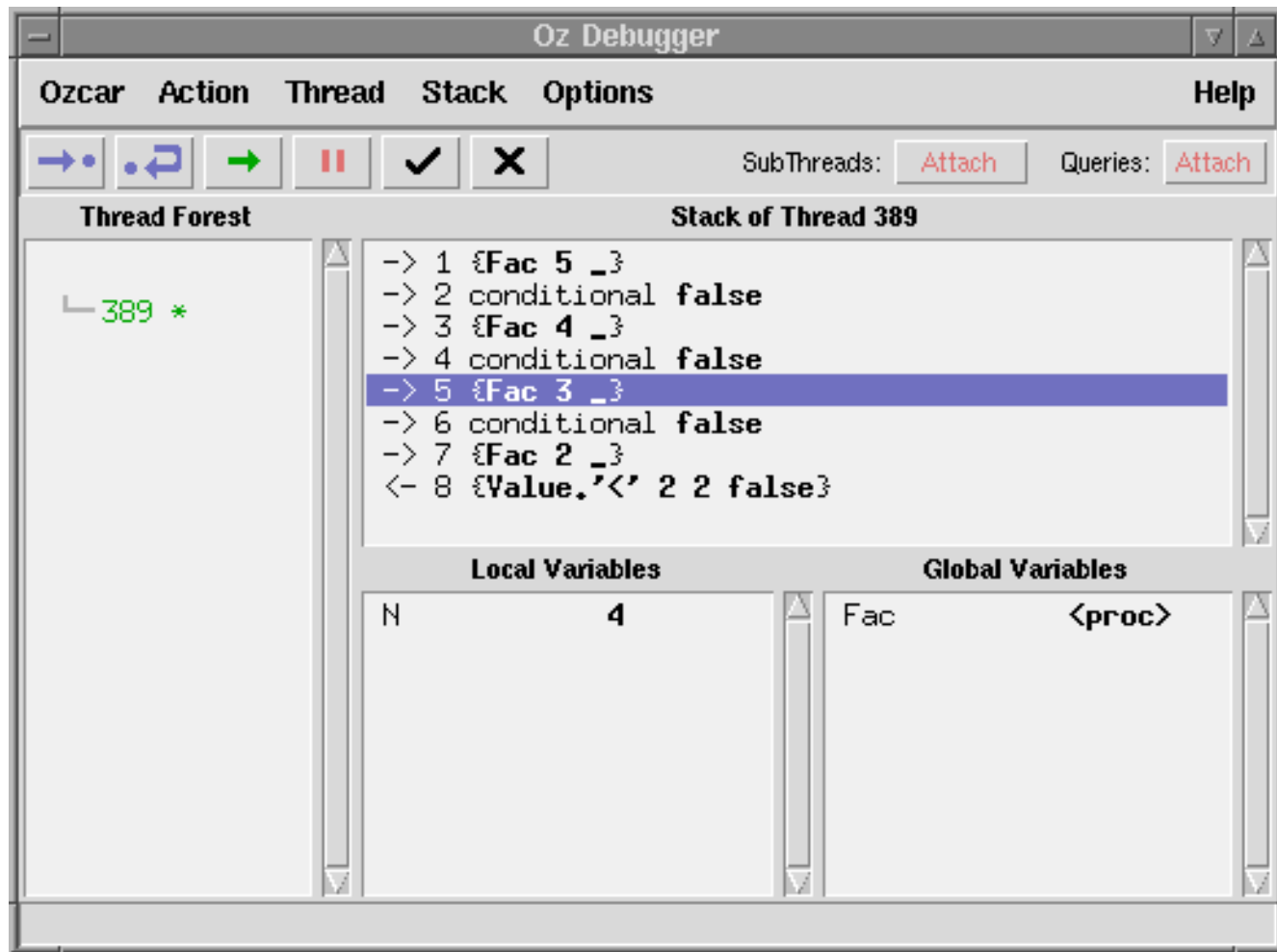

Suppose you already pressed the  button some times, so that there was build a nice stack already, and that you have marked stack frame 5 by clicking on it. What you will see is something like this: You decide to directly compute the value of `{Fac 3}`,

