The Mozart Debugger

Benjamin Lorenz
Leif Kornstaedt

Version 1.3.2
June 15, 2006

Abstract

This manual describes Ozcar, a symbolic debugger which provides well-known debugging features such as breakpoints, single-stepping, and environment inspection. It supports debugging of multiple threads and debugging of distributed applications. Concurrent behaviour can be observed and manipulated.

Ozcar can be used from within the Oz Programming Interface or it can be used as a standalone program to debug Oz applications. It uses Emacs as a source viewer, highlighting the current position in a debugged program, and allowing to set breakpoints at arbitrary source lines.

Credits

Mozart logo by Christian Lindig

License Agreement

Contents

1 Introduction

Thank you for choosing Ozcar, your friendly debugger. This manual will help you to make first steps in using Ozcar, hopefully enabling you to make proper use of it for your daily development work in Oz, and will serve as a reference to its functionality and peculiarities.

Manual Structure Chapter 2 describes how Ozcar can be started, followed by a description of the components of its user interface in Chapter 3. Chapter 4 aims to convey an intuition of how to interact with this user interface; if you found Chapter 3 a hard read, you might want to look at it again after this. The fundamental knowledge of how to control debugged programs and how to examine their data is presented in Chapter 5 and Chapter 6. The more advanced features of post-mortem debugging in the presence of unhandled exceptions and remote debugging of distributed applications are dealt with in Chapter 7 and Chapter 8, respectively. Finally, Chapter 9 summarizes limitations to remember when working with Ozcar.

Appendices The reference section of this manual offers detailed descriptions of all of Ozcar's menu items in Appendix A, and of the Application Programming Interface in Appendix B.

Invoking the Debugger

There are several ways to start Ozcar: from the Oz Programming Interface, during interactive development; and from the shell, to debug standalone applications. Furthermore, an API is provided to enable user applications to control the debugger.

Startup Effects When Ozcar is started, the following actions will be performed:

- The interaction window is opened.
- Concurrent threads are observed: When a thread reaches a breakpoint or raises an unhandled exception, the debugger takes control of this thread. This is called attaching a thread.
- When running under the OPI, the compiler's switches are configured to generate code with debug information.

2.1 Debugging within the Oz Programming Interface

Ozcar can directly be started from the OPI using the Oz menu. This executes the Emacs command `oz-debugger` (see also Chapter *Interacting With the Development Tools, (The Oz Programming Interface)*), which is also bound to the key sequence `C-. C-. d` by default. Any code you feed within the OPI will now run under control of the debugger.

Suspending Ozcar With a prefix argument (`C-u C-. C-. d`), Ozcar is suspended and its window is closed. Note that this means that all attached threads remain under the control of the debugger (but no new threads will be attached). Starting Ozcar again will allow you to continue debugging exactly where you stopped.

2.2 Debugging Standalone Applications

It is also possible to run standalone applications under control of the debugger. For this purpose, the `ozd` application is provided (which, incidentally, is itself an Oz application). For instance, to debug an application `foo`, run `ozd` as follows:

```
ozd -E foo -- args
```

This invokes Ozcar, with an associated Emacs for source handling (since the `-E` option was given), and attaches the main thread of the application `foo`.

Command Line Options For more information on `ozd`'s command line options, refer to Chapter *The Oz Debugger:* `ozd`, *(Oz Shell Utilities)*.

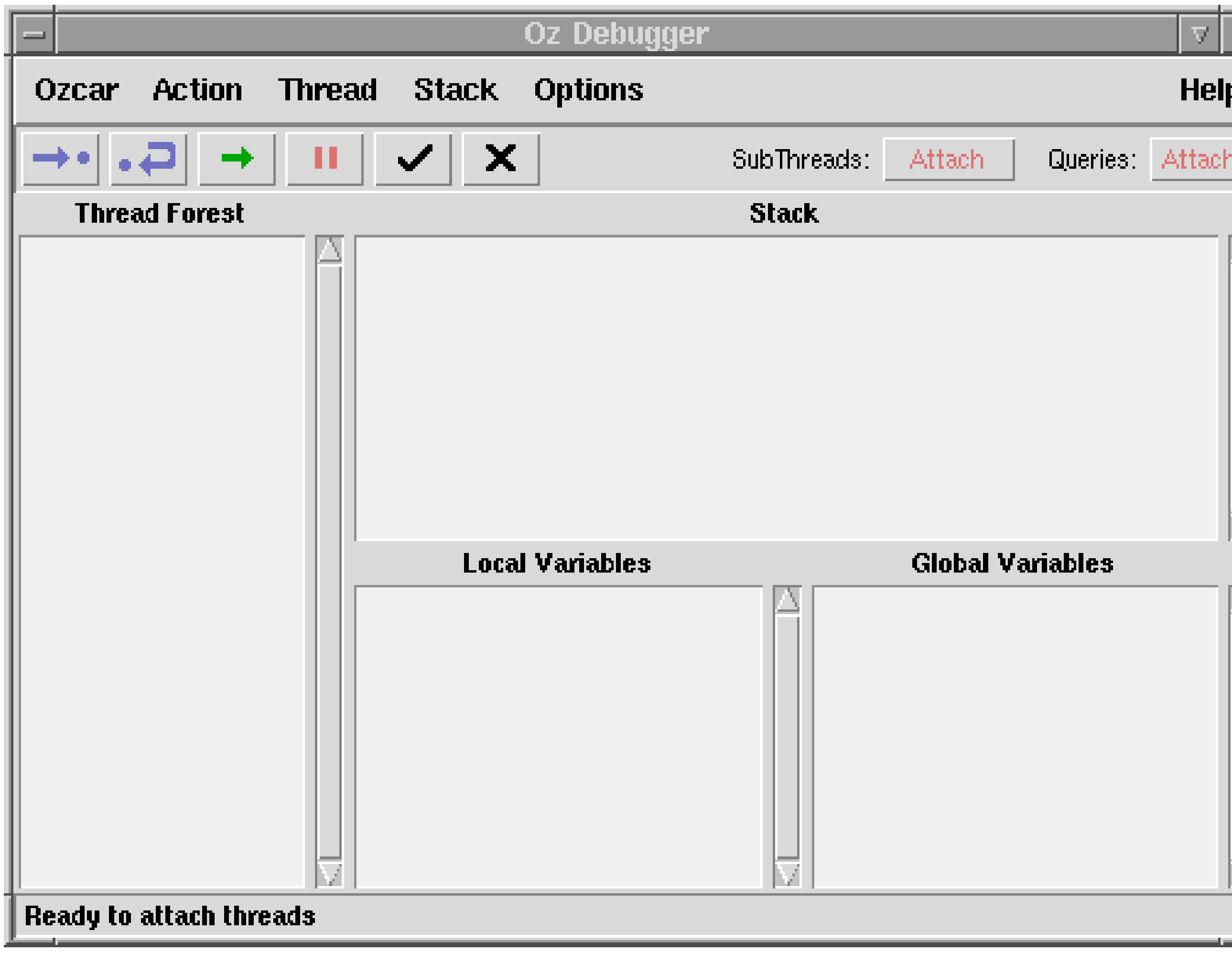
2.3 Invoking Ozcar through its API

Ozcar is implemented as an Oz functor and as such is immediately available to the application programmer. After importing the module `Ozcar` from its URL `x-oz://system/Ozcar`, Ozcar can be opened by `{Ozcar.open}` and suspended again by `{Ozcar.close}`. A detailed description of the API is given in Appendix B.

Ozcar's User Interface

Ozcar's main window, depicted in Figure 3.1 as it appears when Ozcar is first invoked, displays a number of views: the Thread Forest View, the Stack View, and the Environment Views. Below these is the Status Line, and Emacs serves as the Source View. In the following, each of these elements of Ozcar's user interface will be described.

Figure 3.1 The Main Window



Ready to attach threads

3.1 Thread Forest View

The pane on the left, labeled `Thread Forest`, gives an overview of all currently attached threads. The fact that every thread has been created by some other thread (its parent) defines a tree-shaped relation between threads. The attached threads form a number of partial subtrees of the whole thread hierarchy–this is why we refer to this view as a thread forest. The way threads are displayed reflects this tree structure: Children are inserted below their parent and indented to the right.

