Noether programming language

- Strictly obj-cap, typed, parallel and concurrent, potentially highly optimizable.
- Nested subsets of increasing {freedom, expressiveness, power}, decreasing {symmetry, order, tractability}.
- This talk considers only semantics, not syntax.
- Why a new language?
  - puts fewer constraints on optimizing symmetry–freedom.
  - existing languages have features (e.g. exception handling, concurrency) that we want to do or combine in a different way
  - address previously unsolved or solved-only-as-research problems: precise resource control, partial failure (different solution to E), flow control in async message passing, etc.
  - security + expressiveness + speed is already very hard; adding compatibility with some existing language would make it impossible.
  - compatibility is overrated; familiarity is more important
  - *more fun* (must be more fun than Jacaranda)

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. Also some of Emmy Noether’s other work on abstract algebra is indirectly relevant.
- Hard to believe this name hasn’t already been snapped up for a programming language! (but Google says not)

Nested subsets

(slightly out of date)

```
 ________________________________________________________________________
| 0: core                               |  |  |  |  |  |  |  |  |  |  |  |
|                                serial |  |  |  |  |  |  |  |  |  |  |  |
|---------------------------------------'  |  |  |  |  |  |  |  |  |  |  |
| P: parallelism                           |  |  |  |  |  |  |  |  |  |  |
|                                stateless |  |  |  |  |  |  |  |  |  |  |
|------------------------------------------'  |  |  |  |  |  |  |  |  |  |
| L: very-local mutable state                 |  |  |  |  |  |  |  |  |  |
|                                   fail-stop |  |  |  |  |  |  |  |  |  |
|---------------------------------------------'  |  |  |  |  |  |  |  |  |
| E: backtracking exceptions                     |  |  |  |  |  |  |  |  |
|                                         strict |  |  |  |  |  |  |  |  |
|                       acyclic, initial algebra |  |  |  |  |  |  |  |  |
|                                 noninteractive |  |  |  |  |  |  |  |  |
|------------------------------------------------'  |  |  |  |  |  |  |  |
| Z: laziness                                       |  |  |  |  |  |  |  |
|                   strongly normalizing/convergent |  |  |  |  |  |  |  |
|---------------------------------------------------'  |  |  |  |  |  |  |
| T: nonterminating recursion                          |  |  |  |  |  |  |
|                                                 pure |  |  |  |  |  |  |
|                                        Church-Rosser |  |  |  |  |  |  |
|------------------------------------------------------'  |  |  |  |  |  |
| O: output reducers                                      |  |  |  |  |  |
|                                              functional |  |  |  |  |  |
|---------------------------------------------------------'  |  |  |  |  |
| M: mutable task state and general reducers                 |  |  |  |  |
|                                     N*serial space-bounded |  |  |  |  |
|------------------------------------------------------------'  |  |  |  |
| F: [generalizes Z] pure futures                               |  |  |  |
|                                          sequential semantics |  |  |  |
|---------------------------------------------------------------'  |  |  |
| D: dataflow concurrency                                          |  |  |
|                                                    deterministic |  |  |
|------------------------------------------------------------------'  |  |
| N: nondeterministic merge                                           |  |
|                                         only bounded nondeterminism |  |
|---------------------------------------------------------------------'  |
| C: message passing concurrency                                         |
|                        no nondeterministic deadlock or low-level races |
|                                              pervasively transactional |
|                                                           finite delay |
'------------------------------------------------------------------------'
```