Figure 4.1: Before the action unleash 5



so what Ozcar needs to do is to continue the thread's execution until stack frame 5 is just to be removed from the stack. Unleash 5, activated by clicking on the  button, does exactly this:

4.6 Breakpoints

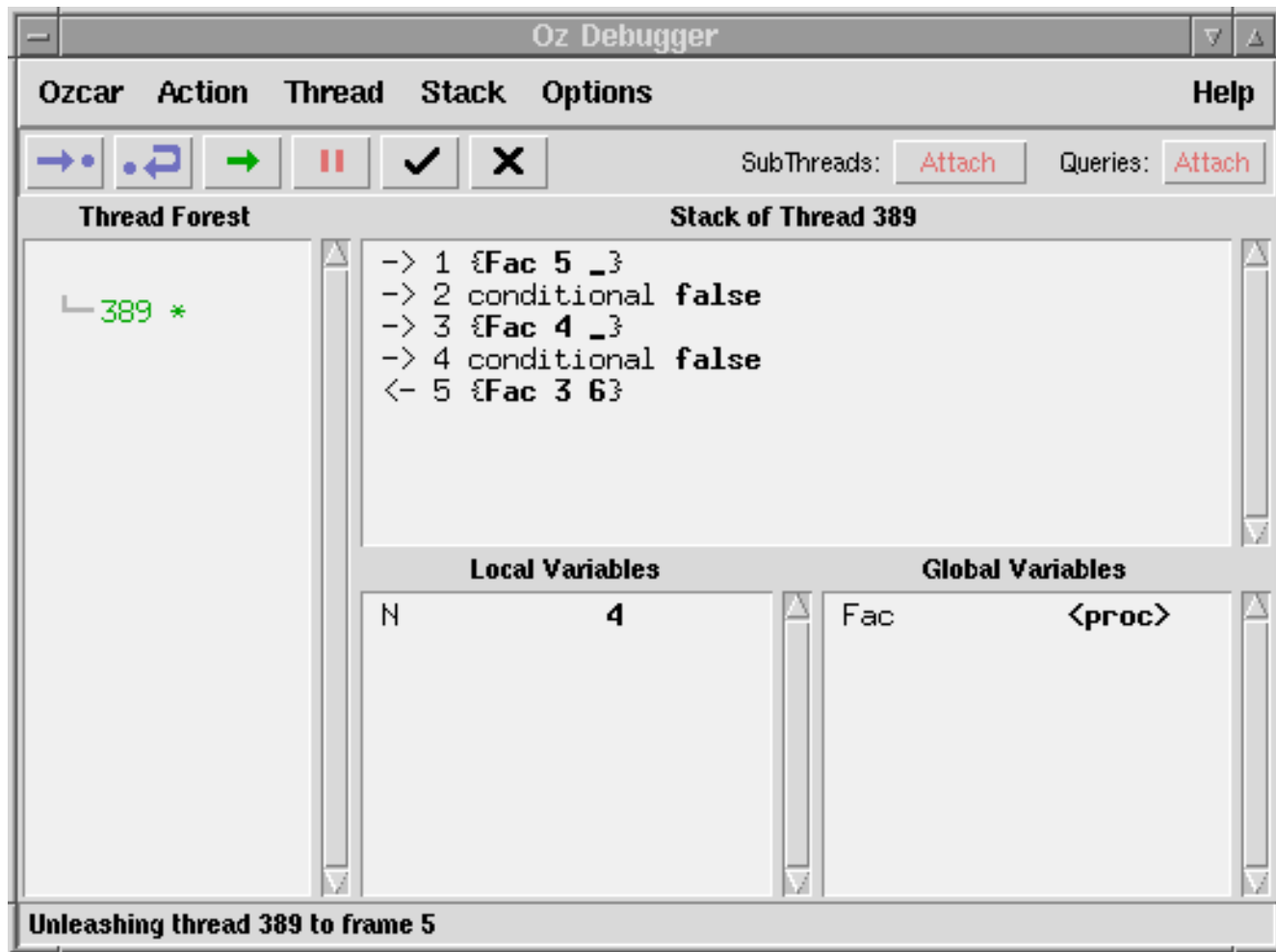
Single stepping is nice, but often somewhat inconvenient, because you need a lot of steps until the interesting section of your program is reached. Breakpoints help you here. Ozcar supports two flavours of them: static and dynamic ones.

