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 the 1970s!

How to Make Error Handling Less Error-Prone

- Introduction
- Sequential error handling
    - Error status; Return codes; Error values; Handlers
    - Exceptions; Restarts
    - Recovery blocks
- Rollback error recovery
    - Motivation; conflicts between encapsulation and having enough information to recover a consistent state
    - Optimisations
- Concurrent error handling
    - Fault masking
    - The ***Ken*** protocol
    - Fate sharing
    - Supervision hierarchies

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 or program defines to be errors;
  - cases where the language implementation or program is unable to meet its specification.
- *Error handling* is the response to errors.
  - An *error handling mechanism* is a feature of a language designed to enable responding to errors.

    An *error handling pattern* is a pattern of use of a language to respond (or not!) to errors.
- This talk is about engineering of error handling mechanisms and patterns for dependability.
  - *Dependability* means consistently doing something reasonable, that other code or a user could depend on, in the face of potential errors.
- Let's consider some common error handling mechanisms and patterns, and what can go wrong with them.

Error state

- Hold the “last error” in a per-process or per-thread variable. Maybe access it via a function, maybe directly.
- Used in: ***C***/Unix (errno); Windows API (GetLastError); ***Perl*** (you don’t want to know); ***PHP*** (error_get_last); various shell languages
- Problems:
  - Programmers may misunderstand the defined conventions for use of the state variable, which are often subtle or badly documented.

    For example, POSIX says: “The setting of errno after a successful call to a function is unspecified unless the description of that function specifies that errno shall not be modified.” So the commonly used pattern (recommended in footnote 202 of the ***C99*** and ***C11*** standards) of setting errno to zero before a function call and then checking that it is non-zero afterward is *incorrect*; in almost all cases it’s necessary to check the return value after each call instead.

    Windows’ GetLastError is even less consistent than this.
  - The path of least resistance is to omit checking for an error. This is very common in ***C*** and Windows programs.
  - In “cleanup” code before an error is propagated to the caller, another operation might clobber the error state.
    - Languages that require less explicit cleanup are in a better position here.
  - Because error checking is programmed explicitly, it's easy to get it wrong.

Return codes

- Return an error code, or a value indicating success, from each function that might fail.
- Used in: ***C***; most operating system APIs; most library APIs in languages without exceptions.
- Problems:
  - If the language doesn't directly support multiple return values, you might be unable to also return some more natural result, 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.
  - The path of least resistance is to omit checking for an error.
  - If cleanup is needed, it is easy to fail to propagate the error correctly.

- The cause of the GnuTLS certificate checking bug was a combination of several of these problems. There was a function called `check_if_ca` that was defined to return non-zero if given a CA certificate, but this conflicted with the usual convention in GnuTLS of returning a negative integer for an error. The bug was introduced by a refactoring that changed many functions, most of which used the usual return convention. Cleanup handling was also a factor because it resulted in the return being further from the code that checked for errors.
- In the Apple OpenSSL "goto fail;" bug, the cleanup code was written incorrectly, resulting in part of the necessary code for certificate checking being bypassed.

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 missing 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 "missing" or "not applicable" values with error values, even if their semantics should be different.
  - On the other hand, 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).

Fun with NaNs

- 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));
    }
}
```

Fun with NaNs

- 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
```

(In main, "==" is value equality on doubles, which is false for NaN == NaN. In equals_itself and equals, "==" is 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.)

Handlers with per-process/thread extent

- A per-process or per-thread variable points to a handler for a given subset of errors.
- Used in: ***Unix*** (for example sigfpe and other signals); ***PL/I***; some ***BASIC***s
- Problems:
  - There is a mismatch between the context available to the handler (e.g. lexical context if the language has closures), and the extent in which it is enabled.
  - The handler usually has little if any information about which call failed.
  - A handler may accidentally be left enabled for longer than it should be, or the previous handler may not be restored.
  - Handlers can't be easily combined; only one is valid at a time for a given error category.
  - Since the handler can be called at essentially any time, it is very restricted in what it can safely do.
  - It's not possible to construct a dependable error handling mechanism by invoking a handler *after* there has already been undefined behaviour, as is the case for some ***Unix*** signals.

Handlers associated with objects or contexts

- Set a handler on an object that is called if an operation on that object fails.
- Used in: gtk, OpenSSL
- Problems:
  - It can be tricky to ensure that handlers are enabled on all relevant objects/contexts.
  - It's not clear that handling usually depends on the object (as opposed to the calling context).
  - Handlers can't be easily combined.
  - Some other error handling mechanism is also needed if we want to affect the control flow of the calling context.

