The bright side of exceptions

The Lisp Condition System

Didier Verna

didier@lrde.epita.fr
http://www.lrde.epita.fr/~didier
http://www.facebook.com/didierverna
@didierverna

ACCU 2013 – Saturday, April 13th

What is an exception?

The Java™ Tutorials (`docs.oracle.com`)

Definition: An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

When an error occurs [...]

If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler[...], the runtime system[...] terminates.

- Unfortunately, "exception" really means "error".

Benefits of exception-based error handing

1. Separation of concerns

```
open (file);
if (opened)
{
  parse (stream);
  if (parsed)
  {
    interpret (contents);
    if (interpreted)
      act ();
    else
      return interpretation_error;
  }
  else
    return parse_error;
}
else
  return open_error;
```

```
try
{
  open (file);
  parse (stream);
  interpret (contents);
}
catch
{
  open_error: handle_it ();
  parse_error: handle_it ();
  interpretation_error: handle_it ();
}
```

2. Up-stack propagation
3. (OO-like) Hierarchies
4. User-defined exceptions

Drawbacks of the usual try/catch/throw model

1. Mandatory stack unwinding
   Execution context is lost
2. 2D Separation of Concerns
   - Throwing an exception
   - Handling it

   Hardwired handler selection

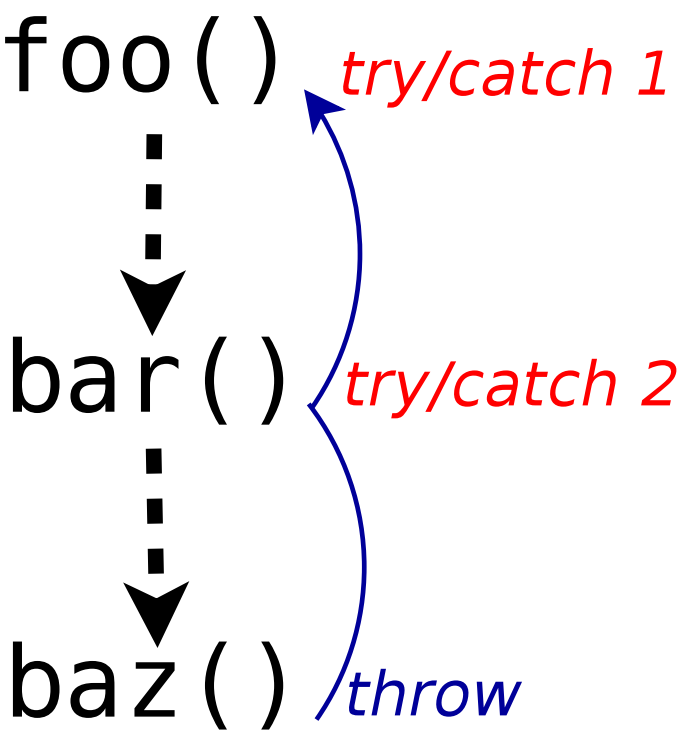


Table of contents

1. Basic Error Handling
   - Catching and throwing errors
   - Defining errors
2. Advanced Error Handling
   - Restarts
   - Dynamic error management
3. Beyond Error Handling
   - Example 1: warnings
   - Example 2: general signals

Basic error handling

Equivalent to try/catch

```
(handler-case form          ;; try
  (error1 (var1) form1)     ;; catch
  (error2 (var2) form2)
  ...)
```

Equivalent to finally

```
(unwind-protect form
  cleanup                   ;; finally
  ...)
```

- Non-local transfer to an exit point
  - `tagbody/go`
  - `catch/throw`

User-defined errors

User-level definitions

```
(define-condition name (supers...)
  ((slot-name options...)
   (slot-name options...)
   ...)
  options...)
```

