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

Maxim Sokhatsky[1,2,a),c)] and Pavlo Maslianko[1,b)]

[1] *National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnical Institute"*
[2] *Groupoid Infinity*

[a)] maxim@synrc.com
[b)] mppdom@i.ua
[c)] URL: https://groupoid.space

Abstract. This paper presents design of **Om** language and implementation of its type checker and bytecode extraction to Erlang. Om is an intermediate language based on a pure type system with infinite number of universes, so it is known to be consistent in dependent type theory. The typechecker can be switched between predicative and impredicative hierarchies of universes. The need to natively support Erlang platform dictated the look and feel of this work. This system is expected to be usable as trusted core for certified applications which could be run inside Erlang virtual machines LING and BEAM. The syntax is compatible with Morte language and supports its base library, however it extends the indexed universes. We show how to program in this environment and link with Erlang inductive and coinductive free structures. A very basic prelude library is shipped as a part of the work. We briefly describe the top-level language which compiles to pure type system core. As the results we will show lambda evaluation performance on BEAM virtual machine.

Introduction

As a part of verification and validation process, according to IEEE[1] standard and ESA[2] regulatory document, exists a lot of tools and approaches. Most advanced techniques involves mathematical languages and notations. The age of verified math was started by de Bruin AUTOMATH prover and Martin-Löf[1] type theory and today we have Coq, Agda, Lean, Idris, F* languages based on Calculus of Inductive Constructions or CiC[2]. The core of CiC is Calculus of Constructions or CoC[3]. The further development leds to Lambda Cube[4] and Pure Type Systems (Henk[5],Morte[3]). Pure Type Systems are custom languages based on CoC with single Pi-type and possible other extensions. The known extensions are ECC, ECC with Inductive Types[6], K-rules[7]. The main motivation of Pure Type Systems is an easy reasoning about core, strong normalization and trusted external verification due to compact typecheckers. Imagine one can implement their own typechecker to run certified programs retrieved over untrusted channels. The applications of such minimal cores are: 1) Blockchain smart-contract languages, 2) certified applications kernels, 3) payment processing, etc.

Generating Trusted Programs

According to Curry-Howard correspondence inside Martin-Löf Type Theory[1] proofs or cerfificates are lambda terms of particular types or specifications. As both specifications and implemntations are done in typed language with de-

[1] IEEE Std 1012-2016 — V&V Software verification and validation
[2] ESA PSS-05-10 1-1 1995 – Guide to software verification and validation
[3] Gabriel Gonzalez. Haskell Morte Library

pendent types we can extract target implementation of certified program just in any programming language. These languages could be so primitive as untyped lambda calculus which manifests (usually implements) as untyped interpreters (JavaScript, Erlang, PyPy, LuaJIT, K). The most advanced usage is code generation to higher-level languages such as C++ and Rust (whish is already language with trusted features on memory, variable accessing, linear types, etc.). In this work we presents a simple code extraction to Erlang programming language as target interpreter. However we have working also on C++ and Rust targets as well.

System Architecture

Om as a programming language has a core type system, the **PTS**$^{\infty}$ — the pure type system with infinite number of universes. This type system represents the core of the language. Higher languages forms a set of front-ends to this core. Here are example of possible languages: 1) Language for inductive reasning, based on CiC with extensions; 2) Homotopy Core with interval [0,1] for proving J and funExt; 3) Stream Calculus for deep stream fusion (Futhark); 3) Pi-caclulus for linear types, coinductive reasoning and runtime modeling (Erlang, Ling, Rust). These languages desugar to **PTS**$^{\infty}$ as an intermediate language before extracting to target language[4].

Not all terms from higher languages could be desugared to PTS. As was shown by Geuvers[9] we cannot build induction principle inside PTS, we need a fixpoint extension to PTS. And also we cannot build the J and funExt terms. But still PTS is very powerful, it's compatible with System F libraries. The properties of that libraries could be proven in Higher Languages with Induction and/or [0,1] Homotopy Core. Then runtime part could be refined to PTS, extracted to target and runned in environment.

We see two levels of extensions to PTS core: 1) Inductive Types support; 2) Homotopy Core with [0,1] and its eliminators. We will touch a bit this topic in the last section of this document.

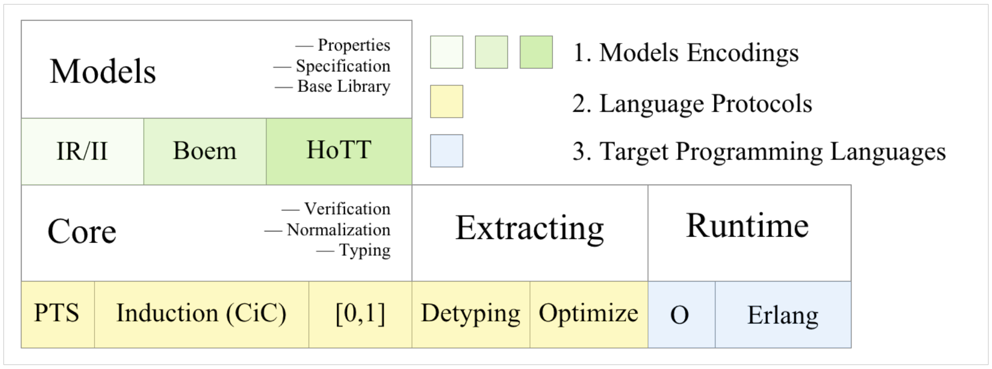

FIGURE 1. Process of Model Verifications

Place among other Languages

The product is an regular Erlang/OTP application, that provides dependent language services to the Erlang environment: 1) typechecking; 2) normalization; 3) extraction. All parts of **Om** compiler are written in Erlang language and target/runtime language is Erlang.