Expressiveness and restraint

- Noether is a “bondage and discipline” language :-)
- Expressiveness properly refers to support for the expression of correct programs.
  - Example: languages that permit undefined behaviour are permissive, but programs that exhibit undefined behaviour are incorrect for any nontrivial specification, so the possibility of undefined behaviour never *by itself* increases expressiveness.
  - Expressiveness by itself is not sufficient, because we don't just want to support the writing of correct programs; we want to decrease the likelihood of writing incorrect ones. (Astonishingly considering the prevalence of incorrect programs, this is actually a controversial position!)
- Restraint is a language-supported programming discipline: try to express a specification using the language subset that results in the most tractable correct program.
  - This is *usually* the least expressive subset in which we know how to meet the specification without undue circumlocution.
  - Language expressiveness is negatively correlated with tractability of programs because more expressive languages typically have fewer symmetries. A symmetry is a preserved property (such as type safety) or equivalence (such as evaluation-order independence).
  - Some symmetries are so useful that they should be preserved globally at all costs (for example: memory safety).
  - *Principle of Restraint* is related to the *Principle of Least Power* or *Principle of Least Expressiveness*, but not identical because *only* tractability is important. For a specific desired behaviour, the most tractable expression might not use the least expressive subset (example: the Same Fringe problem is pure-functional but more tractably expressed using generators).

Expressiveness and restraint (continued)

- Disadvantages:
  - Some features 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 Restraint* 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.
  - Refactoring that imposes a smaller subset constraint on existing code can be difficult. (On the other hand, we try not to impose unnecessary refactoring burdens, for example allowing 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 subset we may not in general assume that it will never interact with code from a larger subset. (Example: evaluation order of pure-functional code 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.

Noether-0

- Strict, deterministic, pure functional core:
  - no divergence, all computations terminate! See Total Functional Programming [Turner]
  - no cyclic data (consequence of strictness) or mutable data
  - not Turing-complete, but more powerful than primitive-recursion (proof: can express the Péter function).
- We allow failures, but no failure *handling* within the subset:
  - more realistic — **a nontrivial programming language semantics that doesn’t allow failure cannot be implemented on any real computer**. Failures can be due to insufficient memory, killing from outside, detected but uncorrected hardware errors, assertion and variant checks, etc.
  - no artificial definitions to make functions total (as in Turner’s paper)
  - no need for dependent types (as in Epigram)
  - can use unproven assertion and variant checks at run-time.
- Termination proving can be completely automatic, even for mutually recursive functions (just apply existing research; termination provers got really good and no-one noticed).

Noether-0

- Lack of divergence and mutable state allows equational reasoning. Allowing failures does not compromise equational reasoning:
  - anything fails $\Rightarrow$ whole compution fails
  - whole computation does not fail $\Rightarrow$ can assume that nothing failed
- Therefore, all definitions in a succeeding computation are equalities (not true if we have exception handling or mutable state)
- 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 recover from it (*outside* the subset); we can’t recover from arbitrary nontermination bugs without using timeouts. (Timeouts are crap: they introduce false positives and nondeterminism, and are slow to recover even when they work.)
- Absence of algorithmic failures (as opposed to hardware failures or being killed) in any subcomputation can then be proven separately and modularly.
- This is not a subsitute for complexity analysis; terminating functions can take a *long* time even for “small” inputs (e.g. Péter function!)
- Conjecture: writing functions to be provably terminating 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.

Noether-P

- Adds fork/join parallelism, but with *sequential semantics* — just delete the annotations to get an equivalent Noether-0 program.
- spawn, parallel for, sync constructs as in Cilk. (No “reducers” in Noether-P, will add those in a larger subset.)
- Works because of confluence properties — different evaluation orders produce the same results (in this subset, and also in larger subsets). Read the Cilk / Cilk Plus papers for details.
- Advantages of Cilk:
  - adapts to the available hardware parallelism with very little overhead. Useful speed-up in practice with just two processor cores (unlike STM!), 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 — allows sequential debugging for example.
  - bound on space consumption (N × serial on N cores).
- Improvements over Cilk:
  - no possibility of race conditions. (Trivially in Noether-P, because no mutable state. Will return to this point when we add state.)
  - therefore, parallel semantics are *identical* to serial semantics, and equally tractable.
  - larger subsets of Noether will provide true concurrency, so restrictions are less burdensome.