4.6.1 Static Breakpoints

Let's assume you need to debug the base case of the recursion. This can easily be achieved by inserting a special breakpoint procedure, like this:

```
local
```


Figure 4.2: After the action unleash 5



```

fun {Fac N}
  if N < 2 then
    {Ozcar.breakpoint} 1
  else
    N * {Fac N-1}
  end
end
in
  {Show {Fac 5}}
end

```

After feeding the code and pressing the  button two times, you are directly at the desired position. Static breakpoints are useful if you want the breakpoint to survive multiple invocations of the mozart system. They are inserted before you feed your code.

On the other hand, you might decide to insert a breakpoint after you feeded the code. Then you need dynamic breakpoints.

4.6.2 Dynamic Breakpoints

This flavour of breakpoints can be set directly from Emacs: You position the cursor on the line and column where you want to change breakpoint information and press (C-x space) to set or (C-u C-x space) to delete a breakpoint.

Unfortunately, there is no information yet about all currently defined dynamic breakpoints. You just have to keep them in mind. This should be changed in the future.

Environment Inspection

In this chapter you learn how to investigate the environment, i.e, how to access the values of variables. In fact, we must distinguish between two kinds of environments in Oz. First, there is the toplevel environment. Here you find all the variables which are defined right after you started the mozart system. The procedure `Show` is an example, the module `String` is another. Second, there exist environments for each stack frame (i.e. each called procedure) of a thread. They consist of local variables, which are the union of locally defined variables and formal parameters, and global variables, which are referenced inside the procedure, but defined outside.

Environment inspection is easy with Ozcar. In fact, it is done automatically for you. Whenever you select a stack frame, the variables which are visible inside this frame are printed in the variable windows. You can inspect the values by clicking on their type information.

5.1 The Query Dialog

There might be situations where you have to operate on some values which are found in the local or global environment. For example, you want to convert a data structure, or to bind an unbound variable which causes your program to hang.

For this purpose the query dialog exists. You can open it by selecting the Query... menu entry in the Stack menu.

