Quanterall HQ Varna, Bulgaria 2019

Namdak Tonpa

The Languages

Groupoid Infinity

About Speaker

— PhD student, 3-rd year of education (https://cubical.systems)
— Author of 8 programming languages and 2 runtime cores
— But more famous for N2O framework (https://n2o.dev)
— Love to create programming languages and talk about them
— Know how to convert open source to money
— Aware of all operating systems/programming languages (~100/~1000)

Github Organizations

— GROUPOID — The Language of Space
— SYNRC — Application Layer Formal Specification and Implementations
— VOXOZ — Virtual Machines and Network Infrastructure

Talk Structure

The Languages

I. Languages

— Main Contributions
— Industrial Compilers
— Fast Interpreters
— Formal Verification

II. Processing

— History
— Workflow Languages
— Financial Languages
— Contract Languages

Main Contributions

— John McCarthy [LISP]
— Robin Milner [ML, Pi Calculus, HOL]
— Simon Peyton Jones [Haskell]
— Xavier Leroy [OCaml]
— Niko Matsakis [Rust] Linear Types
— Joe Armstrong [Erlang] … and many others

— Nicolaas Govert de Bruijn [AUTOMATH]
— Thierry Coquand [Coq]
— Ulf Norell [Agda]
— Leonardo de Moura [Lean] … and not so many

Industry

- V8, WebAssembly (any)
- LuaJIT (nginx)
- JVM (Oracle)
- CLR (MS) … and other JITs
- IR/MIR/LLVM (clang, rust)
- OCaml/GHC
- SPIRAL

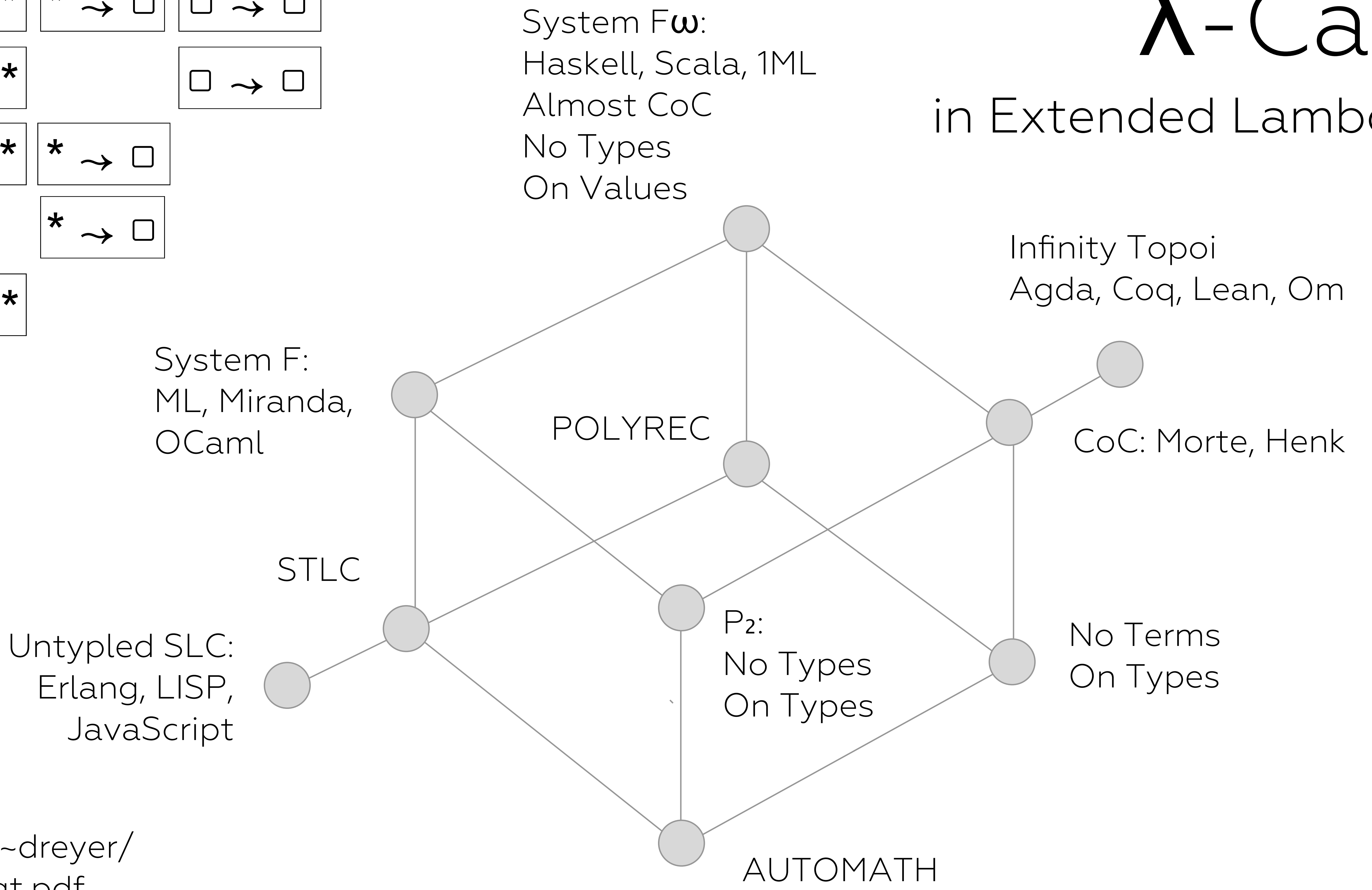
MOTTO 1: If you have compact language that fits L1 cache along with its interpreter, then you don’t need JIT! However you still need vectorization.

