In solidarity with the people of Ferguson
and their protests against police violence

How to Make Error Handling Less Error-Prone

or...

BACK TO THE FUTURE

Rescuing error handling ideas from 1975!

(the year I was born :-)

How to Make Error Handling Less Error-Prone

- Introduction
- Sequential error handling
    - Error status; Return codes; Error values; Trapping; ...
    - Exceptions (history, restarts, ...)
- Concurrent error handling
    - Supervision hierarchies
    - Fate-sharing
- Rollback error recovery
- Motivation; conflicts between encapsulation and having enough information to recover a consistent state;
- how automatic rollback solves the encapsulation problem;
- how this form of rollback differs from database-style transactions or transactional memory;
- optimisations
- the Ken protocol;
-

Introduction

- “Error” comes from a Latin verb meaning “to stray”.

  For our purposes, *errors* are cases where the execution of a program strays from its usual path.
- We'll be considering errors in relation to high-level language programming and distributed systems.
  - Errors at other layers –operating systems, hardware, etc.– are important but not what this talk is focussed on.
- This includes:
  - cases that the language specification defines to be errors, or where the language implementation is unable to meet its defined specification;
  - cases that the program defines to be errors, or where the program is unable to meet its defined specification.
- *Error handling* is the response to errors.
- An *error handling mechanism* is a feature of a language designed to enable responding to errors.
  - For example: exceptions; ...
- An *error handling pattern* is a pattern of use of a language to respond (or not!) to errors.
  - For example: return code checking; ...
- This talk is about engineering of error handling mechanisms and patterns for reliability.
- *Reliability* in this context requires consistently doing something reasonable in the face of potential errors.

Error state

- Hold the "last error" in a global variable. Maybe access it via a function, maybe directly.
- Used in: ***C***/Unix (errno), Windows (GetLastError), ***Perl*** (), ***PHP*** (error_get_last)
- In systems that added threads, the global variable typically became per-thread to avoid the obvious conflict between threads. (***C***'s errno is typically a macro that accesses a per-thread variable.)
- Problems:
  - If the variable is only set when an error occurs, you need to reset it before any call that can fail.
  - If there are multiple calls between resetting and checking, you don't know which one failed.
  - Omitting to check for an error.
  - Confusion about which call failed.
  - In error handling code, a call might clobber the error value intended to be seen by a caller.

Return codes

- Return the error code from each function that might fail.
- Problems:
  - If the language doesn't directly support multiple return values, you might be unable to return some more natural value, or have to use an uglier API.
  - Conventions for what return codes mean (which code is used for success and which codes for particular errors) can be inconsistent, especially across libraries and code written by different people or teams.
  - Refactoring is hazardous: if the meaning of a return code changes then all callers must change.
  - It is easy to omit checking for an error.
  -

“goto fail”

- “goto fail” is an error handling pattern typically combined with error status or return codes. It separates recovery code from code in the usual path.
- Two recent bugs …
- OpenSSL:
- …
- Apple GnuTLS:
-

Error values

- Add an extra value of a type to indicate an error.
- Operations that are given an error value for any of their inputs, usually propagate it to their output.
- Examples:
  - ***IEEE 754*** NaN values
  - ***SQL*** NULL, sort of (its semantics are more suited to representing incomplete information)
  - as an object-oriented programming pattern (not to be confused with the Null Object pattern)
- Problems:
  - If only some types have error values, we get unintuitive behaviour (e.g. due to having an error value for double but not for boolean).
  - If all types have error values (as in ***SQL***), we need a three-valued logic which is unfamiliar.
  - There is a temptation to conflate "unknown", "missing" and "error" values...
  - ... but additional complexity can arise from having multiple kinds of error value (for example "quiet" and "signalling" NaNs in ***IEEE 754***).
  - Operations that take an error value as input may not produce one as output (e.g. the 2008 ***IEEE 754*** standard says that pow(1, quietNaN) and pow(quietNaN, 0) should return 1).
  - Error values can end up being stored in unanticipated places, delaying the detection of an error to arbitrarily long and far away from its introduction.
  - Although we could in principle record debug information about where the error value was first introduced, typically that isn't done.
  - It is easy to omit checking for an error unless "signalling" error values are used...
  - ... but those require an additional error handling mechanism for the signalling.

Error values

- What does this ***Java*** program print?

```
public class P {
    static <T> boolean equals_itself(T x) { return x == x; }
    static <T> boolean equals(T x, T y) { return x == y; }
    public static void main(String[] args) {
        double d = Double.NaN;
        System.out.println("d == d: "              + (d == d));
        System.out.println("equals_itself(d): " + equals_itself(d));
        System.out.println("equals(d, d): "      + equals(d, d));
    }
}
```

Error values

- What does this ***Java*** program print?

```
public class P {
    static <T> boolean equals_itself(T x) { return x == x; }
    static <T> boolean equals(T x, T y) { return x == y; }
    public static void main(String[] args) {
        double d = Double.NaN;
        System.out.println("d == d: "             + (d == d));
        System.out.println("equals_itself(d): " + equals_itself(d));
        System.out.println("equals(d, d): "      + equals(d, d));
    }
}
```

- java P

```
d == d: false
equals_itself(d): true
equals(d, d): false
```

(The uses of == in equals_itself and equals are reference equality on boxed Double objects. So equals_itself will always return true, while equals(x, x) will return false for primitive types, since boxing is repeated for each argument. In main, == is value equality on doubles, which is false for NaN == NaN.)

Handler subroutines

- Pass the … that is called if it fails.
- Advocated by Parnas in …
-
- Problems:
    - (minor) Needs closures in order for the handler subroutine to have sufficient context. This might be why it wasn't widely adopted.
    - …
    -
    - Confusion about which call failed.

Handlers associated with objects

- Set a handler on an object that is called if an operation on that object fails.
-
- Problems:
  - (minor) Needs closures in order for the handler subroutine to have sufficient context.
  - ...
  -

ON ERROR GOTO/GOSUB

- Register a handler to be called ...
- Used in ***PL/I***, several “structured” ***BASIC*** dialects, ...
- Problems:
    - Typically the registration of handlers was global.
    -

Recovery blocks

- Proposed by Randall in 1975
  -

```
ensure accept_condition
by P1
else by P2 ...
else by Pn
else error
```

Exceptions

- Proposed by John B. Goodenough in 1975.
- Goodenough's January 1975 paper is a classic, *very* thorough and well-written, and it mentions *all* of the mechanisms in the previous slides. Highly recommended!
- Exceptions in recent languages differ from Goodenough's proposal in the following ways:
    - Typically, they are only designed to be used to signal errors. Goodenough discussed using them as a general flow control mechanism, and possibly being raised from every call to a routine. He also discussed applications such as progress monitoring that would be a poor fit to exceptions as implemented in most current languages.

