Issue III: Pure Type System

Maxim Sokhatsky

[1] National Technical University of Ukraine
Igor Sikorsky Kyiv Polytechnical Institute
April 22, 2025

Abstract

This paper presents the design of the **Henk** language and an implementation of its type checker and bytecode extractor to Erlang. **Henk** is an intermediate language based on a pure type system with the infinite number of universes, so it is known to be consistent in dependent type theory. This **Henk** language is a core part of the language family for verification purposes. The type checker is built upon MLTT principles and can be switched between predicative and impredicative hierarchies of universes. 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 with infinite I/O operations which immediately enables **Henk** for long living runtime verified applications. We briefly describe the top-level language which compiles to pure type system (PTS) core as the further work. As the results, we will show lambda evaluation performance on BEAM virtual machine. PTS approach take its roots starting from AUTOMATH, MLTT, CoC, but extracting to performant untyped lambda interpreters is a novel way of cerifying applications. As a full vision we propose a stack of langauges, where **Henk** takes a central part.

Contents

1 Introduction

IEEE[1] standard and ESA[2] regulatory documents define a number of tools and approaches for verification and validation processes. The most advanced techniques involve mathematical languages and notations. The age of verified math was started by de Bruin's AUTOMATH prover and Martin-Löf[14]'s type theory. Today we have Coq, Agda, Lean, Idris, F* languages which are based on Calculus of Inductive Constructions or CiC[16]. The core of CiC is Calculus of Constructions or CoC[7]. Further development has lead to Lambda Cube[2] and Pure Type Systems by Henk[13] and Morte[3]. Pure Type Systems are custom languages based on CoC with single Pi-type and possibly other extensions. Notable extensions are ECC, ECC with Inductive Types[15], K-rules[3]. The main motivation of Pure Type Systems is an easy reasoning about core, strong normalization and trusted external verification due to compact type checkers. A custom type checker can be implemented 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.

1.1 Generating Trusted Programs

According to Curry-Howard, a correspondence inside Martin-Löf Type Theory[14] proofs or certificates are lambda terms of particular types or specifications. As both specifications and implementations are done in a typed language with dependent types we can extract target implementation of a certified program just in any programming language. These languages could be so primitive as untyped lambda calculus and are usually implemented as untyped interpreters (JavaScript, Erlang, PyPy, LuaJIT, K). The most advanced approach is code generation to higher-level languages such as C++ and Rust (which is already language with trusted features on memory, variable accessing, linear types, etc.). In this work, we present a simple code extraction to Erlang programming language as a target interpreter. However, we have also worked on C++ and Rust targets as well.

1.2 System Architecture

Henk as a programming language has a core type system, the $\mathbf{PTS}^{\infty}$ — the pure type system with the infinite number of universes. This type system represents the core of the language. Higher languages form a set of front-ends to this core. Here is example of possible languages: 1) Language for inductive reasoning, 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);

[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] Gabriella Gonzalez. Haskell Morte Library

3) Pi-calculus 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[10] 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 run in an 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.

1.3 Place among other languages

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

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

2 Consistent PTS as Intermediate Language

The **Henk** language is a dependently typed lambda calculus **PTS**$^\infty$, an extension of Coquand' Calculus of Constructions[7] with the 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 the sequence of universes **Sorts** and complexity of the dependent term is equal to the maximum complexity of term and its dependency **Rules**. The universe system is completely described by the following PTS notation due to Barendregt[2]:

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

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

Table 1: List of languages, tried as verification targets

Target	Class	Intermediate	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**	**Henk**	**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

The **Henk** language is based on Henk languages described first by Erik Meijer and Simon Peyton Jones in 1997[13]. Later on in 2015 Morte implementation of Henk design appeared in Haskell, using the Boem-Berrarducci encoding of non-recursive lambda terms. It is based only on one type constructor Π, its intro λ and apply eliminator, infinite number of universes, and β-reduction. The design of Om language resemble Henk and Morte both in design and in implementation. This language intended to be small, concise, easy provable and able to produce the verifiable piece of code that can be distributed over the networks, compiled at the target platform with a safe linkage.

2.1 BNF and AST

Henk 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)
```

2.2 Universes

As **Henk** has infinite number of universes it should include metatheoretical **Nat** inductive type in its core. **Henk** 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 a universe with the star function. If an argument is not a universe it returns $\{error, \_\}$.