MOTTO 2: At enterprise scale you still need types or ULC targeted extraction.

λ-Calculi

in Extended Lambda Cube

CoC:	$* \to *$	$\square \to *$	$* \to \square$	$\square \to \square$
Fω:	$* \to *$	$\square \to *$		$\square \to \square$
P2:	$* \to *$	$\square \to *$	$* \to \square$	
AUT:	$* \to *$		$* \to \square$	
F:	$* \to *$	$\square \to *$		



http://ttic.uchicago.edu/~dreyer/course/papers/barendregt.pdf

Formal Verification

Mathematical Formal Software Verification unveils the inner structure of phenomena and avoid wide range of errors.

1) Mars Climate Orbiter (1998), conversion inch/met — $80M;
2) Ariane Rocket (1996), downcast from 64 to 16 bit — $500M;
3) FPU DIV Error Pentium (1994) — $300M;
4) Business Contract Error EVM — $50M;
5) Error in SSL (heartbleed) — $400M.

1) IEEE Std 1012-2016 — V&V Software verification and validation (4 layers)
2) ESA PSS-05-10 1-1 1995 — Guide to software verification and validation;
3) ISO/IEC 13568:2002 — Z formal specification notation.

Attempts to Fix C/C++

Expensive and long way of doing things...

— Coq: VST, DeepSpec
— Haskell, HOL: L4
— Even Manual Proofs!!!

Deep Embedding

... seems a better way exist — direct certified extraction with no imtermediate proofs!

— Coq: The best macroassembler
— Coq.io — OCaml/Lwt bindings
— Agda x86
— Clash, Lambda to VDHL/Verilog

History of Processing Languages

- EMAIL: FSM
- Event-Condition-Action Reactive Rule Engines
- Expert Systems: RETE Engine, Prolog
- Workflow Standards of the past: XPDL, BPML, OpenWFE, WWF and jBPM
- Workflow Standard After 2008: BPMN
- Trading: TpML, Fix, business contract routers, cross-exchanges, arbitrage
- Business Contacts Virtual Machines: EVM, Script VM, aebytecode
- Business Contract Languges: Sophia, Solidity, Plutus
- MLTT Frameworks: Dhall
- Iinteraction Networks Evaluators: Formality, Moonad
- Stream Processing: Oz, Erlang, np/ling, Futhark

What is the Language?

Prerequisites for bootstrapping are algebraic data types: strust (*) and union (+) from C/C++

Logic Core:

```
data pts = star (n: nat)
         | var (l: nat)
         | pi (l: nat) (d c: pts)
         | lambda (l: nat) (d c: pts)
         | app (f a: pts)
```

Runtime Core:

```
data ulc = var (l: nat)
         | lambda (l: nat) (d c: ulc)
         | app (f a: ulc)
```

Is that enough?

No, we need Inductive Types!

Inductive Core:

```
data tele     (A: U) = emp | tel (n: name) (b: A) (t: tele A)
data branch (A: U) = br (n: name) (args: list name) (term: A)
data label    (A: U) = lab (n: name) (t: tele A)
data ind = data_  (n: name) (t: tele lang) (labels:     list (label lang))
         | case   (n: name) (t: lang)      (branches: list (branch lang))
         | ctor   (n: name)                (args:       list lang)
```

IO

And we need Effects to access to business rules!