Noether-P (continued)

- Adds fork/join parallelism, but with *sequential semantics* — just delete the annotations to get an equivalent Noether-0 program.
- This is not concurrency: there's no way to write programs that depend on concurrent operations, since serial execution is a valid implementation.
- 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 semantics are the same, why can't the compiler add annotations?
  - More understandable performance model.
  - When we add state, there will be restrictions on it that depend on the annotations, so the programmer must choose them.
  - Don't want to have to implement a Sufficiently Smart Compiler (*wouldn't be enough fun*).
  - No need for this to be in the compiler, could be 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.
- 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 substituting their current values. 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 frame-local 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-PLE

- Adds failure handling, with try/handle construct. Now we can potentially distinguish different reasons for failure.
- A reason is a pure value (and will remain so when we add more general state).
- Handling a failure rolls back state to the try point. (In Noether-PLE this can only apply to frame-local mutable state, but in larger subsets it will apply to all task-local state.)
  - 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.
  - Security property: functions can assume intermediate states are not observable on failure. This holds in the full language, and should make writing defensively consistent code *much* easier.
- Noether-PLE is still terminating, 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 LTR CBV rather than making evaluation order unspecified, because we want failure handling to be available in a deterministic subset.

Noether-PLET

- Adds Turing completeness.
- Arbitrary recursion and non-terminating 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-PLET is still 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 x = loop x + 1
```

does not imply