Thread IDs For the purpose of debugging, threads are assigned integer IDs internally. These IDs do not carry any semantic significance and are only used to allow easier recognition of individual threads by the user, as well as to construct the tree representation. Note however that thread IDs need not be unique. In the thread forest, each thread node is depicted by the textual representation of its ID. (In the case of distributed debugging, the remote site's node name and process ID are also included.)

Thread States The color of each node in the thread forest reflects the corresponding thread's state, similar to what the `Thread.state` operation returns:

green: runnable The thread's computation can proceed.

yellow: blocked The thread waits for a synchronization condition, i.e., it currently cannot proceed.

gray: terminated The thread is dead.

red: crashed The thread died of an exception it did not handle.

Stopped Threads For the purpose of debugging, thread state actually has an additional dimension, namely whether the thread has currently been stopped by the debugger or not. This is reflected by a normal font (thread is stopped) or a bold font (thread is running), respectively.

Selected Thread If at least one thread is attached, then one thread will always be the selected thread; its node is marked with an asterisk. This is the thread that thread actions operate on, and whose information is displayed in the other views. You can select a thread by clicking its node with the left mouse button, or you can navigate in the thread forest using the left and right cursor keys.

3.2 Stack View

Stack Frames On the top right, the pane labeled `Stack` or `Stack of Thread` *id* displays the stack of the currently selected thread. The stack is a sequence of frames, where each frame describes a nested construct or procedure activation. The stack provides a trace of how the computation reached its current point of execution; the latter is called the topmost frame.

Stack Display The Stack View displays the stack as one line of text per frame, where the first entry corresponds to the oldest activation frame and the last entry to the most recent frame (the *topmost* one). Each line consists of three columns. The first column contains an arrow indicating whether control is at the entry (right arrow) or at the exit of the corresponding nested construct, the second column gives the number of the stack frame, and the third column a description. The description may contain arguments (values), printed in bold face, which can be clicked on for examination with the Inspector.

Selected Frame Some debugging actions refer to specific stack frames. To this purpose, stack frames can be clicked (in a place not displaying a value) to become the selected frame. The selected frame is indicated by a blue background. The up and down cursor keys also permit navigation through the stack.

Disabled Stacks When the selected thread is running, its stack is grayed out to indicate that it is out of date, but contained values will still react to clicks.

3.3 Environment Views

Below the Stack View, there are two panes labeled `Local Variables` and `Global Variables`, displaying the values of local and global variables of the selected stack frame (or the topmost frame if no frame is selected). The variables of the local environment are sorted by order of introduction in the source code; the global environment is sorted alphabetically. Clicking the values causes them to be inspected.

3.4 Status Line

The Status Line at the bottom of the main window provides feedback about actions performed and the current state of the debugger.

3.5 Source View

The Source View is not located within Ozcar's main window; instead, Emacs is instructed to display the source position corresponding to any selected frame.

Color Coding Emacs highlights the line of source code corresponding to the selected frame in a color reflecting the selected thread's state: When the thread is running, highlighting will be done in gray; when it is stopped, in blue; and when the thread died of an exception or stopped at a breakpoint, in red.

Sample Debugging Session

In the following, we will perform a first sample session to get a feeling of how threads behave under the control of Ozcar. For this, we'll use the following small program:

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

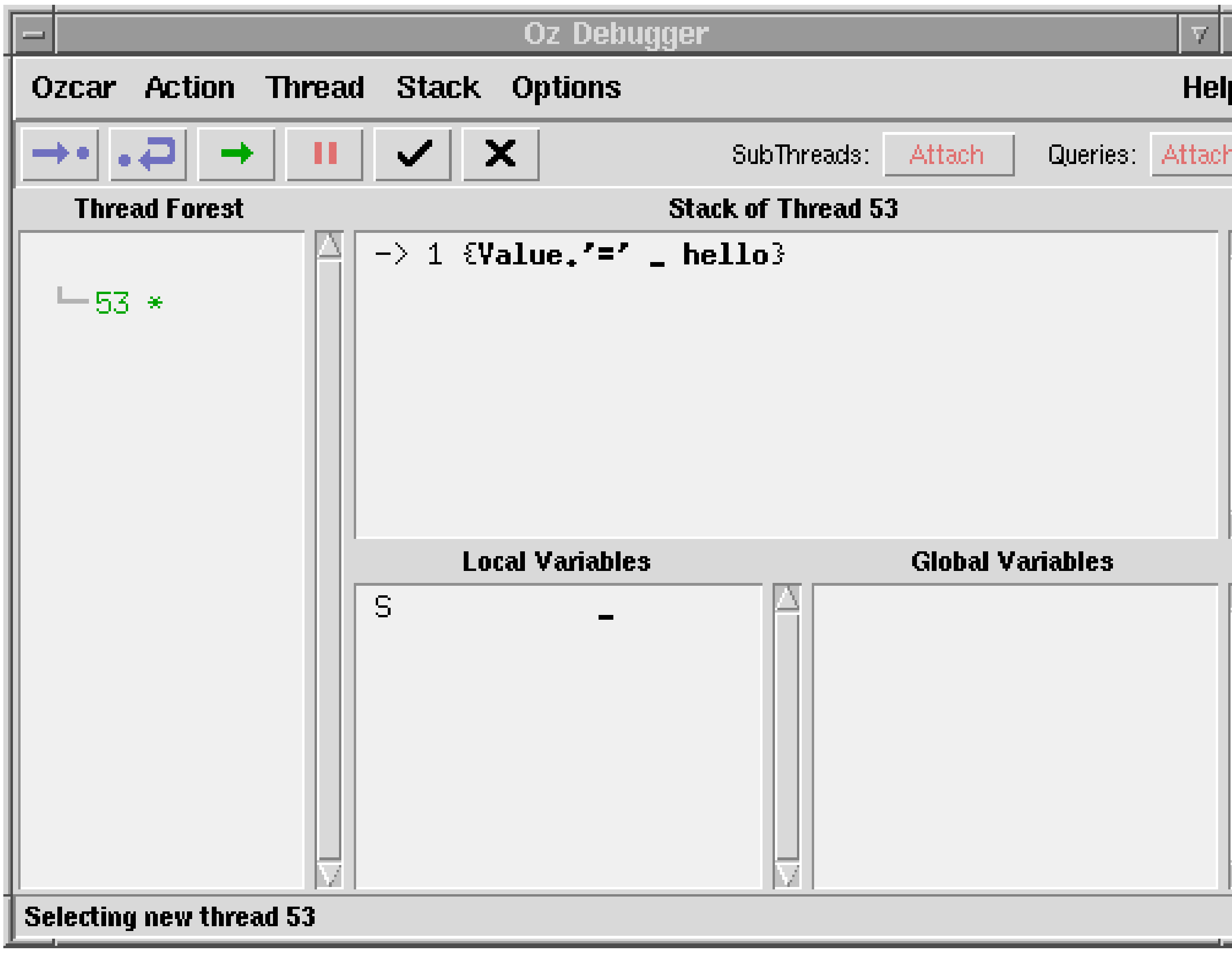
Attaching the Thread Let's check how this code is executed–step by step. By default, Ozcar attaches all threads created by feeding code in the OPI, so this is what we're going to do, for example using `oz-feed-paragraph` (`C-. C-p`). Ozcar responds as shown in Figure 4.1. The status line informs us that a new thread with ID 53 has been attached, and it was immediately selected. Accordingly, the thread forest contains a singleton tree with a node labeled 53 and marked with an asterisk.

Stack and Environment The thread has been stopped immediately after its creation, so the Stack View displays a single frame, containing the first application in the program, `S = 'hello'`, which, expressed in Oz core syntax, corresponds to `{Value.'=' _ hello}`. Instead of the variable name the variable's value is displayed in the Stack View (where the underscore stands for a logic variable). If we explicitly needed to check what the value of `S` is, we could look it up in the Local Variables View.

Source View If we look over to Emacs, we'll find that the source line which corresponds to `{Value.'=' _ hello}` is highlighted, as shown in Figure 4.2.