Exceptions

- Proposed by John B. Goodenough in 1974/5.
- There were other proposals with similar properties at about the same time or slightly earlier, for example the signalling mechanism in section 8.1 of [LMS 1974].
- Goodenough's 1975 paper is a classic, *very* thorough and well-written, and it mentions all of the mechanisms in the previous slides. Highly recommended!
- I'm going to assume knowledge of how exceptions work in recent languages. Goodenough's original proposal differed in the following ways:
  - Typically, exceptions are now 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.
  - Common Lisp's condition/restart mechanism is probably provides the nearest thing to this original proposal that has been implemented in a widely used language.
  - In other languages, applications for this extra functionality would probably be better implemented via continuations or generators.
- 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.
  - The exception handler only runs in the error case, so it is impossible for it to mess up the non-error case.
  - In some languages, it is possible to statically declare which exceptions a routine can raise.

Exceptions

- Exceptions are "good enough" (groan) for most purposes. Use them in preference to the mechanisms described earlier!
- Remaining problems:
  - Designing a good hierarchy of exception classes can be difficult.
  - Exceptions don't solve the problem of maintaining a consistent state.

$$\text{Exceptions} + \text{Mutable state}$$
$$\Downarrow$$
$$\text{Abstraction leaks}$$

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, it's difficult to a strong or nothrow operation from it. Abrahams' paper suggests copying, something like this:
  - strong → { *f* = deepfrozen(); try { *m2* = copy(m1); f(m2) } catch all { strong() } }
- But we may not want to copy! The goal may be to modify the original object/structure.
- 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.

Recovery blocks

- In Goodenough's 1975 paper, it was briefly suggested that "Sometimes as a side-effect of terminating the operation, it is necessary to undo all effects of attempting the operation."

  However, this was not taken up by any of the mainstream languages that implemented exceptions.
- *Recovery blocks* are another mechanism in which rollback is an essential part of the semantics. They were proposed around the same time as exceptions (1974), by Brian Randell's research group at the University of Newcastle upon Tyne (which evolved into the Dependability Research Group; see [Randell 1997] for the history).
- A recovery block looks like this:

```
ensure    acceptance_test
by        Alt₁
[else by  Alt₂
            ⋮
 else by  Altₙ]
else error
```

- First Alt_1 is executed. If it fails or if the acceptance condition is not true after its execution, Alt_2 is executed, and so on. If none of the alternates succeed in making the acceptance condition true, then the ensure construct fails.
- Note that the alternate blocks can fail independently of the acceptance test. This can happen if all alternatives of a nested block fail, or if there is some error signalled by the language or program.
- The acceptance condition can refer to variables both before and after the execution of the alternates.

Recovery blocks *vs* try/catch

- Compare:

```
ensure   true
by       Alt₁
else by  Alt₂
else error
```

```
try        { Alt₁ }
catch all  { Alt₂ }
```

- The difference is that in the recovery block, state changed by Alt_1 will be rolled back before executing Alt_2, if Alt_1 fails.
- If recovery blocks were used consistently in place of exceptions, the effect would be that all code would satisfy the strong exception guarantee. (It is unobservable whether the rollback is done by each function that is called when it propagates an exception, or by ensure constructs, provided that it is never possible to catch an exception without performing rollback.)
- The acceptance condition and the ability to catch only a subset of errors are inessential differences: we could extend try/catch with an acceptance condition, or provide a way for alternates in a recovery block to depend on which error is raised by a previous alternate. (Randell *et al* quite deliberately avoided the latter, but it does increase expressiveness.)
- The intent is also different however: the role of alternates in a recovery block is to provide alternative implementations for an operation, whereas exceptions are normally used to clean up and/or to divert control flow.

Recovery blocks in terms of try/rollback

- Imagine we have a try/rollback construct, which works like try/catch but rolls back any state changes before the rollback block executes.
- Then we have the following equivalence:

```
ensure accept                  def accept() { … }
       by Alt₁                 try                { Alt₁; assert accept() }
else by Alt₂                   rollback all { try { Alt₂; assert accept() }
…                     ⇔        …
else by Altₙ                   rollback all       { Altₙ; assert accept() } … }
else error
```