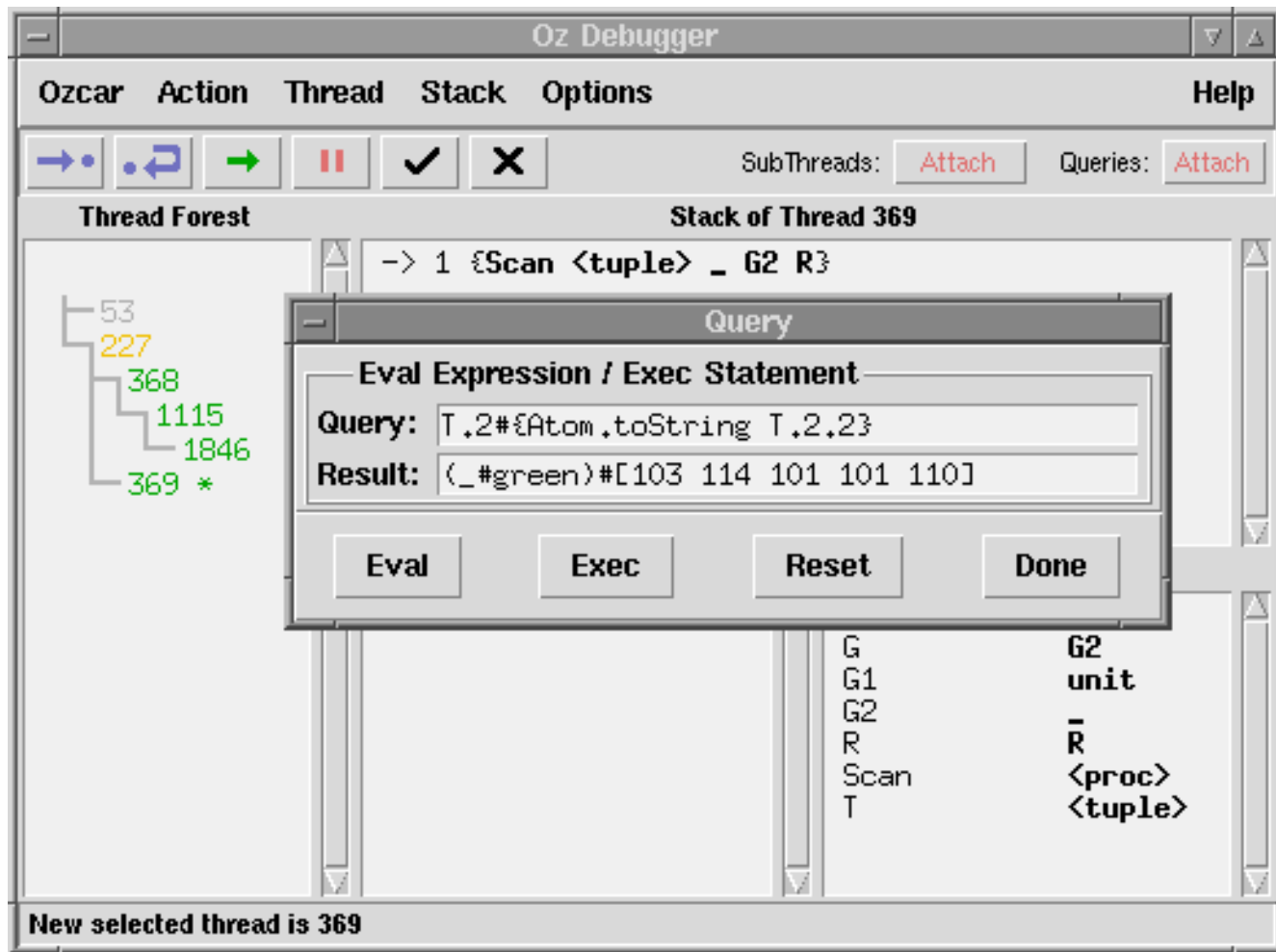
5.1.1 Evaluation of Expressions

Using this dialog, you can evaluate arbitrary Oz expressions. For example, if you type

```
{fun {$ A B C} A+B*C end 1 2 3}
```

in the Query line and press the Eval button, you get the expected result, 7, in the Result line. Another example shows the following picture:

Figure 5.1: You can evaluate arbitrary expressions

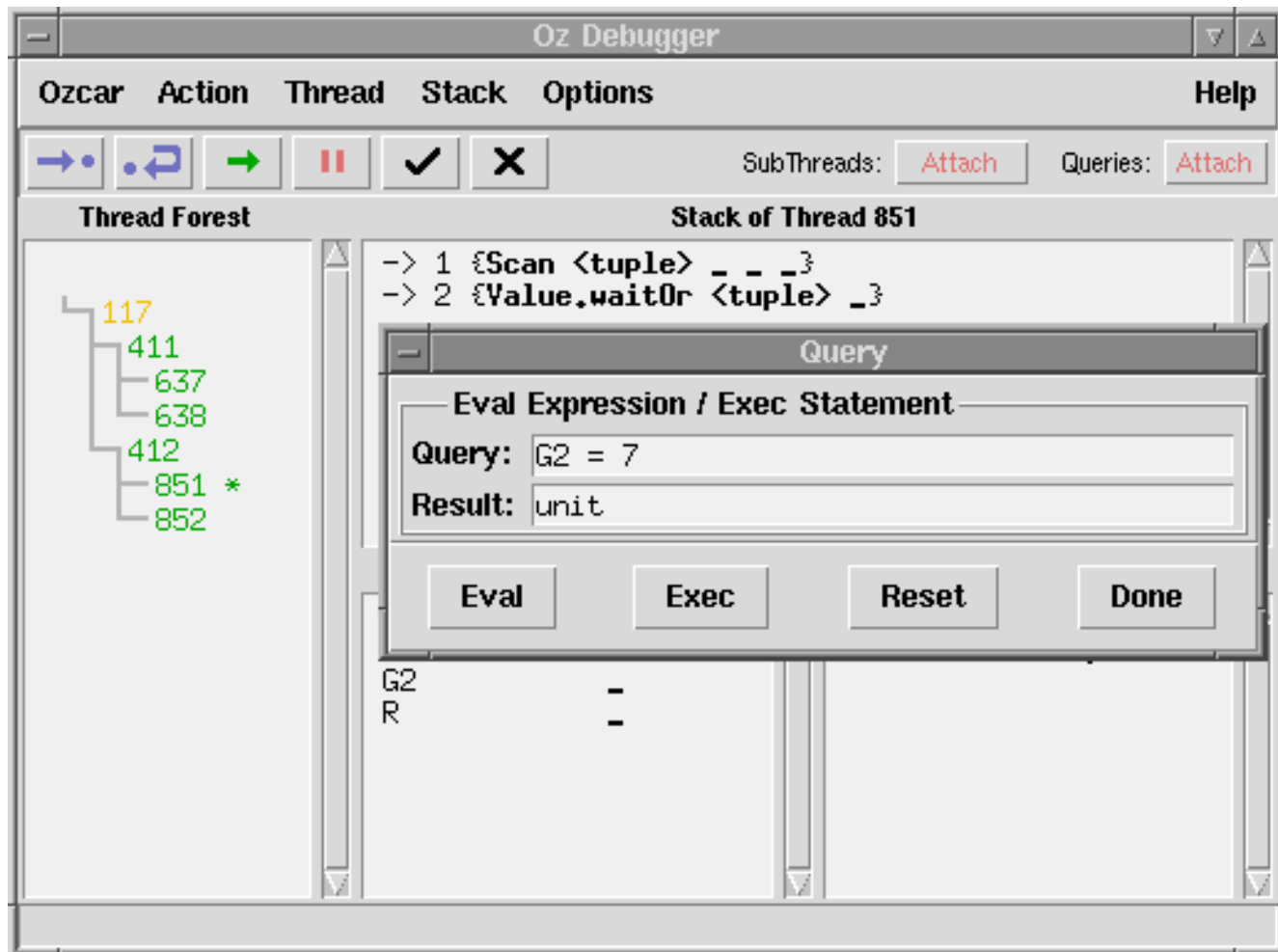


5.1.2 Execution of Statements

Something other you can do with the Query dialog is to execute Oz statements. For example, when you type `{Inspect hugo}` and press the Exec button, the Inspector pops up and prints the obvious data. Note that the result of executing a statement is `unit`. Another example comes with the next picture. An unbound variable, `G2`, is bound to the value `7`. You would have to reprint the variable view's content (by reselecting the stack frame) in order to update its information.

You can open as many Query dialogs as you like. This makes it possible to evaluate several expressions more than one time without always re-entering them.

Figure 5.2: You can execute all kinds of statements



Exceptions

Errors in programs often manifest themselves by threads throwing unhandled exceptions. It would be nice to have a post mortem stack to check how the position of the big bang was reached. Moreover, the environment should be visible.

Ozcar provides this information. Exceptions which would normally be printed in the Emulator buffer are caught by Ozcar, which prints the post mortem stack, together with an explanation of the exception in the status line. All stack frames provide variable information.