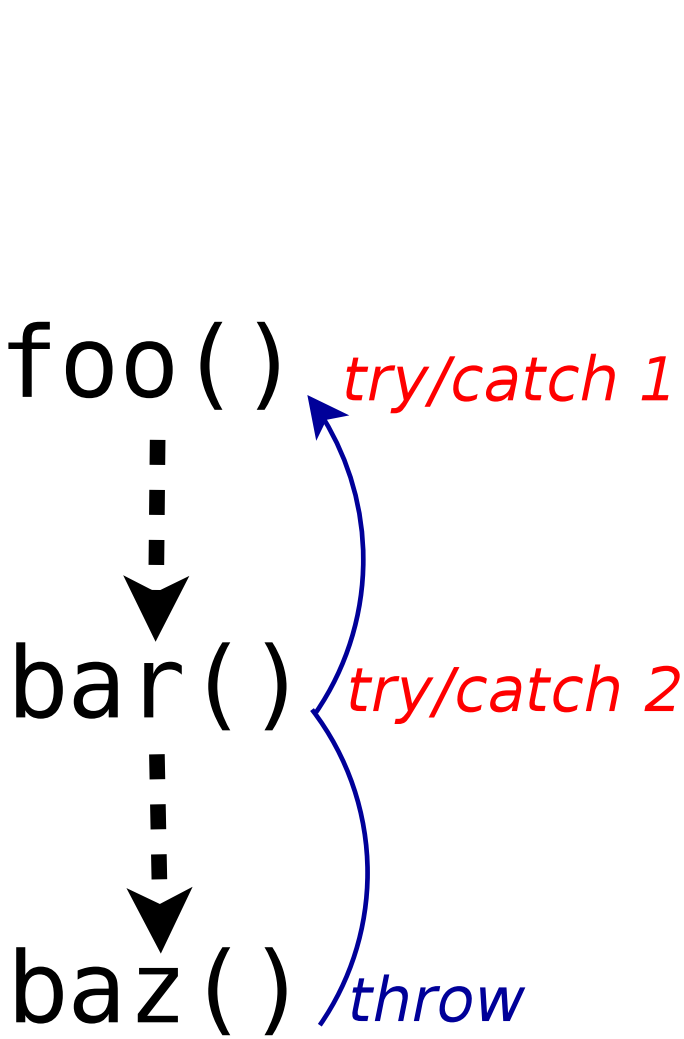
User-level throwing

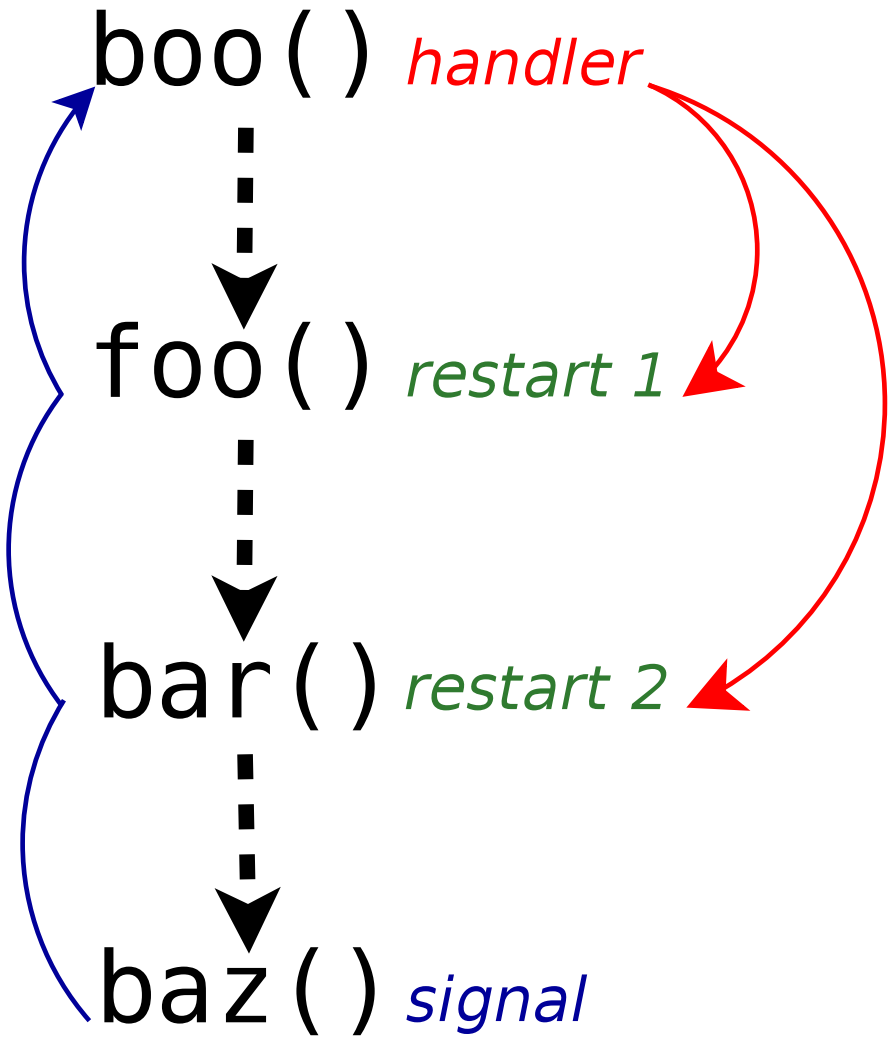
```
(error datum arguments...)
```

- **Jargon:**
  - We don't "throw an error"
  - We "signal a condition"

3D Separation of Concerns

No mandatory stack unwinding





User-defined restarts

Restart definition

```
(restart-case form
  (restart1 (args...) options... body)
  (restart2 (args...) options... body)
  ...)
```

Restarts management

Restart invocation

```
(invoke-restart restart args...)
```

Stack-preserving handlers

```
(handler-bind
    ((error1 handler-function) ;; you may call
     (error2 handler-function) ;; invoke-restart
     ...)                      ;; here...
  body)
```

Dynamic error management

Restart definition

```
(restart-case form
  (restart (args...) options... body)
  ...)
```

Conditional availability

```
;; The :test option to restarts
(find-restart restart) ;; Is restart available?
(compute-restarts) ;; Get all available restarts
```

Interactive calling

```
;; The :interactive option to restarts
(invoke-restart-interactively restart)
```

The truth about `error`

- **The function `error`:**
  1. signals a *condition*
  2. looks for an appropriate handler
     - may handle the error (non-local exit)
     - may not (decline by returning)
  3. eventually invokes the debugger
- **The function `cerror`:**
  - "continuable error"
  - invokes the debugger but...
  - ...establishes a restart named `continue`

- What about *not* invoking the debugger *at all* ?

Example 1: Warnings

- **The function `warn`:**
  1. signals a *condition*
  2. looks for an appropriate handler
     - may handle the error (non-local exit)
     - may not (decline by returning)
  3. eventually *prints* the condition and *returns*
- **Also establishes a `muffle-warning` restart**

The Truth about `warn`

The true truth about `error`

- **The function `warn`:**
  1. establishes a `muffle-warning` restart
  2. calls `signal` to signal the condition
  3. eventually *prints* the condition and *returns*
- **The function `[c]error`:**
  1. [establishes a `continue` restart]
  2. calls `signal` to signal the condition
  3. eventually invokes the debugger
- **The function `signal`:**
  1. looks for an appropriate handler
     - may handle the error (non-local exit)
     - may not (decline by returning)
  2. simply returns otherwise

