Cleaning up after yourself

M. Anton Ertl*
TU Wien

Abstract

Performing cleanup actions such as restoring a variable or closing a file used to be impossible to guarantee in Forth before Forth-94 gave us `catch`. Even with `catch`, the cleanup code can be skipped due to user interrupts if you are unlucky. We introduce a construct that guarantees that the cleanup code is always completed. We also discuss a cheaper implementation approach for cleanup code than using a full exception frame.

1 Introduction

A frequent programming problem is to restore some state, free a resource, or perform some other cleanup reliably. Typical examples are:

- Restore `base` after a temporary change.
- Close a file.

In Forth-94 we can use `catch` to ensure that such cleanup actions happen under most (but not all) circumstances.

In this paper we explore ways to improve on this state of affairs in the following ways:

- Provide a more reliable mechanism that works even in the presence of asynchronous exceptions (e.g., user interrupts).
- Avoid the cost of a full-blown exception frame where possible.

2 Running Example

As a running example, we will use a word `hex.` that prints a number in hex base without changing `base`. And that word will be used in the following context:

```
: foo
  ... hex. ...
  ... .     ... ;

decimal foo
hex foo
```

Note that, in addition to printing a number in hex, `foo` also prints a number in the current base.

3 Standard Forth solutions

3.1 Thinking Forth approach

In the old days, Forth did not have `catch`, so one would write, e.g.:

```
: hex. ( u -- )
  base @ >r
  hex u.
  r> base ! ;
```

But even in the old days Forth had non-local exits via `abort`, and `quit`, as well as user interrupts. If any of these non-local exits from `u.` happened, `base` would not be restored.

So, in the old days, cleanup could not be performed reliably. So, various ways to work around this situation were developed and practised, as discussed in the Thinking Forth, Chapter 7, Section "Saving and Restoring a State" [Bro84]; in particular, this section cites Charles Moore as follows:

> You really get tied up in a knot. You're creating problems for yourself. If I want a hex dump I say `HEX DUMP`. If I want a decimal dump I say `DECIMAL DUMP`. I don't give `DUMP` the privilege of messing around with my environment.
>
> There's a philosophical choice between restoring a situation when you finish and establishing the situation when you start. For a long time I felt you should restore the situation when you're finished. And I would try to do that consistently everywhere. But it's hard to define "everywhere." So now I tend to establish the state before I start.
>
> If I have a word which cares where things are, it had better set them. If somebody else changes them, they don't have to worry about resetting them.
>
> There are more exits than there are entrances.

*Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; `anton@mips.complang.tuwien.ac.at`

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in: EuroForth 2008 Conference proceedings, p. 35-38

Unfortunately, this workaround is not even usable in our running example: What is the situation that should be established before the call to `.` in `foo`?

And this workaround does not help at all with other cleanup tasks like closing files and freeing other resources.

3.2 Using `catch`

Fortunately, with the introduction of `catch` in Forth-94, the situation changes: There is only one exit from `catch`, and we can use that property to make the cleanup more reliable:

```
: hex.-helper ( u -- )
  hex u. ;

: hex. ( u -- )
  base @ >r
  ['] hex.-helper catch
  r> base ! throw ;
```

This makes sure that `base` will be restored even if an exception (of any kind) happens while `hex.-helper` is executed.

Unfortunately, there is still one chink in our cleanup armour: If an exception (e.g., a user interrupt) happens during the restoration of `base`, the cleanup code would not complete, and `base` would be left in the wrong state.

4 Advanced solutions

4.1 `Try...restore...endtry`

The current development version of Gforth offers a construct `try` *`code1`* `restore` *`code2`* `endtry`. If any exception happens anywhere between `try` and `endtry` (including in *`code2`*), the stack depths are reset to the depth at `try`, the throw value is pushed on the data stack, and execution jumps right behind the `restore`.

With this construct there is not just only one exit, it also guarantees that *`code2`* is executed from start to end.