```
star (:star ,N) → N
          _ → (:error , "∗")
```

2.3 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 with a type checker parameter.

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

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

2.4 Impredicative Universes

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

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

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

2.5 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 returns max universe or universe of the last term respectively.

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

h A B → dep A B :impredicative
```

2.6 Contexts

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

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

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

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

2.7 Single Axiom Language

This language 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 of substitution and lambda application. Operational semantics in that way defines the rewrite rules of computations.

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

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

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

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

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

The theorems (specification) of PTS could be embedded in itself and used as Logical Framework for the Pi type. Here is the example in the 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 [2]. The equalities of computational semantics presented here as **Path** types in the higher language.

The **Henk** language is the extention of the **PTS**$^\infty$ with the remote AST node which means remote file loading from trusted storage, anyway this will be checked by the type checker. We deny recursion over the remote node. We also add an index to var for simplified de Bruijn indexes, we allow overlapped names with tags, incremented on each new occurrence.

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

Our typechecker differs from cannonical example of Coquand[6]. We based our typechecker on variable **Substitution**, variable **Shifting**, term **Normalization**, definitional **Equality** anf **Type Checker** itself.

2.8 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**.

```
type (:star,N)          D → (:star,N+1)
     (:var,N,I)         D → :true = proplists:is_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)
```

2.9 Shifting

Shift renames var N in B. Renaming means adding 1 to the nat component of variable.

```
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)
```

2.10 Substitution

Substitution replaces 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)
```

2.11 Normalization

Normalization performs substitutions on applications to functions (beta-reduction) by recursive entrance over the lambda and pi nodes.

```
norm (:star,X)        → (:star,X)
     (:var,X)         → (:var,X)
     (:remote,N)      → cache (norm N [])
     (:pi,N,0,I,O)    → (:pi,N,0,norm I,norm O)
     (:fn,N,0,I,O)    → (:fn,N,0,norm I,norm O)
     (:app,F,A)       → case norm F of
                            (:fn,N,0,I,O) → norm (subst O N A)
                                       NF → (:app,NF,norm A) end
```

2.12 Equality

Definitional Equality simply checks the equality of Erlang terms.

```
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))
```

3 Language Usage

Here we will show some examples of **Henk** language usage. In this section, we will show two examples. One is lifting PTS system to MLTT system by defining **Sigma** and **Equ** types using only **Pi** type. We will use Bohm inductive dependent encoding[4]. The second is to show how to write real world programs in **Henk** that performs input/output operations within Erlang environment. We show both recursive (finite, routine) and corecursive (infinite, coroutine, process) effects.

```
$ ./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."}]

$ ./om print fst erase norm a "#List/Cons"
   \ Head
-> \ Tail
-> \ Cons
-> \ Nil
-> Cons Head (Tail Cons Nil)
ok
```

3.1 Sigma Type

The PTS system is extremely powerful even without **Sigma** type. But we can encode **Sigma** type similar how we encode **Prod** tuple pair in Bohm 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 in Aaron Stump [17]. Here we will show desugaring to $\mathbf{PTS}^{\infty}$.

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

-- Sigma/snd
   \ (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)
```

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

-- Equ/Refl
   \ (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 is they are equal, however, you may want to use built-in definitional equality and normalization feature of type checker 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}}}}
```

3.3 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 used both on LING and BEAM Erlang virtual machines. This language allows you to define trusted operations in System F and extract this routine to Erlang/OTP platform and plug as trusted resources. As the 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)
```

3.3.1 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>
```

3.3.2 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>
```

4 Higher Language with Inductive Types

As was shown by Herman Geuvers[10] the induction principle is not derivable in second-order dependent type theory. However there a lot of ways doing this. For example, we can build 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[17]. In any case for derivable induction principle in PTS^{∞} we need to have fixpoint somehow in the core.

So-called Calculus of Inductive Constructions[16] 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 intended to be a language extension to PTS system. CiC is allowing fixpoint for any terms, and base checking should be performed during type checking 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.

4.1 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
```

4.2 AST

The AST of higher language is formally defined using itself. Here you can find telescopes (context lists), split and its branches, 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[5] and could be reached at Github repository[6].

4.3 Inductive Type Encoding

There are a number of inductive type encodings: 1) Commutative square encoding of F-algebras by Hinze, Wu [12]; 2) Inductive-recursive encoding, algebraic type of algebraic types, inductive family encoding by Dagand [8]; 3) Encoding with motives inductive-inductive definition, also with inductive families, for modeling quotient types by Altenkirch, Kaposi [1]; 4) Henry Ford encoding or encoding with Ran,Lan-extensions by Hamana, Fiore [11]; 5) Church-compatible Bohm-Berarducci encoding Bohm, Berarducci [4]. Om is shipped with base library in Church encoding and we already gave the example of IO system encoded with runtime linkage. We give here simple calculations behind this theory.

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

4.4 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 is known as a recursive expression that has a base of recursion In Chuch-Bohm-Berarducci encoding type are store 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 constructor as parameters to a pure function.

4.5 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 $f = (\!|[c, h]|\!) : List(A) \to C$ 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 its desugared version in **Henk** language.

