Noether: Symmetry in Programming Language Design

- What problems am I trying to solve with this language?
- Symmetry
- Expressiveness
- Coherence
- The sublanguages, in order of decreasing symmetry
- Along the way we'll talk briefly about how to implement transactions efficiently.
- Bonus slides:
    - Rejected features

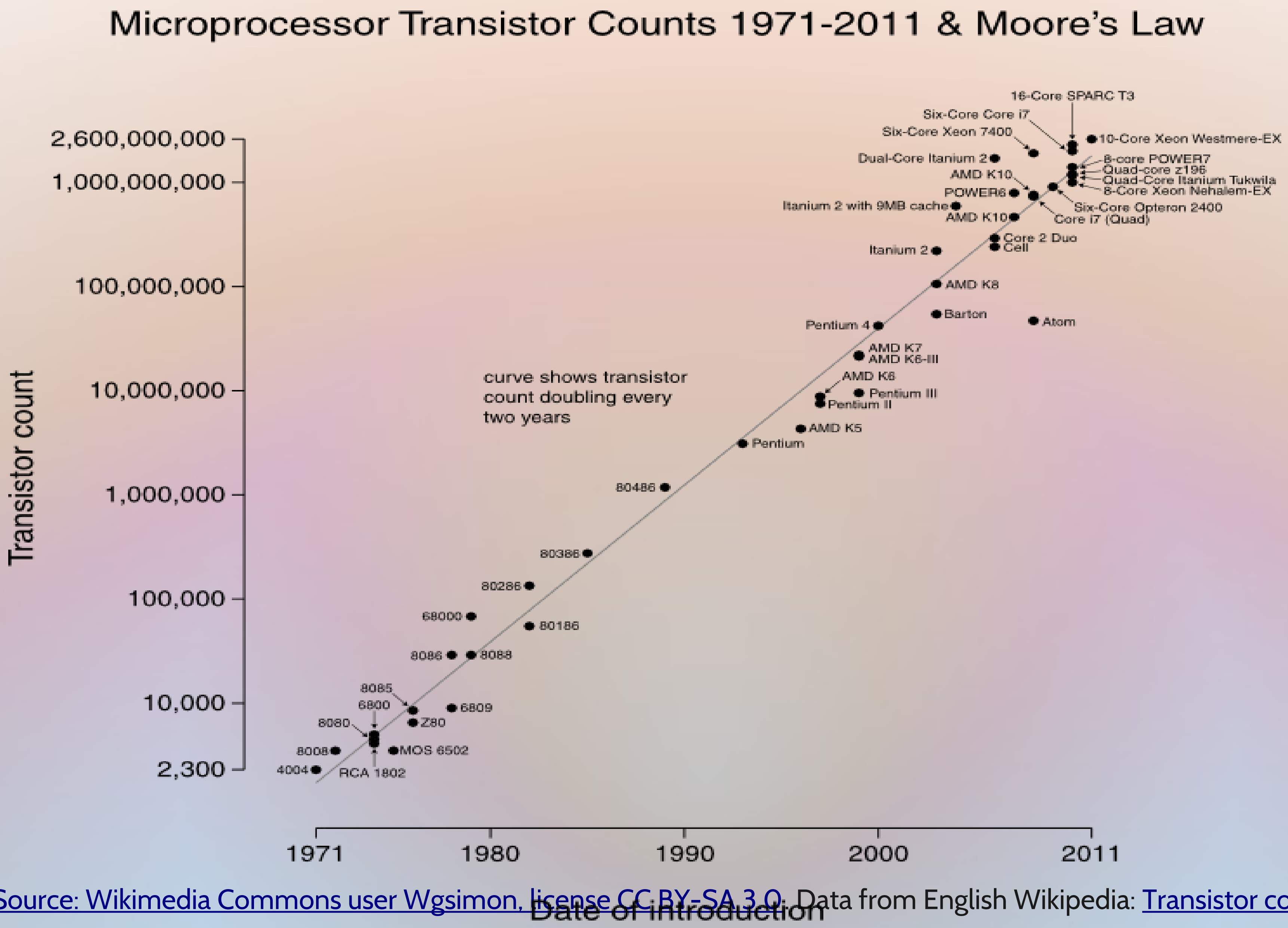
How to program gigantic computers?

"… as the power of available machines grew by a factor of more than a thousand, society's ambition to apply these machines grew in proportion, and it was the poor programmer who found [their] job in this exploded field of tension between ends and means. The increased power of the hardware, together with the perhaps even more dramatic increase in its reliability, made solutions feasible that the programmer had not dared to dream about a few years before. And now, a few years later, [they] *had* to dream about them and, even worse, [they] had to transform such dreams into reality! Is it a wonder that we found ourselves in a software crisis?"

– Edsger W. Dijkstra,

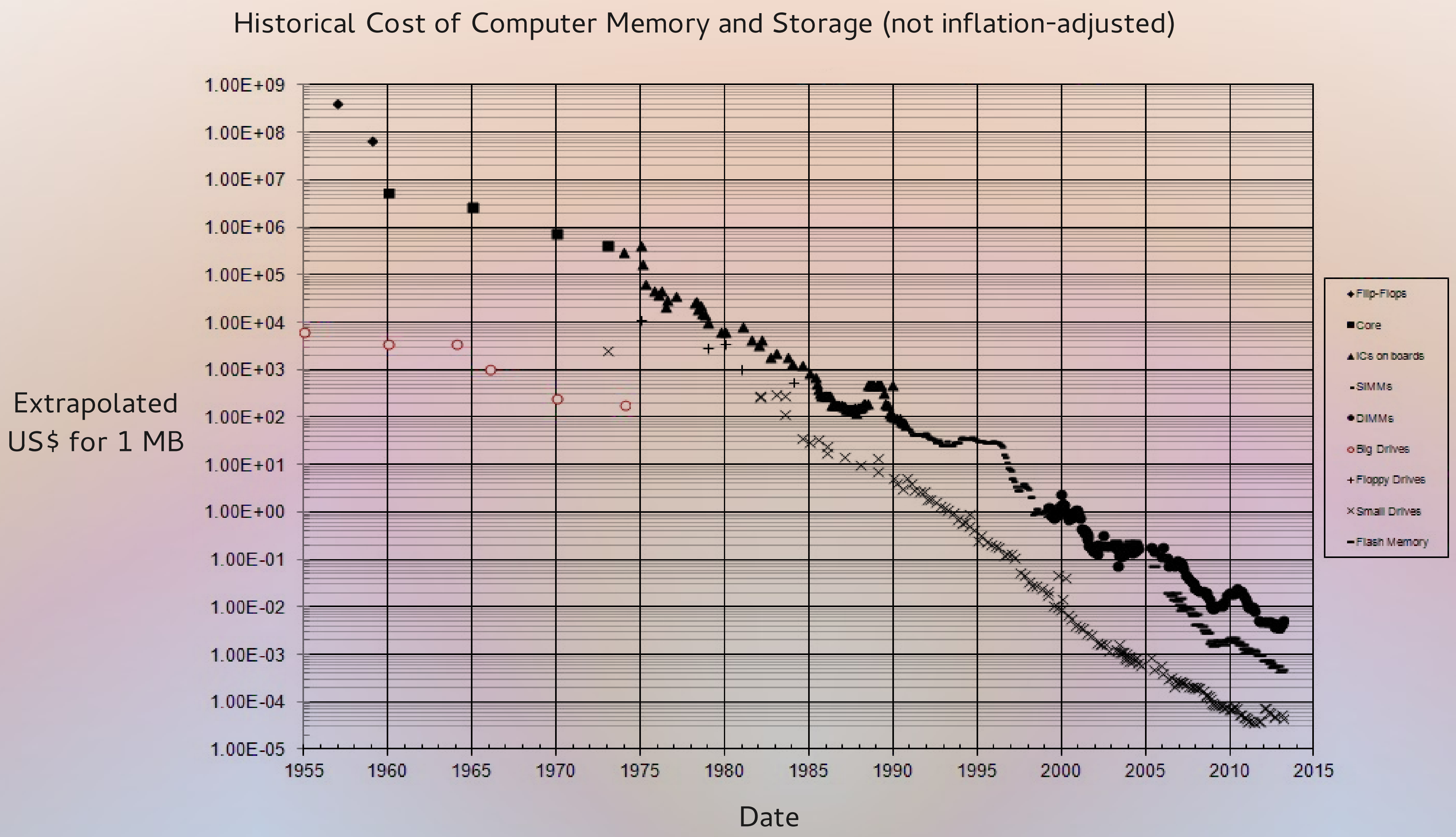
The Humble Programmer (EWD340), ACM Turing Lecture 1972

How to program gigantic computers?



[Source: Wikimedia Commons user Wgsimon, license CC BY-SA 3.0.] Data from English Wikipedia: [Transistor count].

How to program gigantic computers?



Source: jcmit.com, © 2001, 2013 John C. McCallum ("You can use the material here. Please acknowledge the source.")