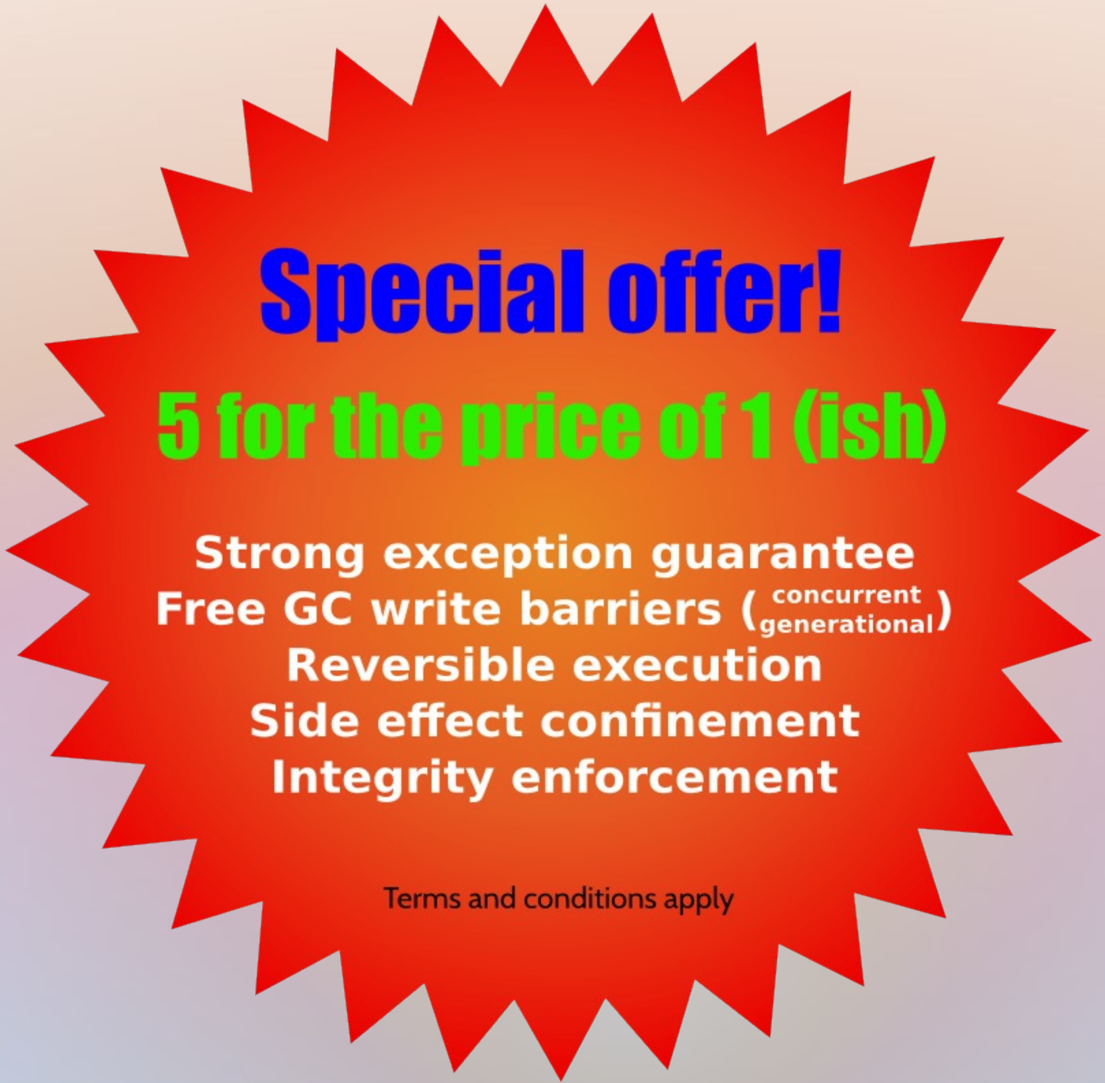
- This is a local macro transformation, so the original recovery block syntax doesn't provide any greater expressiveness.
- However it has a nice syntax that doesn't require the repetition of "assert accept()", and that guides programmers toward using the same acceptance test for each of the alternate blocks. This is highly desirable because it aids understanding of the program independently of the details of all the blocks.
- Allowing an acceptance test to refer to variables both before and after the alternate block can be supported 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)
Reversible execution
Side effect confinement
Integrity enforcement
Terms and conditions apply

Text reads: "Special offer! 5 for the price of 1 (ish): Strong exception guarantee, Concurrent GC write barrier, Reversible execution, Side effect confinement, Integrity enforcement. Terms and conditions apply."