▶ User-defined protocols on top of `signal`

Example 2: coroutines (sort of) and `yield`

Generators / iterators, really

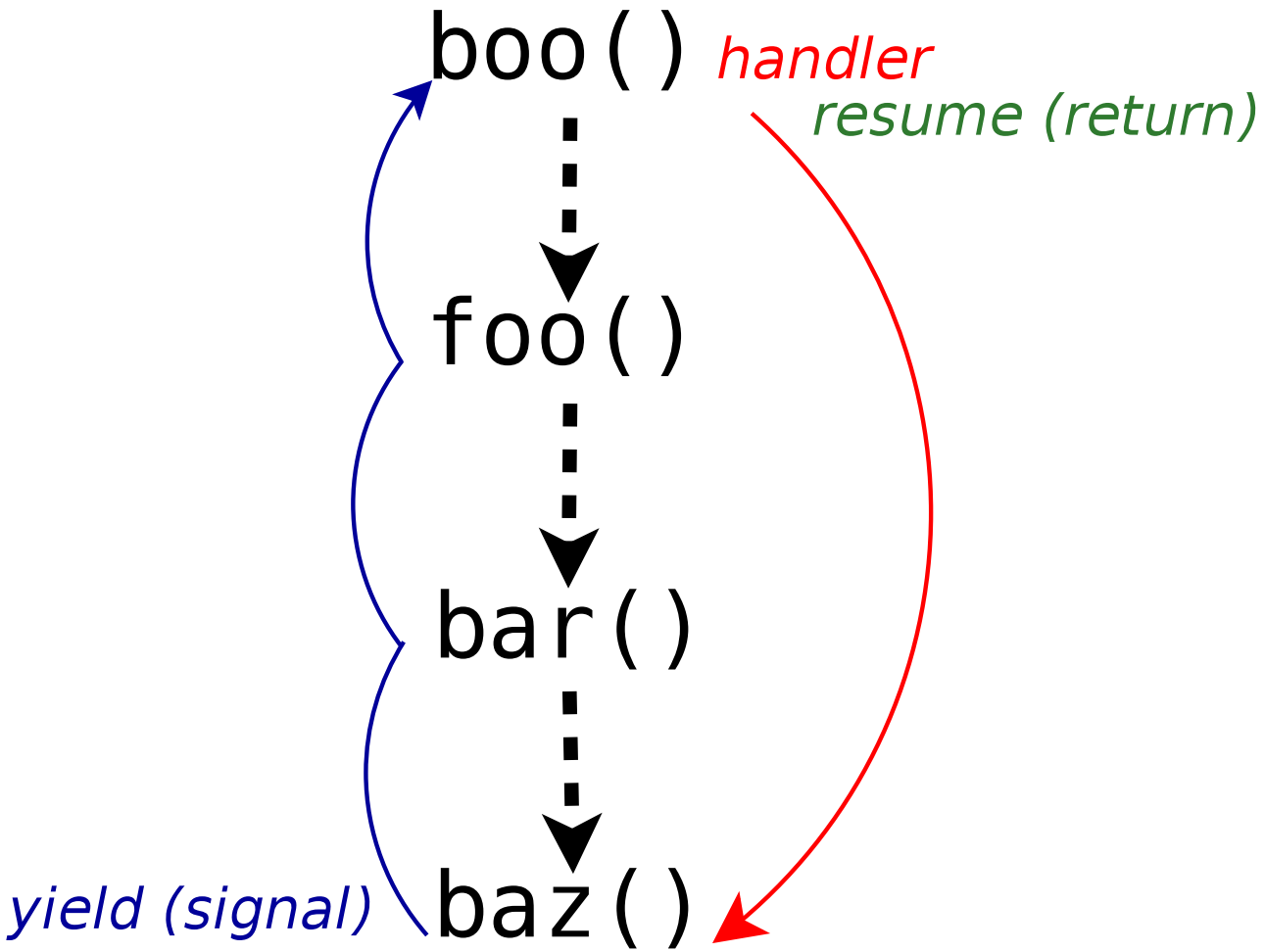
- **Coroutines:**
  - "yield" values
  - retain their
    - state
    - position

Generator example

```
(defun squares ()
  (loop :for i :from 0
        :do (yield (* i i))))
```

- Retain state and position $\Longleftrightarrow$ *don't unwind the stack*
- yield values $\Longleftrightarrow$ *signal them*
- **The trick:**
  1. yield values by signalling a condition
  2. use them in a condition handler (by side-effect)
  3. let the handler return normally (pretend it declined) !!

The big picture
Worth a thousand words



Conclusion

There is more to exceptions than meets the eye...

- Traditional approach is limited
  - Stack unwinding
  - 2D separation of concerns
- Improvements
  - No (mandatory) stack unwinding
  - 3D separation of concerns

- Better error management
- No restricted to errors
- Not even restricted to exceptions
- User-defined condition-based protocols