The problem

- Languages and tools have improved greatly over that time, but have they improved enough?
- The "Software Crisis" is still here:
    - Programs are too large, complex, and difficult to maintain.
    - No-one knows how to write correct, secure, or reliable programs.
    - No-one knows how to review a given program for correctness, security, and reliability, with economical effort.
- Correctness, security, and reliability are three sides of the same coin.
- Some techniques, e.g. pure functional programming or (in different ways) object programming, seem to help.
- How do they help?

Symmetry

- Thesis: many programming language techniques/disciplines help by imposing symmetries, which aid reasoning about programs.
- A *symmetry* gives a set of possible transformations that do not change a given property.
- In Geometry: translation (e.g. of an infinite lattice); rotation; reflection.

  In Physics: time- and location-invariance in (e.g.) general relativity; CPT symmetry.

  In Programming (all of these are language-dependent):
    - Confluence: in a pure language, evaluating in different orders produces the same result.
    - Moving a call of a pure function to a different time, or task, does not change its result.
    - Reflexive/symmetric/associative operators and functions.
    - Alpha renaming.
    - Adding and removing parallelism annotations; maybe other annotations.
    - Expansion and abstraction of {functions, macros, modules, type synonyms, ...}.
    - Common subexpression elimination.
    - Adding and removing staging.
    - Trait flattening.
    - Representation changes.
    - Adding and removing dead code.
    - Comments and formatting.

Interfering features

- Many programming language features/properties interfere with symmetries we'd like to have.
- Some are essential features:
  - Side effects; local/task state
  - Failure handling
  - Concurrency
- Some are not:
  - Dynamic scoping
  - Overloading
  - Implicit coercion
  - Introspection (stack, local variables, function code, objects, ...)
  - "Unhygienic" or textual macros
  - Anything that violates memory safety
  - Anything that violates Actor locality
  - Colliding namespaces
  - Global state; modes (e.g. floating point mode/state)
- A design wart in a feature (e.g. module or macro system) can stop it having desired symmetries.
- With care, similar or better *expressiveness* can be achieved without problematic features, or with restricted variations on them... but what do we do about the essential ones?

Symmetry “breaking”

- We want the strongest symmetries possible for any given expressiveness.
- Solution: stratified languages.
- Not a new idea (e.g. ***Erlang***, ***Oz***, ***Haskell***), but ***Noether*** takes it further.
- At each level, we add one or more features and break a symmetry.
- Some symmetries are so useful that they should be preserved globally (for example: memory safety and Actor locality laws).
- There's a trade-off between optimizing the symmetry-expressiveness profile, and keeping the number of levels small to reduce complexity.
- If we start by designing with one level per broken symmetry, we can always collapse levels together later.
- The sublanguages don't *have* to be nested, but we make them nested:
  - for simplicity of understanding and teaching the full language;
  - for simplicity of inferring which sublanguage we're in and conveying that to the programmer;
  - to make the design problem of which features depend on others tractable.

  We could relax this constraint if it was important, but haven't needed to for ***Noether***.

Coherence

- Getting strong symmetries when a whole program is in a small sublanguage isn't enough.
- Nontrivial programs are intended to be a mixture of subprograms at different levels. We need to know whether symmetry properties are maintained when mixing levels.
- Another physical analogy:
  - Sublanguage with strong symmetries ⇔ highly ordered phase of matter, e.g. a solid.
  - Sublanguage with weaker symmetries ⇔ less ordered phase of matter, e.g. a fluid.
  - A material is coherent if it maintains its phase when moved within a less ordered phase.
    - Example: most solids maintain their phase and shape when moved in a fluid...
    - ... unless, say, the fluid corrodes the solid.
  - A sublanguage is coherent if code in that sublanguage maintains its symmetry properties when used anywhere within a larger language.
    - Example: a pure function can be executed anywhere in an impure program without losing symmetry (confluence within the function, determinism, etc.)...
    - ... unless, say, the impure language has reflective features that break opaqueness of pure functions.

Noether programming language

- Object-capability, typed, parallel and concurrent, potentially highly optimizable.
- Stratified sublanguages of increasing expressiveness and decreasing symmetry.
- This talk considers mainly semantics rather than syntax.
- Why a new language?
  - fewer constraints on optimizing symmetry–expressiveness profile.
  - existing languages have features (e.g. exception handling, concurrency) that we want to specify and combine in different ways.
  - address previously unsolved or solved-only-as-research problems: precise resource control, partial failure, flow control in asynchronous message passing, etc.
  - security + expressiveness + speed is already very hard; adding a requirement for compatibility with some existing language would make it impossible. (The problem is running existing programs *with expected performance*.)
  - compatibility is overrated; familiarity is more important.
  - "a language really defines its own universe, and that a language designer has the power to control what can or cannot happen within that universe." – Tom Van Cutsem
  - *Programming language design is fun!*

Emmy Noether

- Mathematician and theoretical physicist, Jewish, expelled from Nazi Germany in 1933.
- Einstein in 1935: "In the judgement of the most competent living mathematicians, Fräulein Noether was the most significant creative mathematical genius thus far produced since the higher education of women began. In the realm of algebra, in which the most gifted mathematicians have been busy for centuries, she discovered methods which have proved of importance in the development of the present-day younger generations of mathematicians. Pure mathematics is, in its way, the poetry of logical ideas. One seeks the most general ideas of operation which will bring together in simple, logical and unified form the largest possible circle of formal relations. In this effort toward logical beauty spiritual formulas are discovered necessary for the deeper presentation into the laws of nature."
- Appropriate for ***Noether-0*** because it can be viewed as a Noetherian (terminating) rewriting system. Some of Emmy Noether's other work on abstract algebra is indirectly relevant, and we can use analogies with physical symmetry properties to explain the language.
- Hard to believe this name hasn't already been snapped up for a programming language! (Google says not)

Symmetry properties

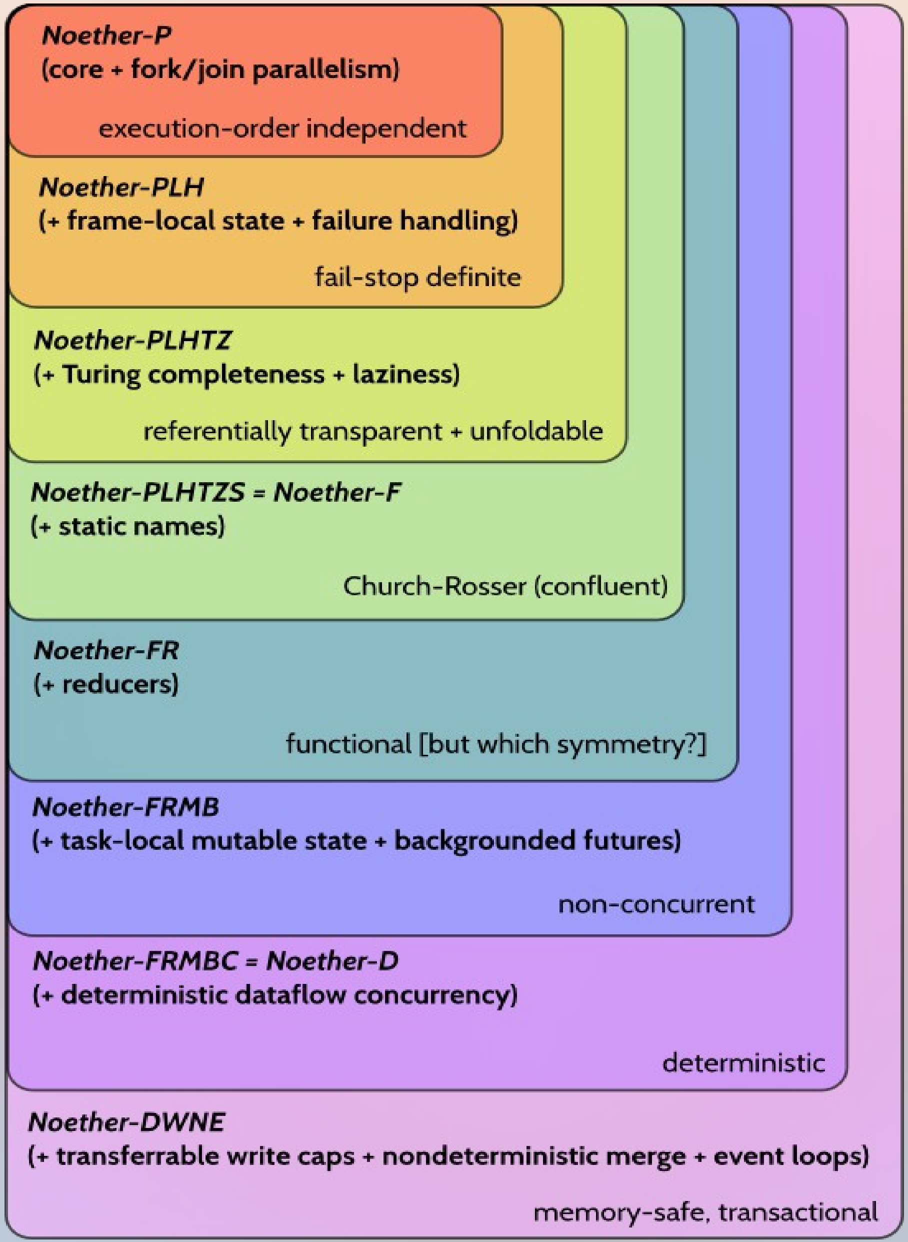
- Because ***Noether*** is in general nondeterministic , we need to be particularly careful in how we define properties such as referential transparency, confluence, etc.
- We model evaluation as a rewriting system ($\mathcal{A}$, $\rhd$), where $\mathcal{A}$ is a set of terms and $\rhd$ is an irreflexive binary relation on $\mathcal{A}$.
- For ***Noether***, let $\mathcal{A} = \mathcal{R} \cup \mathcal{F} \cup \mathcal{S} \cup \{\perp\}$, where $\mathcal{R}$ is the set of reducible terms (i.e. that can be rewritten by $\rhd$), $\mathcal{F}$ is the set of failure terms, $\mathcal{S}$ the set of success terms, and $\perp$ a special term denoting nontermination. Each failure term is associated with a reason which is always a success term.
- A ***Noether*** term other than $\perp$ can be represented as a program fragment and a "store", which maps locations to other program fragments.
- Notation:
  - T is terminating under $\rhd$, written T *stops*, iff there are no infinite $\rhd$-chains starting from T. When this is true for all T in $\mathcal{A}$, ($\mathcal{A}$, $\rhd$) is noetherian.
  - $\rightarrow \;\overset{\text{def}}{=}\; \trianglerighteq^*$
  - $\leftrightarrow \;\overset{\text{def}}{=}\; (\trianglelefteq \cup \trianglerighteq)^*$
  - $T \twoheadrightarrow U \;\overset{\text{def}}{=}\; T \rightarrow U \wedge U \notin \mathcal{R}$ [U is a *normal form* of T]
  - $T\downarrow \;\overset{\text{def}}{=}\; \{ U : T \twoheadrightarrow U \}$, if T *stops*
  - $\overset{\text{def}}{=}\; \{ U : T \twoheadrightarrow U \} \cup \{\perp\}$, otherwise
  - T *fails* $\overset{\text{def}}{=}\; T\downarrow \subseteq \mathcal{F}$
  - $S \approxeq T \;\overset{\text{def}}{=}\; S\downarrow = T\downarrow$
  - $T \rightarrowtail U \;\overset{\text{def}}{=}\; T\downarrow \setminus \mathcal{F} = \{ U \}$