Single-Stepping The right arrow at the beginning of the current stack frame means: We are about to apply the function, but we haven't yet! One way to do this is to use the Step Into (➡) action, which continues execution, but only until the next stop within the body of the applied procedure. In this case, `Value.'='` is a primitive procedure, which is why execution only stops when the application returns, indicated by the arrow in the stack frame turning into a left arrow. Note also that the arguments of the procedure have been updated.

Figure 4.1 The Main Window after Feeding the Program



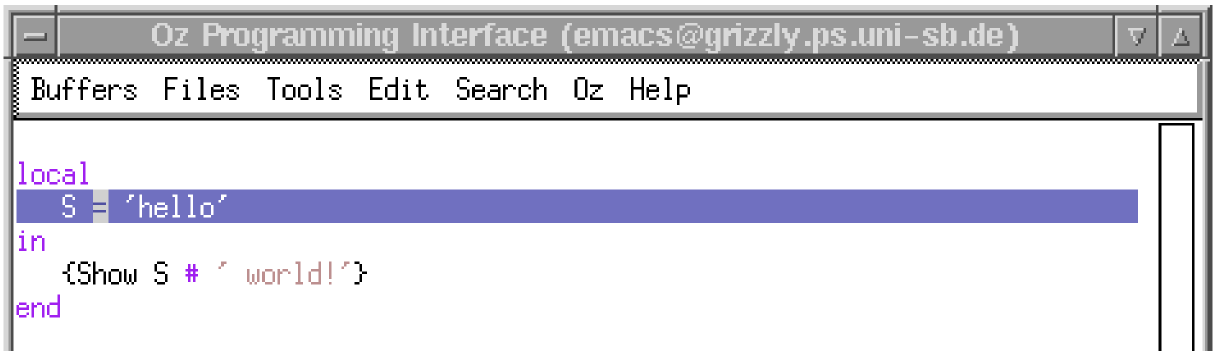
```
53 *
```

```
-> 1 {Value.'=' _ hello}
```

```
S            _
```

Selecting new thread 53

Figure 4.2 Emacs Showing the Position of the Stopped Thread



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

Detaching the Thread Performing the Step Into action another two times, we also evaluate the application of the primitive procedure `Show`, after which the stack becomes empty, and the thread terminates. Accordingly, the thread's node in the thread forest turns gray. We'll now detach this thread using the Detach (✓) action.

5

Execution Control

This chapter describes the various possibilities of exercising fine-grained control over the execution of programs. To do so, we'll first define at what program points thread execution can be stopped, the so-called step points.

5.1 Step Points

Step points are defined in terms of syntactic constructs in source code (in particular, debugging is not line-oriented). Many Oz constructs are designated to define step points, and in fact each such construct defines *two* step points: one at the entry and one at the exit. The entry point creates a new stack frame, which is popped at the corresponding exit point. When stopping at a step point, Emacs highlights the line containing it and inserts an additional mark within the line to indicate its exact position (where the precise syntactic construct begins or ends).

Constructs The following gives a non-exhaustive list of what constructs constitute step points and how they are described in stack frames.

Definitions The definition of a procedure, function or class constitutes a step point. The entry point is the `proc`, `fun`, or `class` keyword, respectively, while the exit is the `end` keyword. The frame description consists of the single word `definition`.

Applications Procedure, function, object and method applications are step points. The entry and exit points are the opening and closing braces (or the comma in the case of a method application). The frame description mimics application syntax, displaying the procedure name (or `$` if it is anonymous) and the argument values.

Conditionals Boolean and pattern-matching conditionals are step points. The entry point is the `if` or `case` keyword (or `elseif`, `elsecase`, or `catch`, to be precise), the exit point the corresponding `end`. The frame description consists of the word `conditional`, followed by the value tested by the conditional, if it could be determined.

Thread Creation Thread creation using the `thread` ... `end` construct is a step point, with the obvious entry and exit points. Thread creation frames consist of the word `thread`.

Installation of Exception Handlers The `try ... end` construct defines step points (unless it has neither `catch` nor `finally`, in which case it is ignored), with a frame description of `exception handler`. The actual handling of exceptions is a step point by virtue of the fact that conditionals are step points.

Critical Sections The `lock ... end` constructs are step points, with the frame description containing both the word `lock` and the lock itself.

Loops The `for ... end` construct defines step points, with frame description `loop`.

5.2 Actions

Step Into The simplest way to control the execution of a thread is to single-step, i.e., to proceed from one step point to the next. This is exactly what Step Into ( ) does. This is the most fine-grained visualization of program execution.

Step Over In contrast, the Step Over action ( ), when performed at an entry point, causes thread execution to next stop at the corresponding exit point, not stopping at any nested step points. On an exit point, it behaves exactly as Step Into does.

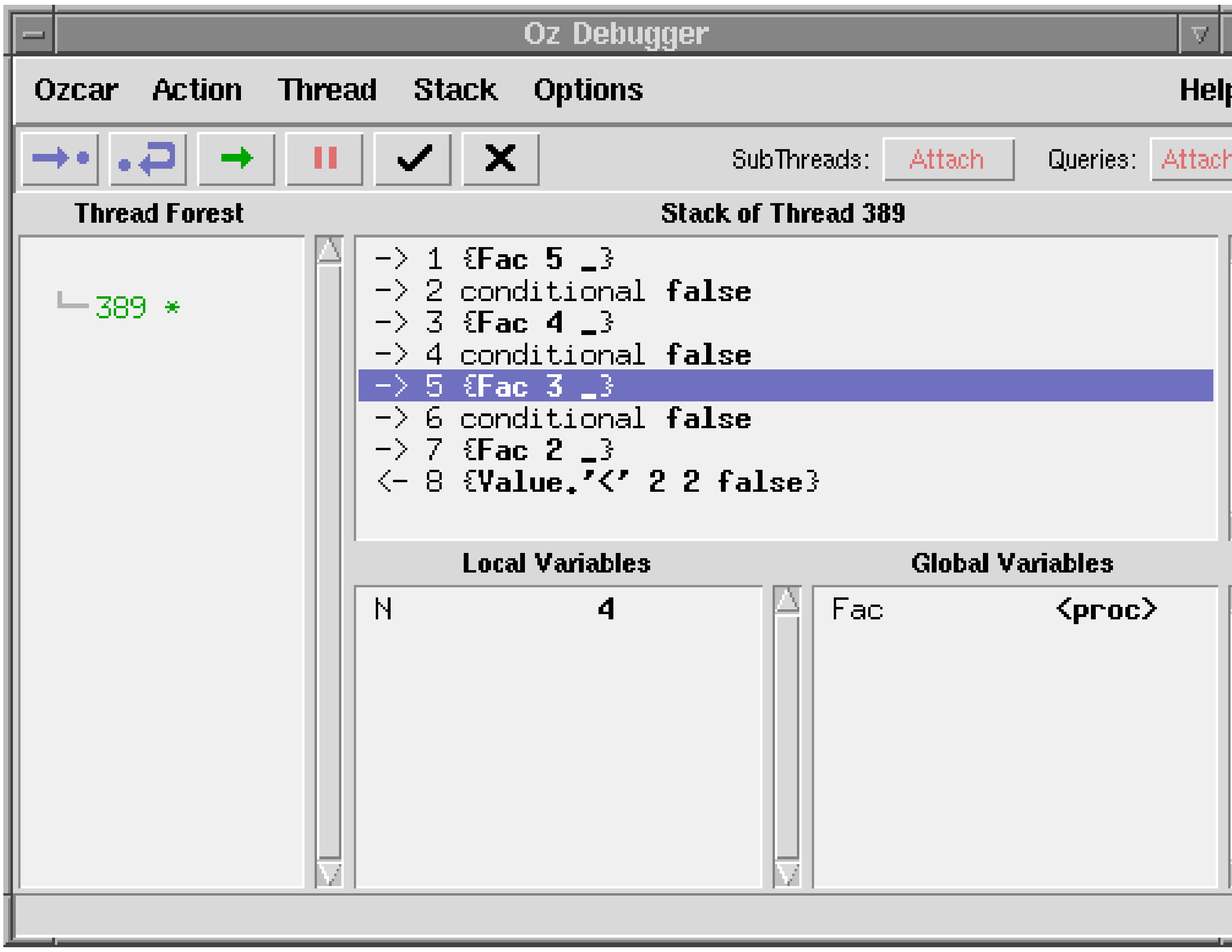
Action Unleash Actually, Step Over is a special case of a more generic action, called Unleash ( ). It causes execution of the thread to continue until the *selected* stack frame is about to be popped from the stack (or until the thread has finished executing the whole stack if no stack frame is selected); this makes Step Over identical to an Unleash with the topmost frame selected. Remember you can select a stack frame by clicking on it or by walking to it using the up and down cursor keys.