- Level 0 — certified vectorized interpreter
- **Level 1 — consistent pure type system for typechecking and normalization**
- Level 2 — higher language for theorem proving and models property checking

[4] Note that extracting from [0,1] Homotopy Core is an open problem

TABLE 1. List of languages, tried as verification targets

Target Language	Class	Higher Language	Type Theory
C++	compiler/native	HNC	System F
Rust	compiler/native	HNC	System F
JVM	interpreter/native	Java	F-sub^5
JVM	interpreter/native	Scala	System F-omega
GHC Core	compiler/native	Haskell	System D
GHC Core	compiler/native	Morte	CoC
Haskell	compiler/native	Coq	CiC
OCaml	compiler/native	Coq	CiC
BEAM	**interpreter**	**Om**	**PTS$^\infty$**
O	interpreter	Om	PTS$^\infty$
K	interpreter	Q	Applicative
PyPy	interpreter/native	N/A	ULC
LuaJIT	interpreter/native	N/A	ULC
JavaScript	interpreter/native	PureScript	System F

Consistent Pure Type System as Intermediate Language

The **Om** language is a dependently typed lambda calculus **PTS$^\infty$**, an extension of Coquand' Calculus of Constructions[3] with predicative hierarchy of indexed universes. There is no fixpoint axiom, so there is no infinite term dependence, the theory is fully consistent and has strong normalization property.

All terms respect ranking **Axioms** inside sequence of universes **Sorts** and complexity of the dependent term is equal to maximum complexity of term and its dependency **Rules**. The universe system is completely described by the following PTS notation (due to Barendregt[4]):

$$\begin{cases} Sorts = Type.\{i\},\ i : Nat \\ Axioms = Type.\{i\} : Type.\{inc\ i\} \\ Rules = Type.\{i\} \rightsquigarrow Type.\{j\} : Type.\{max\ i\ j\} \end{cases}$$

The **Om** language is based on Henk languages described first by Erik Meijer and Simon Peyton Jones in 1997[5]. Leter on in 2015 Morte impementation of Henk design appeared in Haskell, using Boem-Berrarducci encoding of non-recursive lambda terms. It is based only on one type constructor Π, its intro λ and apply eliminator, infinity number of universes, and β-reduction. The design of Om language resemble Henk and Morte both in design and in implementation. This language indended to be small, concise, easy provable and able to produce verifiable peace of code that can be distributed over the networks, compiled at target with safe trusted linkage.

BNF and AST

Om syntax is compatible with CoC presented in Morte and Henk languages. However it has extension in a part of specifying universe index as a **Nat** number. Traditionally we present the language in Backus-Naur form. Equivalent AST tree encoding from the right side.

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

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

Universes

As **Om** has infinite number of universes it should include metatheoretical **Nat** inductive type in its core. **Om** supports predicative and impredicative hierarchies.

$$U_0 : U_1 : U_2 : U_3 : ...$$

Where U_0 — propositions, U_1 — sets, U_2 — types and U_3 — kinds, etc.

$$\frac{}{Nat} \tag{I}$$

$$\frac{o : Nat}{Type_o} \tag{S}$$

You may check if a term is an universe with the star function. If argument is not an universe it returns $\{error, \_\}$.

```
star (:star,N) → (:ok, N)
star _ → (:error, ”*”)
```

Predicative Universes

All terms obey the **Axioms** ranking inside the sequence of **Sorts** universes, and the complexity **Rules** of the dependent term is equal to a maximum of the term's complexity and its dependency. Note that predicative universes are incompatible with Church lambda term encoding. You choose either predicative or impredicative universes by a typechecker parameter.

$$\frac{i : Nat, j : Nat, i < j}{Type_i : Type_j} \tag{A_1}$$

$$\frac{i : Nat, j : Nat}{Type_i \to Type_j : Type_{max(i,j)}} \tag{R_1}$$

Impredicative Universes

Propositional contractible bottom space is the only available extension to predicative hierarchy which doesn't lead to inconsistency. However there is another option to have infinite impredicative hierarchy.

$$\frac{i : Nat}{Type_i : Type_{i+1}} \tag{A_2}$$

$$\frac{i : Nat, \quad j : Nat}{Type_i \to Type_j : Type_j} \tag{R_2}$$

Hierarchy Switching

Function **h** returns the target Universe of B term dependence on A. There are two dependence rules known as the predicative one and the impredicative one which return max universe or universe of last term respectively.

```
dep A B impredicative → B
dep A B predicative   → max A B

h A B → dep A B om:hierarchy(impredicative)
```