Symmetry properties

- These symmetries always hold:
  - Adding unused bindings to a store
  - Alpha-renaming of locations
  - Alpha-renaming of bound variables
- Other definitions relating to pure functional languages:
  - *Church-Rosser*: $S \leftrightarrow T \;\Rightarrow\; \exists U \, (S \rightarrow U \;\wedge\; T \rightarrow U)$
  - *Fail-stop deterministic*: $|\, T{\downarrow} \;\setminus\; \mathcal{F} \,| \;\leq\; 1$
  - *Fail-stop definite*: $T \textit{ stops} \;\wedge\; |\, T{\downarrow} \;\setminus\; \mathcal{F} \,| \;\leq\; 1$
  - Strachey's definition of referential transparency: "if we wish to find the value of an expression which contains a sub-expression, the only thing we need to know about the sub-expression is its value."
  - This is a bit unsatisfactory; see [Søndergaard and Sestoft 1990] for details.
  - *Referential transparency*: $S \approxeq T \;\vdash\; V[S] \approxeq V[T]$

    (if two expressions are equivalent, substituting them at the same point in a containing expression preserves equivalence).
  - *Unfoldability*: $\mathrm{def}\; f\,(\,x\,) = V\,[\mathrm{x}] \;\vdash\; S \approxeq T \;\Rightarrow\; \mathrm{f}\,(\,S\,) \approxeq V\,[T]$
  - ***Noether-0*** is Church-Rosser, fail-stop definite, referentially transparent, and unfoldable.

Noether sublanguages and their symmetries


Noether-P
(core + fork/join parallelism)
execution-order independent
Noether-PLH
(+ frame-local state + failure handling)
fail-stop definite
Noether-PLHTZ
(+ Turing completeness + laziness)
referentially transparent + unfoldable
Noether-PLHTZS = *Noether-F*
(+ static names)
Church-Rosser (confluent)
Noether-FR
(+ reducers)
functional [but which symmetry?]
Noether-FRMB
(+ task-local mutable state + backgrounded futures)
non-concurrent
Noether-FRMBC = *Noether-D*
(+ deterministic dataflow concurrency)
deterministic
Noether-DWNE
(+ transferrable write caps + nondeterministic merge + event loops)
memory-safe, transactional

Loopholes in “Safe” Languages

- Causes ***OCaml*** runtime to dump core (tested with `ocamlopt`):
  - let *x* = Marshal.to_string [1;2;3] [];;
  - let *y* = Marshal.from_string x 0;;
  - # val y : 'a = <poly>
  - let *z1* = (y : in_channel);;
  - let *z2* = input_line z1;;
- Causes ***Glasgow Haskell*** runtime to dump core (tested with GHC version 7.4.1):

  ```
  import System.IO.Unsafe (unsafePerformIO)
  import Data.IORef (IORef, newIORef, writeIORef, readIORef)
  test = unsafePerformIO $ newIORef []
  main = do { writeIORef test [42] ;  bang <- readIORef test :: [() -> IO ()] ;  (head bang) () }
  ```

- The underlying problem (as opposed to the immediate cause of unsafety) is lack of expressiveness:
  - Marshall is memory-unsafe because ***OCaml*** lacks a Dynamic type that could be used to write a typesafe unmarshalling function. [details]
  - unsafePerformIO is memory-unsafe because ***Haskell*** lacks a safe way to bypass the IO monad.

Symmetries “keep us honest”

- For example, we intend to have no implicit coercions in ***Noether***.
- But, it might not always be obvious that some behaviour amounts to an implicit coercion.
- If it does, then it will break equational reasoning in sublanguages that need it:
  - Example from the slide Taut and Loose types:

    “In order to preserve equational reasoning, loosening conversions must explicitly specify the wrapper type.

    Δ(•) without a specified type would not be a function, since

    obj:S = obj:T $\not\Rightarrow$ Δ(obj:S) = Δ(obj:T).”

    In this example, Δ is not implicit in the alternative design, but the type was

    (Δ being implicit was not even considered.)
- This means we have to make a choice in the design between:
  - Explicitly breaking the symmetry for sublanguages that allow that behaviour, or
  - Avoiding the behaviour in *all* sublanguages
- Leaving the behaviour in without acknowleging the symmetry break is not an option.

More Subtle Pitfalls

- Many bugs are due to two kinds of surprises:
  - success with a "wrong answer"
  - inconsistent state after a failure.
- Example in ***ECMAScript*** (due to Mark Miller; recently fixed):
  - Nat(a); Nat(b); Nat(a + b); assert(Nat(a) + Nat(b) === Nat(a + b))
  - But $(\mathrm{a} \twoheadrightarrow a \wedge \mathrm{b} \twoheadrightarrow b \wedge \mathrm{a} + \mathrm{b} \twoheadrightarrow c) \nRightarrow a + b = c$ (when $c = 2^{53} + 1$ for example*)*
  - Led to a bug in the showcase "Mint" example
- Underlying problem is divergence from intuition:
  - In ***Secure ECMAScript***, Nat is supposed to be the subset of numbers on which exact arithmetic is possible, but + is not fail-stop exact, so it satisfies *fewer symmetries than expected*.
  - The question of what symmetries are expected is messy and subjective, but ignoring it leads to security and correctness bugs.***Noether*** addresses these by:
  - as far as possible, not having operations that succeed with a wrong answer (e.g. integer arithmetic is fail-stop exact by default, and there are no implicit conversions)
  - supporting programming without state
  - pervasive transactionality: inconsistent states are rolled back on failure

“But it’s too expensive to get the right answer!”

- Failure is an alternative.
- It really isn’t too expensive to either get the right answer or fail.
- Pervasive transactionality isn’t too expensive either.
- We should start taking computing seriously.

“The first principle [of the ***Elliott Algol*** implementation, *circa* 1962] was security: The principle that every syntactically incorrect program should be rejected by the compiler and that every syntactically correct program should give a result or an error message that was predictable and comprehensible in terms of the source language program itself. Thus no core dumps should ever be necessary. [...]

I note with fear and horror that even in 1980, language designers and users have not learned this lesson. In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law.”

