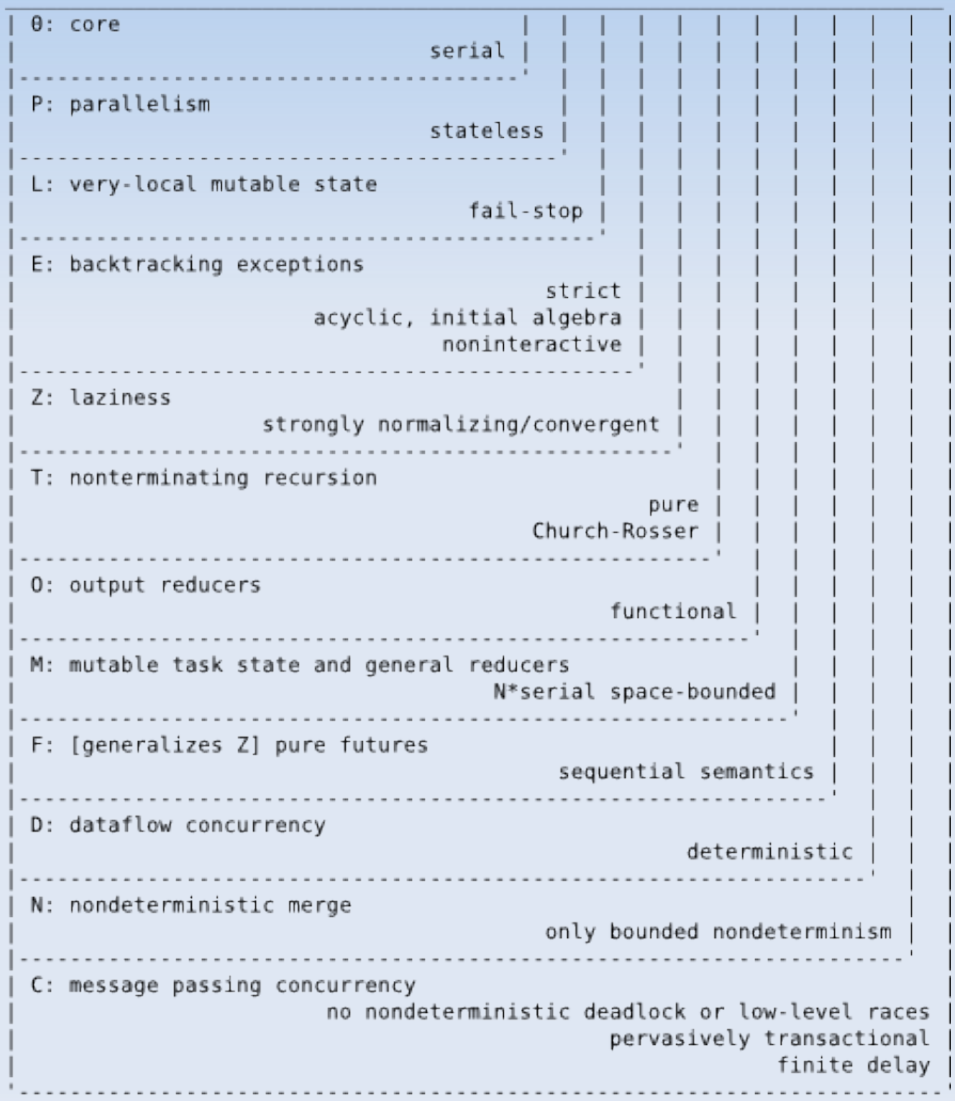
Noether programming language

- Strictly obj-cap, typed, parallel and concurrent, (potentially) highly optimizable.
- Nested subsets of increasing expressive power, decreasing tractability.
- This talk considers only semantics, not syntax.
- Why a new language?
  - puts fewer constraints on optimizing expressiveness *vs* tractability
  - 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



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
- We allow failures, but no exception *handling* within the subset:
  - more realistic — **a nontrivial programming language semantics that doesn't allow failure cannot be implemented on any real computer**. Can fail due to insufficient memory, killed from outside, detected but uncorrected hardware errors, etc.
  - easier — no artificial definitions to make functions total (as in Turner's paper), no need for dependent types (as in Epigram)
- Termination proving can be completely automatic, even for mutually recursive functions (just apply existing research; termination provers got really good and no-one noticed).
- Lack of divergence and mutable state allows equational reasoning. Allowing failures does not compromise equational reasoning:
  - anything fails ⇒ whole compution fails
  - whole computation does not fail ⇒ can assume that nothing failed
- Therefore, all definitions in a succeeding computation are equalities (not true if we have exception handling or mutable state)

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, e.g. …
- 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, serial semantics are *identical* to parallel 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 *very 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 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 might use more space (absent heroic optimisation efforts), so state is not just a convenience.
- Some loss of tractability for functions that use 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 very-local mutable variable reference into a lambda isn't valid. (The variable could be promoted to a full mutable variable, but that correspondance only holds in a larger subset.)

Noether-PLE

- Adds exception handling, with familiar try/catch syntax.
- 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).
- Catching an exception rolls back state to the try point. (In Noether-PLE this can only apply to very-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.
- Noether-PLE is still terminating, like Noether-0.
- Exceptions increase Felleisen-expressiveness, since transforming exceptions 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 any function could raise an exception that is then caught.
- Exception handling makes additional details of evaluation order visible, because it affects *which* exception occurs when there is more than one potential failure in evaluating an expression. We choose not to make the evaluation order unspecified, because determinism is important.

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 refs. In larger subsets “refs” will be used to model mutable state, references across tasks, etc., but here they refer to a lazy future.
- The ref is a read-only view. The function that will compute its value is specified when it is created, and can either execute in parallel (like MultiLisp futures), or on the strand that first needs it.
- Forcing is explicit. This allows execution time to be analysed by techniques similar to those used in Purely functional data structures [Okasaki].
- In Noether-PLE, 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 refs, need to take into account that the time at which a ref gets forced affects the program's space usage.
- Lazy refs 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.
- Noether-PLETZ is still a functional language by most definitions.

Noether-PLETZO

- Laziness 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 for example 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. What we want is effectively a safe version of Haskell's `unsafePerformIO` (which is not memory-safe!)
- To do that we add “output 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.
- An output reducer has first-class update refs, but the output can't be accessed from within the Noether-PLETZO subset.
- Why would that restriction be useful? Tractability. Since the output-so-far can't be read, it doesn't affect the course of the computation.
- The output depends on the evaluation order of the program, but is independent of parallel execution.
- The update operation of a reducer can block, applying backpressure to the updating program (this affects timing but not results, so is still as tractable).

Noether-PLETZO...

- Full mutable state and general reducers
- Parallel futures
- Deterministic dataflow concurrency
- Nondeterministic merge
- Message-passing concurrency
- I don't have slides for these yet, sorry.
- Go back to slide 3 to see the diagram of subsets.