IO Core:

```
data IO (A: U)
     = getLine (_: String → IO)
      | putLine (_: String)
      | pure (_: A)
```

Secure Storage:

```
data KV (A: U)
     = get (_: String → IO)
      | put (_: String)
      | sign (_: String → IO)
      | verify (_: String → IO)
      | pure (_: A)
```

Infinity IO

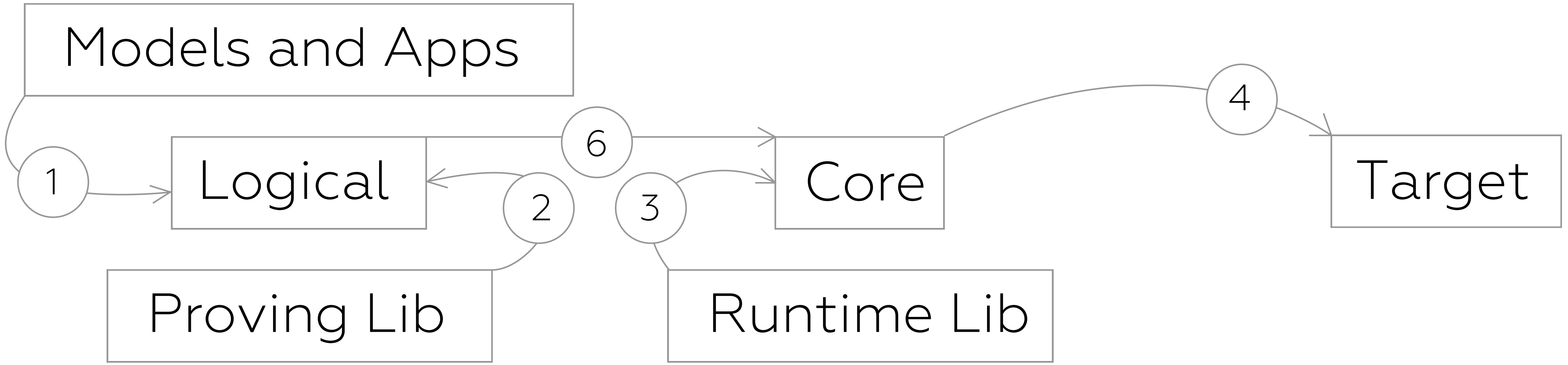
What about Infinitary IO?

```
data IOI.F (A State: U) = getLine (_: String → State)
                        | putLine (_: String) (_: State)
                        | pure (_: A)

data IOI (A State U) = intro (_: State). (_: State -> IOI.F A State)
```

Infinitely Running Processes

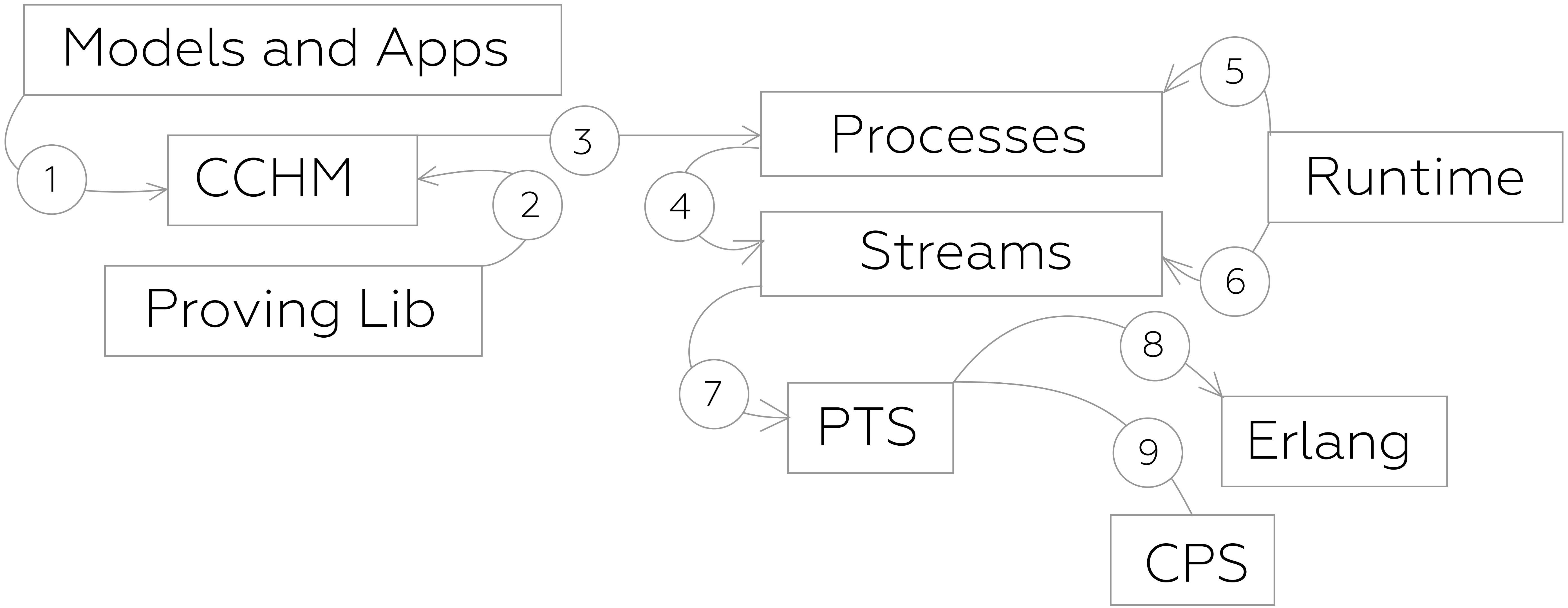
```
process : U = (protocol state: U) * (current: prod protocol state)
                                   * (act: id (prod protocol state))
                                   * (storage (prod protocol state))

spawn (protocol state: U) (init: prod protocol state)
      (action: id (prod protocol state)) : process

receive (p: process) : protocol p
send (p: process) (message: protocol p) : unit
execute (p: process) (message: protocol p) : process
```