– C. A. Hoare,

The Emperor’s Old Clothes, ACM Turing Lecture 1980

Noether-0

- Strict, deterministic, pure functional core:
  - all computations are definite (succeed with a deterministic result or fail in finite time)
  - therefore not Turing-complete, but more powerful than primitive-recursion (proof: can express the Péter/Ackermann function)
  - inspired by Total Functional Programming [Turner], but functions need not be total
  - no mutable data
  - no cyclic data (this is a consequence of strictness and immutability)
- We allow failures, but no failure handling within the sublanguage:
- This is more realistic than claiming to prohibit failure – a nontrivial programming language semantics that doesn't allow failure cannot be implemented on any real computer.
  - Failures can be due to insufficient memory, termination from outside, detected but uncorrected hardware errors, assertion and variant checks, etc.
  - Programs can use unproven assertion and variant checks at run-time.
  - no need for dependent types (as in ***Epigram***).
  - no artificial definitions to make functions total (as in Turner's paper).

Noether-0 symmetries

- ***Noether-0*** has many symmetries associated with equational reasoning.
- An "outcome" of executing a program fragment is a result value or a failure.
- A program fragment is *pure* iff it has no side effects, and only one possible outcome for given values of its inputs.
- A program fragment is *deterministic* iff its outcome and side effects are dependent only on its input state.
- Time/location invariance:
    - Informally, if you move a fragment to a different "place" in a program, it has the same outcome.
    - This also applies if you move ***Noether-0*** fragments across sparks or tasks in larger sublanguages.
-

Noether-0

- Lack of divergence or mutable state allows equational reasoning: *in a successful computation*, all definitions are equations.
  - This would not be true if we had exception handling or mutable state.
- Definiteness is necessary as shown by this example:

  ```
  loop : int → int
  loop x = loop x + 1
  ```

  subtracting loop x from both sides (well-typed since loop returns int) gives:

  $0 = 1$
- Overall success is necessary as shown by:

  ```
  bomb : int → int
  bomb x = bomb (x – 1), if x > 0
  ```

  subtracting bomb (x – 1) from both sides (bomb returns int) and substituting $x = 1$ gives:

  $\text{bomb } 1 - \text{bomb } 0 = 0$

  But actually the left-hand side fails, it does not evaluate to 0.

Noether-0

- So what have we gained by enforcing definiteness without enforcing success?
- We can always substitute equivalent expressions whenever the result is a valid program. So if $e_1 \approx e_2$, we can substitute e_2 in place of e_1 (even to substitute e_2 into the definition of *f* which is called by e_2).
- Reasoning as though failure does not occur in any subcomputation gives a property like partial correctness, but replacing possibility of nontermination with possibility of failure.
- Failure is more useful than nontermination, since we can reliably recover from it (*outside* the sublanguage); we can't recover from arbitrary nontermination bugs without using timeouts.
  - Timeouts introduce false positives and nondeterminism, and are slow to recover even when they work as intended.
- Absence of algorithmic failures (as opposed to hardware failures or being killed) in any subcomputation can then be proven separately and modularly.
- Termination proving can be completely automatic, even for mutually recursive functions (just apply existing research; termination provers got really good and no-one noticed).
- But what about bugs that cause excessive run-time?
  - Using a definite sublanguage is not a subsitute for complexity analysis; definite functions can take a *long* time even for "small" inputs (e.g. Péter function).
  - Conjecture: writing functions to be *provably* definite helps to make the parameters on which their algorithmic complexity depends more explicit, and reduces the scope for mistakes that would cause higher-than-intended complexity.
  - Later we'll see how to confine a computation to limit its CPU and memory usage.

Noether-P

(by the way, these sublanguage names are provisional)

- Adds fork/join parallelism, but with *sequential semantics*.
- Deleting any parallel annotation is a symmetry; deleting all parallel annotations gives an equivalent sequential ***Noether-0*** program.
- spawn, parallel for, sync constructs as in ***Cilk***. (Will add "reducers" in a larger sublanguage.)
- Works because different evaluation orders produce the same results (confluence symmetry).
  - even when we add state, because the state is required to be separated.
- Read the Cilk / Cilk Plus papers for details.
- Advantages of fork-join parallelism:
  - It adapts to the available hardware parallelism with very little overhead. Existing implementations have a useful speed-up in practice with just two processor cores (unlike STM), and it scales well to at least hundreds of cores given sufficient "parallel slack" in the program.
  - Serial execution is a valid implementation of the parallel semantics – this allows sequential debugging for example.
  - Space consumption is bounded to N × serial on N cores.
- Improvements over ***Cilk***:
  - no possibility of race conditions.
  - therefore, parallel semantics are *identical* to serial semantics, and equally tractable.
  - larger sublanguages of ***Noether*** will provide true concurrency, so restrictions are less burdensome.

Noether-P (continued)

- Fork/join parallelism is not concurrency: there is no way to write programs that depend on concurrent operations, since serial execution is a valid implementation.
- The parallel computations are called "sparks".
- Concurrency is not *necessary* for parallelism. Sometimes it helps tractability (*cf* Observer example in MarkM's thesis), but *concurrency just to enable parallelism* hinders tractability.
- Prediction: ***Future hardware will be capable of more parallelism than can be exploited via the "natural concurrency" of most programs.***
- If the semantics are the same, why does the compiler need the annotations?
  - It doesn't for pure functions (that do not accept or close over mutable state). We need experience with an actual implementation to see whether inferring parallelism in that case is a good idea.
  - When we add state, there will be restrictions on it that depend on the annotations, so the programmer must choose them.
  - Explicit annotations give a more understandable performance model.
  - We don't want to have to implement a Sufficiently Smart Compiler.
  - There is no need for this to be in the compiler; annotations could be added by an external refactoring tool.

Noether-PL

- Adds *frame-local* mutable state, but no references-as-values.
- Mutable variables/structures are not first-class, they can only be referenced within the function that declares them.
    - No possibility of aliasing, no "spooky effects at a distance."
    - No race conditions between parallel spawned function invocations, because they cannot access the parent's state.
- A ***Noether-PL*** function that uses frame-local state can be locally rewritten to a pure function. So we can consider this to still be a pure functional computation model.
- However, a pure-functional rewriting will allocate more space (absent heroic optimisation efforts), so state is not just a convenience.
- There is some loss of tractability for functions that use frame-local state. Simple equational reasoning fails for expressions directly involving mutable variables, but can be restored by locally rewriting the program to be functional. Uses of mutable variables and their exact types are statically apparent.
- In a function that uses state, statements are evaluated in program order. That may sound obvious, but in ***Noether-P*** the evaluation order was unobservable, and in ***Noether-PL*** it is still unobservable within a statement.
- Mild violation of Tennent-Landin correspondence principle because moving a disjoint mutable variable reference into a lambda isn't valid. (The variable could be promoted to a full mutable variable, but that correspondence only holds in a larger subset.)

Noether-PLH

- Adds *transactional* handling of *deterministic, explicit* failures, using an escape construct.
- Now we can distinguish reasons for failure. A reason is represented as a pure value (also in the full language).
- Handling a failure rolls back state to the escape point.
    - In ***Noether-PLH*** this can only apply to frame-local state, so information needed to roll back changes can be discarded as mutable structures go out of scope. When we add task-local state, that state will also be rolled back; later we'll discuss implementation and efficiency consequences of that.
    - Any information that needs to be extracted from the failing context must be in the reason. This will be unfamiliar to programmers, but I expect it to be easy to get used to.
- We have the "strong exception guarantee" as a guaranteed property of the language for *every* function, rather than something that programmers have to document and ensure.
- Symmetry/security property: ***intermediate states are not observable on failure***. This holds in the full language, and should make writing defensively consistent code *much* easier.
- ***Noether-PLH*** is still fail-stop definite, like ***Noether-0***.
- Failure handling increases Felleisen-expressiveness, since transforming failures into error returns, with added code for propagation, would be a non-local transformation.
- But, reasoning gets more complicated than in ***Noether-PL***, because we need to take into account that failures raised from any function can arbitrarily affect the course of the computation.
- Failure handling makes details of evaluation order observable, by making visible *which* failure occurs when there could be more than one. We choose left-to-right call-by-value rather than making evaluation order unspecified as in ***Haskell***, because we want failure handling (and full mutable state) to be available in a deterministic sublanguage. We choose CBV for the efficiency reasons described in AIM 349 §4 pp 24–26 [Sussman and Steele].

Noether-PLH (continued)

- However, the DarpaBrowser security review found that traditional exception handling results in information leakage outside explicit capability channels [Mark Miller].
- Kevin Reid [#erights freenode 2012-08-18 02:15Z]: "one level's explanation of the caller's incorrect arguments is the next level's private data leak." In other words, it is wrong to attempt to classify reasons as sensitive or non-sensitive, or to rely on avoiding sensitive information in reasons.
- We cannot use the "ejector" solution adopted by ***E-on-CL*** unchanged: it is unclear how to reconcile non-local exits with rollback, and we need static typing of the reason value.
- Taking inspiration from ***Standard ML***'s (and ***OCaml***'s) exception mechanism, we make ejector types generative: each evaluation of 'escape' yields a new local type. This ensures only the right caller can unwrap the reason, and is necessary for type safety when the reason type is polymorphic.
- An ejection specifies the exception type and reason. It is not expressed as a call, since that would expose the current state to the callee before the rollback.
  - `escape E of T { … raise E of reason … } handle E of r { … r … }`
- The ejector type E is a second-class value that can be passed down to the function that needs to eject (but not stored or closed over). The 'escape' construct defines a handler for each ejector, which can unseal the reason value.
- Ejector types are generative existential types, which are well-understood.