Exceptions

- Proposed by John B. Goodenough in 1975.
- Goodenough's January 1975 paper is a classic, *very* thorough and well-written, and it mentions *all* of the mechanisms in the previous slides. Highly recommended!
- Exceptions in recent languages differ from Goodenough's proposal in the following ways:
    - Typically, they are only designed to be used to signal errors. Goodenough discussed using them as a general flow control mechanism, and possibly being raised from every call to a routine. He also discussed applications such as progress monitoring that would be a poor fit to exceptions as implemented in most current languages.

Exceptions

- Exceptions were a huge improvement on previous techniques:
    - By default, if a caller includes no error handling code, the exception will be propagated to its caller.
    - In some languages, it is possible to statically declare which exceptions a routine can raise. (This is not particularly popular with ***Java*** programmers. We'll get to why in a later slide.)
- Summary of remaining problems:
    - Designing a good hierarchy of exception classes can be difficult.
    - ...

Exceptions

- Exceptions were proposed in 1975. They were adopted in “mainstream” languages in the 1990s. It's now 2014.
- There can be *no excuses* for ... known-broken error handling techniques in security libraries nearly 40 years after .... It's an embarrassment.
- Okay. End of rant. Calm now.
- How can we do better than exceptions?

Exception “safety”

- The following system of exception contracts was proposed by David Abrahams in the context of standardizing the ***C++*** standard library.
- The system classifies operations according to the guarantees they make about exceptions and state:
  - The basic guarantee: that the invariants of the component are preserved, and no resources are leaked.
  - The strong guarantee: that the operation has either completed successfully or thrown an exception, leaving the program state exactly as it was before the operation started.
  - The no-throw guarantee: that the operation will not throw an exception.
- This is essentially a static effect system.

Exception "safety"

- Unfortunately, this system does not compose well in general.
- The { nothrow, strong } subset composes well:
  - nothrow → nothrow()
  - strong → nothrow()
  - strong → strong()
  - nothrow → try { strong() } catch all { nothrow() }
- Adding a deepfrozen classifier for things that reference no preexisting state, also composes well:
  - strong → deepfrozen()
  - strong → *f*, *m* = deepfrozen(); try { f(m) } catch all { strong() }

  (The name and definition of deepfrozen comes from the auditing system of ***E***.)
- But if we have a basic operation, we have no way to use it to construct a strong or nothrow operation. (Abrahams' paper gives an example that claims to show otherwise but it is actually an implicit use of deepfrozen. XX need to refine this argument, explicitly take account of copying.)
- In the original context of the ***C++*** standard library, this problem is not so visible because the classifiers are being specified only for "leaf" operations.
- But for application programming, it defeats the general applicability of the whole idea – *unless* the language provides support for ensuring that mutating operations can be given the strong guarantee in general.

Exceptions

Exceptions + Mutable state

$\Downarrow$

Abstraction leaks

State rollback

- In Goodenough's 1975 paper [LINKME], it was briefly suggested that an exception could optionally cause state to be rolled back to the point … :
    - “”
- This was also an essential part of the definition of recovery blocks in Randell's paper: “”
- However, this was not taken up by any of the mainstream languages that implemented exceptions.
- Outside database systems and not counting Software Transactional Memory, recovery blocks have not been implemented in any commonly used language. [??]
-

Recovery blocks in terms of try/rollback

-
  ```
  ensure accept
          by P1
  else by P2

  ...

  else by Pn
  else error
  ```

  ⇒

  ```
  def accept() { ... }
  try              { P1; assert accept() }
  rollback all { try { P2; assert accept() }
  ...
  rollback all     { Pn; assert accept() } ... }
  ```

- This is a local macro transformation, so the original recovery block syntax doesn't provide any greater expressiveness.
- However it has a nice syntax that doesn't require the repetition of "assert accept()", and that guides programmers toward using the same acceptance test for each of the alternate blocks. This is highly desirable because it aids understanding of the program independently of the details of all the blocks.
- Randell's description allowed an acceptance test to refer to variables both before and after execution of the alternate block. This is slightly tricky with our anticipated implementation, but we can support it by capturing the variables referred to by the test.

Why isn't rollback supported more widely?

- If rollback-based error recovery was proposed in 1975 and is a good idea, why is it only supported in databases and a few research systems, rather than a common feature of general-purpose programming languages?
- Possible reasons:
  - Failure to fully appreciate costs of unreliable error handling in general-purpose programming;
  - Perceived performance cost;
  - Perceived narrow applicability (to "fault-tolerant software", or only to concurrent software, rather than error handling in general);
  - Perceived implementation complexity;
  - Perceived usage complexity (invalid! It's really simple);
  - Problems of interaction with I/O (solved);
  - Failure to recognise synergies between features enabled by rollback;
  - The computing community is collectively pretty forgetful of many good ideas.
- I'd like to address performance costs and synergies between features in the next few slides.

Features for cheap

Special offer!

5 for the price of 1 (ish)

Strong exception guarantee
Free GC write barriers (concurrent / generational)
Reverse debugging
Integrity enforcement
Distributed error recovery

Terms and conditions apply

Text reads: "Special offer! 5 for the price of 1 (ish): Strong exception guarantee, Concurrent GC write barrier, Reverse debugging, Integrity enforcement, Distributed error recovery. Terms and conditions apply."

Implementing rollback

- *Trailing* is a technique used in many ***Prolog*** implementations to implement backtracking search. Rollback is sufficiently similar to backtracking that we can use the same technique.
    - This similarity was already pointed out in Randell's 1975 paper [LINKME] on recovery blocks. As he notes, recovery points are likely to be less frequent than are *choice points* in backtracking.
- In typical programs, reads are more common than writes. [Vechev & Bacon 2004] says in the context of ***Java*** that "reads tend to outnumber writes by around five to one". This ratio will be higher for functional programming styles and languages.
- We have to accept the overhead of a *write barrier* (extra code associated with writes), but would like to avoid needing a *read barrier*. We'd also like to optimise out as much of the write barrier overhead as possible.
- This description will be in terms of a memory model that has *values* stored in *locations.* Allocation always produces fresh locations. (In practice we can represent locations as pointers, which might need to be rewritten if an object is moved in memory.) Note that a write is not necessarily a language-level mutation; writes also occur in the execution of pure functional programs.
- Trailing works by keeping a *trail stack* of (location, old value) pairs recorded by the write barrier.
- To roll back, we restore old values from the trail stack in reverse order.
- For each location, the trail stack only needs to store the old value for the first write after a recovery point, but there is no harm other than memory use caused by storing entries for additional writes.

