2nd International Conference on
Mathematical Methods & Computational Techniques in Science & Engineering

The Systems Engineering of Consistent Pure Language with Effect Type System for Certified Applications and Higher Languages

PTS-infinity

Murray Edwards College, University of Cambridge, UK, February 16-18, 2018
MMCTSE 2018 144 Maksym Sokhatskyi, Pavlo Maslianko

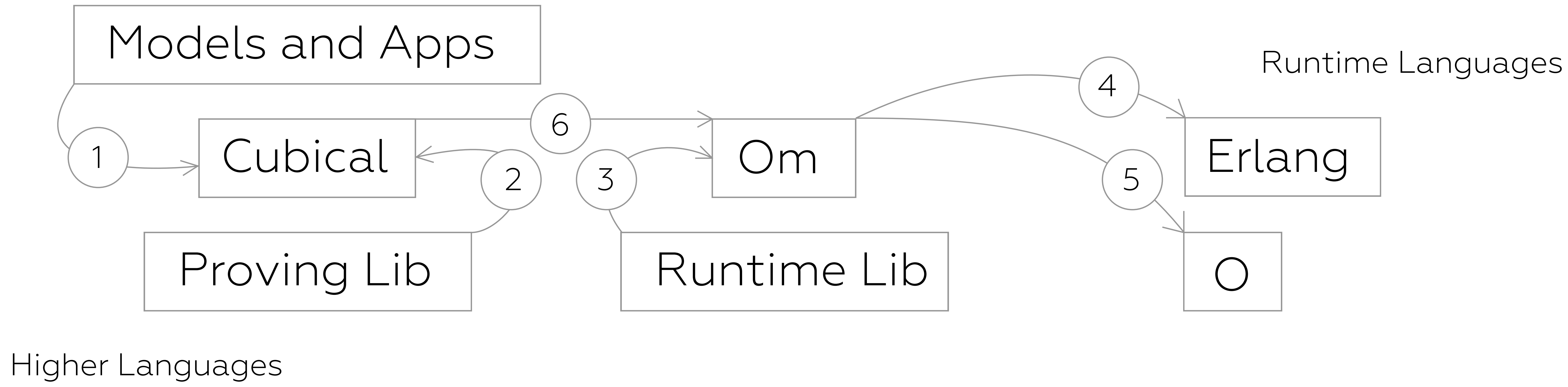
Abstract

PTS Language for Erlang virtual machines

Om — pure type system language as minimal core for evaluating Erlang programs for BEAM and LING virtual machines. The main motivation is trusted clean implementation in 260 lines of code that easily can be replicated by the authority site. The engineered system is built compatible with most lambda evaluators syntactically (Morte, Caramel) and semantically (Henk, CoC, PTS). Om language is member of languages family for proving and running verified applications.