```
0 = 1
```

Noether-PLETZ

- Adds lazy capabilities. In larger subsets 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. (So we technically add concurrent tasks here, although most aspects of concurrency are not observable.)
- Note that implementing memoization within the language would require mutable state, which we do not have in this subset (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.
- In Noether-PLH, we had the same space bound property as 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 subsets, all data structures were acyclic. That property helps in proving termination, which is why we dropped the termination requirement before adding laziness.

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 strand or a child strand) without affecting the result.
- Reducers are implemented, as in Cilk, by splitting the state of each reducer when strands fork, and combining states when strands 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-FO

- Noether-PLETZ is still a functional language (by most definitions). Abbreviate it to Noether-F.
- 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!)
- 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. To support logging, for example, run the program as a future computation with 'log' as update on a free monoid.
- 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 (in another task), so doesn't affect it.
- By default, the update operation blocks until the next value is needed. We'll generalize this later.
- 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 ML.)

Noether-FOM

- This subset 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.
- When an exception is caught, state is rolled back to the try point. This creates a problem: when can we discard information needed to roll back?
  - Answer: when the state that would occur after rollback is 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. The stack and rollback information are discarded at a boundary call.
  - If the top level of any task program is a loop, either with no try/catch 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), so this will in practice limit 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.

Noether-FOMB

- 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-FOMBD

- Strictly deterministic dataflow concurrency.
- What conditions are needed for it to be deterministic?
  - Cannot wait for any of several concurrent events; must decide which event to wait for.
  - Any modifications of concurrently accessible state must be monotonic.
- 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-FOMBD 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 can only happen if it has been transferred), the write fails.
  - Otherwise, if the variable has never been written, the write succeeds.
  - Otherwise, writing the same value has no effect, and writing a conflicting value fails.
- 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-FOMBD'

- Fail-stop deterministic dataflow concurrency.
- Dataflow variables need not be owned by only one task.
- 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.
- However, we'd like to be able to ensure determinism of this subset even for untrusted code. To do that, we can confine a computation in the subset, and ensure that any conflict failure within the confinement 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.

Noether-FOMBDN

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

Noether-FOMBDNC

- Message-passing concurrency and event loops:
- Recall from slide 14: 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.

Noether-FOMBDNC (continued)

- Message-passing concurrency and event loops:
- 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 out of the local TCB.
- 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 preserved property or equivalence. For Noether's type system, we are concerned with symmetries of values. For example, the property that all values 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. 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. Our type system allows functions that have this kind of polymorphic symmetry.
- [TODO: more detail]

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 inherit *A*'s priority, if higher. "Priority propagation" is also implemented.
- 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".

Spaces

- Objects are allocated in spaces. Spaces can contain other spaces, forming a tree. 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 spaces can have multiple parents, seems unnecessary.
- Each task has a home space, which stores its stack.
- A space can be deleted. This also, as-if atomically, deletes any subspaces and kills any tasks whose home space was deleted.
- Attempting to dereference a value in a deleted space 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 space 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 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.)
- References into read-only spaces can be shared between tasks and accessed synchronously. It is not required that the task's home space be a parent of the space it references. Read-only values can be computed lazily.

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 spaces.
- The main focus is on efficient resource control/accounting and destruction of confined computations.
- A confined set of spaces and tasks is called a “bubble”. Code inside the bubble only has the authority to create subspaces, 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 spaces. (The measurements are nondeterministic.)
- Spaces 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.
  - “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.
  - Brackets surround code fragments.
  - Escapes can be used to subsitute or “splice” one code fragment into another.
- Note that this is essentially 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”. Stages before the run stage must be in Noether-P (hence terminating).
- A program is correctly staged if earlier stages do not depend on values that will only be known in a later stage.
- 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].

Macros

- Staging is not as expressive as general macros.
- Macro systems used well can allow a program to be closer to its specification.
- Noether supports analytic macros (that perform arbitrary analysis and transformation of input code) as well as generative macros (that only construct and combine code).
    - TODO: study relative expressiveness of staging and generative macros; they may be equivalent.
- The *Principle of Restraint* says we should prefer generative macros where possible; these are sufficient to express most syntactic abstractions.
- "Built-in" but non-primitive syntactic abstractions can be specifed using generative macros.
- The output subset 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 makes expansions more readable; quasiparsing makes code analysis in analytic macros more readable.
- 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.
- Noether-0 is sufficient to express most useful macros. In this subset it is easy to write generative macros that are guaranteed to be hygienic.
- Generative macros, with some restrictions, can be proven to generate only typesafe code; one possible approach is given in Macros as Multi-Stage Computations [Ganz, Sabry and Taha].

Jonathan Shapiro’s objections to Macros

- [bitc-dev] DISCUSS: To macro or not to macro?, July 2008:
- “... expanders are very easy to get wrong and very hard to understand [more so for a typed language]”
  - Yes. Code fragments can be typed in terms of the type they produce and the environment they assume. They unfortunately have complicated refinement types, which in general might have to be checked at elaboration (but we check as much as possible statically).
  - Supporting staging separately from macros incurs this burden only when macros are needed.
  - Staging can also be used within a macro expansion, which can make it easier to see which aspects of a complicated macro depend on rewriting, and which are “just” partial evaluation (for which the staging elision property holds).
- “A macro has to "run" somewhere [at compile-time] ... what environment does it run in? Should a macro-expander be able to call helper procedures ... defined above it in the program?”
  - Each invocation of the expander runs in a confined bubble and can call anything in the bubble. The complexity cost of supporting confinement is a sunk cost. The head of the macro declaration can import pure functions from the program into the bubble.
- “... how eagerly does polyinstantiation need to run?”
  - Polyinstantiation is never a required implementation technique for Noether. If used, it is probably best to apply it as late as possible.
- “Macro systems are closely tied to the AST scheme ...”
  - Expanders expressed *only* with quasiquotation and inferred types, are not dependent on the AST.

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.
  - 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.

Rejected feature: monadic I/O / 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* 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 I/O monad, as GHC 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, but Noether already has those features in forms that are more structured and more easily optimizable (because the runtime “knows about” the concurrency).
  - 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].
- Delimited continuations
  - 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.
  - The literature on delimited continuations pays little or no 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 subset it should go in or what symmetries it breaks.
  - I don’t care about this feature, so leave it out for simplicity.
- 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 strands 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.
- Rolling back unshared but not shared state would be a disaster. ("What did I do last night?")
- 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 – consider a transactional function calling a nontransactional one.
- 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. Would not be fun to compromise.