How far?

- Every write adds a location pointer and old value to the trail stack. To maintain compositionality, we would ideally like *all* calls to satisfy the strong exception guarantee. But retaining all the information needed to roll back to the beginning of execution would introduce a space leak.
- So how far do we need to be able to roll back?
- One possible answer assumes that the program is structured as an *event loop* (or set of concurrent event loops). This is a popular programming model with many advantages. The event loop structure can be implemented in a framework or by the program itself.
- In event-loop models, execution in each task or "vat" is . Vats do not share mutable state, which allows us to ...
- When can we discard information needed to roll back?
  - Answer: we can discard representation states that are unobservable. This can be because there is no enclosing 'escape' or 'try', or because the task will be killed on failure. (When we add concurrency, this will depend on state not being shared between tasks.)
  - We introduce boundary calls: `boundary f(...)` ≈ `try { f(...); die PastBoundary } catch e { die e }`, to make the latter case explicit. [FIXME this equivalence makes no sense since we haven't described catch or die.] The stack and rollback information are discarded at a boundary call.
  - If the top level of any task program is a loop, with no 'escape' or 'try' outside the loop or with looping achieved by a boundary tail-call, then rollback state can be discarded on each iteration.
  - When we add concurrency, we will want fairly short event-loop turns for other reasons (e.g. responsiveness to incoming messages), which in practice limits the size of rollback information.
- Alternatively, we could just flush the compacted trail stack to persistent storage, taking advantage of the fact that disk is cheap and that earlier portions of the trail are unlikely to be accessed.

Optimising trailing

- We have two sources of ideas: optimisations for trailing in ***Prolog***, and *write barrier elision* for concurrent GC.
- Most of the trailing optimisations are based on the idea that we only need to retain sufficient information to get the state back to a recovery point, not any arbitrary execution point.
- Suppose that a location L has been allocated since the last recovery point. If we roll back, then L will no longer exist. So we don't need to trail writes to L (yet, until the next recovery point if L is still live then).
- Writes to locations that no longer exist can be discarded.
- We can reduce the number of trailed writes by applying optimizations from ***Prolog*** implementations, from the concurrent/incremental GC literature, and generic caching optimizations.
- Although it may be possible to elide trailing based on run-time tests, the cost of the test offsets the saving from avoided trailing, so this is not worthwhile (or wasn't for ***Prolog*** [Schrijvers and Demoen 1988 §2.1]). The most important optimizations come from static analysis:
    - Some code is statically known not to contain escape points.
    - Writing to a location only requires a trail entry if the location existed at the last escape point, and only for the first write after an escape point.
    - We can hoist trailing above loops that contain no escape points.
    - If a loop is probably going to write to a range of locations, we can trail that range, avoiding some overhead associated with trailing each location. Similarly for stereotyped patterns of writes to an object's fields.
    - If we know that a procedure is passed a cell that it writes to, we can change all callers to trail the cell before the call, and now the callee does not need to. (The advantage is that we might be able to further optimize by hoisting, merging etc. in the callers.)
    - If the previous value of the cell is known to be null, we can store just a tagged location rather than

Transaction overhead

Concurrent GC

- In general, garbage collection works by *conservatively* identifying at least all live objects, and then discarding objects not in that set.
  - Although strictly speaking only a conservative *set* of objects needs to be identified, it's easier to understand many of these algorithms as identifying a *graph* that includes at least all live objects.
- In concurrent GC, the aim is to keep the mutator running most of the time while the conservative graph is constructed.
- In "snapshot-at-the-beginning" concurrent GC algorithms, the conservative graph includes all edges that were in the live object graph at the time of the snapshot.
- If the mutator changes an edge in the current object graph, the old edge must be kept. This is typically done using a Yuasa write barrier (named for Taiichi Yuasa) which records the old value if it is a non-null pointer.
- A trailed write barrier is similar to a Yuasa write barrier, except that it must also record the location of the update, and must operate for non-pointer writes.
- Since the trail stack contains all the pointers that a Yuasa barrier would have recorded, we get the required write barrier for concurrent garbage collection "for free".
- Another way of thinking of this is that if we are able to roll back to a recovery point, we necessarily retain the live set at that recovery point.

Optimising concurrent GC

- If we assume that we have an event-loop system and are using trailing, there are ways to optimise GC to reduce the amount of tracing we need to do.
- The recovery point at a turn boundary is a good place to do a GC snapshot, because there is no stack and typically less "ephemeral" live data.
    - If a GC in a conventional system happens to run when there are many objects reachable from the stack, then it will need to trace those objects even if they would have been dead soon afterward. In event-loop systems it's common to allocate objects that are guaranteed to be dead at the end of the turn. Unless we need an "emergency" GC during a turn, the policy of scheduling GC at turn boundaries will ensure these objects are never traced.
- Generational GC is almost always a win (see ...). Assume we're doing that, and we have per-task nurseries, with nursery collection for a task being independent of other tasks. Assume that a particular nursery collection is snapshotted at a turn boundary. How can we reduce its cost?
    - We're not going to try to collect anything in the heap (i.e. the older generation(s)), just the nursery. What is the minimum we need to trace?
    - Pointers into the nursery from messages sent in this turn
    - Pointers into the nursery that were stored into the heap ← all of these are on the trail stack!
    - ...

Reverse debugging

- If we can roll back to any recovery point in normal execution, we can also do so when debugging.
- We can go back further than the last turn boundary if we take checkpoints.
- We can also roll forward from any recovery point. To follow the same path as the original execution, we need to have logged nondeterministic inputs (such as message arrival order, randomness, and ...) should have been logged.
-

Log analysis

- XX how programming languages can support the use of error handling for debugging and log analysis.
- XX not sure I have time to finish this

Integrity enforcement

- Integrity enforcement:
    - The trail stack also tells us which potentially-live objects have been changed at the end of a turn.
    - Therefore, we can ..., and fail the turn. (We can also replay the ...)