Contexts

The contexts model a dictionary with variables for typechecker. It can be typed as list of pairs or **List Sigma**. The elimination rule is not given here as in our implementation the whole dictionary is destroyed after typechecking.

$$\frac{}{\Gamma : Context} \qquad \text{(Ctx-formation)}$$

$$\frac{\Gamma : Context}{Empty : \Gamma} \qquad \text{(Ctx-intro}_1\text{)}$$

$$\frac{A : Type_i, \quad x : A, \quad \Gamma : Context}{(x : A) \ \vdash \ \Gamma : Context} \qquad \text{(Ctx-intro}_2\text{)}$$

Single Axiom Language

This langauge is called one axiom language (or pure) as eliminator and introduction rules inferred from type formation rule. The only computation rule of Pi type is called beta-reduction. Computational rules of language are called operational semantics and establish equality on substitution and application to lambda. Operational semantics in that way defines the rewrite rules of computations.

$$\frac{x : A \vdash B : Type}{\Pi\ (x : A) \to B : Type} \qquad \text{(}\Pi\text{-formation)}$$

$$\frac{x : A \vdash b : B}{\lambda\ (x : A) \to b : \Pi\ (x : A) \to B} \qquad \text{(}\lambda\text{-intro)}$$

$$\frac{f : (\Pi\ (x : A) \to B) \quad a : A}{f\ a : B\ [a/x]} \qquad \text{(}App\text{-elimination)}$$

$$\frac{x : A \vdash b : B \quad a : A}{(\lambda\ (x : A) \to b)\ a = b\ [a/x] : B\ [a/x]} \qquad \text{(}\beta\text{-computation)}$$

$$\frac{\pi_1 : A \quad u : A \vdash \pi_2 : B}{[\pi_1/u]\ \pi_2 : B} \qquad \text{(subst)}$$

The theorems (specification) of PTS could be embedded in itself and used as Logical Framework for the Pi type. Here is example in higher language.

```
record Pi (A: Type) :=
        (intro:  (A → Type) → Type)
        (lambda: (B: A → Type) → pi A B → intro B)
        (app:    (B: A → Type) → intro B → pi A B)
        (applam: (B: A → Type) (f: pi A B) → (a: A) →
                 Path (B a) ((app B (lambda B f)) a) (f a))
        (lamapp: (B: A → Type) (p: intro B) →
                 Path (intro B) (lambda B (λ (a:A) → app B p a)) p)
```

The proofs intentionally left blank, as it proofs could be taken from various sources. The equalities of computational semantics presented here as **Path** types in higher language.

We extend the $PTS\infty$ with remote AST node which means remote file loading from trusted storage, anyway this will be checked by typechecker. We deny recursion over remote node. We also add index to var for simplified de Bruijn indexes, we allow overlapped names with tags, incremented on each new occurance.

```
data om = star             (n: nat)
        | var    (n: name) (n: nat)
        | remote (n: name) (n: nat)
        | app                       (f a: om)
        | lambda (x: name)          (d c: om)
        | arrow                     (d c: om)
        | pi     (x: name)          (d c: om)
```

Functions

Func returns true if the argument is a functional space. Otherwise it returns {*error*, _}.

```
func ((:forall,),(I,O)) → true
func T                  → (:error,(:forall,T))
```

Variables

Var returns true if the var *N* is defined in dictionary *B*. Otherwise it returns {*error*, _}.

```
var N B             → var N B (proplists:is_defined N B)
var N B true        → true
var N B false       → (:error,(”free var”,N,proplists:get_keys(B)))
```

Shift

Shift renames var N in B. Renaming means adding an 1 to an index of that name.

```
sh (:var,(N,I)),N,P) when I>=P    → (var,(N,I+1))
sh ((:forall,(N,0)),(I,O)),N,P)   → ((:forall,(N,0)),sh I N P,sh O N P+1)
sh ((:lambda,(N,0)),(I,O)),N,P)   → ((:lambda,(N,0)),sh I N P,sh O N P+1)
sh (Q,(L,R),N,P)                  → (Q,sh L N P,sh R N P)
sh (T,N,P)                        → T
```

Substitution

```
sub Term Name Value                      → sub Term Name Value 0
sub (:arrow,           (I,O)) N V L → (:arrow,           sub I N V L,sub O N V L);
sub ((:forall,(N,0)),(I,O)) N V L → ((:forall,(N,0)),sub I N V L,sub O N(sh V N 0)L+1)
sub ((:forall,(F,X)),(I,O)) N V L → ((:forall,(F,X)),sub I N V L,sub O N(sh V F 0)L)
sub ((:lambda,(N,0)),(I,O)) N V L → ((:lambda,(N,0)),sub I N V L,sub O N(sh V N 0)L+1)
sub ((:lambda,(F,X)),(I,O)) N V L → ((:lambda,(F,X)),sub I N V L,sub O N(sh V F 0)L)
sub (:app,             (F,A)) N V L → (:app,sub F N V L,sub A N V L)
sub (:var,             (N,L)) N V L → V
sub (:var,             (N,I)) N V L when I>L → (:var,(N,I-1))
sub T                         _ _ _ → T.
```