In the following example there was forgotten an `else` branch in the case statement:

```
local
  proc {Check X}
    case X
    of foo then {Show 'This is a foo'}
    [] bar then {Show 'This is a bar'}
    end
  end
in
  {Check foobar}
end
```

Emacs prints the location in the source where the error happened. Note that the bar is printed in red, as you reached the position unexpectedly.

Note that the thread is just about to die; if you do a single step into, it will actually terminate, and the post mortem stack vanishes.

Figure 6.1: An unhandled exception has been raised

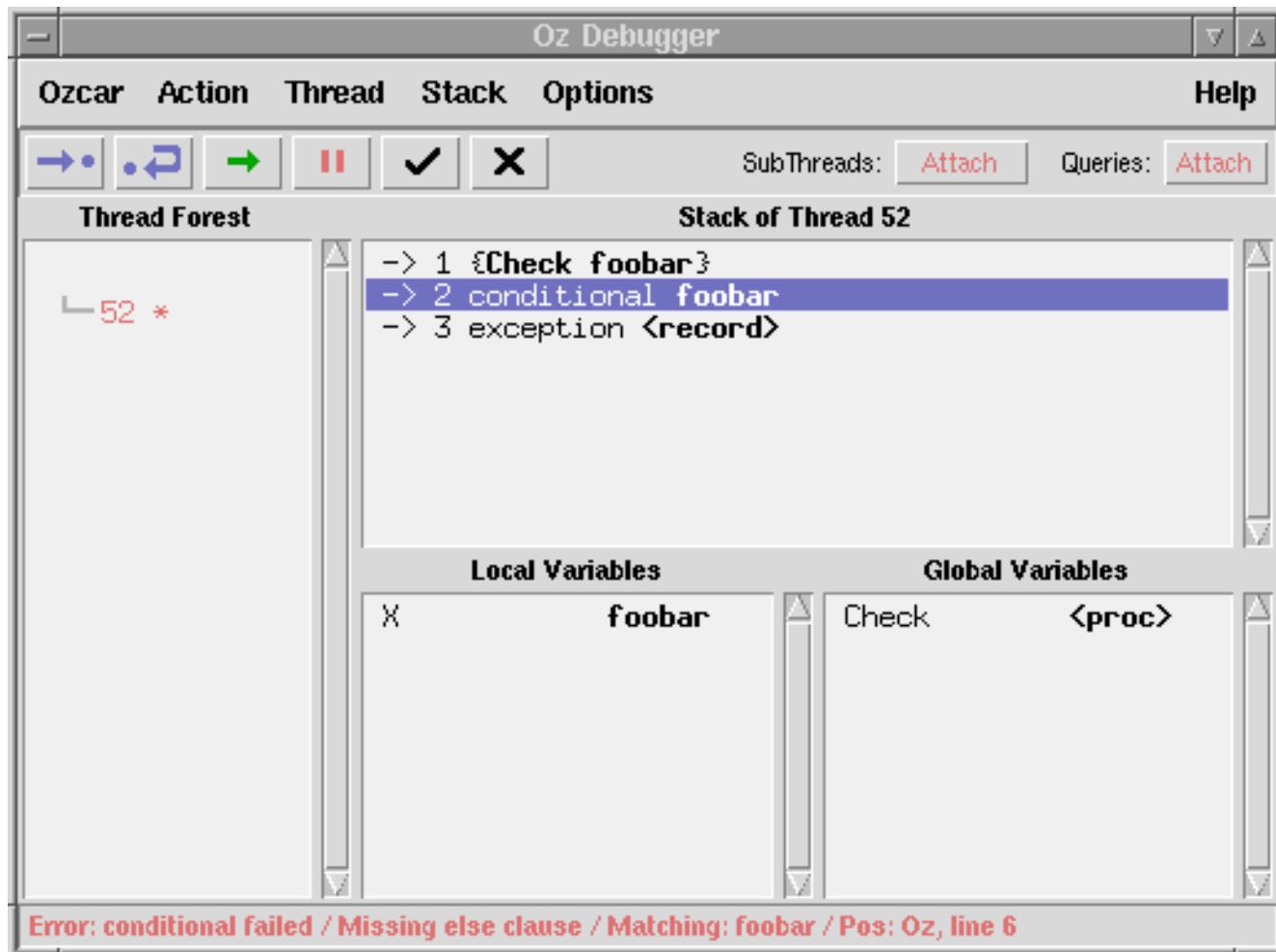
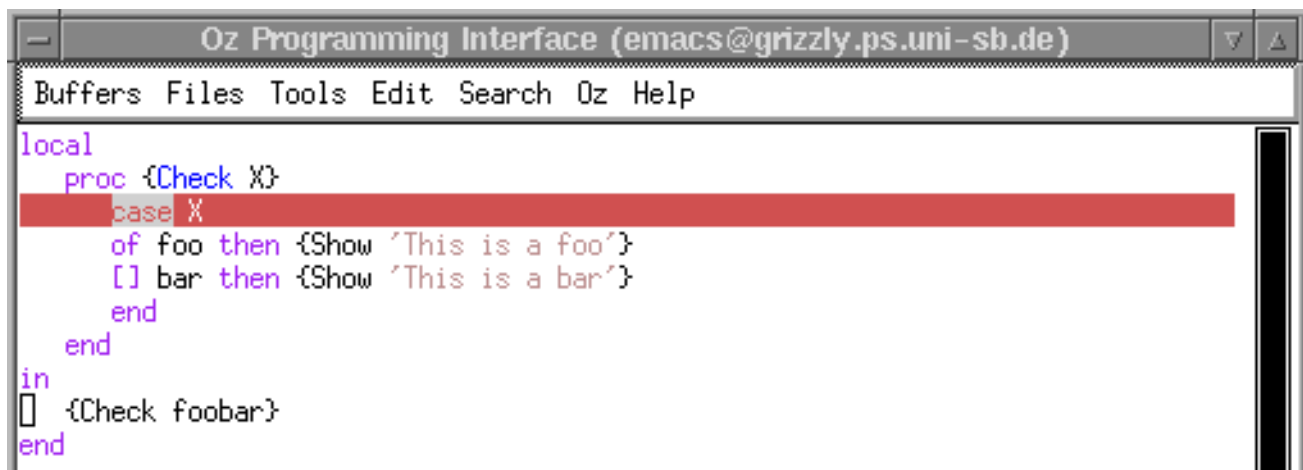