This construct can be used to solve our problem as follows:

```
: hex. ( u -- )
  base @ { oldbase }
  try
    hex .
    0      \ value for throw
  restore
    oldbase base !
  endtry
  throw ;
```

The old base is stored in a local, because we cannot use the return stack for this purpose (`try` pushes an exception frame on the return stack). However, Instead of using a local, we could use the data stack, as follows:

```
: hex. ( u -- )
  base @
  try
    over hex .
    0      \ value for throw
  restore
    over base !
  endtry
  throw
  2drop ;
```

Note how we use `over` twice to keep the values on the data stack intact, so we can use them in the restoration code. We only drop these values after `endtry`.

This construct requires some care in usage:

- As shown above, one has to be careful not to remove items from the stacks that are needed in the restoration; one must not even remove them during restoration.
- The restoration code must not throw an exception, at least not every time it executes. Otherwise the system will go into an infinite loop of start-restoration...throw-exception. And not even a user interrupt can be used to break out of that loop. Instead, the user then has to stop the system by using some more brutal methods (e.g., in Unix by sending a `SIGTERM` to the Gforth process).
- The restoration code must be idempotent, i.e., executing it multiple times (starting at the same stack depths) should have the same effect as executing it once.

However, Forth programmers are used to taking responsibility for their programs, so these caveats should not be a problem.

The idempotence requirement may be hard or impossible to satisfy in some cases, e.g., when the cleanup involves `close-file` or `free`. In such cases it is usually preferable to have a small chance of not cleaning up than to try to clean up several times. One can achieve this by writing the non-idempotent part between the `endtry` and the `throw`.

In cases where a variable is changed and restored, the idempotence requirement is easy to achieve.

An example of a non-idempotent use is:

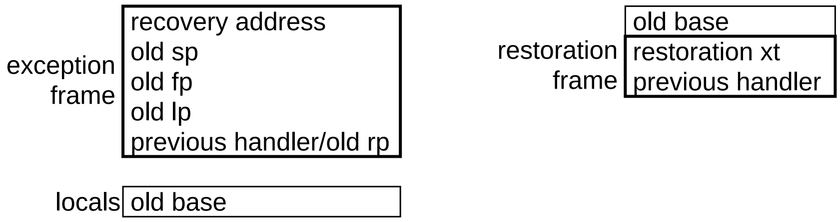

Figure 1: A general-purpose exception frame used for restoring `base` compared to a restoration frame

```
... open-file throw { f }
try
  ... f read-file throw ...
0 restore
endtry
f close-file throw
throw
```

4.2 Special-purpose words

Gforth also has special-purpose words for a few frequent or dangerous purposes:

`base-execute ( i*x xt u -- j*x )` executes *xt* while `base` is set to *u*.

`infile-execute ( i*x xt file-id -- j*x )` executes *xt* while `key` etc. read their input from *file-id*.

`outfile-execute ( i*x xt file-id -- j*x )` executes *xt* while the output of `type` etc. is redirected to *file-id*.

Given that all these words take an xt from the stack, and the xt is nearly always a constant, it is probably better to define future words of this kind such that they take the xt from the top-of-stack.

5 Efficiency

An exception frame costs five return stack cells in Gforth (and probably a similar amount in other systems), and constructing and consuming it costs a bit of time. For the purpose of cleanup a full exception frame is overkill. We don't really need to restore the depths of all stacks in this case: If we enter the restoration in the normal way, the stack depths are not restored anyway; and if we enter the restoration code through a `throw`, we are going to throw the error further on, so we don't need the stack depths, either; we just need access to the restoration data.

So we could implement a lighter-weight mechanism for restoration. Two cells for the restoration frame itself would be enough (see Fig. 1). The restoration frames would be kept on the return stack and chained in a linked list. `Throw` would process all the restoration frames that are above the next exception frame on the return stack, then process that exception frame as usual.

The downside is that the code for the restoration, and for setting up the restoration data would have to be even more aware of the restoration mechanism, because the restoration data cannot directly be transferred through a stack, but has to be accessed through the restoration frame. E.g., the restoration word for restoring `base` might look as follows:

```
: restore-base ( addr -- )
  dup @ base !
  next-restoration ;
```

Here, *`addr`* is the address of the user part of the restoration frame. `Next-restoration ( addr -- )` removes the current restoration frame from the chain. Any non-idempotent cleanup code would happen after `next-restoration`.