Structure

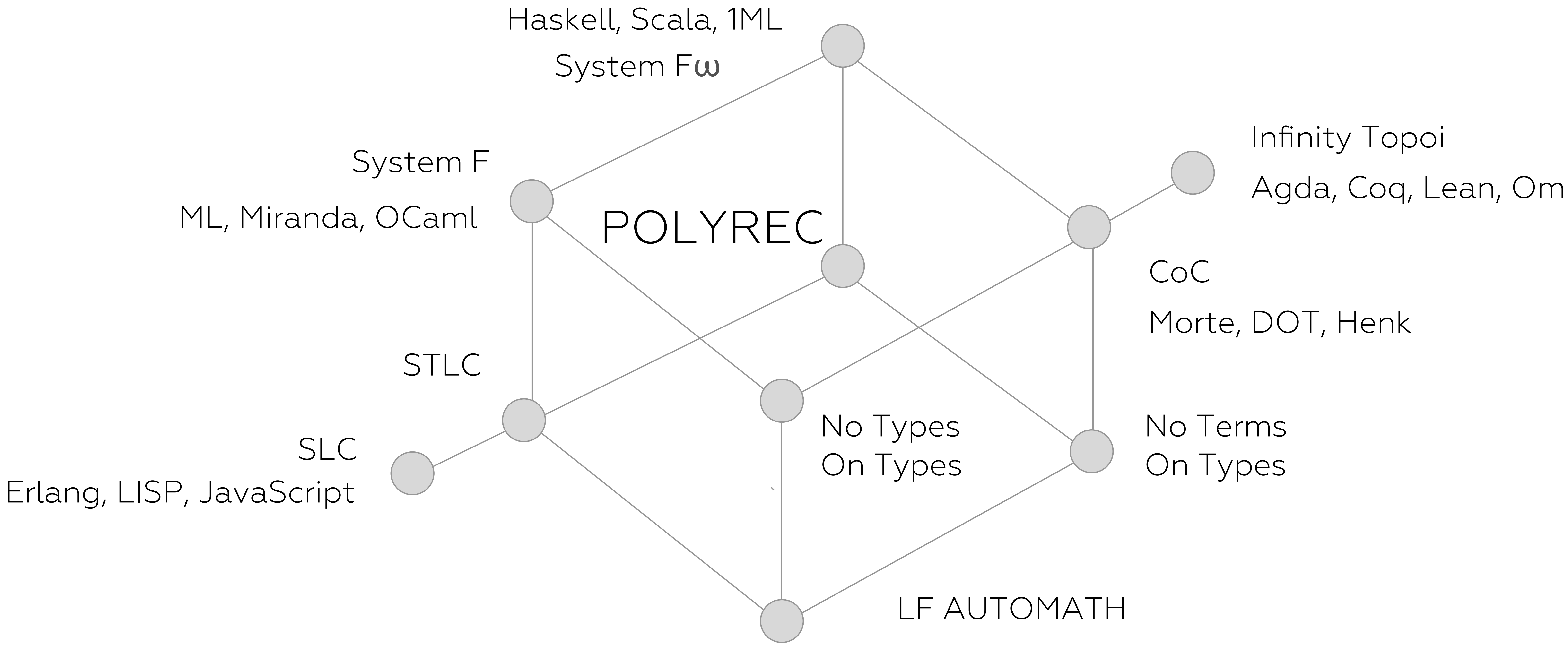
Models, Languages, Libraries, Applications



[3,4] cover the presented work, [1,2,5,6] cover the future works.

Programming Languages

in Extended Lambda Cube



Model Verification Process

From proving to Extraction, Linking and Running

1) Models

— IR/II
— Bohm
— HoTT

2) Core — Infinity Language

— Model Verification
— Normalization
— Bidirectional Checking

— Pure Type System (Om)
— Identity
— Induction
— Homotopy Interval [0,1]

3) Extraction

— LLVM
— Interpreters
— Detyping
— Optimization
— Linking

4) Runtimes

— O
— Erlang
— V8
— JVM

Runtime Languages

Through a Prism of Engineering

JIT	Interpreters	LLVM	Non-LLVM
LuaJIT	K	Rust	OCaml
V8	LING/Erlang	Julia	GHC
SpiderMonkey	O	C/C++	Spiral
EDGE			
JVM/HotSpot			
CLR			

Higher Languages

for Proving and Model Checking

Target	Class	Higher Language	Type Theory
CPU	Non-LLVM	Spiral	System F
JVM	JIT	Scala	System F-omega
GHC	Non-LLVM	Morte	CoC
Erlang	Interpreter	Om	PTS-infinity
O	Interpreter	Om	PTS-infinity
Haskell	Extract	Coq/Agda	CiC

MLTT — CT — Proof Theory

Syntax — Semantics — Logic

$x : A$ — x is a object of type A
$y = [\ x : A\ ]$ — x and y are definitionaly
equal objects of type A

Nat : U n — constant functor
List (A : U n) : U n — functor
List Nat : U n — constant functor

$\text{List } A = 1 \longrightarrow \text{List } A \longrightarrow \text{List } A \longleftarrow A : U$
$\text{nil} : 1 \longrightarrow \text{List } A$
$\text{cons} : A \longrightarrow \text{List } A \longrightarrow \text{List } A$