Implementing rollback

- Randell *et al*'s 1974 paper describes a possible implementation of rollback, using a *recursive cache* to record the values of words that have been modified. Experiments with this approach in a ***Pascal*** system in 1978 gave a measured overhead of between 1% and 11% [Randell 1997].
- This may not accurately reflect what the overhead would be in a modern language implementation that optimises the rest of the code more aggressively. I'd like to suggest an alternative implementation that should reduce the overhead further. (This is only one possible technique; another that would work is the copy-on-write approach used in [AGV 2011].)
- *Trailing* is a technique used in many ***Prolog*** implementations to implement backtracking search. The earliest published description of trailing I can find is [Warren & Pereira 1977], describing the "old ***Prolog*** Engine" (later versions of which became known as ***Warren's Abstract Machine***).
- Rollback is sufficiently similar to backtracking that we can use almost the same technique. (In trailing we only undo bindings; for rollback we need to restore old values.)
  - The similarity to backtracking was already pointed out in [Randell 1975]. As he notes, recovery points are likely to be less frequent than are *choice points* in backtracking.
- In typical programs, writes are less common than reads. [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.

Implementing rollback

- 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.
- We keep an *undo stack* of (location, old value) pairs recorded by the write barrier.
- To roll back, we restore old values from the undo stack in reverse order.
- For each location, the undo 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.
  - This is the main difference between this implementation and the recursive cache used by Randell *et al*. Their write barrier was conditional and required tracking which words had been modified since the last entry to a recovery block.
  - Unconditional barriers are faster, and in practice we will be able to remove most barriers by static analysis.

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.
- Since vats do not share mutable state, we can record rollback information independently for each vat.
- Typically each event-loop turn is triggered by processing of an incoming message, initially with an empty stack. Therefore, we need only be able to roll back to the beginning of the last turn boundary in order to maintain the strong exception guarantee.
- Fairly short event-loop turns are desirable 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 rollback

- We have two main 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.
- Although it may be possible to elide write barriers based on run-time tests, the cost of the test offsets the saving, 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 recovery points.
  - Writing to a location only requires an undo entry if the location existed at the last recovery point, and only for the first write after a recovery point.
  - We can hoist barriers above loops that contain no recovery 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 location that it writes to, we can change all callers to trail that location 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.)

Optimising memory usage

- Writes to locations that no longer exist can be discarded.
- If the previous value at a location is known to be null, we can store just a tagged location rather than also storing the null.
- We should make sure that the undo stack reuses the same memory between turn boundaries to avoid cache pollution (similar argument to [Gonçalves and Appel 1995]).
- We can “compress” the undo stack (possibly during GC) to remove redundant entries.

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 beneficial. 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 locations are on the trail stack!

Reversible execution and debugging

- If we can roll back to any recovery point in normal execution, we can also do so when debugging.
- The earliest reference I can find to this idea is a 1973 CACM paper by Marvin Zelkovitz. That paper also notes the connection to backtracking. It referenced an implementation for ***PL/I*** using a write barrier, with an overall time overhead of roughly 70% and a space overhead of 30-60%.
    - This compares to gdb's recent implementation of reverse debugging which is thousands of times slower (apparently because it uses single-stepping?)
- We can go back further than the last turn boundary if we take checkpoints, or if we keep the trail stack and the final state.
- We can also roll forward from any recovery point. To follow the same path as the original execution, we need to have logged all inputs to the vat (i.e. incoming messages and sources of nondeterminism).
- We can potentially reify checkpoints and replay logs and expose them to applications.

Confining side effects

- Some languages have type systems and/or effect systems that can enforce that some code does not have side effects (such as mutation or sending messages).
- However, what if we are not in such a language, *or* we want to confine the effects of code that doesn't satisfy these constraints?
- For example: at the Future of Programming Workshop at the pre-conference there was a fantastic presentation by Joel Galenson on CodeHint, which does a backtracking search for code satisfying given criteria.
- In order to do this it needs to actually execute the code, but it also has to undo any side effects. On the JVM this incurs significant overhead. With support from the language implementation, we can implement this much more efficiently.
  - Caveat: in order to prune duplicate states from the search, the undo stack isn't quite sufficient and this application may need a write barrier with slightly heavier overhead. However, most of the optimisations we've described are still applicable.
- Side effect confinement can be used as part of a more general capability-based confinement mechanism.