Verification Process #1



1. Model Specification
2. Model Checking
3. Runtime Linkage
4. Target Machine Code Extraction

Verification Process #2



Research Subject

Classification of Languages use in
Specification, Formalization and Verification process

1) Specification Languages (Z, UML, MLTT);
2) Model Checkers (TLA+, Twelf, Dedukti, Z3);
3) General Purpose Languages (Haskell, OCaml, Erlang, Scala, LISP);
4) Theorem Provers (Agda, Coq, HOL, ACL2);
5) Unified Execution Environments (HaLVM, LING, Mirage);
6) Contract Machines and Languages (EVM, Script VM, Sophia, Plutus)
7) Worflow Languages (BPMN)
9) Exchange Trading Languages (TpML)

Plutus Review

IOHK Certified Language for Haskel Embedding and Development

1) Certification and Formalization (Agda): NbE, Extraction
2) Plutus IR (Lisp): Intermediate Language, Fix, No Pattern Match Compiler
3) Plutus Core: CEK, L machines
4) Scott Encoding of Data Types
5) Marlowe: Business Contracts (Alexander Nemish)
6) Plutus TxCompiler: Haskell Code to Plutus (getPlc)

Plutus IR AST

```
data Term tyname name a
    = Let a Recursivity [Binding tyname name a] (Term tyname name a)
    | Var a (name a)
    | TyAbs a (tyname a) (Kind a) (Term tyname name a)
    | LamAbs a (name a) (Type tyname a) (Term tyname name a)
    | Apply a (Term tyname name a) (Term tyname name a)
    | Constant a (PLC.Constant a)
    | Builtin a (PLC.Builtin a)
    | TyInst a (Term tyname name a) (Type tyname a)
    | Error a (Type tyname a)
    | IWrap a (Type tyname a) (Type tyname a) (Term tyname name a)
    | Unwrap a (Term tyname name a)
```

Plutus IR Sample

IOHK Certified Language for Haskel Embedding and Development

```
(lam pubkey (con bytestring)
 (lam signed (con bytestring)
[ { (abs a (type) (lam b (all a (type) (fun a (fun a a)))
    (lam t (fun (all a (type) (fun a a)) a) (lam f (fun (all a (type) (fun a a)) a)
[ [{b(fun(alla(type)(funaa))a)}tf ] (abs a (type) (lam x a x)) ] ) ) ) )
  (all a (type) (fun a a)) } [ (builtin verifySignature) signed txhash pubkey ] (lam
u (all a (type) (fun a a)) (abs a (type) (lam x a x)) )
(lam u (all a (type) (fun a a)) (error (all a (type) (fun a a)))) ] ))
```

Pure Core/CoC/Morte/Om

Theoretical Mimimum Scholarship Language Development
Toy Dependently Typed Language for Typechecking and Extraction

— CoC, Morte, Om (Pure Core)
— Further Evolution (Inductive Types): Dhall, Formality

Specially created for Erlang deployment!
Real Monads Extracted from CoC to Erlang bytecode!

```
> 'Monad':
'[<=<]'/0     '[=<<]'/0     '[>=>]'/0     '[>>=]'/0     forM/0
forM_/0       join/0        mapM/0        mapM_/0       module_info/0
module_info/1  replicateM/0  replicateM_/0  sequence/0   sequence_/0
```

Formality

Interaction Networks based Evaluator (Run-time Fusion)
GPU Backend, Rust Implementation
Faster that GHC

id(1000000000(List<Bool>, map(Bool, Bool, not), list))

Flips every bit in a list of 100 bits, a billion times. It prints the correct output in 0.03s. You could increase that to beyound the number of stars in the universe, and it'd still output the correct result, instantly.

https://github.com/moonad/whitepaper