$\text{Nat} = 1 \longrightarrow N \longrightarrow N : U$
$\text{zero} : 1 \longrightarrow N$
$\text{succ} : N \longrightarrow N$

Pure Type System

Infinity Topoi

$U_0 : U_1 : U_2 : U_3 : \ldots \infty$

U_0 — propositions
U_1 — sets
U_2 — types
U_3 — sorts

$S\ (n : nat) = U\ n$
$A_1\ (n\ m : nat) = U\ n : U\ m$ where $m > n$ — cumulative
$R_1\ (m\ n : nat) = U\ m \longrightarrow U\ n : U\ (\max\ m\ n)$ — predicative

$A_2\ (n : nat) = U\ n : U\ (n + 1)$ — non-cumulative
$R_2\ (m\ n : nat) = U\ m \longrightarrow U\ n : U\ n$ — impredicative

$\text{Prop} = \text{Large}\ \Omega_0 = U_0$

$\boldsymbol{\Sigma} = \text{Large}\ \Omega_2 = U_2$

Pi Type

Inductive Type, AST, Logical Framework

data O_1 := U : nat → O_1
  | Var: Ident → O_1
  | App: O_1 → O_1 → O_1
  | Lambda: Binder → O_1 → O_1 → O_1
  | Arrow: O_1 → O_1 → O_1
  | Pi: name → O_1 → O_1 → O_1.

```
<> ::= #option
V ::= #identifier
S ::= ∗ < #number >
O ::= S | V | ( O ) | O O | O → O
      | λ (I: O) → O | ∀ (I: O) → O
```

```
record Pi (A: Type) :=
    intro: (A → Type) → Type :=
    fun: (B: A → Type) → ∀ (a: A) → B a → intro B
    app: (B: A → Type) → intro B → ∀ (a: A) → B a
    app-fun (B: A → Type) (f: ∀ (a: A) → B a): ∀ (a: A) → app (fun f) a = f a
    fun-app (B: A → Type) (p: intro B): fun (λ (a: A) → app p a) = p
```

Shifting

Modified version of De Bruin indeces

```
sh (:star, X)          N P → (:star, X)
   (:var, N, I)        N P → (:var, N, I+1) when I >= P
                           → (:var, N, I)
   (:remote, X)        N P → (:remote, X)
   (:pi, N, 0, I, O)   N P → (:pi, N, 0, sh I N P, sh O N (P+1))
   (:fn, N, 0, I, O)   N P → (:fn, N, 0, sh I N P, sh O N (P+1))
   (:app, L, R)        N P → (:app, L, R)
```

Substitution

Replacing variable occurance in terms

```
sub (:star, X)          N V L → (:star, X)
    (:var, N, L)        N V L → V
    (:var, N, I)        N V L → (:var, N, I−1) when I > L
    (:remote, X)        N V L → (:remote, X)
    (:pi, N, 0, I, O)   N V L → (:pi, N, 0, sub I N V L, sub O N (sh V N 0) L+1)
    (:pi, F, X, I, O)   N V L → (:pi, F, X, sub I N V L, sub O N (sh V F 0) L)
    (:fn, N, 0, I, O)   N V L → (:fn, N, 0, sub I N V L, sub O N (sh V N 0) L+1)
    (:fn, F, X, I, O)   N V L → (:fn, F, X, sub I N V L, sub O N (sh V F 0) L)
    (:app, F, A)        N V L → (:app, sub F N V L, sub A N V L)
```

Normalization

Replacing variable occurance in terms

```
type (:star, N)        D → (:star, N+1)
     (:var, N, I)      D → :true = proplists:defined N B, om:keyget N D I
     (:remote, N)      D → om:cache(typeND)
     (:pi, N, 0, I, O) D → (:star ,h (star (type I D)), star (type O [(N,norm I)|D]))
     (:fn, N, 0, I, O) D → let star (typeID), NI=norm I
                                in (:pi,N,0,NI,type(O,[(N,NI)|D]))
     (:app, F, A)      D → let T = type(F,D), (:pi,N,0,I,O) = T, :true = eq I (type AD)
                                in norm (subst O N A)
```