Noether-PLH (continued)

- Ejector types are values, but cannot be first-class, because having first-class values with identity would break equational reasoning in ***Noether-PLH***:

  -
    ```
    def foo() {
       escape E of () { return E } handle E of r {}
    }
    ⊢  foo() != foo()  ↠  true  ↯
    ```

  To fix this, we make the ejectors second-class in this sublanguage, i.e. they can only be passed as arguments and used to raise failures, not returned, closed over, or used for general sealing. There are no values corresponding to sealed reasons, only unsealed ones.
- Allowing ejector types to be only closed over, not returned, would still break equational reasoning:

  -
    ```
    def make() {
       ejector E of ()
       def raiser() { raise E of () }
       def handler(f) { try { f() } handle E of r {} }
       return (raiser, handler)
    }
    def (raiser1, handler1) = make()
    def (raiser2, handler2) = make()
    ⊢  raiser1 ≋ raiser2  ∧  handler1(raiser1) ↠ ()  ∧  handler1(raiser2) fails  ↯
    ```
- In larger sublanguages that allow sealers/unsealers, it is possible to use an "ejectoid" sealer – possibly cryptographic in the remote case – in place of an ejector type. This follows Kevin Reid's proposal for ***E***. These objects are not permitted to close over mutable state in the task in which the failure occurred.

Noether-PLHT

- Adds Turing completeness.
- Arbitrary recursion and nonterminating loops are allowed.
  - Functions that are proven to terminate are identified by their type.
  - For functions that cannot be *proven* to terminate, the programmer should document whether they are *intended* to terminate, possibly under certain conditions on their inputs.
- Despite being Turing-complete, ***Noether-PLHT*** is still deterministic and noninteractive, like a standard Turing a-machine: the whole input of a program must be available in advance, and output is only available when and if the program terminates.
- Equational reasoning fails in general due to divergence, e.g.

```
loop : int → int
loop x = loop x + 1
```

does not imply

```
0 = 1
```

Noether-PLHTZ

- Adds lazy capabilities. In larger sublanguages capabilities will be used to model mutable state, references across tasks, etc., but here we only have caps for lazy futures.
- The lazy cap is a read-only view. The function that will compute its value is specified when it is created.
- The function is executed as-if in a new task that is initially suspended.
  - Even though we haven't added more general forms of concurrency yet, viewing lazy evaluation as being in a separate task is simpler and preserves more symmetries. (It ensures non-interference of future computations with the main computation and each other.)
- Note that implementing memoization within the language would require mutable state, which we do not have in this sublanguage (and for efficiency, this state should not be rolled back on failure).
- Forcing is explicit. Since it is syntactically apparent when forcing can occur, analysis of execution time is simpler, as argued in *Purely functional data structures [Okasaki]*. Alternative syntaxes could use implicit forcing without changing the semantics.
- If the lazy computation fails, then each forcing of the value fails with the same reason.
- If a lazy computation attempts to force a thunk that is being computed (a "black hole"), it fails.
- In ***Noether-PLH***, we had the same space bound property as in ***Cilk***, that an execution on N processors requires no more than N times the space of a serial execution. With lazy caps, we must take into account that the time at which a cap gets forced affects the program's space usage.
- Lazy caps can be used to represent effectively cyclic data structures. In smaller sublanguages, all data structures were acyclic. That property helps in proving termination, which is why we dropped the termination requirement "before" adding laziness.