Example To illustrate the unleash action, we'll consider the following program which computes the factorial function:

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

Go and activate Step Into ( ) a couple of times, so as to build up a bit of stack, and select Frame 5. Your Ozcar should look similar to Figure 5.1. Let's decide to immediately compute the value of `{Fac 3}`, in other words, continue the thread's execution until Frame 5 is about to be removed from the stack. Unleash ( ) with Frame 5 selected does exactly this. Figure 5.2 displays the result of this action.

Figure 5.1 Before the Action 'Unleash 5'



```
└─389 *
```

```
-> 1 {Fac 5 _}
-> 2 conditional false
-> 3 {Fac 4 _}
-> 4 conditional false
-> 5 {Fac 3 _}
-> 6 conditional false
-> 7 {Fac 2 _}
<- 8 {Value.'<' 2 2 false}
```

```
N        4
```

```
Fac      <proc>
```

5.3 Breakpoints

Single-stepping is nice, but often somewhat inconvenient, because you may need a lot of steps until you reach the interesting section of your program. This is where breakpoints come in. Ozcar supports them in two flavours: static breakpoints and dynamic breakpoints.

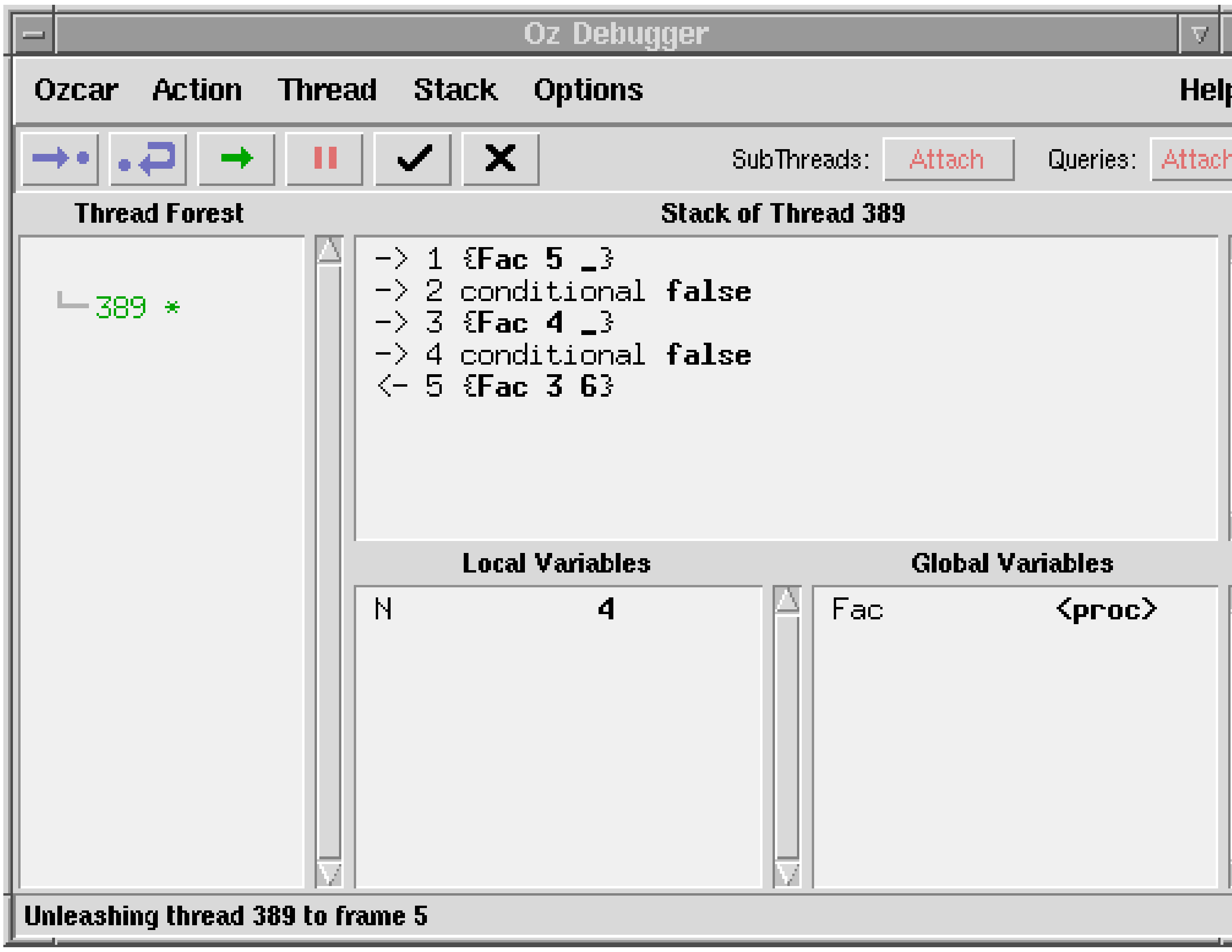
5.3.1 Static Breakpoints

A static breakpoint can be defined by editing source code to contain an application `{Ozcar.breakpoint}` and recompiling the program (which is why they're called *static*). This also means that static breakpoints will persist across multiple debugging sessions.

Example Let's take the factorial example again and assume we need to debug the base case of the recursion. We can do this with a static breakpoint as follows:

```
local
```

Figure 5.2 After the Action ‘Unleash 5’



```
-> 1 {Fac 5 _}
-> 2 conditional false
-> 3 {Fac 4 _}
-> 4 conditional false
<- 5 {Fac 3 6}
```

```
N        4
```

```
Fac      <proc>
```

Unleashing thread 389 to frame 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 performing the Unleash (➡) action twice, we arrive directly at the desired program point.

5.3.2 Dynamic Breakpoints

Sometimes we’ll want to insert a breakpoint in the course of a debugging session. Then we need dynamic breakpoints.

Setting Breakpoints Dynamic breakpoints can easily be set from within Emacs: Position the cursor on the line where you want to set or reset breakpoints and press `C-x space` to set or `C-u C-x space` to delete breakpoints. Note that this will affect *all entry points* on the corresponding line (there may be several, or none).

Currently, there is no way to list all currently defined dynamic breakpoints.

6

Environments

In this chapter we'll learn how to examine and use the environment, i.e., how to access the values of variables. In fact, we need to distinguish between different kinds of environments. First, there is the toplevel environment. This is the environment relative to which code was compiled (in the case of `ozc`, this normally is the Base Environment defined in *"The Oz Base Environment"*). The procedure `NewName` is one example, the module `String` is another. Then there exist environments for each procedure (and accordingly, for each frame of a thread's stack). For one there is a procedure's global environment, which contains variables referenced within the procedure, but defined outside it (i.e., global to the procedure); for the purpose of debugging, we consider this to not contain all toplevel variables since they are always visible. Finally, there come the procedure's local variables, which include its formal parameters.

6.1 Inspecting the Environment

Environment inspection is easy with Ozcar. In fact, it is done automatically for you. Whenever you select a stack frame, the variables visible at the program point corresponding to this frame are displayed in the Environment Views. You can further examine the values using the Inspector by clicking on them.