Distributed error recovery

- Fault tolerance in distributed message passing and/or shared memory systems has been a well-studied field for at least 40 years. However, adoption of fault-tolerant protocols has been limited.
- This is probably due to the complexity of most existing protocols that require close coordination between nodes.
- One of the main problems is the “domino effect”, in which error recovery on a given node can require cascading rollbacks on other nodes. The folk wisdom on distributed fault tolerance has tended to assume that the domino effect is inevitable.
- We need something simpler and more composable!
- ***Ken*** is a family of protocols that provide error recovery for detected faults without the domino effect.
- The first implementation was ***Waterken*** (for ***Java/Joe-E*** programs) developed by Tyler Close, followed by ***C-Ken*** for ***C*** programs.
- Research at Hewlett-Packard uses the term ***COVR*** (Composable Output-Valid Resiliency) to describe the properties of this protocol family.

Output validity

- "Output validity" is the property that the outputs of a distributed system in the presence of faults are outputs that could also have occurred under fault-free operation.
  - Note that this doesn't require that faults have no influence on the output.
  - Distributed message-passing systems are normally nondeterministic, because the arrival order of messages depends on timing. Faults can influence which nondeterministic choices are made, but this should not affect correctness.
- Assumptions:
  - *Messages* are *transmitted* and *received* by *nodes*, over a communication medium that sometimes loses or duplicates messages. So a node may transmit a message that is not received, or that is received more than once by the destination node.
  - We assume that messages are only received at their intended destination node as specified by the transmitter. A good way to do this is for each message to identify its recipient by an unforgeable *capability*, where a capability implies a particular destination node. If capabilities can also be sent in messages, this allows dynamic patterns of collaboration to be established securely.
  - Over an open communication medium, meeting these assumptions requires cryptography, including a key management system that assigns keys to nodes.
  - We only claim to recover from detected faults that do not involve loss of a node's persistent state.

How Ken/COVR works

- All communication between nodes is by message passing according to the properties below.
- Each node has a state that is checkpointed reasonably frequently, and independently. The state includes incoming and outgoing messages.
- Checkpoints are written durably to the node's persistent storage.
- Outgoing messages are *released* in the same atomic action as checkpointing.
- Released messages are periodically retransmitted until they are acknowledged. They can be deleted from the state only after they are acknowledged.
- A received message is not acknowledged until it has been recorded in a checkpoint as an incoming message.
- If a node fails, it restores from the last complete checkpoint. (Its behaviour in this case can't be distinguished from a node that had been running more slowly.)
- Received messages that a node already knows about are reacknowledged and then discarded.
- Duplicate acknowledgements are discarded.
- That's it.

What about deterministic errors?

- ***COVR*** protocols are suited to recovering from errors that are nondeterministic (such as hardware failures or temporary resource exhaustion), so that there is a good chance of avoiding the error on recovery.
- We should use other mechanisms to recover from deterministic asynchronous errors. Possibilities include:
  - ***Erlang***'s supervision hierarchies and process linking
  - ***E***'s
- Importantly, we don't need to know which errors are nondeterministic in advance. We can try to use a ***COVR*** protocol to recover, and then fall back to a different error handling mechanism if the error appears to be deterministic. Of course this violates output validity, but in a controlled way.

Thanks to

- This talk directly mentions ideas due to John B. Goodenough, Brian Randell, Tyler Close, Alan Karp, Marc Stiegler, Terence Kelly, Kevin Reid, and Mark S. Miller.
- Special thanks to Nathan Wilcox, the friam group, and the Lambda Ladies for their support and encouragement.
- Thanks to Zooko Wilcox O'Hearn for being an awesome boss and giving me sufficient time to work on this presentation, ***Noether***, and other fun stuff.
- Thanks also to Brian Warner, Zancas Wilcox, Meredith Patterson, Jonathan Shapiro, Dean Tribble, Norm Hardy, Carl Hewitt, Marc Stiegler, Mark Seaborn, Matej Kosik, Toby Murray, David Wagner, Charles Landau, Ben Laurie, Bill Frantz, Doug Crockford, Chip Morningstar, Constantine Plotnikov, David Barbour, Sandro Magi, David Mercer, Fred Speissens, Ka-Ping Yee, Jed Donnelley, Ihab Awad, Ivan Krstić, Jonathan Rees, Kris Zyp, Mike Samuel, Nick Szabo, Peter Van Roy, Pierre Thierry, Rob Jellinghaus, Tom Van Cutsem, Vijay Saraswat, Mike Stay, Patrick D. Logan, Amber Wilcox-O'Hearn, and Sophie Taylor.
- Last but never least, thanks to my partners Allie, Aura and Samantha.

In solidarity with the people of Ferguson
and their protests against police violence

How to Make Error Handling Less Error-Prone

or…

BACK TO THE FUTURE

Rescuing error handling ideas from 1975!

(the year I was born :-)

How to Make Error Handling Less Error-Prone

- Introduction
- Sequential error handling
  - Error status; Return codes; Error values; Trapping; ...
  - Exceptions (history, restarts, ...)
- Concurrent error handling
  - Supervision hierarchies
  - Fate-sharing
- Rollback error recovery
- Motivation; conflicts between encapsulation and having enough information to recover a consistent state;
- how automatic rollback solves the encapsulation problem;
- how this form of rollback differs from database-style transactions or transactional memory;
- optimisations
- the Ken protocol;
-

Introduction

- "Error" comes from a Latin verb meaning "to stray".

  For our purposes, *errors* are cases where the execution of a program strays from its usual path.
- We'll be considering errors in relation to high-level language programming and distributed systems.
  - Errors at other layers –operating systems, hardware, etc.– are important but not what this talk is focussed on.
- This includes:
  - cases that the language specification defines to be errors, or where the language implementation is unable to meet its defined specification;
  - cases that the program defines to be errors, or where the program is unable to meet its defined specification.
- *Error handling* is the response to errors.
- An *error handling mechanism* is a feature of a language designed to enable responding to errors.
  - For example: exceptions; ...
- An *error handling pattern* is a pattern of use of a language to respond (or not!) to errors.
  - For example: return code checking; ...
- This talk is about engineering of error handling mechanisms and patterns for reliability.
- *Reliability* in this context requires consistently doing something reasonable in the face of potential errors.