Noether-PLHTZS = Noether-F

- Adds static names. A static name can be used to look up private properties on an object, providing a form of sibling communication between objects defined in the scope of the name declaration. Static names may also be used as security tokens, and to construct other forms of rights amplification.
- Equational reasoning is weakened, because instances of identical code fragments at different points in a program refer to observably different static names:

  -
    ```
    def make1() { name N; def t { to N() { return () } }; def n(o) { return o.N }; return (t, n) }
    def make2() { name N; def t { to N() { return () } }; def n(o) { return o.N }; return (t, n) }
    def (t1, n1) = make1()
    def (t2, n2) = make2()
    ⊢  make1 ≈ make2  ∧  t1 ≈ t2  ∧  n1(t1) ↠ ()  ∧  n1(t2) fails  ↯
    ```

  Notice the similarity of this example to the one for ejectors on a previous slide. The difference is that static names are *not* generative, so we need both make1 and make2 in order to force the contradiction. In an example with a single make, then there would be a single N, so n1(t2) would evaluate to (), which would not contradict equational reasoning.
- If all static names are uniquely α-renamed, full equational reasoning is restored (even though name values are first-class).
- ***Noether-PLHTZS*** is still a functional language by most definitions. Abbreviate it to ***Noether-F***.

Noether-F (continued)

- Static names imply rights amplification *without* identity or mutable state. This may be unfamiliar because traditional rights amplification mechanisms use abstractions such as sealers and unsealers that have identity. In ***Noether-F***, it is possible to define identity-free ("selfless") analogues of these abstractions, for example we can say:
  - name *N*; def (*sealer*, *unsealer*) = makeSealerUnsealerPair(N)

    ⊢ unsealer(sealer(obj)) ↠ obj

  where makeSealerUnsealerPair is a pure function, i.e. it always returns effectively the same sealer and unsealer derived from the single name N.
- As soon as we have first-class tokens, they can be used in place of static names, with the same implementations of rights amplification mechanisms, to give the traditional abstractions that have identity. This is useful since it makes it easy to parameterize library code to use either, and the library code itself remains pure-functional if it otherwise would be.
- Rights amplification, and sibling communication in particular, should be used with care, since it may enable confused deputy attacks. The preferred encapsulation mechanism in most cases is instance encapsulation provided by lexical scoping.

Reducers

- "Reducers" are another concept from ***Cilk***. They specify a semigroup with an associative operation that is used to combine a new value with the current state:
  - $\forall\, a, b, c \in T:\ (a \oplus b) \oplus c = a \oplus (b \oplus c)$
- In ***Cilk***, reducers specify a monoid, which is a semigroup with an identity. If there is a "natural" identity for a given reducer, it can be specified, but if there isn't, we can add one to any semigroup in order to form a monoid:
  - $T' \overset{\text{def}}{=} T \cup \{O\}$ where $\forall\, a \in T:\ a \oplus O = O \oplus a = a$
- A "free monoid" is a monoid over a sequence type, with $\oplus$ as concatenation and natural identity the empty sequence.
- Reducers effectively are a form of mutable state. The reason to distinguish them from other state is to support parallel execution. Because the reducer operation is associative, the runtime system can dynamically choose the granularity of parallel execution (whether each spawned function call executes on the same spark or a child spark) without affecting the result.
- Reducers are implemented, as in ***Cilk***, by splitting the state of each reducer when sparks fork, and combining states when sparks join.
- Each reducer has an update cap and a read cap. The update cap may be given to (i.e. closed over or passed to) spawned functions, but the read cap may not.

Noether-FR

- ***Noether-F*** supports laziness, which could in principle be used to model interactivity: give the program a lazy stream as input and have it produce one as output.
- However, we don't expect programs to be written that way because it's too clunky. Consider a purely functional program that just logs intermediate results for debugging. To express the log as the lazily computed result of the program would require turning its control flow inside out. We want something like a safe version of ***Haskell*** unsafePerformIO (which is not memory-safe).

  [Bad example because we support debug logging more simply. FIXME find a better example.]
- To do that we generalize futures, by allowing the value of a future to be the final output of a reduction. A special case is the free monoid which just appends all updates.
- The future computation is not functional in the sense of Sabry, because the output depends on left-to-right call-by-value evaluation order. However this is more tractable than general mutable state. Evaluation is independent of parallel execution.
- The output-so-far can't be read by the future computation, so doesn't affect it.
- By default the update operation blocks until the next value is needed, but that can be generalized.
- The unsafePerformIO type-safety and memory-safety problem doesn't occur, because we restrict polymorphic generalization to stateless values. (This is similar but less restrictive than the value restriction of ***Standard ML***.)

Noether-FRM

- This sublanguage adds task-local mutable state.
- Spawned function calls cannot pass general mutable caps as arguments, and the function cannot close over mutable state. (It *can* be given reducer update caps.)
- Mutable state is deceptively simple to describe, but has many interactions with other features, for example with failure handling.
- When an exception is caught, state is rolled back to the escape point. This creates a problem: when can we discard information needed to roll back?
    - Answer: we can discard states that are unobservable. This can be because there is no enclosing ‘try’, or because the task will be killed on failure. Notice that this depends on mutable 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 `try` 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 ‘try’ or ‘escape’ 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.
- Boundary calls can also act as hints that this is a good point to do GC (just after discarding the stack and rollback information, and before the call).
- Future computations run in their own task typically with no enclosing ‘try’, so need not record rollback information.

Transaction overhead

- We claimed that pervasive transactions were not too expensive. This is mainly because only rollback is needed, not conflict detection between concurrent transactions.
- The problem is closer to backtracking in logic languages such as ***Prolog***, than to Software Transactional Memory or database transactions. An escape point is analogous to a *choice point*.
- This means transactions don't need to maintain read sets, which is what makes STM expensive [CITATION].
- Barriers are only needed for mutations (writes to mutable cells), which are much less frequent than reads in mostly-functional languages.
- ***Prolog*** implementations typically use trailing: when mutating a cell, record the location and old cell contents in a *trail stack*. To backtrack, unapply the mutations by popping the trail entries in reverse order.
- Naively costs memory proportional to the number of writes in a transaction, but writes to the same location between escape points can be merged if we run out of memory. Writes to locations that no longer exist can be discarded.

Transaction overhead (continued)

- We can reduce the number of trailed mutations 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 cell only requires a trail entry if the cell 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.
  - If the previous value of the cell is null, we can store just a tagged location rather than also storing the null.
  - We should make sure that the trail stack reuses the same memory between boundaries to avoid cache pollution (similar argument to [Gonçalves and Appel 1995]).
  - It's also important to make sure that the code generated for trailing doesn't cause unnecessary processor stalls.

Transaction overhead and GC

- Apply write barrier elision optimizations, using analogy of a trail buffer to a *sequential store buffer*.
- “Snapshot-at-the-beginning” concurrent garbage collection algorithms retain sufficient information to recover a conservative approximation to the set of live objects at the time of the snapshot.
- A trailed write is similar to a Yuasa barrier typically used to implement snapshot-at-the-beginning GC, except that it must store 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”.
- (Retaining enough information to roll back to a escape point automatically retains the root set at that escape point. Escape points at the start of turns are good places to do a GC snapshot, because there is no stack and typically less “ephemeral” live data.)

Noether-FRMB

- This generalizes lazy futures, to allow them to be computed in the background, rather than only when needed.
- We add a ‘trigger’ operation that hints that a future task should start executing, but does not return its value. Triggering a future at the same time as creating it, is just a special case.
- The runtime only follows the hint when it can obtain more concurrency by doing so, i.e. when it can use processors that would otherwise be idle. This can be implemented by scheduling future computations as low-priority tasks.
- The actual memory usage becomes more difficult to predict. To make progress, the runtime need only compute one future at a time per explicit task, but background computation allows it to compute more than one.
- Under memory pressure, the total memory taken by future computations can be throttled:
  - queue new computations until existing ones have finished
  - pause existing computations (all but one in each task, starting with the most recent)
  - discard existing computations and restart them later. This is a last resort because it entails repeating work.
- The actual performance also becomes more difficult to predict.

Noether-FRMBC = Noether-D

- This adds dataflow concurrency, in a form that is strictly deterministic.
- What conditions are needed for it to be deterministic?
  - A task cannot wait for any of several concurrent events; it must decide which event to wait for.
  - Any reads or writes of concurrently accessible state must be monotonic. [XXX DEFINE]
- We already have tasks; they were added without fanfare in ***Noether-PLHTZ***. However, it was not possible to pass any mutable state into a task.
- ***Noether-FRMBC*** is obtained by adding "owned dataflow variables", which are first-class variables for which write authority can be transferred linearly between tasks.
- Transferring write authority creates a fresh write cap and passes it in a message to another task; the old write cap immediately becomes invalid via all aliases.
- On an attempted write:
  - If the write cap is invalid (which only happens if it has been transferred), the write fails.
  - Otherwise, writing a value that is *unifiable* with the existing value succeeds. Writing a value that is not unifiable with the existing one fails.
- Reading a dataflow variable only obtains information that cannot change after the read has completed.
- Unlike dataflow concurrency in ***Oz***, tasks cannot race to perform the first write (and transfers are deterministic), so this does not introduce nondeterminism *even when writes fail*.

Noether-DW

- This sublanguage removes the constraint that dataflow variables are owned by a single task.
- This gives a form of dataflow concurrency that is fail-stop deterministic, but not strictly deterministic.
- Tasks holding a write cap can race to set the variable. This introduces nondeterminism, since the write that wins the race will succeed and the others (if conflicting) will fail.
- Programs that are nondeterministic because they make conflicting writes can be considered incorrect (as in ***Oz***).
- However, we'd like to be able to ensure determinism of this sublanguage even for untrusted code. To do that, we can confine a computation in the sublanguage, and ensure that any conflict failure within the confined bubble destroys the bubble, thus hiding the nondeterminism.
- In order to get information out of the bubble deterministically, we need a termination condition. One obvious possibility is that all remaining tasks in the bubble are quiescent. At this point the final state is fully deterministic. It is also deterministic whether or not the bubble has a conflict failure (even though it is nondeterministic which race causes it).
- Other kinds of nondeterministic failure in a confined bubble will also destroy the bubble. To maintain determinism, the handler must only handle conflict failures, and must propagate other kinds of failure. Alternatively, we can accept the nondeterminism resulting from handling other failures.

Noether-DWN

- Nondeterministic merge
- Allows waiting on multiple *explicitly* specified dataflow variables (which can represent lazy input channels).
- This is not equivalent to message passing concurrency (introduced next), which would require a vat to wait on any message directed to one of its objects. To do that in this sublanguage, the wait would need to explicitly specify all objects in the vat.
- The constraint is useful for certain styles of dataflow programming, particularly mostly-static dataflow networks and Flow-Based Programming.
- Compared to message passing concurrency, it is:
    - Easier to bound storage requirements
    - Easier to analyse for datalock and gridlock.

Noether-DWNE

- Message-passing concurrency and event loops:
- If the top level of any task program is a loop, either with no try/handle outside the loop or with looping achieved by a boundary tail-call, then rollback state can be discarded on each iteration.
- A task that follows this pattern and processes one or more incoming messages on each iteration is called an "event-loop task", and each iteration is called a "turn".
- So, turns determine the granularity of transactions. This makes sense because turns are the unit of interleaving: a program must ensure in any case that invariants hold at each turn boundary.
- The event-loop structure is not primitive; a task need not use an event loop, and it may implement its own event loop (perhaps with different failure handling, prioritization of received messages, intercession on messages, or transactional processing of multiple messages).
- The message ordering constraint is ***E***-order.
- Communication between tasks requires remote caps. The type system keeps track of which caps must be local.
- As in ***E***, values passed in messages are logically copied into the receiving task's home zone (~ vat), stopping at remote or mutable caps. The implementation is expected to share immutable structures (at least when they are large) between zones on the same machine.

Noether-DWNE (continued)

- The main difference between message passing semantics of ***Noether*** and ***E*** is in the failure model; ***Noether*** implements the ***Ken*** protocol (or any protocol that provides output validity), which allows some failures to be masked.
- This requires sent messages to be queued and only released when the turn commits.
- Speculation is allowed as long as it does not transmit values marked secret (see slide 38) out of the local TCB.
    - The restriction for secret values does not prevent leakage of control flow information, e.g. if a task includes code to send a message or not depending on a secret value, and the execution of that code is rolled back, then this bit of information may still be leaked. We believe this is an acceptable trade-off for the improvement in latency allowed by speculation.
- We do not require the ***Ken*** implementation to tolerate all failures or retry indefinitely. The application can partly control retry policy.
- The scheduler detects lost progress due to cycles of closed waits. Not all lost progress bugs can be detected if there are cycles of waits that include open waits.
- Lost progress bugs will probably most often be datalocks, but the full language is not deadlock-free when concurrency styles are mixed (an unfortunate but apparently unavoidable consequence of allowing multiple concurrency styles).
    - Experience is needed to see whether this is actually a problem.

Type-and-symmetry system

- Recall that a symmetry is a transformation that preserves an equivalence. For ***Noether***'s type system, we are concerned with symmetries on terms. For example, the property that all terms reachable from V are pure and acyclic is a symmetry of V, preserved under operations that extract subvalues.
- Symmetries of a program are reflected as symmetries of the corresponding function and code values.
- Sometimes symmetries can be inferred from plain types; sometimes we need or want to declare a symmetry explicitly.
- Refinement types are supported. Tentatively, full dependent types are not.
- Type checking is decideable.
- The symmetry of a function is often dependent on the weakest symmetry of its input arguments. For example, map is pure only if it is mapping a pure function over a pure list. ***Noether***'s type system allows functions that have this kind of polymorphic symmetry.
- Otherwise, it is a fairly conventional type system similar to those of ***Standard ML*** and ***OCaml***, with Hindley-Milner-based type inference (perhaps requiring more declarations than ***OCaml***) and row polymorphism.

Parametricity and casting

- Parametricity is a useful symmetry/security property of type systems. Technically, parametricity can be emulated by use of type-enforcing wrappers or sealers, but direct support for parametricity has the following advantages:
    - The obvious expression of parametric algorithms will have the parametricity property.
    - Parametricity proves useful "free theorems." (In terminating sublanguages, we don't need to make further assumptions about strictness for these to hold.)
    - There is no avoidable run-time overhead (which could be optimized out but probably wouldn't be) or conceptual overhead due to the use of wrappers.
    - Some cases would only be expressible using sealer/unsealers. These require static names, which weaken equational reasoning, whereas direct support for parametricity would not.
- There is a related reviewable-security problem in type systems that include subtyping with downcasting. Consider a mutable object that is passed via a read-only interface. The receiver can downcast and modify the object, but it is not *locally* apparent, based on review of types, that this will be possible.
- Preventing downcasting or type inspection in all cases reduces expressiveness. Can we "have our cake and eat it," by making it explicit when downcasting and type inspection will be allowed, and obtaining the security and reasoning benefits of parametricity when these facilities are not needed?

Taut and Loose Types

- Our solution is a type modifier that specifies whether a type is "taut" or "loose". This makes it locally apparent whether downcasting will be possible. For each taut type T, we have a loose type ΔT, and an loosening conversion ΔT(•).
    - The types and type variables introduced by Hindley-Milner inference are taut; only declared types and type variables may be loose.
- Suppose that we upcast from A to B and then to ΔC. It should not be possible to downcast past B. This requires a type-enforcing wrapper. Such wrappers do not affect equational reasoning and are introduced only at a loosening conversion. Multiple layers of wrapping are never needed.
- In order to preserve equational reasoning, loosening conversions must explicitly specify the wrapper type. Δ(•) without a specified type would not be a function, since obj:S = obj:T $\not\Rightarrow$ Δ(obj:S) = Δ(obj:T).
- The resulting type system feature is similar to Dynamic, but:
    - Loose types can be more precise than a single Dynamic type.
    - 'typecase' is not needed; tightening conversions (from ΔC to C) are implicit.
- It is unclear at the moment whether upcasts will be explicit or implicit. Tentatively, upcasts will be explicit and type variable unifications implicit.
    - Explicit upcasts have the advantages of making a security-relevant operation locally apparent in the code, and of simplifying type inference (as in ***OCaml*** and other type systems with both Hindley-Milner-style inference and subtyping).
    - Explicit upcasts may be too verbose. Experience is needed to see how much of a problem that is.

Task priorities

- Tasks have a numerical priority.
- The scheduler implements priority inheritance for closed waits: if one task *A* is in a closed wait for another (single) task *B*, task *B* will implicitly inherit *A*'s priority, if higher. "Priority propagation" is also implemented.
  - There are some cases where you want to wait for a task and explicitly avoid raising its priority. Since implicit priority inheritance is normally more useful, it is the default.
- There is some controversy over whether the priority inheritance *vs* priority ceiling approach is preferable in shared state models. In a message-passing concurrency model, priority inheritance has fewer problems because it is needed in fewer cases:
  - The priority ceiling/highest locker protocol comes essentially for free because each resource is implemented by a task, and that task's priority acts as the resource's priority ceiling.
  - A high-priority task that sends a message asynchronously can continue with other processing until the reply is available.
  - Priority inheritance is only needed for remaining cases in which a higher-priority task *blocks* waiting for a task with lower priority. If following the design guidelines for the priority ceiling approach this will never happen, but it is sometimes useful.
- Suppose a task is waiting to receive messages sent by several other tasks of different priorities. It will not process a message from a lower-priority task in preference to one from a higher-priority task, unless forced to do so by message order constraints.
- Since future and generator computations are run in separate tasks, we can control their priority. The 'trigger' operation on a future just raises the priority of the future's task from "suspended".

Debugging

- It's important to support debugging of parallel and concurrent executions.
- A sequential debugging log is not sufficient because the control structure of the execution is not sequential.
- The overall control structure of a ***Noether*** execution is a forest, with each tree representing a task, and each branch representing a spark. All the branches "grow" in parallel.
- For on-line debugging, we need a debugger that can interactively explore this forest.
- Rollbacks can cause some branches to "die". It should still be possible to explore these in order to determine the reason for the failure that triggered the rollback.
- The log should be annotated to include dependencies between tasks, in a similar way to Causeway [Stanley, Close and Miller 2009].
    - We'll start by generating Causeway's language-neutral log format so that we can use it directly, but we eventually want an on-line debugger for ***Noether***.
    - *Event abstraction* allows patterns of messages caused by a stable abstraction to be presented in a form that hides irrelevant detail.

Zones

- Objects are allocated in zones. Zones can contain subzones, forming a tree. Subtrees cannot span nodes. Limits can be put on the total resources accessible to any subtree.
  - Rationale for tree structure: we want to be able to limit the resources used by a confined activity, but it should be able to do the same thing for subactivities. This requires the structure to be at least a tree. A more general graph, in which zones can have multiple parents, seems unnecessary.
- Zones can each be under different memory management regimes, not only garbage collection.
- Each task has a home zone, which stores its stack.
- A zone can be deleted. This also, as-if atomically, deletes any subzones and kills any tasks whose home zone was deleted.
- Attempting to dereference a value in a deleted zone fails. This is a runtime failure; there are intentionally no type system constraints that would prevent it.
  - References by a task into an ancestor of its home zone cannot fail, because the task would be killed before it can observe the deletion of the ancestor.
- If a task that owns exclusive write authority to a promise or dataflow variable is killed, the promise or variable becomes broken, and any closed waits for its value will fail.
  - This implements promise contagion as in **E**. It can also be used to support an **Erlang**-style supervision hierarchy.
- Capabilities to immutable structures in other zones can be shared between tasks on the same node (anywhere in the node's zone tree) and accessed synchronously. Lazily computed values count as immutable, and can be triggered by an access from another task.

Confinement

- Since **Noether** is a pure obj-cap language, it is possible to build non-primitive confinement mechanisms.
- However, here we consider the built-in confinement mechanism which is based on zones.
- The main focus is on efficient resource control/accounting and destruction of confined computations.
- A confined set of zones and tasks is called a "bubble". Code inside the bubble only has the authority to create subzones, so we can destroy the whole bubble efficiently.
- We can measure and limit total CPU usage policy and maximum priority for tasks in the bubble, and total memory usage for zones. (The measurements are nondeterministic.)
- Zones inside the bubble can have references to shared deep-immutable data outside it. This memory is also accounted to the bubble, because it *might* be the only thing keeping the shared data live.
- Other features useful for confinement (details to be worked out):
  - Flows – a flow allows reflection on messages between some subset of senders and receivers. The total bandwidth and space consumption of a flow may be limited.
  - A "module system" supporting dependency injection.

Staging

- Staging annotations are a highly structured way to separate the execution of code into stages.
- The annotations are "brackets" and "escapes" as in ***MetaML***. (In some papers these are called defer and splice operators respectively.)
    - Brackets surround code fragments.
    - Escapes perform *capture-avoiding* substitution of one code fragment into another.
- Note that this is essentially capture-avoiding quasiquotation, but we use the ***MetaML*** terminology and semantics.
- The "unstaged elision" of a valid program is the program obtained from it by deleting all staging annotations.
- Valid programs must be well-typed and "correctly staged" (also called "well-leveled"). A program is correctly staged if earlier stages do not depend on values that will only be known in a later stage.
- Stages before the run stage must be in ***Noether-P***, hence definite.
- Under these conditions, any valid program *P* has the same type and possible behaviour as its unstaged elision *U*, apart from changing when (effect-free) work is done.
- For more detail see Multi-Stage Programming: Its Theory and Applications [Taha].