6.2 Using Values

Ozcar allows you to access the value you have last clicked to inspect it. It can be requested by the expression

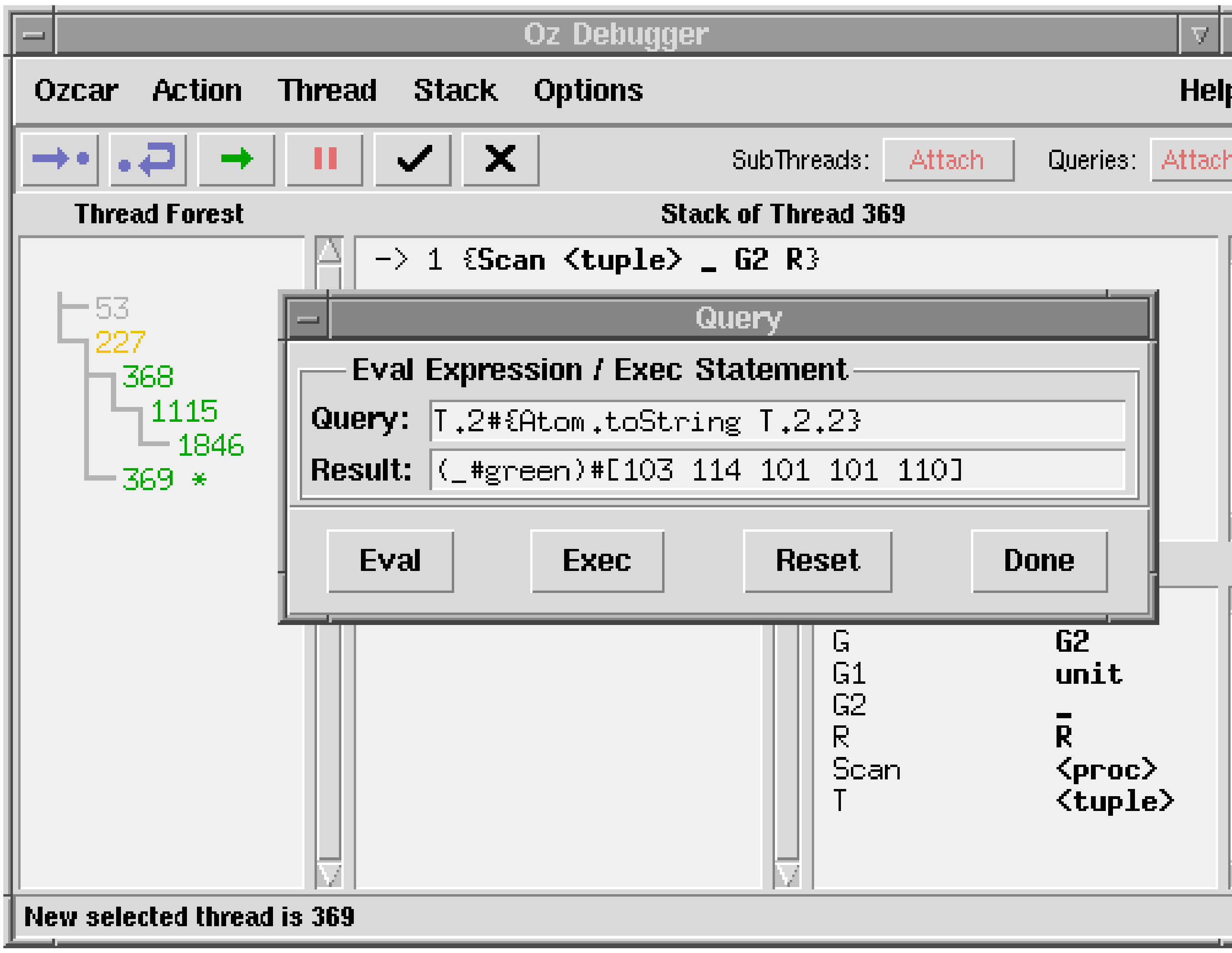
```
{Ozcar.object lastClickedValue($)}
```

6.3 Compiling against the Environment

There are situations where you want to operate on the values found in the local or global environment, for example, to convert a data structure, or to bind a logic variable which causes your program to hang. In these cases, you can use the Query dialog, opened by selecting the `Query` ... entry in the `Stack` menu.

Evaluating Expressions Using the Query dialog, you can evaluate arbitrary Oz expressions in the environment of the selected stack frame. Just enter the expression to evaluate on the `Query` line and press the `Eval` button to make the result appear on the `Result` line. Figure 6.1 shows an example.

Figure 6.1 Evaluating Arbitrary Expressions



```
-> 1 {Scan <tuple> _ G2 R}
```

Query: `T.2#{Atom.toString T.2.2}`
Result: `(_#green)#[103 114 101 101 110]`

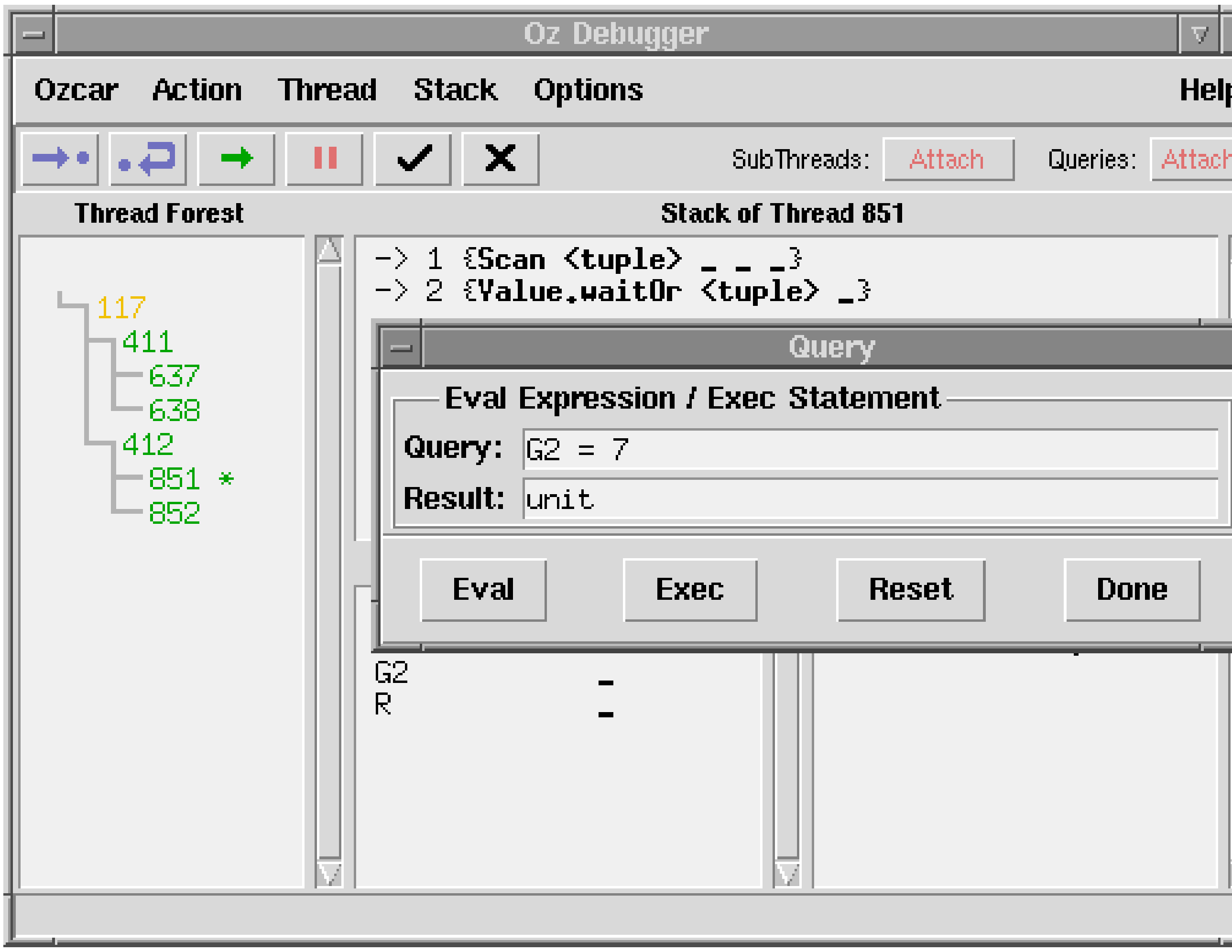
```
G       G2
G1      unit
G2      _
R       R
Scan    <proc>
T       <tuple>
```

New selected thread is 369

Executing Statements Besides evaluating expressions, the Query dialog allows to execute statements using the `Exec` button. The result of executing a statement will always be **`unit`**. Figure 6.2 shows an example scenario: An unbound variable, `G2`, is bound to the value `7`. (Note that the Environment Views are only updated when the stack frame is selected again.)