Figure 6.2: Emacs shows the position where the error occurred



Reference Section

This chapter gives a complete description of all the menus Ozcar provides.

7.1 The Main Menu

The main menu is located on the top of Ozcar's main window.

7.1.1 Ozcar

About...

Pops up an info box containing author and compilation information

Destroy

Closes the internal Ozcar object. Use this menu entry only if Ozcar hangs, as all information about currently attached threads gets lost.

Suspend


C-x

Closes the main window, sets the mozart emulator to non debug mode, and causes the compiler to generate non debug code. Information about currently attached threads is preserved.

7.1.2 Action


Step Into

s

Do exactly one step (stop again at the next step point, enter procedures). Identical to pressing the  button.

Step Over

n


Do one step, jump over procedures or other blocks which constitute a step point. Identical to pressing the  button.

Unleash

c

Continue execution until the marked stack frame is about to be removed from the stack, or until the stack is empty if no frame is marked. Identical to pressing the  button.


Stop	z
------	----------

Stop the current thread at the next step point it reaches. Identical to pressing the  button.

7.1.2.1 The Detach submenu

This menu contains some useful entries to detach one or more attached threads:

Current	d
---------	----------

Detach the current thread. Identical to pressing the  button.

All But Current	C-d
-----------------	------------

Leave the current thread alone in Ozcar's thread forest. Detach all the others, let them continue to run (if they are not dead).

All	M-d
-----	------------

Detach all threads, let them continue to run (if they are not dead).


All Dead	M-u
----------	------------

Detach all dead threads.