Type Inferenece

Type Checker

```
type (:star, N)         D → (:star, N+1)
     (:var, N, I)       D → :true = proplists:defined N B, om:keyget N D I
     (:remote, N)       D → om:cache(type N D)
     (:pi, N, 0, I, O)  D → (:star ,h(star(type I D)),star(type O [(N,norm I)|D]))
     (:fn, N, 0, I, O)  D → let star (type I D), NI = norm I
                                in (:pi,N,0,NI,type(O,[(N,NI)|D]))
     (:app, F, A)       D → let T = type (F,D), (:pi,N,0,I,O) = T, :true = eq I (type A D)
                                in norm (subst O N A)
```

Equality

Definitional, built into Type Checker

```
eq (:star ,N)         (:star ,N)          → true
   (:var,N,I)         (:var,(N,I))        → true
   (:remote ,N)       (:remote ,N)        → true
   (:pi,N1,0,I1,O1)   (:pi,N2,0,I2,O2)    → let :true = eq I1 I2
                                              in eq O1 (subst (shift O2 N1 0) N2 (:var,N1,0) 0)
   (:fn,N1,0,I1,O1)   (:fn,N2,0,I2,O2)    → let :true = eq I1 I2
                                              in eq O1 (subst (shift O2 N1 0) N2 (:var,N1,0) 0)
   (:app,F1,A1)       (:app,F2,A2)        → let :true = eq F1 F2 in eq A1 A2
   (A,                B)                  → (:error ,(:eq,A,B))
```

Language Usage

Erlang or UNIX shell

Om implemented as escript file accessible from UNIX command line, resembling the Erlang API for manipulating Om terms and contexts.

```
$ ./om help me
[{a,[expr],"to parse. Returns { , } or {error , }."} ,
{type,[term],"typechecks and returns type."},
{erase ,[ term ] ," to untyped term . Returns {   ,   }."} ,
{norm,[term],"normalize term. Returns term's normal form."},
{file ,[name],"load file as binary."},
{str ,[binary],"lexical tokenizer."},
{parse ,[ tokens ] ," parse given tokens into {   ,   } term ."} ,
{fst ,[{x,y}],"returns first element of a pair."},
{snd ,[{x,y}] ," returns second element of a pair ."} ,
{debug ,[ bool ] ," enable / disable debug output ."} , {mode,[name],"select metaverse folder."},
{modes ,[] ," list all metaverses ."}]
```

Language Usage

Erlang or UNIX shell

```
$ ./om print fst erase norm a "#List/Cons"

    \ Head
-> \ Tail
-> \ Cons
-> \ Nil
-> Cons Head (Tail Cons Nil)

ok
```

Applications: Sigma Types

Syntax and Model

```
data O₂ := O₁
        | Sigma: name → O₂ → O₂ → O₂
        | Pair: O₂ → O₂ → O₂
        | Fst: O₂ → O₂
        | Snd: O₂ → O₂.
```

$\Sigma\, x : A, B\, x : U$ — formation rule

$\mathrm{pair}\ (x : A)\ (y : B\, x)$ — introduction

$pr_1\, s : A$ — elimination

$pr_2\, s : B\, x$ — elimination

```
data  Sigma (A: Type) (P: A -> Type) (x: A): Type =
       intro: P x -> Sigma A P
```

Sigma Types

Typing and Introduction Rules

```
-- Sigma/@ \ (A: *)
-> \ (P: A -> *)
-> \ (n: A)
-> \/ (Exists: *)
-> \/ (Intro: A -> P n -> Exists)
-> Exists
```

```
-- Sigma/Intro \ (A: *)
-> \ (P: A -> *)
-> \ (x: A)
-> \ (y: P x)
-> \ (Exists: *)
-> \ (Intro: \/ (x:A) -> P x -> Exists)
-> Intro x y
```