Compilation Environment The exact environment used for compiling the code in the `Query` line is the currently active toplevel environment of the OPI (or the full default OPI environment when running standalone). Over this the selected frame's global environment is adjoined, and again the local environment is adjoined. Since the exact toplevel environment relative to which the code currently being debugged has

Figure 6.2 Executing Statements


Oz Debugger
Ozcar Action Thread Stack Options Help
SubThreads: Attach Queries: Attach
Thread Forest
Stack of Thread 851
Query
Eval Expression / Exec Statement
Eval Exec Reset Done

```
117
  411
    637
    638
  412
    851 *
    852
```

```
-> 1 {Scan <tuple> _ _ _}
-> 2 {Value.waitOr <tuple> _}
```

Query: G2 = 7

Result: unit

```
G2          _
R           _
```

been compiled is not available, this may be an incorrect approximation. Note also that the environment is constructed using the frame selected at the time the `Eval` or `Exec` button is pressed, not when the Query dialog is opened.

Multiple Queries You can have as many Query dialogs as you like. This makes it possible to evaluate multiple pieces of code more than once without retyping them.

Unhandled Exceptions

Errors in programs often manifest themselves by unhandled exceptions. Ozcar supports post-mortem inspection of threads that died of a system or error exception. (User exceptions do not provide debugging information–work around this by raising your exceptions as error exceptions, using `Exception.raiseError`.)

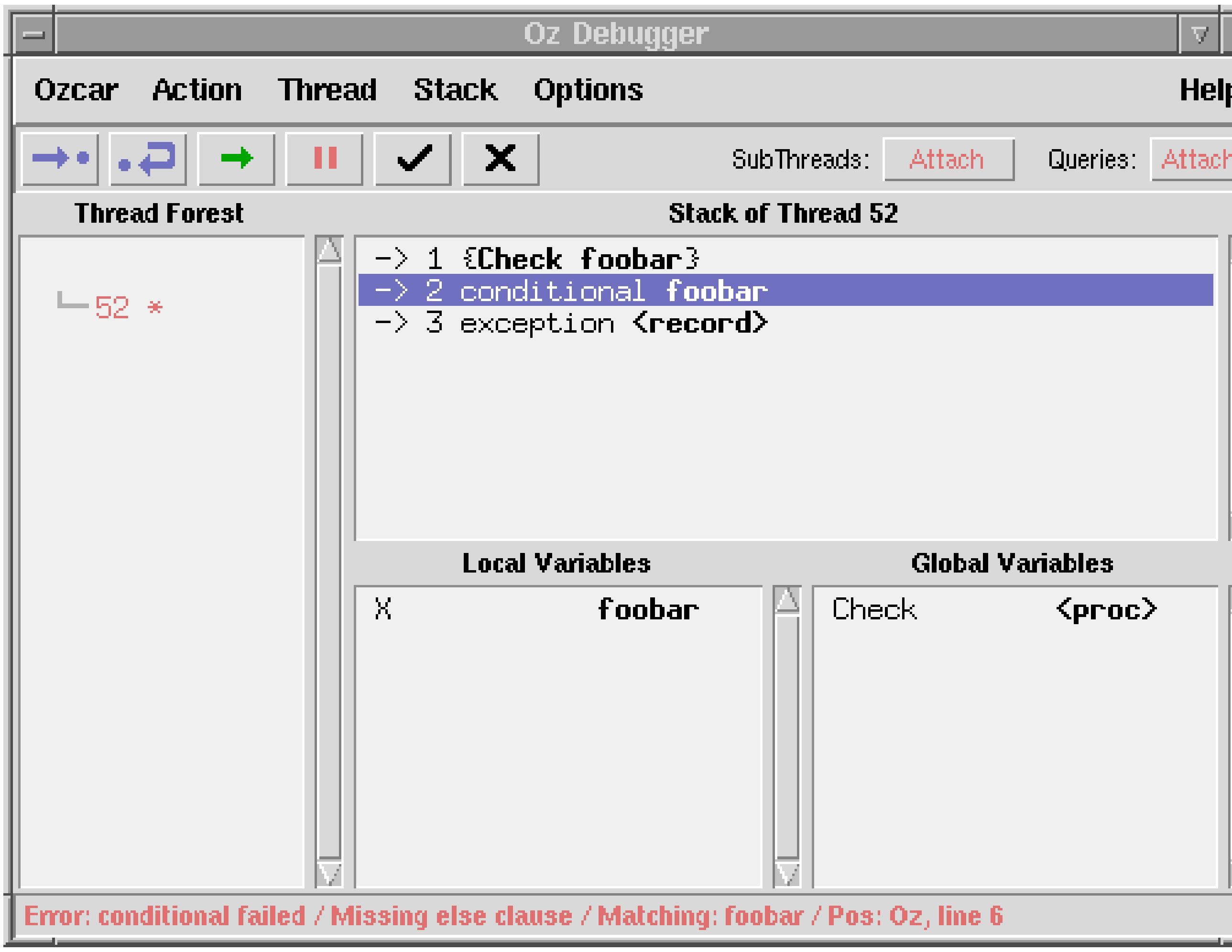
Exception Handling Any unhandled exception, which is normally just printed out to standard error, is caught by Ozcar. It attaches the dying thread and displays the stack from the time the exception was raised (or re-raised). The topmost frame gets the description `exception` followed by the exception value itself. Note that all stack frames include environment information. The description of the exception itself is provided in the Status Line.

Example Consider the following code. Its author forgot to handle some value in his patterns, leading to a missing `else` exception, as depicted in Figure 7.1:

```
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
```

Source View Emacs displays the location in the source code where the exception was raised. Note that the bar is colored red, as the position has been reached unexpectedly.

Figure 7.1 An Exception has not been Handled



```
└─52 *
```

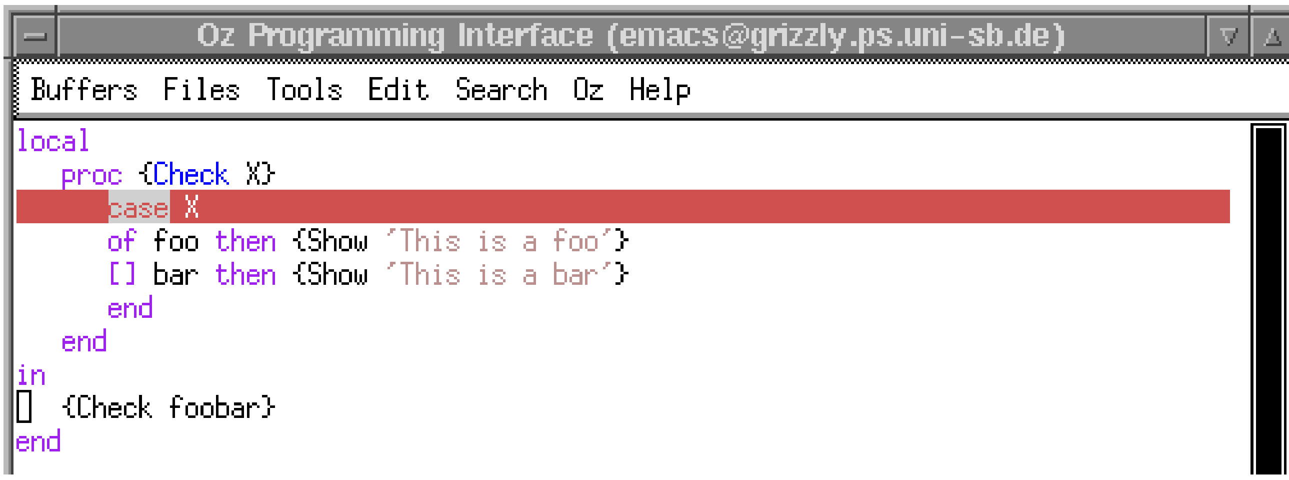
```
-> 1 {Check foobar}
-> 2 conditional foobar
-> 3 exception <record>
```

```
X            foobar
```

```
Check        <proc>
```

```
Error: conditional failed / Missing else clause / Matching: foobar / Pos: Oz, line 6
```

Figure 7.2 Emacs Showing the Error Position


Oz Programming Interface (emacs@grizzly.ps.uni-sb.de)
Buffers Files Tools Edit Search Oz Help

```
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
```

Remote Debugging

So far, we have only used Ozcar to debug threads running in the same process as the debugger itself. However, Ozcar also supports the same features for threads running remotely, and thus also the debugging of distributed applications.