Error state

- Hold the “last error” in a global variable. Maybe access it via a function, maybe directly.
- Used in: ***C***/Unix (errno), Windows (GetLastError), ***Perl*** (), ***PHP*** (error_get_last)
- In systems that added threads, the global variable typically became per-thread to avoid the obvious conflict between threads. (***C***'s errno is typically a macro that accesses a per-thread variable.)
- Problems:
  - If the variable is only set when an error occurs, you need to reset it before any call that can fail.
  - If there are multiple calls between resetting and checking, you don't know which one failed.
  - Omitting to check for an error.
  - Confusion about which call failed.
  - In error handling code, a call might clobber the error value intended to be seen by a caller.

Return codes

- Return the error code from each function that might fail.
- Problems:
  - If the language doesn't directly support multiple return values, you might be unable to return some more natural value, or have to use an uglier API.
  - Conventions for what return codes mean (which code is used for success and which codes for particular errors) can be inconsistent, especially across libraries and code written by different people or teams.
  - Refactoring is hazardous: if the meaning of a return code changes then all callers must change.
  - It is easy to omit checking for an error.
  -

“goto fail”

- “goto fail” is an error handling pattern typically combined with error status or return codes. It separates recovery code from code in the usual path.
- Two recent bugs …
- OpenSSL:
- …
- Apple GnuTLS:
-

Error values

- Add an extra value of a type to indicate an error.
- Operations that are given an error value for any of their inputs, usually propagate it to their output.
- Examples:
  - ***IEEE 754*** NaN values
  - ***SQL*** NULL, sort of (its semantics are more suited to representing incomplete information)
  - as an object-oriented programming pattern (not to be confused with the Null Object pattern)
- Problems:
  - If only some types have error values, we get unintuitive behaviour (e.g. due to having an error value for `double` but not for `boolean`).
  - If all types have error values (as in ***SQL***), we need a three-valued logic which is unfamiliar.
  - There is a temptation to conflate "unknown", "missing" and "error" values...
  - ... but additional complexity can arise from having multiple kinds of error value (for example "quiet" and "signalling" NaNs in ***IEEE 754***).
  - Operations that take an error value as input may not produce one as output (e.g. the 2008 ***IEEE 754*** standard says that pow(1, quietNaN) and pow(quietNaN, 0) should return 1).
  - Error values can end up being stored in unanticipated places, delaying the detection of an error to arbitrarily long and far away from its introduction.
  - Although we could in principle record debug information about where the error value was first introduced, typically that isn't done.
  - It is easy to omit checking for an error unless "signalling" error values are used...
  - ... but those require an additional error handling mechanism for the signalling.

Error values

- What does this ***Java*** program print?

```
public class P {
   static <T> boolean equals_itself(T x) { return x == x; }
   static <T> boolean equals(T x, T y) { return x == y; }
   public static void main(String[] args) {
      double d = Double.NaN;
      System.out.println("d == d: "             + (d == d));
      System.out.println("equals_itself(d): " + equals_itself(d));
      System.out.println("equals(d, d): "      + equals(d, d));
   }
}
```

Error values

- What does this ***Java*** program print?

```
public class P {
  static <T> boolean equals_itself(T x) { return x == x; }
  static <T> boolean equals(T x, T y) { return x == y; }
  public static void main(String[] args) {
    double d = Double.NaN;
    System.out.println("d == d: "              + (d == d));
    System.out.println("equals_itself(d): " + equals_itself(d));
    System.out.println("equals(d, d): "       + equals(d, d));
  }
}
```

- java P

```
d == d: false
equals_itself(d): true
equals(d, d): false
```

(The uses of == in equals_itself and equals are reference equality on boxed Double objects. So equals_itself will always return true, while equals(x, x) will return false for primitive types, since boxing is repeated for each argument. In main, == is value equality on doubles, which is false for NaN == NaN.)

Handler subroutines

- Pass the ... that is called if it fails.
- Advocated by Parnas in ...
-
- Problems:
  - (minor) Needs closures in order for the handler subroutine to have sufficient context. This might be why it wasn't widely adopted.
  - ...
  -
  - Confusion about which call failed.

Handlers associated with objects

- Set a handler on an object that is called if an operation on that object fails.
-
- Problems:
    - (minor) Needs closures in order for the handler subroutine to have sufficient context.
    - ...
    -

ON ERROR GOTO/GOSUB

- Register a handler to be called ...
- Used in ***PL/I***, several "structured" ***BASIC*** dialects, ...
- Problems:
    - Typically the registration of handlers was global.
    -

Recovery blocks

- Proposed by Randall in 1975
    -

```
ensure accept_condition
by P1
else by P2 ...
else by Pn
else error
```

Exceptions

- Proposed by John B. Goodenough in 1975.
- Goodenough's January 1975 paper is a classic, *very* thorough and well-written, and it mentions *all* of the mechanisms in the previous slides. Highly recommended!
- Exceptions in recent languages differ from Goodenough's proposal in the following ways:
    - Typically, they are only designed to be used to signal errors. Goodenough discussed using them as a general flow control mechanism, and possibly being raised from every call to a routine. He also discussed applications such as progress monitoring that would be a poor fit to exceptions as implemented in most current languages.
    -
-

Exceptions

- Proposed by John B. Goodenough in 1975.
- Goodenough's January 1975 paper is a classic, *very* thorough and well-written, and it mentions *all* of the mechanisms in the previous slides. Highly recommended!
- Exceptions in recent languages differ from Goodenough's proposal in the following ways:
    - Typically, they are only designed to be used to signal errors. Goodenough discussed using them as a general flow control mechanism, and possibly being raised from every call to a routine. He also discussed applications such as progress monitoring that would be a poor fit to exceptions as implemented in most current languages.
    -
-

Exceptions

- Exceptions were a huge improvement on previous techniques:
  - By default, if a caller includes no error handling code, the exception will be propagated to its caller.
  - In some languages, it is possible to statically declare which exceptions a routine can raise. (This is not particularly popular with ***Java*** programmers. We'll get to why in a later slide.)
- Summary of remaining problems:
  - Designing a good hierarchy of exception classes can be difficult.
  - ...

Exceptions

- Exceptions were proposed in 1975. They were adopted in “mainstream” languages in the 1990s. It's now 2014.
- There can be *no excuses* for … known-broken error handling techniques in security libraries nearly 40 years after …. It's an embarrassment.
- Okay. End of rant. Calm now.
- How can we do better than exceptions?