Sigma Types

Dependent Eliminators

```
-- Sigma/fst \ (A: *)
-> \ (B: A -> *)
-> \ (n: A)
->\ (S: #Sigma/@ABn)
-> S A ( \(x: A) -> \(y: B n) -> x)
```

```
-- Sigma/snd \ (A: *)
-> \ (B: A -> *)
-> \ (n: A)
->\ (S: #Sigma/@ABn)
-> S (B n) ( \( : A) -> \(y: B n) -> y )
```

Equ Type a la Martin-Löf

```
record Id (A: Type): Type :=
    Id: A → A → Type
    refl (a: A): Id a a
    Predicate: ∀ (x,y: A) → Id x y → Type
    Forall (C: Predicate): ∀ (x,y: A) → ∀ (p: Id x y) → C x y p
    Δ (C: Predicate): ∀ (x: A) → C x x (refl x)
    axiom-J (C: Predicate): Δ C → Forall C
    computation (C: Predicate) (t: Δ C): ∀ (x: A) → (J C t x x (refl x)) ==> (t x) )

record Subst (A: Type): Type :=
    intro (P (a: A): Type) (a1,a2: A) : Id a1 a2 → P a1 → P a2 :=
        Id.axiom-J (λ a1 a2 p12 → P a1 → P a2))
```

Equ Type in Om

```
-- Equ/Refl \ (A: *)
-> \ (x: A)
-> \ (Equ: A -> A -> *)
-> \ (Refl: \/ (z: A) -> Equ z z)
-> Refl x
```

```
-- Equ/@
\ (A: *)
-> \ (x: A)
-> \ (y: A)
-> \/ (Equ: A -> A -> *)
-> \/ (Refl: \/ (z: A) -> Equ z z) -> Equ x y
```

Effects Protocol

Type Spec

```
String: Type = List Nat

data IO: Type =
        getLine: (String -> IO) -> IO
        putLine: String -> IO
        pure: () -> IO
```

```
-- IO/@
     \ (a : *)
-> \/ (IO : *)
-> \/ (GetLine : (#IO/data -> IO) -> IO)
-> \/ (PutLine : #IO/data -> IO -> IO)
-> \/ (Pure : a -> IO) -> IO
```

Replication

Church Encoded Identity Folding

```
-- IO/replicateM
    \ (n: #Nat/@)
-> \ (io: #IO/@ #Unit/@)
-> #Nat/fold n (#IO/@ #Unit/@)
               (#IO[>>] io)
               (#IO/pure #Unit/@ #Unit /Make)

— Nat/fold
#id #Nat/@
```

Runtime Recursion Sample

Recursion Elimination

```
-- Recursion
((#IO/replicateM #Nat/Five)
  ((((#IO/[>>=] #IO/data) #Unit/@) #IO/getLine)
    #IO/putLine))
```

Infinity I/O

Corecursion Fixpoint

```
-- IOI/@
      \   (r : *)
->  \/ (x : *)
-> (\/ (s : *)
-> s
-> (s -> #IOI/F r s) -> x)
-> x
```

```
-- IOI/F
    \ (a : *)
-> \ (State : *)
-> \/ (IOF : *)
-> \/ (PutLine : #IOI/data -> State -> IOF)
-> \/ (GetLine : (#IOI/data -> State) -> IOF)
-> \/(Pure :a->IOF)
-> IOF
```

Infinity I/O Construction

Corecursion Introduction

```
-- IOI/MkIO
    \ (r : *)
-> \ (s : *)
-> \ (seed: s)
-> \ (step: s -> #IOI/F r s)
-> \ (x : *)
-> \ (k : forall (s : *) -> s -> (s -> #IOI/F r s) -> x)
-> k s seed step
```