An implementation of `base-execute` with such a mechanism might look as follows:

```
: base-execute ( i*x xt u -- j*x )
  base @ >r
  ['] restore-base >restore
    base ! execute
  restore>
  r> drop ;
```

Here `>restore` would push a restoration frame on the return stack and add it to the restoration chain. `Restore>` would execute the restoration xt (i.e., `restore-base`) and drop it from the return stack. The old base would have to be dropped explicitly.

This mechanism has not been implemented. While it would be relatively easy to implement, it is unclear if it is worth the documentation and support load to provide it as a feature to the users. Here are a number of points to consider:

- In my experience nearly all uses of `catch` are for restoration/cleanup. So most exception frames could be replaced by lighter-weight restoration frames.

- Exception frames and their handling have not shown up as performance bottlenecks, but then I have not performed any measurements.

6 Related work

I am not aware of other advanced solutions in Forth. However, this is a common programming problem, so other languages have developed a wide variety of approaches for solving it.

6.1 Dynamic Scoping

Some of the problems addressed in this paper, e.g., our `base`-based running examples can be seen as customizing the execution environment. Hanson and Proebsting [HP01] argue that dynamically-scoped variables have the right properties for this usage and that programmers in languages without dynamic scoping resort to simulating dynamic scoping, and they point out the similiarity between exceptions (a dynamically scoped control structure) and dynamically scoped variables (which explains why we and others use exception-catching to implement them).

A significant number of programming languages and systems provide dynamically-scoped variables. Lisp is a well-known example. But a probably more widely-used example is environment variables in Unix and Windows processes.

Another language with dynamically-scoped variables is Postscript; there programmers perform dynamic scoping by (in Forth terminology) constructing wordlists dynamically, and pushing them on the search order stack; because name lookup in Postscript happens at run-time, this results in dynamic scoping. However, the Postscript dynamic control-flow words (`exit`, `stop`) do not affect the depth of the control-flow stack, so these features cannot be combined safely.

6.2 Cleanup

Lisp has the `unwind-protect` special form[1]: `(unwind-protect` *`protected cleanup`*`)` makes sure that *`cleanup`* is executed in any case, even if there is an abnormal exit from *`protected`*. However, unlike `try...restore...endtry`, it does not protect against abnormal exits from *`cleanup`*.

Java has a similar feature in the form of the `try ... finally` construct, and C++ in `try ... catch`.

C++ also provides destructors that can be used to automatically release resources and perform other cleanup when the scope of a variable is exited. Stroustrup[Str01] gives a good overview of what kind of exception safety are desirable, and how the various features of C++ may be used to achieve them.

In a similar vein, Java finalizers perform cleanup actions when an object is garbage-collected. However, because the finalizer may be executed a long time after a destructor would have been executed, it is often recommended to favor other approaches over using finalizers.

Many other languages have similar features.

7 Conclusion

The introduction of `catch` in Forth-94 provided a good basis for writing code that cleans up after itself rather than requiring every piece of code to clean up all the trash that all other code may have left behind.

However, in the presence of user interrupts and other asynchronous exceptions this is not sufficient. We propose the `try ... restore ... endtry` construct that can be used to solve this problem completely for some, but not all uses. We also discuss a more light-weight implementation technique.

References

[Bro84] Leo Brodie. *Thinking Forth*. Fig Leaf Press (Forth Interest Group), 100 Dolores St, Suite 183, Carmel, CA 93923, USA, 1984.

[HP01] David. R. Hanson and Todd A. Proebsting. Dynamic variables. In *SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 264–273, 2001.

[Str01] Bjarne Stroustrup. Exception safety: concepts and techniques. In *Advances in exception handling techniques*, pages 60–76. Springer LNCS 2022, 2001.

[1] `http://www.lispworks.com/documentation/HyperSpec/Body/s_unwind.htm`