Type Checker

For sure in a pure system we should be careful with **:remote** AST node. Remote AST nodes like **#List/Cons or #List/map** are remote links to files. So using trick one should desire circular dependency over **:remote**. This is denied in the system. The same notes apply to normalization and definitional equality.

```
type (:star,N)                  D → (:star,N+1)
type (:var,(N,I))               D → let true = var N D in keyget N D I
type (:remote,N)                D → cache type N D
type (:arrow,(I,O))             D → (:star,h(star(type I D)),star(type O D))
type ((:forall,(N,0)),(I,O))    D → (:star,h(star(type I D)),star(type O [(N,norm I)|D]))
type ((:lambda,(N,0)),(I,O))    D → let star (type I D)
                                        NI = norm I
                                     in ((:forall,(N,0)),(NI,type(O,[(N,NI)|D])))
type (:app,(F,A))               D → let T = type(F,D), true = func T,
                                        ((:forall,(N,0)),(I,O)) = T, Q = type A D
                                        true = eq I Q in norm (subst O N A)
```

Normalization

Normalization performs substitutions on applications to functions (beta-reduction), searching over all function spaces, performing recursive normalization over the lambda and pi nodes.

```
norm :none                      → :none
norm :any                       → :any
norm (:app,(F,A))               → case norm F of
                                   ((:lambda,(N,0)),(I,O)) → norm (subst O N A)
                                                        NF → (:app,(NF,norm A)) end
norm (:remote,N)                → cache (norm N [])
norm (:arrow,           (I,O))  → ((:forall,(”_”,0)),(norm I,norm O))
norm ((:forall,(N,0)),(I,O))    → ((:forall,(N,0)),   (norm I,norm O))
norm ((:lambda,(N,0)),(I,O))    → ((:lambda,(N,0)),   (norm I,norm O))
norm T                          → T
```

Definitional Equality

Definitional Equality simply checks the equality of Erlang terms.

```
eq ((:forall,(”_”,0)), X) (:arrow,Y)      → eq X Y
eq (:app,(F1,A1))          (:app,(F2,A2)) → let true = eq F1 F2 in eq A1 A2
eq (:star,N)               (:star,N)      → true
eq (:var,(N,I))            (:var,(N,I))   → true
eq (:remote,N)             (:remote,N)    → true
eq ((:farall,(N1,0)),(I1,O1))
   ((:forall,(N2,0)),(I2,O2)) →
   let true = eq I1 I2
    in eq O1 (subst (shift O2 N1 0) N2 (:var,(N1,0)) 0)
eq ((:lambda,(N1,0)),(I1,O1))
   ((:lambda,(N2,0)),(I2,O2)) →
   let true = eq I1 I2
    in eq O1 (subst (shift O2 N1 0) N2 (:var,(N1,0)) 0)
eq (A,B)                             → (:error,(:eq,A,B))
```

Language Usage

In this section we will lift PTS system to MLTT system by defining **Sigma** and **Equ** types using only **Pi** type. We will use Böhm inductive dependent encoding[15].

Sigma Type

Here we will show you some examples of **Om** Language usage. The PTS system is extremely powerful even without **Sigma** type. But we can encode **Sigma** type similar how we encode **Prod** tuple pair in Böhm encoding. Let’s formulate **Sigma** type as an inductive type in higher language.

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

The Sigma-type with its eliminators appears as example staring from Aaron Stump[8]. Here we will show desugaring to **PTS**$^{\infty}$. In the comments you can find the AUTOMATH-like version of terms[6].

[6] Pi type is denoted as [x: A] B x and lambda function is denoted as (x: A) M x

```
-- Sigma/@: (A:U) (P:A->U) (x:A) [exists:U] [intro:A->P x->e] e
   \ (A: *)
-> \ (P: A -> *)
-> \ (n: A)
-> \/ (Exists: *)
-> \/ (Intro: A -> P n -> Exists)
-> Exists

-- Sigma/Intro: (A:U) (P:A->U) (x:A) (y:P x) (e:U) (i:A->P x->e) e = i x y
   \ (A: *)
-> \ (P: A -> *)
-> \ (x: A)
-> \ (y: P x)
-> \ (Exists: *)
-> \ (Intro: \/ (x:A) -> P x -> Exists)
-> Intro x y

-- Sigma/fst: (A:U) (P:A->U) (x:A) (s:sig A P x) A = s A (z:A) (y:P x) z
   \ (A: *)
-> \ (B: A -> *)
-> \ (n: A)
-> \ (S: #Sigma/@ A B n)
-> S A ( \(x: A) -> \(y: B n) -> x)

-- Sigma/snd: (A:U) (P:A->U) (x:A) (s:sig A P x) P x = s (P x) ((z:A) (y:P x) y)
   \ (A: *)
-> \ (B: A -> *)
-> \ (n: A)
-> \ (S: #Sigma/@ A B n)
-> S (B n) ( \(_: A) -> \(y: B n) -> y )
```

```
> om:fst(om:erase(om:norm(om:a(”#Sigma/test.fst”)))).
{{λ,{’Succ’,0}},
 {any,{{λ,{’Zero’,0}},{any,{var,{’Zero’,0}}}}}}}
```