7.1.2.2 The Terminate submenu

This menu contains some useful entries to detach *and kill* one or more attached threads:

Current	t
---------	----------

Detach and kill the current thread. Identical to pressing the  button.

All But Current	C-t
-----------------	------------

Leave the current thread alone in Ozcar's thread forest. Detach and kill all the others.

All	M-t
-----	------------

Detach and kill all threads.

7.1.3 Thread

Previous	Left
----------	-------------

Select the thread which is located above the currently selected thread in the thread view. If the current thread is the first thread in the thread view, the last thread (at the bottom) is selected.

Next	Right
------	--------------

Select the thread which is located below the currently selected thread in the thread view. If the current thread is the last thread in the thread view, the first thread (at the top) is selected.

Status	C-s
--------	-----

Print some useful information about the currently attached threads, for example: 2 *attached threads, currently selected: 58/1 (stopped, runnable)*. The first part of the information is obvious. The two numbers associated with the selected thread are the thread id and the parent thread id. In parentheses, information is given if the selected thread is stopped or running, and if it is runnable, blocked or terminated.

7.1.4 Stack

Previous Frame	Up
----------------	----

Select the previous stack frame (if it exists) of the currently selected thread. Note that the stack grows to the bottom of the window, so you will reach an older frame.

Next Frame	Down
------------	------

Select the next stack frame (if it exists) of the currently selected thread; you will reach a younger frame.

Re-Calculate	C-1
--------------	-----

Update the stack view. You may see special, compiler-generated stack frames then, as well as some variables (which have been bound in the meanwhile) appear with their value instead of an underscore. (There is *no* automatic update which forces the stack view to display a variable's value as soon as it becomes known.)

Update Env	v
------------	---

Immediately re-calculates and re-displays the environment of the selected stack frame.

Query...	e
----------	---

Opens a dialog box where (small) statements can be executed or (small) expressions can be evaluated in the context of the currently selected stack frame. If the computation needs some more time, you see a spinner turning around until the computation ends. If it does never end (for instance because of a blocking thread), use the Reset button to cancel the operation.

7.1.5 Options

7.1.5.1 The Value Printing submenu

In this menu you can determine how to display values in the stack and variable windows. There is a short form which only prints type information, and a long form which prints the actual value (up to a given print width/depth, which can be set in the Preferences dialog box, see below).

Types Only	<
------------	---

This is the default setting. You just see type information, which leads to a very compact display style.

Complete >

You see the actual value, using the function `Value.toVirtualString`. This display mode can be quite unhandy if the values to be printed are large tuples or records. So you should carefully adjust the print depth and print width to your needs.

Use Emacs C-e

If turned off there are no bars printed within Emacs, so you don't get any position information.

Env Auto Update C-a

If you debug threads with huge environments, it may be a good idea to turn of the auto update of the environment (re-calculation and re-display every time you make a step) in order to save time and memory. While the variables display is not up to date, it is printed in grey. You can always request the newest environment information by pressing 'v' (see above, function 'Update Env').

Preferences... C-o

Opens a dialog box to adjust some self explanatory options. For example you can adjust the print width/depth of values in the stack and variable windows.

7.2 The SubThreads Menu

This menu is located on the right side of the button bar, below the menu bar. You can select the mode how child threads of *already attached* threads should be treated.

Ignore

They will not be attached at all (but can silently run of course).

Attach

They are attached and immediately stopped.

Unleash 0

They are attached and the command 'Unleash 0' is executed. This gives you a good feeling about the dependencies between all your threads in a concurrent application. They are already dead (if they do not block somewhere), but are still printed in the thread view.

Unleash 1

They are attached and the command 'Unleash 1' is executed. This makes it possible to even explore the environment of every child thread just before it terminates.

7.3 The Queries Menu

This menu is located on the right side of the button bar, right beside the SubThreads menu. You can select the mode how a thread which is born by setting up an Emacs query is treated by Ozcar.

Ignore

It is not attached, neither are any of its subthreads (except, of course, they run into a breakpoint).

Attach

It is attached and immediately stopped.