Infinity I/O Process

Corecursion Elimination

```
-- Corecursion
( \ (r: *1) -> ( (((#IOI/MkIO r) (#Maybe/@ #IOI/data))
        (#Maybe/Nothing #IOI/data))
    ( \ (m: (#Maybe/@ #IOI/data))
      -> (((((#Maybe/maybe #IOI/data) m)
            ((#IOI/F r) (#Maybe/@ #IOI/data)))
            ( \ (str: #IOI/data)
              -> ((((#IOI/putLine r ) (#Maybe/@ #IOI/data)) str)
                    (#Maybe/Nothing #IOI/data))))
                  (((#IOI/getLine r ) (#Maybe/@ #IOI / data ))
                    (#Maybe/Just #IOI/data))))))
```

Application 1. Logic

```
data Proper (A: Type) (R: A → A → Prop) (m: A): Prop := intro: R m m
data Inhabited (A: Type): Prop := intro: A → Inhabited A
data True: Prop := intro: () → True
data False: Prop := ()
data Eq (A : Type): A → A → Type := refl: ∀ (x: A) → Eq A x x
data Exists (A: Type): A → Type → Type := intro: ∀ (P: A → Type) →
                                              ∀ (x: A) → P x → Exists A P

record SKI := S : (p → q → r) → (p → q) → p → r
              K :  p → (q → p)
              I : p -> p
```

Application 2. Control

```
record pure (P: Type → Type) (A: Type): Type := return: P A
record functor (F: Type → Type) (A,B: Type): Type := map: (A → B) → F A → F B
record applicative (F: Type → Type) (A,B: Type): Type :=
         pure: pure F A
         functor: functor F A B
         ap: F (A → B) → F A → F B

record monad (F: Type → Type) (A,B: Type): Type :=
         pure: pure F A
         functor: functor F A B
         join: F (F A) → F B
```

Benchmarks

Operation	Type	Time
Pack/Unpack 1M	Inductive Nat	776407 us
Pack/Unpack 1M	Inductive List	1036461 us
Pack/Unpack 1M	Erlang/OTP	148084 us
Type Checking	Om ShadowTrans	4.972s
Type Checking	Morte ShadowTrans	57.867s

Components

Erlang/OTP Modules

File	LOC	Description
om_tok	54	Handcoded Tokenizer
om_parse	81	Inductive AST Parser
om_type	60	Term normalization and typechecking
om_erase	36	Delete information about types
om_extract	34	Extract Erlang Code

Runtime Library

Simple Types shipped with Compiler

Bool	Lazy	Exec	String
Cmd	Leibnitz	Nat	Sum
Equ	List	Path	Unit
Either	Maybe	Prod	Vector
Frege	Mon	Prop	
IO	Monad	Sigma	
IOI	Monoid	Simple	

Higher Language

```
<> ::= #option
[] ::= #list

| ::= #sum
1 ::= #unit
I ::= #identifier
U ::= Type < #nat >
T ::= 1 | ( I : O ) T
F ::= 1 | I : O = O , F
B ::= 1 | [ | I [ I ] → O ]

O ::= I | ( O ) | U
        | O → O
        | fun ( I : O ) → O
        | snd O
        | J O O O O O
        | ( I : O ) * O
        | data I T : O := T
        | O O
        | fst O
        | id O O O
        | let F in O
        | ( I : O ) → O
        | record I T : O := T
        | case O B
```

Bibliography

1. Barendregt. The Lambda Calculus with Types http://5ht.co/pts.pdf
2. Martin-Löf. Intuitionistic Type Theory http://5ht.co/mltt.pdf
3. Awodey. Category Theory http://5ht.co/cat.pdf
4. Hermida, Jacobs. Fibrations with indeterminates http://5ht.co/completeness.pdf
5. Jacobs. Categorical Logic http://5ht.co/fibrations.pdf
6. Streicher. The groupoid interpretation of type theory http://5ht.co/groupoid.pdf
7. Voevodsky et all. Homotopy Type Theory http://5ht.co/hott.pdf
8. Huber, Coquand. Cubical Type Theory http://5ht.co/cubicaltt.pdf