For using **Sigma** type for Logic purposes one should change the home Universe of the type to **Prop**. Here it is:

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

Equality Type

Another example of expressiveness is Equality type a la Martin-Löf.

```
data Equ (A: Type): A -> A -> Type :=
     (refl (a: A): Equ A a a)

-- Equ/@: (A: U) (x: A) (y: A) [equ: A -> A -> U] [refl: [z:A] equ z z] equ x y
   \ (A: *)
-> \ (x: A)
-> \ (y: A)
-> \/ (Equ: A -> A -> *)
-> \/ (Refl: \/ (z: A) -> Equ z z)
-> Equ x y

-- Equ/Refl: (A: U) (x: A) (equ: A -> A -> U) (refl: [z: A] equ z z) refl x
```

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

You cannot construct a lambda that will check different values of A type for equality, however you may want to use built-in definitional equality and normalization feature of typechecker to actually compare two values:

```
> om:print(
  om:type(
  om:a(”(\\ (z: #Equ/@ #Nat/@ #Nat/One #Nat/One) -> #Prop/True)”++
       ” (#Equ/Refl #Nat/@ (#Nat/Succ #Nat/Zero))”))).
    \/ (True: *0)
-> \/ (Intro: True)
-> True
ok

> om:print(
  om:type(
  om:a(”(\\ (z: #Equ/@ #Nat/@ #Nat/One #Nat/One) -> #Prop/True)”++
       ” (#Equ/Refl #Nat/@ #Nat/Zero)”))).
** exception error: no match of right hand side value
    {error,{”==”,
            {app,{{var,{'Succ',0}},{var,{'Zero',0}}}},
            {var,{'Zero',0}}}}
```

Effect Type System

This work is expected to compile to a limited number of target platforms. For now Erlang, Haskell and LLVM are awaiting. Erlang version is expected to be useful both on LING and BEAM Erlang virtual machines. This language allows you to define trusted operations in System F and extract this routines to Erlang/OTP platform and plug as trusted resourses. As example we also provide infinite coinductive process creation and inductive shell that linked to Erlang/OTP IO functions directly.

IO protocol. We can construct in pure type system the state machine based on (co)free monads driven by **IO/IOI** protocols. Assume that **String** is a **List Nat** (as it is in Erlang natively), and three external constructors: getLine, putLine and pure. We need to put correspondent implementations on host platform as parameters to perform the actual IO.

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

Infinity I/O Type

Infinity I/O Type Spec.

```
-- IOI/@: (r: U) [x: U] [[s: U] -> s -> [s -> #IOI/F r s] -> x] x
    \ (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

-- 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

-- IOI/data
#List/@ #Nat/@
```

Infinite I/O Sample Program.

```
-- Morte/corecursive
( \ (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))))))
```

Erlang Coinductive Bindings.

```
copure() ->
    fun (_) -> fun (IO) -> IO end end.

cogetLine() ->
    fun(IO) -> fun(_) ->
        L = ch:list(io:get_line(”> ”)),
        ch:ap(IO,[L]) end end.

coputLine() ->
    fun (S) -> fun(IO) ->
        X = ch:unlist(S),
        io:put_chars(”: ”++X),
        case X of ”0\n” -> list([]);
                     _ -> corec() end end end.

corec() ->
    ap(’Morte’:corecursive(),
        [copure(),cogetLine(),coputLine(),copure(),list([])]).
```

```
> om_extract:extract(”priv/normal/IOI”).
ok
> Active: module loaded: {reloaded,’IOI’}

> om:corec().
> 1
: 1
> 0
: 0
#Fun<List.3.113171260>
```

I/O Type

I/O Type Spec.

```
-- IO/@
   \ (a : *)
-> \/ (IO : *)
-> \/ (GetLine_ : (#IO/data -> IO) -> IO)
-> \/ (PutLine_ : #IO/data -> IO -> IO)
-> \/ (Pure_ : a -> IO)
-> IO

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

Guarded Recursion I/O Sample Program.

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

Erlang Inductive Bindings.

```
pure() ->
     fun(IO) -> IO end.

getLine() ->
     fun(IO) -> fun(_) ->
          L = ch:list(io:get_line(”> ”)),
          ch:ap(IO,[L]) end end.

putLine() ->
     fun (S) -> fun(IO) ->
          io:put_chars(”: ”++ch:unlist(S)),
          ch:ap(IO,[S]) end end.

rec() ->
     ap(’Morte’:recursive(),
          [getLine(),putLine(),pure(),list([])]).
```

Here is example of Erlang/OTP shell running recursive example.