Integrity enforcement

- Suppose that we have defined invariants for certain abstractions, but we consider checking the invariants too expensive to repeat for every new object or every mutation.
    - Note that invariants can span large structures of objects, which is one reason why they may be expensive to compute.
- Instead, we want to check integrity only of object structures that become part of the persistent state after an event-loop turn.
- The undo stack tells us which objects those are (that is, which objects have changed or been created, and are *potentially* live).
- Therefore, we can enumerate those objects, check their invariants, and roll back the turn if this checking fails.
- We can also replay the turn with more precise and expensive invariant checking turned on, and signal an error from precisely the point at which an invariant fails.
- If it fails within a recovery block, and the alternate implementation of the block is correct, this could allow the error to be corrected – and then the program can resume at full speed.
- This approach makes orthogonal persistence more attractive, since we can be more confident that the long-term persistent state will not become corrupted.
- There are some similarities with work on *GC assertions* and *asynchronous assertions*, and it may be useful to combine these ideas.

Distributed error recovery

- Clarification: “distributed” does not necessarily mean across a network; communicating nodes can be on the same machine. In a vat-based event loop model, we can treat any subset of vats as a “node”.
- 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.
- Implementations:
  - ***Waterken*** for ***Java/Joe-E*** programs, developed by Tyler Close
  - ***CKen*** for ***C*** programs
  - ***MaceKen***, integrated with the ***Mace*** distributed-systems toolkit by Hyoun Kyu Cho
  - ***SchemeKen*** for ***Scheme***
  - ***v8-ken***, for Google’s ***v8*** implementation of ***Javascript***.
- 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 (crash-restart) 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 rules below.
- Each node has a state that is checkpointed reasonably frequently, and independently of other nodes. The state includes incoming and outgoing messages.
- Checkpoints are written durably to the node's persistent storage.
- 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.)
- Outgoing messages are *released* in the same atomic action as checkpointing.
  - We need this anyway for local error recovery. We are incurring a latency cost for delaying release of outgoing messages until a turn is committed, but we are paying this cost only once for local error recovery and for ***Ken***.
  - We need to checkpoint every time we release messages to another node, but not necessarily at every turn.
- Released messages are periodically *retransmitted* until they are *acknowledged*. They can be deleted from the state only after they are acknowledged. Duplicate acknowledgements are discarded.
- A received message is not acknowledged until after it has been recorded in a checkpoint.
  - ([HPL-2010-155] says acknowledgements should be delayed until the message has been *processed*. This isn't necessary for output validity if applications do not see acknowledgements.)
- Received messages that a node has already recorded are *reacknowledged* and then discarded.
- It may also be desirable to enforce message ordering guarantees such as pairwise FIFO ordering.
- 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 smashed promises and sturdy/live ref distinction
- 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.
- This will eliminate nondeterministic failures where a detected error happened in the same checkpoint period as its nondeterministic cause. All remaining failures will be deterministic or at least easily reproducible, and therefore will be easier to debug!
- Of course this means that output validity does not apply to deterministic errors.

Broken promises

- A *promise* is a reference to a value that need not be available yet. The promise can be used as a first-class value, even before it is *resolved*.
  - It differs from a lazy reference in that its value can be resolved asynchronously. *Forcing* a promise can be implicit (typically blocking), or explicit (either blocking or non-blocking).
  - Sometimes the capability to read the promise's value and the capability to *resolve* it are separated.
  - Promises are becoming increasingly popular in mainstream languages, especially those using message passing concurrency or event loops.
- Error handling for promises in the ***E*** language:
  - A possible state for a promise is *broken* or *smashed*. This is essentially an error value, but it avoids some of the problems of error values, because the operation of waiting for a promise to resolve is explicit, using a when/catch construct.
  - When a node is partitioned from another node, any promises it has to objects on the inaccessible node become broken.
  - This ensures that we never have a case where a message is not received, and then another message which was assumed to depend on it *is* received. (Other conditions on message ordering, even in the absence of errors, are also necessary to achieve this.)
  - *Sturdy refs* are references that can survive a partition. Messages cannot be sent directly to sturdy refs. A *live ref* can be constructed explicitly from a sturdy ref, allowing recovery after a partition.

Thanks

- This talk directly mentions ideas and systems due to John B. Goodenough, Brian Randell, Jim Horning, Hugh C. Lauer, P. Michael Melliar-Smith, David H. D. Warren, Marvin Zelkovitz, Edsger Dijkstra, Taiichi Yuasa, Tyler Close, Alan Karp, Marc Stiegler, Terence Kelly, Hyoun Kyu Cho, Sunghwan Yoo, Charles Killian, Steven Plite, Joel Galenson, Kevin Reid, Carl Hewitt, and Mark S. Miller.
- Special thanks to Nathan Wilcox, Brian Warner, Annie Ogborn, Kevin Reid, the Lambda Ladies, and the friam group 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.
- Last but never least, thanks to my partners Allie, Aura and Samantha.