Exception “safety”

- The following system of exception contracts was proposed by David Abrahams in the context of standardizing the ***C++*** standard library.
- The system classifies operations according to the guarantees they make about exceptions and state:
  - The basic guarantee: that the invariants of the component are preserved, and no resources are leaked.
  - The strong guarantee: that the operation has either completed successfully or thrown an exception, leaving the program state exactly as it was before the operation started.
  - The no-throw guarantee: that the operation will not throw an exception.
- This is essentially a static effect system.

Exception "safety"

- Unfortunately, this system does not compose well in general.
- The { nothrow, strong } subset composes well:
  - nothrow → nothrow()
  - strong → nothrow()
  - strong → strong()
  - nothrow → try { strong() } catch all { nothrow() }
- Adding a deepfrozen classifier for things that reference no preexisting state, also composes well:
  - strong → deepfrozen()
  - strong → *f*, *m* = deepfrozen(); try { f(m) } catch all { strong() }

  (The name and definition of deepfrozen comes from the auditing system of ***E***.)
- But if we have a basic operation, we have no way to use it to construct a strong or nothrow operation. (Abrahams' paper gives an example that claims to show otherwise but it is actually an implicit use of deepfrozen. XX need to refine this argument, explicitly take account of copying.)
- In the original context of the ***C++*** standard library, this problem is not so visible because the classifiers are being specified only for "leaf" operations.
- But for application programming, it defeats the general applicability of the whole idea – *unless* the language provides support for ensuring that mutating operations can be given the strong guarantee in general.

Exceptions

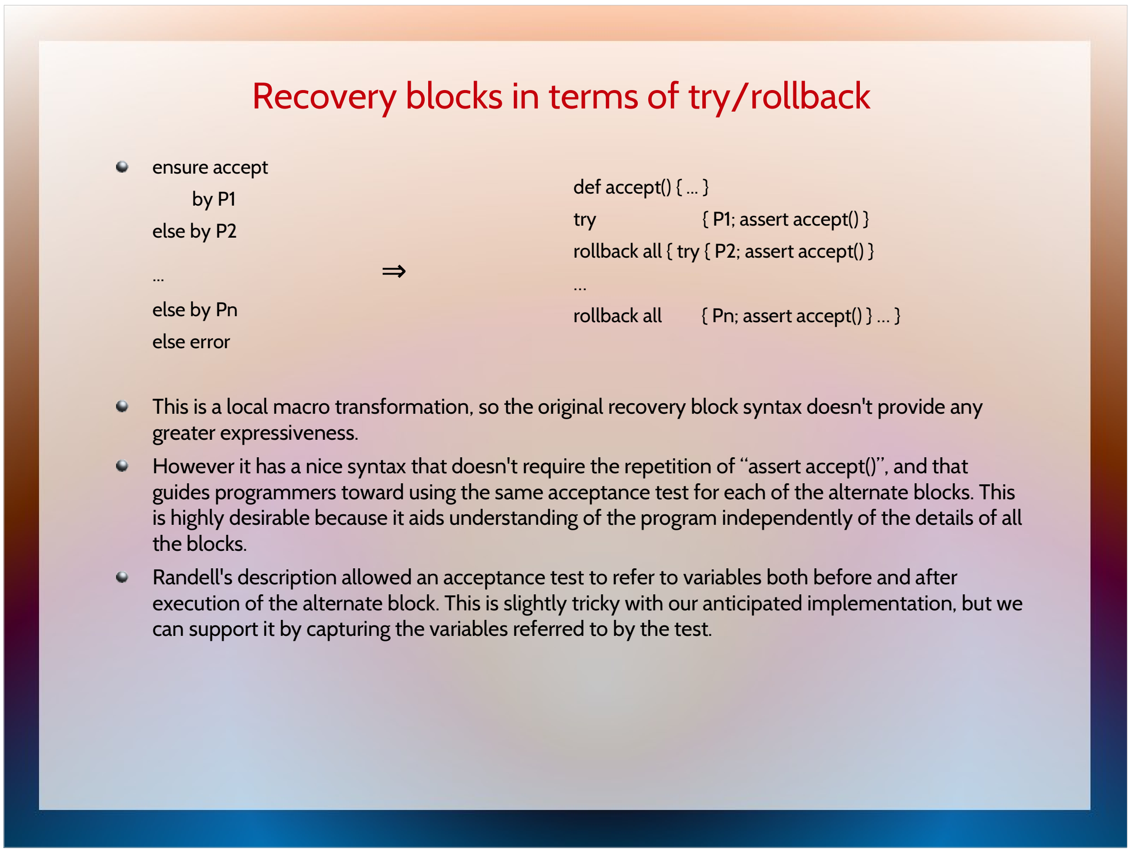
Exceptions + Mutable state

$\Downarrow$

Abstraction leaks

State rollback

- In Goodenough's 1975 paper [LINKME], it was briefly suggested that an exception could optionally cause state to be rolled back to the point ... :
    - “”
- This was also an essential part of the definition of recovery blocks in Randell's paper: “”
- However, this was not taken up by any of the mainstream languages that implemented exceptions.
- Outside database systems and not counting Software Transactional Memory, recovery blocks have not been implemented in any commonly used language. [??]
-



Recovery blocks in terms of try/rollback

```
• ensure accept                                def accept() { … }
        by P1                                  try              { P1; assert accept() }
  else by P2                                   rollback all { try { P2; assert accept() }
                              ⇒
  …                                            …
  else by Pn                                   rollback all     { Pn; assert accept() } … }
  else error
```

- This is a local macro transformation, so the original recovery block syntax doesn't provide any greater expressiveness.
- However it has a nice syntax that doesn't require the repetition of "assert accept()", and that guides programmers toward using the same acceptance test for each of the alternate blocks. This is highly desirable because it aids understanding of the program independently of the details of all the blocks.
- Randell's description allowed an acceptance test to refer to variables both before and after execution of the alternate block. This is slightly tricky with our anticipated implementation, but we can support it by capturing the variables referred to by the test.

Why isn't rollback supported more widely?