```
> om:rec().
> 1
: 1
> 2
: 2
> 3
: 3
> 4
: 4
> 5
: 5
#Fun<List.28.113171260>
```

Higher Language with Inductive Types

Despite we can encode inductive types in PTS, the best usage of inductive types comes with recursors and fixpoint type that allow recursive typecheck for special cases. As was shown by Herman Geuvers[9] the induction principle in not derivable in second-order dependent type theory. However there a lot of ways doing this. For example we can built in induction principal into the core for every defined inductive type. We even can allow recursive type check for only terms of induction principle, which have recursion base — that approach was successfully established by Peng Fu and Aaron Stump[8]. In any case for derivable induction principle in PTS$^{\infty}$ we need to have fixpoint somehow in the core.

So called Calculus of Inductive Constructions[2] is used as a top language on top of PTS to reason about inductive types. Here we will show you a sketch of such inductive language model which intendent to be a language extension to PTS system. CiC is allowing fixpoint for any terms, and base checking should be performed during typechecking such terms.

Our future top language is a general purpose functional language with Π and Σ types, recursive algebraic types, higher order functions, corecursion, and a free monad to encode effects. It compiles to a small MLTT core of dependent type system with inductive types and equality. It also has an Id-type (with its recursor) for equality reasoning, Case analysis over inductive types.

BNF

```
<> ::= #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               | O O
         | fun ( I : O ) → O   | fst O
         | snd O               | id O O O
         | J O O O O O         | let F in O
         | ( I : O ) ∗ O       | ( I : O ) → O
         | data I T : O := T   | record I T : O := T
         | case O B
```

AST

The AST of higher language is formally could be defined using itself. Here you can find telescopes (context lists), split and its bracnhes, inductive data definitions.

```
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
   = star                           (n: nat)
   | var    (n: name)               (i: nat)
   | app              (f a: ind)
   | lambda (x: name) (d c: ind)
   | pi     (x: name) (d c: ind)
   | sigma  (n: name) (a b: ind)
   | arrow            (d c: ind)
   | pair             (a b: ind)
   | fst              (p:   ind)
   | snd              (p:   ind)
   | id               (a b: ind)
   | idpair           (a b: ind)
   | idelim           (a b c d e: ind)
   | data_  (n: name) (t: tele ind) (labels:   list (label ind))
   | case   (n: name) (t: ind)      (branches: list (branch ind))
   | ctor   (n: name)               (args:     list ind)
```

The Erlang version of parser encoded with OTP library **yecc** which implements LALR-1 grammar generator. This version resembles the model and slightly based on cubical type checker by Mortberg[10] and could be reached at Github repository[7].

Inductive Type Encoding

There are a number of inductive type encodings: 1) Commutative square encoding of F-algebras by Hinze, Wu[11]; 2) Inductive-recursive encoding, algebraic type of algebraic tupes, inductive famility encoding by Dagand[12]; 3) Encoding with motives inductive-inductive definition, also with inductive families, for modeling quotient types by Altenkirch, Kaposi[13]; 4) Henry Ford encoding or encoding with Ran,Lan-extensions by Hamana, Fiore[14]; 5) Church-compatible Böhm-Berarducci encoding Böhm, Berarducci[15]. Om is shipped with base library in Church encoding and we already gave example of IO system encoded with runtime linkage. We give here simple calculations behind this theory.

Polynomial Functors

Least fixed point trees are called well-founded trees. They encode polynomial functors.

Natural Numbers: $\mu\, X \to 1 + X$
List A: $\mu\, X \to 1 + A \times X$
Lambda calculus: $\mu\, X \to 1 + X \times X + X$
Stream: $\nu\, X \to A \times X$
Potentialy Infinite List A: $\nu\, X \to 1 + A \times X$
Finite Tree: $\mu\, X \to \mu\, Y \to 1 + X \times Y = \mu\, X = List\; X$

As we know there are several ways to appear for a variable in a recursive algebraic type. Least fixpoint are known as an recursive expressions that have a base of recursion In Chuch-Böhm-Berarducci encoding type are store as as non-recursive definitions of their right folds. A fold in this encoding is equal to id function as the type signature contains its type constructore as parameters to pure function.

[7] http://github.com/groupoid/infinity/tree/master/priv

List Example

The data type of lists over a given set A can be represented as the initial algebra $(\mu L_A, in)$ of the functor $L_A(X) = 1 + (A \times X)$. Denote $\mu L_A = List(A)$. The constructor functions $nil : 1 \to List(A)$ and $cons : A \times List(A) \to List(A)$ are defined by $nil = in \circ inl$ and $cons = in \circ inr$, so $in = [nil, cons]$. Given any two functions $c : 1 \to C$ and $h : A \times C \to C$, the catamorphism $f = \llparenthesis [c, h] \rrparenthesis : List(A) \to C$ is the unique solution of the simultaneous equations:

$$\begin{cases} f \circ nil = c \\ f \circ cons = h \circ (id \times f) \end{cases}$$

where $f = foldr(c, h)$. Having this the initial algebra is presented with functor $\mu(1 + A \times X)$ and morphisms sum $[1 \to List(A), A \times List(A) \to List(A)]$ as catamorphism. Using this encoding the base library of List will have following form:

$$\begin{cases} list = \lambda\ ctor \to \lambda\ cons \to \lambda\ nil \to ctor \\ cons = \lambda\ x\ \to \lambda\ xs \to \lambda\ list \to \lambda\ cons \to\ \lambda\ nil \to cons\ x\ (xs\ list\ cons\ nil) \\ nil = \lambda\ list \to \lambda\ cons \to \lambda\ nil \to nil \end{cases}$$

Here traditionally we show the **List** definition in higher language and desugared version in **Om**.