Generative Macros

- Macro systems can greatly extend the expressiveness of a language, and used well, can allow a program to be closer to its specification. “Built-in” but non-primitive syntactic abstractions can often be specified and implemented using macros.
- However, *analytic macros*, which perform arbitrary analysis and transformation of input code (as provided by unrestricted macro systems from other languages such as **Common Lisp**) occupy a suboptimal point in the symmetry-expressiveness space:
  - We cannot guarantee that an analytic macro generates well-typed output code from well-typed input.
  - We cannot guarantee, in general, that the results of symmetry inference applied to the input code will not be misleading. This is a potentially severe practical problem for security review, because a reviewer should normally only have to look at the input code, not the output.
- *Generative macros*, which only construct and combine code, can be proven to generate only well-typed code given well-typed input. We follow Macros as Multi-Stage Computations [Ganz, Sabry and Taha], defining the semantics of generative macros in terms of staging, as in ***MacroML***. (Since this transformation is global, generative macros are more Felleisen-expressive than staging.)
- ***Noether-0*** is sufficient to express most useful macros. In this sublanguage it is easy to write generative macros that are guaranteed to be hygienic. The output sublanguage of a generative macro is straightforward to infer from the input fragments and macro body.
- In general, macros transform code values represented as ASTs. Quasiquotation allows these representations to be expressed in more readable code.
- Macros can be debugged by allowing the user to see the code at any level of expansion. When code is copied into an expansion, it can be mapped back to its pre-expanded position.

Language plugins

- An earlier plan was to support *analytic macros* as well as generative macros. However, analytic macros are not expressive enough for whole-program transformations and analyses, while still posing problems for security and robustness as described on the preceding slide.
- Rather than imposing further restraints (a direction that is already adequately covered by generative macros and staging), we increase expressiveness by providing a language plugin API. Unlike an analytic macro, a plugin is conceptually separate from programs that use it, and is expected to be reused across programs.
- Programs can and must specify which plugins they enable for a given span of source code. Such spans are effectively in a different language, providing support for domain-specific languages as well as conservative extensions of ***Noether***.
- A plugin is a *partially trusted* component for each program that enables it. Plugins are subject to least authority, so for example if a plugin is only performing an analysis without generating code, or if it is only supposed to generate code with a given symmetry, that can be enforced.
- Plugins can perform arbitrary analysis and transformation of input code, including global transformations at a specified compilation stage.
    - If the compilation strategy is significantly changed, plugins that declare a dependency on an affected compilation stage might (visibly) break and need to be updated.
- A plugin is required to cooperate with the symmetry inference. It is in a position to declare whether applying symmetry analysis to each input gives meaningful results; and potentially even modify how the analysis is performed. It can also generate its own specialized error messages.
- A plugin cannot affect how its *output* is analysed.

Secret values

- There are advantages to the language implementation being able to distinguish values that are secret:
    - Secret values are not speculatively transmitted outside the current TCB.
    - Secret values are not logged. They can specify a non-secret identifying string to be logged instead.
    - Secret values can be stored only once in physical memory (and not copied by GC, for example). This allows zeroisation of the only in-memory copy to resist cold-boot attacks. They could potentially be stored in specialized memory, hardware permitting.
    - Secret values can be separately checkpointed, possibly with additional security precautions. Alternatively, they can be omitted in checkpoints and an application can handle the failure on accessing them after restore.
    - Operations on secret values can automatically use implementations with greater resistance to side-channel attacks [credit to Zooko Wilcox-O'Hearn for this idea].
- These advantages remove some trivial objections to the use of languages with automatic memory management.
- Applications often don't need to directly mark secret values if they use cryptography libraries that do so, for example.

Disadvantages of stratified languages

- Some features of a stratified language will be rarely used. These features nevertheless impose significant costs in implementation and specification complexity.
- The full language is likely to take longer to teach or learn.
    - We are optimizing for incremental teaching of less commonly used features as they are encountered, rather than teaching the full language in one go.
- Following the *Principle of Greater Symmetry* takes greater short-term effort and discipline.
    - We believe it will more than pay off in debugging time and confidence in correctness.
- In *some* cases additional effort is needed to prove to the compiler that a program has a safety property, such as termination.
- In *some* cases, the program will be more verbose (but not to the point where verbosity interferes with reviewable correctness).
- Refactoring that imposes a smaller sublanguage constraint on existing code can be difficult.
    - On the other hand, we try not to impose unnecessary refactoring burdens, for example we allow subprograms that use mutable state only internally to conform to a pure functional interface.
- Symmetries that are broken in the full language are not necessarily exploitable for optimization: when compiling code from a smaller sublanguage we cannot in general assume that it will never interact with code from a larger sublanguage. Example: evaluation order of pure-functional code is exposed by failure handling.
    - Sometimes this can be worked around by compiling code more than once under different assumptions, at a cost in compilation time and size of object code.

Rejected feature: monadic I/O and monadic state

- ***Noether*** has a type-and-symmetry system that separates pure and impure computations, as a monadic approach does. Unlike the monadic approach, a pure value is always substitutable for an impure one.
- In ***Haskell*** for example, within a 'do' block there is a distinction between '*y* ← f x' (only correct when 'f x' is an action in the monad) and 'let *y* = f x' (only correct when 'f x' is pure). In ***Noether*** both are 'def *y* = f x', and the enclosing function is inferred to be impure if 'f x' is impure.
- No "monadic plumbing" – no 'lift*', 'bind*', 'return*', '>>', '>>=', 'mapM', etc. ⇒ less to remember.
- This also eases refactoring. If we increase symmetry (e.g. make an impure computation pure), there are *no* necessary cascading changes. If we break a symmetry (e.g. make a pure computation impure), then only type declarations incompatible with the broken symmetry need to change, which is what we want. Incidental syntactic changes are not needed in either case.
- The benefit is even greater when we make finer symmetry distinctions such as [a]cyclicity, [a]synchrony, etc. It would be unwieldy to handle all of these using separate monads, and some don't really make sense as monads, but they fit well into a type-and-symmetry system.
- Type inference wants to know precisely what it is dealing with, not make only generic inferences based on the monad structure. More precise inference needs more cases, but in practice should work better.
- Modelling concurrency with the IO monad, as ***Glasgow Haskell*** does, is dubious. The best that can be said of it is that it happens to work –maybe not if you use 'unsafe*'– but it has no satisfactory theoretical basis. (The semantics given by Peyton-Jones, for example, is very operational and *ad hoc*.)
- Monads can do other things, but we could implement them in a library, with 'do' as a macro; no language support is needed.

Rejected feature: continuations

- Full continuations
  - Continuations are sometimes advocated as a way to implement control-flow features such as generators, threads, or exceptions. ***Noether*** already has those features in forms that are more structured and more easily optimizable, because the runtime "knows about" the concurrency.
  - It is unclear how continuations should interact with rollback. Naïve 'call/cc' fails to preserve the strong exception guarantee.
  - Full continuations capture state and allow returning to the same code more than once. Defending against that is too hard, and unnecessarily hard for code that will normally be returned to only once. See A Security Kernel Based on the Lambda-Calculus §4.3.2 [Jonathan Rees].
  - Oleg Kiselyov convincingly argues that full continuations are inferior to delimited ones.
- Delimited continuations
  - I considered implementing rollback in terms of delimited continuations, but couldn't see how to do it simply, and it's not worth doing if complicated.
  - Literature on delimited continuations pays little attention to security, including Rees' argument.
  - Lots of variations on shift/reset operators; no consensus on which to use.
  - Potentially inhibits optimization.
  - Not clear which sublanguage it should go in or what symmetries it breaks.
  - I don't care much about this feature, so leave it out for simplicity.
- However, Yield: Mainstream Delimited Continuations [James and Sabry] is an awesome paper.

Rejected feature: synchronous message passing

- In sync MP, send and receive are as-if simultaneous: one cannot occur without the other.
- This conflicts with transactionality:
  - Suppose turn *A* synchronous-sends to turn *B* in another task. Send+receive is a side effect, so if *A* rolls back its send, *B* must roll back its receive, and *vice versa*. Unlike asynchronous message sends, we cannot transparently queue synchronous sends to the end of a turn. This "entangles" *A* and *B* so that neither turn can commit (and lose the ability to roll back) before the other.
  - Entanglement is implementable, but undesirable: long chains of turns could become entangled (bounded only by the number of tasks), and it would be impractical to ensure that turns are short.
  - It would also be sequentializing: once a task has finished its own work for this turn and is waiting for entangled turns to commit, it cannot do anything, even if there are queued messages for it! (If there are fewer ready sparks than processors, we've lost some parallelism, and even if there are more, we might incur work stealing overhead.) The blocked task could proceed *speculatively*, but we don't want to require that, or to speculate very far since we might have to discard work.
- Sync MP can't be implemented over an unreliable network (Communicating Generals Problem):
  - Groups of tasks on different machines could be separated by membranes, so that they never have direct references to which a message could be sent synchronously. But that would require a visible two-level hierarchy of tasks, which is complicated and would mean that programs relying on sync MP can't be made distributed without significant design changes.
  - For async MP, on the other hand, the ***Ken*** protocol helps programs be transparent to machine failures with fewer and less invasive (if any) design changes. They still need to take account of higher latencies, but that's "only" an efficiency issue.

Rejected feature: shared state

- The problem is lack of composability between code that relies on the strong exception guarantee, and code that can access shared state across tasks.
- Shared state is fatal to ensuring that turns behave as transactions ⇒ fatal to the ***Ken*** protocol.
- It would be disastrous to roll back unshared but not shared state – each task would "forget" information needed for coordination with other tasks. ("What did I do last night?" problem.)
- What about disabling rollback entirely for tasks that can access shared state?
  - Those tasks could not synchronously call code that relies on the strong exception guarantee.
  - We want to rely on that guarantee for *all* standard library code, because we want the TCB to be defensively correct. So, shared-state-accessing tasks would have no standard library.
- What about declaring which functions should be transactional and which should not be?
  - Doesn't work – if transactional functions cannot call nontransactional ones, then this reduces to the same thing as disabling rollback for the task.
- What about entangling turns that access the same state (effectively a form of STM)?
  - Same objections to entanglement as for synchronous message passing.
- Sharing of *immutable* data between tasks is still possible.
- Plenty of languages support shared state; very few support output-valid rollback recovery; no others (?) support the strong exception guarantee for all code.

Thanks to

- Special thanks to the friam group, who walked through a trial run of this presentation with me.
- Would also like to thank:
- Zooko Wilcox-O'Hearn, Brian Warner, Meredith Patterson, Alan Karp, Jonathan Shapiro, Mark S. Miller, Dean Tribble, Norm Hardy, Carl Hewitt, Marc Stiegler, Kevin Reid, 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, Tyler Close, 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