Model In this case, debugging is a client/server application. The debugger itself (i.e., the site where the user interface is running) doubles as a server, accepting connections of clients (i.e., debugging sessions on remote sites). Clients forward events to the server, for example when threads stop at step points or become blocked. All attached threads, whether remote or local, are displayed in the same thread forest: Nodes corresponding to remote threads will be named by their site name and process ID in addition to their thread ID. All control operations are supported on remote threads: Commands are sent to the corresponding site so that the actual operations can be carried out locally.

Reflection Note that when inspecting remote stacks, any contained values will be reflected before they are displayed. This means that an approximating copy will be made on the remote site and the debugger will only be able to operate on the copy.

Chapter Structure This chapter will first take a look at how the server is started, then at how remote debugging sessions are initiated and how clients connect to the server.

8.1 Starting the Server

There are two ways to start a server: using the user interface or the API.

Using the Menus In the `Ozcar` menu, you'll find an entry called `Start Server`. Activating this will cause Ozcar to open a port where it listens for clients that want to connect. This port is identified by a ticket: a string describing the server's location on the net. Clients will need this ticket in order to connect. A dialog will be opened showing the ticket. You can use 'copy and paste' to create clients.

Using the API Another way to start the server is to use the API, for instance to start the server automatically from your application or to implement automatic connection of clients. The expression

```
{Ozcar.startServer}
```

will return the required ticket.

Multiple Activations Once Ozcar has started its server, the server will live as long as the Mozart process it runs in. Clicking `Start Server` or evaluating `{Ozcar.startServer}` again will yield the same ticket.

8.2 Connecting a Client to the Server

Once a server has been started, clients can connect to it. Again, there are two ways to start a client: using the `ozd` application or the API. Note that you should not have started Ozcar on the site where you want to start a server: Both debuggers will attempt to take over control of threads and will confuse each other.

From the Command Line The command line debugger can be used to start a client:

```
ozd --remotedebugger --ticket=ticket url -- args
```

This will use *ticket* to connect to the server. Otherwise, behaviour is similar to `ozd -debugger`, see Chapter *The Oz Debugger:* `ozd`, *(Oz Shell Utilities)* for more information.

Using the API The API provides the module `OzcarClient` at URL `x-oz://system/OzcarCl`
This can also be used to start a client:

```
{OzcarClient.start Ticket}
```

Fault Handling When a client dies, the server will still consider its threads attached. Only when you invoke an operation on a thread will it notice this fact and consider the corresponding thread dead.

9 Limitations

Ozcar currently suffers from a number of limitations. Most of these are not severe, since simple workarounds are available.

Non-debug code Often, it is not be practical or possible to recompile all code of an application (including all used libraries) with debug information. Ozcar will happily attach threads executing mixed debug and non-debug code, but only if they execute a step point within code compiled with debug information *or* a static breakpoint. Furthermore, single-stepping and environment information will not be available within non-debug code.

Tail-calls Long-running applications, such as server applications, depend on *tail-call optimization* to implement their toplevel loop. Tail-call optimization causes stack frames to be popped before the activation of another procedure if the corresponding call is the last thing to happen within a procedure body. Thus, tail-call optimization limits stack growth. Within debug code, tail-call optimization is disabled, and long-running programs as well as programs with deep recursion may abort due to excessive stack growth.

As a workaround, implement the top-level loop of server applications in a separate component compiled without debug information.

Pickling and Garbage Collection Due to maintenance of the runtime environments, values may be referenced from closures that would not be referenced in production code. This can lead to different dynamic behaviour with respect to pickling and garbage collection: Pickling may produce larger pickles or fail due to resources or stateful data structures. Garbage collection may not be able to release as much memory. Finalizers may be called later.

Thread IDs Thread IDs are 16-bit integers, and as such wrap around pretty quickly. As a consequence, they may often not be unique. This can cause the thread forest display and in particular navigation to become confused when debugging massively concurrent applications, because parent-child relationships cannot be correctly determined. Note however that actions performed on and events generated by threads will always be associated correctly.

Spaces Currently, Ozcar will not attach any thread that is not executing in the toplevel space. To debug a search script, execute it on the toplevel as a workaround.

Reflection of remote values When debugging remote threads (or distributed applications), the values displayed in stack frames and environments are only approximations of the actual values on the remote site obtained through reflection. Currently, the Query dialog executes queries locally and thus operates on the approximations instead of executing remotely and operating on the actual values.

A

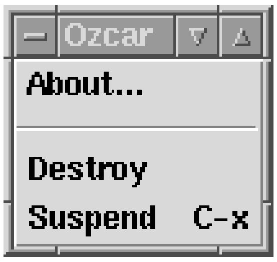
Menu Reference

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

A.1 The Main Menu

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

A.1.1 Ozcar



About. . .

Pops up an info box containing platform and bug reporting information.

Start Server

Starts a server locally (if not already started) to accept incoming connections for remote debugging. Furthermore, a dialog box is popped up displaying the ticket that remote sites need to connect to. Once a server has been started, this operation will always return the same ticket.

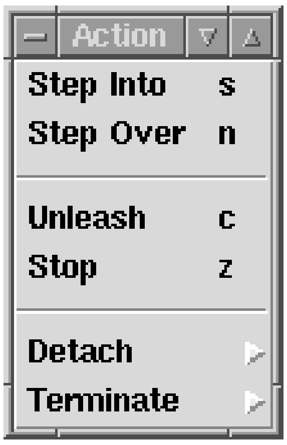
Destroy

Closes the internal Ozcar object. Use this menu entry only if Ozcar hangs, as all information about currently attached threads gets lost. Threads are not killed explicitly, but unless some other reference to them is retained, they are likely to be garbage collected if they had been stopped.

Suspend **`C-x`**

By default, closes the main window, sets the virtual machine to non-debug mode, and causes the compiler to generate non-debug code. Information about currently attached threads is preserved. Can be reconfigured using the API by the `closeAction` parameter.

A.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, not stopping on nested constructs which constitute a step point. Identical to pressing the button.

Unleash **c**

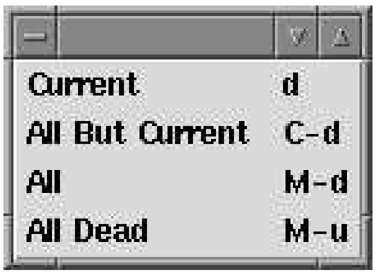
Continue execution until the selected stack frame is about to be popped from the stack, or until the stack is empty if no frame is selected. Identical to pressing the button.

Stop **z**

Stop the selected thread at the next step point it reaches, or immediately if it is blocked. Identical to pressing the button.

A.1.2.1 Detach

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



Current **d**

Detach the selected thread. Identical to pressing the button.

All But Current	**`C-d`**

Leave the current thread alone in the thread forest. Detach all others, letting them run freely (unless dead).

All	**`M-d`**

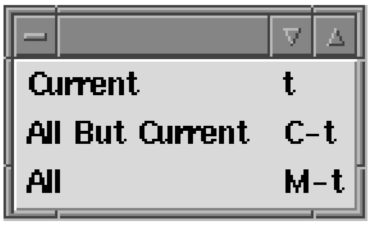
Detach all threads, letting them run freely (unless dead).

All Dead	**`M-u`**

Detach all dead threads.

A.1.2.2 Terminate

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.