```
  data List: (A: *) → * :=
        (Cons: A → list A → list A)
        (Nil: list A)

-- List/@
   \ (A : *)
-> \/ (List: *)
-> \/ (Cons: \/ (Head: A) -> \/ (Tail: List) -> List)
-> \/ (Nil: List)
-> List

-- List/Cons
   \ (A: *)
-> \ (Head: A)
-> \ (Tail:
      \/ (List: *)
   -> \/ (Cons: \/ (Head: A) -> \/ (Tail: List) -> List)
   -> \/ (Nil: List)
   -> List)
-> \ (List: *)
-> \ (Cons:
      \/ (Head: A)
   -> \/ (Tail: List)
   -> List)
-> \ (Nil: List)
-> Cons Head (Tail List Cons Nil)

-- List/Nil
   \ (A: *)
-> \ (List: *)
-> \ (Cons:
      \/ (Head: A)
   -> \/ (Tail: List)
   -> List)
-> \ (Nil: List)
-> Nil
```

```
record lists: (A B: ∗) :=
       (len: list A → integer)
       ((++): list A → list A → list A)
       (map: (A → B) → (list A → list B))
       (filter: (A → bool) → (list A → list A))
```

$$\begin{cases} foldr = \llparenthesis [f \circ nil, h] \rrparenthesis, f \circ cons = h \circ (id \times f) \\ len = \llparenthesis [zero, \lambda\, a\, n \rightarrow succ\, n] \rrparenthesis \\ (++) = \lambda\, xs\, ys \rightarrow \llparenthesis [\lambda(x) \rightarrow ys, cons] \rrparenthesis (xs) \\ map = \lambda\, f \rightarrow \llparenthesis [nil, cons \circ (f \times id)] \rrparenthesis \end{cases}$$

$$\begin{cases} len = foldr\ (\lambda\ x\ n \rightarrow succ\ n)\ 0 \\ (++) = \lambda\ ys \rightarrow foldr\ cons\ ys \\ map = \lambda\ f \rightarrow foldr\ (\lambda x\ xs \rightarrow cons\ (f\ x)\ xs)\ nil \\ filter = \lambda\ p \rightarrow foldr\ (\lambda x\ xs \rightarrow if\ \ p\ x\ then\ cons\ x\ xs\ else\ xs)\ nil \\ foldl = \lambda\ f\ v\ xs = foldr\ (\lambda\ xg \rightarrow\ (\lambda \rightarrow g\ (f\ a\ x)))\ id\ xs\ v \end{cases}$$

Base Library

The base library includes basic type-theoretical building blocks starting from **Unit**, **Bool**, **Either**, **Maybe**, **Nat**, **List** and **IO**. Here some exmaples how it looks like. The full listing of Base Library folder is available at **Om** github repository[8].

```
  data Nat: Type :=
       (Zero: Unit → Nat)
       (Succ: Nat → Nat)

  data List (A: Type) : Type :=
       (Nil: Unit → List A)
       (Cons: A → List A → List A)

record String: List Nat := List.Nil

  data IO: Type :=
       (getLine: (String → IO) → IO)
       (putLint: String → IO)
       (pure: () → IO)

record IO: Type :=
       (data: String)
       ([>>=]: ...)