- If rollback-based error recovery was proposed in 1975 and is a good idea, why is it only supported in databases and a few research systems, rather than a common feature of general-purpose programming languages?
- Possible reasons:
  - Failure to fully appreciate costs of unreliable error handling in general-purpose programming;
  - Perceived performance cost;
  - Perceived narrow applicability (to “fault-tolerant software”, or only to concurrent software, rather than error handling in general);
  - Perceived implementation complexity;
  - Perceived usage complexity (invalid! It's really simple);
  - Problems of interaction with I/O (solved);
  - Failure to recognise synergies between features enabled by rollback;
  - The computing community is collectively pretty forgetful of many good ideas.

- I'd like to address performance costs and synergies between features in the next few slides.

Features for cheap

Special offer!

5 for the price of 1 (ish)

Strong exception guarantee
Free GC write barriers $\left(\substack{\text{concurrent}\\ \text{generational}}\right)$
Reverse debugging
Integrity enforcement
Distributed error recovery

Terms and conditions apply

Text reads: "Special offer! 5 for the price of 1 (ish): Strong exception guarantee, Concurrent GC write barrier, Reverse debugging, Integrity enforcement, Distributed error recovery. Terms and conditions apply."

Implementing rollback

- *Trailing* is a technique used in many ***Prolog*** implementations to implement backtracking search. Rollback is sufficiently similar to backtracking that we can use the same technique.
    - This similarity was already pointed out in Randell's 1975 paper [LINKME] on recovery blocks. As he notes, recovery points are likely to be less frequent than are *choice points* in backtracking.
- In typical programs, reads are more common than writes. [Vechev & Bacon 2004] says in the context of ***Java*** that "reads tend to outnumber writes by around five to one". This ratio will be higher for functional programming styles and languages.
- We have to accept the overhead of a *write barrier* (extra code associated with writes), but would like to avoid needing a *read barrier*. We'd also like to optimise out as much of the write barrier overhead as possible.
- This description will be in terms of a memory model that has *values* stored in *locations.* Allocation always produces fresh locations. (In practice we can represent locations as pointers, which might need to be rewritten if an object is moved in memory.) Note that a write is not necessarily a language-level mutation; writes also occur in the execution of pure functional programs.
- Trailing works by keeping a *trail stack* of (location, old value) pairs recorded by the write barrier.
- To roll back, we restore old values from the trail stack in reverse order.
- For each location, the trail stack only needs to store the old value for the first write after a recovery point, but there is no harm other than memory use caused by storing entries for additional writes.

How far?

- Every write adds a location pointer and old value to the trail stack. To maintain compositionality, we would ideally like *all* calls to satisfy the strong exception guarantee. But retaining all the information needed to roll back to the beginning of execution would introduce a space leak.
- So how far do we need to be able to roll back?
- One possible answer assumes that the program is structured as an *event loop* (or set of concurrent event loops). This is a popular programming model with many advantages. The event loop structure can be implemented in a framework or by the program itself.
- In event-loop models, execution in each task or "vat" is . Vats do not share mutable state, which allows us to ...
- When can we discard information needed to roll back?
    - Answer: we can discard representation states that are unobservable. This can be because there is no enclosing 'escape' or 'try', or because the task will be killed on failure. (When we add concurrency, this will depend on state not being shared between tasks.)
    - We introduce boundary calls: `boundary f(...) ≈ try { f(...); die PastBoundary } catch e { die e }`, to make the latter case explicit. [FIXME this equivalence makes no sense since we haven't described catch or die.] The stack and rollback information are discarded at a boundary call.
    - If the top level of any task program is a loop, with no 'escape' or 'try' outside the loop or with looping achieved by a boundary tail-call, then rollback state can be discarded on each iteration.
    - When we add concurrency, we will want fairly short event-loop turns for other reasons (e.g. responsiveness to incoming messages), which in practice limits the size of rollback information.
- Alternatively, we could just flush the compacted trail stack to persistent storage, taking advantage of the fact that disk is cheap and that earlier portions of the trail are unlikely to be accessed.

Optimising trailing

- We have two sources of ideas: optimisations for trailing in ***Prolog***, and *write barrier elision* for concurrent GC.
- Most of the trailing optimisations are based on the idea that we only need to retain sufficient information to get the state back to a recovery point, not any arbitrary execution point.
- Suppose that a location L has been allocated since the last recovery point. If we roll back, then L will no longer exist. So we don't need to trail writes to L (yet, until the next recovery point if L is still live then).
- Writes to locations that no longer exist can be discarded.
- We can reduce the number of trailed writes by applying optimizations from ***Prolog*** implementations, from the concurrent/incremental GC literature, and generic caching optimizations.
- Although it may be possible to elide trailing based on run-time tests, the cost of the test offsets the saving from avoided trailing, so this is not worthwhile (or wasn't for ***Prolog*** [Schrijvers and Demoen 1988 §2.1]). The most important optimizations come from static analysis:
    - Some code is statically known not to contain escape points.
    - Writing to a location only requires a trail entry if the location existed at the last escape point, and only for the first write after an escape point.
    - We can hoist trailing above loops that contain no escape points.
    - If a loop is probably going to write to a range of locations, we can trail that range, avoiding some overhead associated with trailing each location. Similarly for stereotyped patterns of writes to an object's fields.
    - If we know that a procedure is passed a cell that it writes to, we can change all callers to trail the cell before the call, and now the callee does not need to. (The advantage is that we might be able to further optimize by hoisting, merging etc. in the callers.)
    - If the previous value of the cell is known to be null, we can store just a tagged location rather than

Transaction overhead

Concurrent GC

- In general, garbage collection works by *conservatively* identifying at least all live objects, and then discarding objects not in that set.
  - Although strictly speaking only a conservative *set* of objects needs to be identified, it's easier to understand many of these algorithms as identifying a *graph* that includes at least all live objects.
- In concurrent GC, the aim is to keep the mutator running most of the time while the conservative graph is constructed.
- In "snapshot-at-the-beginning" concurrent GC algorithms, the conservative graph includes all edges that were in the live object graph at the time of the snapshot.
- If the mutator changes an edge in the current object graph, the old edge must be kept. This is typically done using a Yuasa write barrier (named for Taiichi Yuasa) which records the old value if it is a non-null pointer.
- A trailed write barrier is similar to a Yuasa write barrier, except that it must also record the location of the update, and must operate for non-pointer writes.
- Since the trail stack contains all the pointers that a Yuasa barrier would have recorded, we get the required write barrier for concurrent garbage collection "for free".
- Another way of thinking of this is that if we are able to roll back to a recovery point, we necessarily retain the live set at that recovery point.