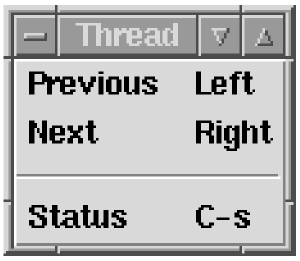
All But Current	**`C-t`**

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

All	**`M-t`**

Detach and kill all threads.

A.1.3 Thread



Previous	**`Left`**

Select the thread which is located above the selected thread in the thread forest. If the selected thread is the topmost thread in the forest, the bottommost thread is selected.

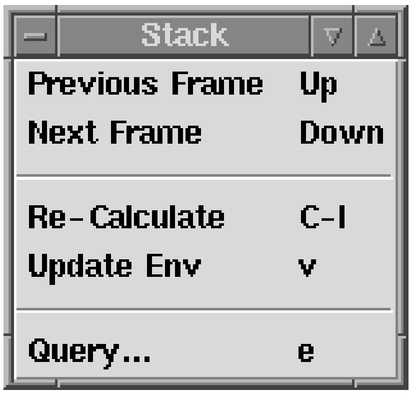
Next **`Right`**

Select the thread which is located below the selected thread in the thread forest. If the selected thread is the bottommost thread in the forest, the topmost thread is selected.

Status **`C-s`**

Print some useful information about the currently attached threads, for example: `2 attached threa`
The two numbers associated with the selected thread are the thread ID and the parent's thread ID. In parentheses, the thread's state is given.

A.1.4 Stack



Previous Frame **`Up`**

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

Next Frame **`Down`**

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

Recalculate **`C-l`**

Update the Stack View. You may see internal stack frames appear, as well updated values in the Environment Views. (There is *no* automatic update to display a value as soon as it changes.)

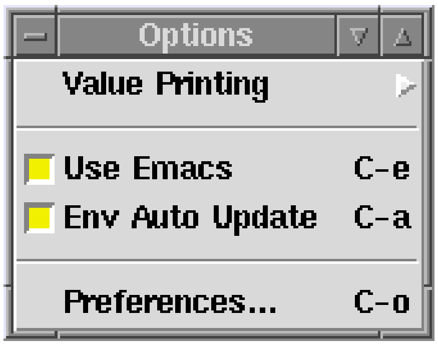
Update Env **`v`**

Immediately recalculates and redisplays the environment of the selected stack frame.

Query. . . **`e`**

Opens a dialog box where statements can be executed or expressions evaluated in the context of the currently selected stack frame's environment. If the computation does not terminate, use the Reset button to cancel the operation and kill the computation.

A.1.5 Options



Use Emacs **C-e**

If turned off Emacs is not used as a Source View, so you don't get any source position information.

Env Auto Update **C-a**

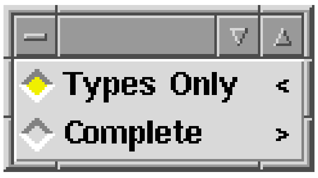
If you debug threads with huge environments, it may be a good idea to turn of the automatic update of the environment (recalculation and redisplay every time you make a step) in order to save time and memory. When the variables display is not up to date, it will be displayed in gray. You can always request the current environment by pressing `v` (see above, item `Update Env`).

Preferences. . . **C-o**

Opens a dialog box to adjust preferences. For instance, many events are handled in batch mode if they arrive in quick succession. The `Idle Time To Perform Action` preferences define what a 'quick succession' is. You can also adjust the width and depth to which values are displayed in the Stack and Environment Views (if `Value Printing` is set to `Complete`). Furthermore, the display of 'system variables' can be suppressed in the Environment Views (system variables are variables starting with a backquote–they are used by the compiler for built-in runtime operations). Application of the primitive procedures `.` (feature selection) and `NewName` can be ignored for purposes of stepping. Last but not least some logging of the workings of Ozcar can be printed to standard output–should you ever want to report a bug in the debugger itself, you should try to include this information in your report.

A.1.5.1 Value Printing

In this menu you can determine how to display values in the Stack and Environment Views. There is the short form which only prints type information for most values, and a long form which always prints the actual value (up to a given print width and 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. You should carefully adjust the print depth and print width to your needs.

A.2 SubThreads

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

Ignore

Children will not be attached at all, but run freely.

Attach

Children will be attached and immediately stopped. This is the default.

Unleash 0

Children will be attached and the command `Unleash 0` be executed. This gives you a good feeling about how threads are created in a concurrent application. They will terminate normally (unless they block somewhere), but will still be included in the thread view.

Unleash 1

Children will be attached and the command `Unleash 1` be executed. This makes it possible to explore the environment of every child thread just before it terminates.

A.3 Queries

This menu is located to the right of the button bar, next to the `SubThreads` menu. You can select how threads created by the OPI are treated by Ozcar.

Ignore

OPI threads are not attached, neither are any of their subthreads (unless, of course, they run into a breakpoint).

Attach

OPI threads are attached and immediately stopped. This is the default.

B

Application Programming Interface

B.1 The `Ozcar` Module

The `Ozcar` module, available at `x-oz://system/Ozcar`, exports the following features.

`object`

    {Ozcar.object +Message}

provides the interface to the running debugger itself. It is described below.

`open`

    {Ozcar.open}

is identical to `{Ozcar.object on}`.

`close`

    {Ozcar.close}

is identical to `{Ozcar.object off}`.

`breakpoint`

    {Ozcar.breakpoint}

sets a static breakpoint (see Section 5.3.1). Execution of `{Ozcar.breakpoint}` causes the current thread to be stopped and attached to the debugger if the debugger has been started, and is ignored otherwise.

`startServer`

    {Ozcar.startServer ?Ticket}

starts a server to accept connections of remote debugging clients (see Chapter 8) and returns a ticket with which clients can connect. Once the server has been started, the expression `{Ozcar.startServer}` will always return the same ticket.

B.2 `Ozcar.object`

`init`

    {Ozcar.object init()}

closes and recreates the internal Ozcar object, discarding all information about attached threads. Identical to the `Destoy` menu item described in Section A.1.1.

on

```
{Ozcar.object on()}
```

starts Ozcar, opening the Ozcar window, setting the virtual machine to debug mode, and configuring the OPI compiler to generate debug code.

off

```
{Ozcar.object off()}
```

by default, suspends Ozcar, closing its window, but keeping all information about attached threads. Restarting Ozcar (using `on()`) will resume a debugging session exactly where it has been left before. Can be reconfigured by `closeAction` (see below).

conf

```
{Ozcar.object conf(FeatureA: +ValueX ...)}
```

configures Ozcar parameters. The method must be a record where each feature `FeatureA` is a configuration parameter and the corresponding subtree `ValueX` the value to set the parameter to. See Section B.3 for a list of parameters.

bpAt

```
{Ozcar.object bpAt(+FileV +LineI +EnableB)}
```

sets or resets dynamic breakpoints. All step points in files with name `FileV` and in line `LineI` are affected. The flag `EnableB` specifies whether dynamic breakpoints at these step points should be enabled (**`true`**) or disabled (**`false`**). See Section 5.3.2 for more information on dynamic breakpoints.

lastClickedValue

```
{Ozcar.object lastClickedValue(X)}
```

returns the value that has last been clicked on in the user interface, causing it to be passed to the Inspector.

B.3 Configuration Options

Feature	Type	Default	Description
`timeoutToSwitch`	integer	100	Timeout in milliseconds before a
`timeoutToUpdateEnv`	integer	200	Timeout in milliseconds before t
`emacsInterface`	**`false`** or an object	The OPI	The object to talk to for source d
`closeAction`	**`unit`** or a nullary procedure	**`unit`**	Configures how the `off()` metho

B.4 The `OzcarClient` Module

start

```
{OzcarClient.start +Ticket}
```

starts a local debug session, connecting to a remote debugger using `Ticket`. This sets the virtual machine to debug mode (if and only if the remote debugger is active) and sets up threads to forward debug events to the remote debugger and to locally execute control operations generated by the remote debugger. This should not be used with a local Ozcar runnning. See Chapter 8 for more information on remote debugging.

Index