record Morte: Type :=
       (recursive: IO.replicateM Nat.Five
                   (IO.[>>=] IO.data Unit IO.getLine IO.putLine))
```

Measurements

The underlying **Om** typechecker and compiler is a target language for higher level languages. The overall size of **Om** language with extractor to Erlang is 265 lines of code.

[8] http://github.com/groupoid/om

TABLE 2. Compiler Passes

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

We benchmarked the unrolling of inductive list type in Church encoding extracted with OM with native erlang **lists:foldl**.

TABLE 3. Performance

Operation	Type	Time
Pack/Unpack 1 000 000	Inductive Nat	776407 us
Pack/Unpack 1 000 000	Inductive List	1036461 us
Pack/Unpack 1 000 000	Erlang/OTP List	148084 us
Typechecking	Om ShadowTrans	4.972s
Typechecking	Morte ShadowTrans	57.867s

Conclusion

We have proposed a modified version of calculus of constructions, the pure type system, with predicative and impredicative switchable infinitary hierarhies. This system is known to be consistent, supports strong normalization and resembles the type system based in foundations of modern provers, like Coq, Lean, Agda.

Discoveries. During this investigation were made following discoveries: 1) baning recursion caused impossiblity of encoding a class of theorems based on induction principle. As was shown by Peng Fu, Aaron Stump[16], the only needed ingridient for induction in CoC is Self-Type, weak form of fixpoint recursion in the core. 2) however for running applications in runtime it is enough System F programs or Dependent Types without Fixpoint. So we can prove properties of these programs in higher languages with fixpoint (and thus induction) and then erase theorems from specification and convert runtime parts of specification into **PTS**$^\infty$ with later extraction to any functional language. 2) there are a lot of theorems, that could be expressed without fixpoint, such as theorems from higher order logic. 3) this system could be naturally translated into untyped lambda interpreters.

Advantages over existing pure languages. 1) refined version of typechecker and clean implementation in 265 LOC. This will make more trust to the core by external institutions. 2) supporting both predicative and impredicative hierarhies of **PTS**$^\infty$ configuration. 3) comparing to other languages, Om is much faster on big terms thanks to fast Erlang lambda evaluations and a cache layer. 4) Om is a production language.

Scientific and Production usage. 1) The language could be used as a trusted core for cetification sensitive parts of applications, such as in finance, math or other domains with requirement for totality. 2) This work could be used as embeddable runtime library. 3) In the academia **PTS**$^\infty$ could be used as teaching instrument for logic, type systems, lambda calculus, functional languges.

Further research perspective. 1) Extend the host languages from Erlang to Rust and prove the Om within Coq or Cubical. 2) Build a theory of compilation and erasing from higher languages to **PTS**$^\infty$. 3) Build a certified interpreter (replace Erlang) in future higher level language. 4) Make Induction Principle switchable with **PTS**$^\infty$ in future.

Acknowledgments

We thank to all contributors of Groupoid Infinity who helped us to avoid mistakes in TeX and Erlang files. We also thank to our spspouses for continuous support.

REFERENCES

[1] P. Martin-Löf and G. Sambin, *Intuitionistic type theory*, Studies in proof theory (Bibliopolis, 1984).

[2] C. Paulin-Mohring, in *All about Proofs, Proofs for All*, Studies in Logic (Mathematical logic and foundations), Vol. 55, edited by B. W. Paleo and D. Delahaye (College Publications, 2015).

[3] T. Coquand and G. Huet, "The calculus of constructions," in *Information and Computation* (Academic Press, Inc., Duluth, MN, USA, 1988), pp. 95–120.

[4] H. P. Barendregt, in *Handbook of Logic in Computer Science (Vol. 2)*, edited by S. Abramsky, D. M. Gabbay, and S. E. Maibaum (Oxford University Press, Inc., New York, NY, USA, 1992) Chap. Lambda Calculi with Types,, pp. 117–309.

[5] S. P. Jones and E. Meijer, "Henk: A typed intermediate language," in *In Proc. First Int'l Workshop on Types in Compilation* (1997).

[6] C.-E. Ore, "The extended calculus of constructions (ecc) with inductive types," in *Information and Computation*, Vol. 99 (1992), pp. 231 – 264.

[7] G. Barthe, "Extensions of pure type systems," in *Typed Lambda Calculi and Applications: Second International Conference on Typed Lambda Calculi and Applications, TLCA '95 Edinburgh, United Kingdom, April 10–12, 1995 Proceedings*, edited by M. Dezani-Ciancaglini and G. Plotkin (Springer Berlin Heidelberg, Berlin, Heidelberg, 1995), pp. 16–31.

[8] A. Stump, "The calculus of dependent lambda eliminations," in *Journal of Functional Programming*, Vol. 27 (Cambridge University Press, 2017).

[9] H. Geuvers, "Induction is not derivable in second order dependent type theory," in *Typed Lambda Calculi and Applications: 5th International Conference, TLCA 2001 Kraków, Poland, May 2–5, 2001 Proceedings*, edited by S. Abramsky (Springer Berlin Heidelberg, Berlin, Heidelberg, 2001), pp. 166–181.

[10] S. H. A. M. Cyril Cohen, Thierry Coquand, in *Cubical Type Theory: a constructive interpretation of the univalence axiom*, Vol. abs/1611.02108 (2017).

[11] R. Hinze and N. Wu, "Histo- and dynamorphisms revisited," in *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming*, WGP '13 (ACM, New York, NY, USA, 2013), pp. 1–12.

[12] P. Dagand, U. of Strathclyde. Department of Computer, and P. t. Information Sciences, *A Cosmology of Datatypes: Reusability and Dependent Types* (2013).

[13] T. Altenkirch and A. Kaposi, "Type theory in type theory using quotient inductive types," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16 (ACM, New York, NY, USA, 2016), pp. 18–29.

[14] M. Hamana and M. P. Fiore, "A foundation for gadts and inductive families: dependent polynomial functor approach," in *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011* (2011), pp. 59–70.

[15] C. Böhm and A. Berarducci, "Automatic synthesis of typed lambda-programs on term algebras," in *Theoretical Computer Science*, Vol. 39 (1985), pp. 135–154.

[16] P. Fu and A. Stump, "Self types for dependently typed lambda encodings," in *Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings* (2014), pp. 224–239.

Contents