Optimising concurrent GC

- If we assume that we have an event-loop system and are using trailing, there are ways to optimise GC to reduce the amount of tracing we need to do.
- The recovery point at a turn boundary is a good place to do a GC snapshot, because there is no stack and typically less "ephemeral" live data.
  - If a GC in a conventional system happens to run when there are many objects reachable from the stack, then it will need to trace those objects even if they would have been dead soon afterward. In event-loop systems it's common to allocate objects that are guaranteed to be dead at the end of the turn. Unless we need an "emergency" GC during a turn, the policy of scheduling GC at turn boundaries will ensure these objects are never traced.
- Generational GC is almost always a win (see ...). Assume we're doing that, and we have per-task nurseries, with nursery collection for a task being independent of other tasks. Assume that a particular nursery collection is snapshotted at a turn boundary. How can we reduce its cost?
  - We're not going to try to collect anything in the heap (i.e. the older generation(s)), just the nursery. What is the minimum we need to trace?
  - Pointers into the nursery from messages sent in this turn
  - Pointers into the nursery that were stored into the heap ← all of these are on the trail stack!
  - ...

Reverse debugging

- If we can roll back to any recovery point in normal execution, we can also do so when debugging.
- We can go back further than the last turn boundary if we take checkpoints.
- We can also roll forward from any recovery point. To follow the same path as the original execution, we need to have logged nondeterministic inputs (such as message arrival order, randomness, and ...) should have been logged.
-

Log analysis

- XX how programming languages can support the use of error handling for debugging and log analysis.
- XX not sure I have time to finish this

Integrity enforcement

- Integrity enforcement:
  - The trail stack also tells us which potentially-live objects have been changed at the end of a turn.
  - Therefore, we can ..., and fail the turn. (We can also replay the ...)

Distributed error recovery

- Fault tolerance in distributed message passing and/or shared memory systems has been a well-studied field for at least 40 years. However, adoption of fault-tolerant protocols has been limited.
- This is probably due to the complexity of most existing protocols that require close coordination between nodes.
- One of the main problems is the "domino effect", in which error recovery on a given node can require cascading rollbacks on other nodes. The folk wisdom on distributed fault tolerance has tended to assume that the domino effect is inevitable.
- We need something simpler and more composable!
- ***Ken*** is a family of protocols that provide error recovery for detected faults without the domino effect.
- The first implementation was ***Waterken*** (for ***Java/Joe-E*** programs) developed by Tyler Close, followed by ***C-Ken*** for ***C*** programs.
- Research at Hewlett-Packard uses the term ***COVR*** (Composable Output-Valid Resiliency) to describe the properties of this protocol family.

Output validity

- "Output validity" is the property that the outputs of a distributed system in the presence of faults are outputs that could also have occurred under fault-free operation.
  - Note that this doesn't require that faults have no influence on the output.
  - Distributed message-passing systems are normally nondeterministic, because the arrival order of messages depends on timing. Faults can influence which nondeterministic choices are made, but this should not affect correctness.
- Assumptions:
  - *Messages* are *transmitted* and *received* by *nodes*, over a communication medium that sometimes loses or duplicates messages. So a node may transmit a message that is not received, or that is received more than once by the destination node.
  - We assume that messages are only received at their intended destination node as specified by the transmitter. A good way to do this is for each message to identify its recipient by an unforgeable *capability*, where a capability implies a particular destination node. If capabilities can also be sent in messages, this allows dynamic patterns of collaboration to be established securely.
  - Over an open communication medium, meeting these assumptions requires cryptography, including a key management system that assigns keys to nodes.
  - We only claim to recover from detected faults that do not involve loss of a node's persistent state.

How Ken/COVR works

- All communication between nodes is by message passing according to the properties below.
- Each node has a state that is checkpointed reasonably frequently, and independently. The state includes incoming and outgoing messages.
- Checkpoints are written durably to the node's persistent storage.
- Outgoing messages are *released* in the same atomic action as checkpointing.
- Released messages are periodically retransmitted until they are acknowledged. They can be deleted from the state only after they are acknowledged.
- A received message is not acknowledged until it has been recorded in a checkpoint as an incoming message.
- If a node fails, it restores from the last complete checkpoint. (Its behaviour in this case can't be distinguished from a node that had been running more slowly.)
- Received messages that a node already knows about are reacknowledged and then discarded.
- Duplicate acknowledgements are discarded.
- That's it.

What about deterministic errors?

- ***COVR*** protocols are suited to recovering from errors that are nondeterministic (such as hardware failures or temporary resource exhaustion), so that there is a good chance of avoiding the error on recovery.
- We should use other mechanisms to recover from deterministic asynchronous errors. Possibilities include:
  - ***Erlang***'s supervision hierarchies and process linking
  - ***E***'s
- Importantly, we don't need to know which errors are nondeterministic in advance. We can try to use a ***COVR*** protocol to recover, and then fall back to a different error handling mechanism if the error appears to be deterministic. Of course this violates output validity, but in a controlled way.

Thanks to

- This talk directly mentions ideas due to John B. Goodenough, Brian Randell, Tyler Close, Alan Karp, Marc Stiegler, Terence Kelly, Kevin Reid, and Mark S. Miller.
- Special thanks to Nathan Wilcox, the friam group, and the Lambda Ladies for their support and encouragement.
- Thanks to Zooko Wilcox O'Hearn for being an awesome boss and giving me sufficient time to work on this presentation, ***Noether***, and other fun stuff.
- Thanks also to Brian Warner, Zancas Wilcox, Meredith Patterson, Jonathan Shapiro, Dean Tribble, Norm Hardy, Carl Hewitt, Marc Stiegler, Mark Seaborn, Matej Kosik, Toby Murray, David Wagner, Charles Landau, Ben Laurie, Bill Frantz, Doug Crockford, Chip Morningstar, Constantine Plotnikov, David Barbour, Sandro Magi, David Mercer, Fred Speissens, Ka-Ping Yee, Jed Donnelley, Ihab Awad, Ivan Krstić, Jonathan Rees, Kris Zyp, Mike Samuel, Nick Szabo, Peter Van Roy, Pierre Thierry, Rob Jellinghaus, Tom Van Cutsem, Vijay Saraswat, Mike Stay, Patrick D. Logan, Amber Wilcox-O'Hearn, and Sophie Taylor.
- Last but never least, thanks to my partners Allie, Aura and Samantha.