```
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))
```

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

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

4.6 Base Library

The base library includes basic type-theoretical building blocks starting from **Unit**, **Bool**, **Either**, **Maybe**, **Nat**, **List** and **IO**. Here some examples how it looks like. The full listing of Base Library folder is available at **Henk** GitHub repository[7].

```
  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))
```

4.7 Measurements

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

[7] 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

5 Conclusion

We have proposed a modified version of CoC, also known as pure type system, with predicative and impredicative switchable infinitary hierarchies. This system is known to be consistent, supports strong normalization and resembles the type system which is the same as foundations of modern provers, like Coq, Lean, Agda.

Discoveries. During this investigation were made following discoveries: 1) baning recursion caused impossibility of encoding a class of theorems based on induction principle. As was shown by Peng Fu, Aaron Stump[9], the only needed ingredient for induction in CoC is Self-Type, weak form of fixpoint recursion in the core. 2) however for running applications at 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 a specification and convert runtime parts of the specification into $\mathbf{PTS}^{\infty}$ 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 type checker and the clean implementation in 265 LOC. This will make more trust to the core by external institutions. 2) supporting both predicative and impredicative hierarchies of $\mathbf{PTS}^{\infty}$ 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 certification sensitive parts of applications, such as in finance, math or other domains with the requirement for totality. 2) This work could be used as embeddable runtime library. 3) In the academia $\mathbf{PTS}^{\infty}$ could be used as teaching instrument for logic, type systems, lambda calculus, functional languages.

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 $\mathbf{PTS}^{\infty}$. 3) Build a certified interpreter (replace Erlang) in future higher level language. 4) Make Induction

Principle switchable with $\mathbf{PTS}^\infty$ in future.

6 Acknowledgments

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

References

[1] Thorsten Altenkirch and Ambrus 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, pages 18–29, New York, NY, USA, 2016. ACM.

[2] H. P. Barendregt. In S. Abramsky, Dov M. Gabbay, and S. E. Maibaum, editors, *Handbook of Logic in Computer Science (Vol. 2)*, pages 117–309, New York, NY, USA, 1992. Oxford University Press, Inc.

[3] Gilles Barthe. *Extensions of pure type systems*, pages 16–31. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.

[4] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed lambda-programs on term algebras. In *Theoretical Computer Science*, volume 39, pages 135–154, 1985.

[5] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. In *Cubical Type Theory: a constructive interpretation of the univalence axiom*, volume abs/1611.02108, 2017.

[6] Thierry Coquand. An algorithm for type-checking dependent types. In *Sci. Comput. Program.*, volume 26, pages 167–177, 1996.

[7] Thierry Coquand and Gerard Huet. The calculus of constructions. In *Information and Computation*, pages 95–120, Duluth, MN, USA, 1988. Academic Press, Inc.

[8] P.É. Dagand, University of Strathclyde. Department of Computer, and PhD thesis Information Sciences. *A Cosmology of Datatypes: Reusability and Dependent Types*. 2013.

[9] Peng Fu and Aaron 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*, pages 224–239, 2014.

[10] Herman Geuvers. *Induction Is Not Derivable in Second Order Dependent Type Theory*, pages 166–181. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[11] Makoto Hamana and Marcelo 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*, pages 59–70, 2011.

[12] Ralf Hinze and Nicolas Wu. Histo- and dynamorphisms revisited. In *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming*, WGP '13, pages 1–12, New York, NY, USA, 2013. ACM.

[13] Simon Peyton Jones and Erik Meijer. Henk: A typed intermediate language. In *In Proc. First Int'l Workshop on Types in Compilation*, 1997.

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

[15] Christian-Emil Ore. The extended calculus of constructions (ecc) with inductive types. In *Information and Computation*, volume 99, pages 231 – 264, 1992.

[16] Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In Bruno Woltzenlogel Paleo and David Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, January 2015.

[17] Aaron Stump. The calculus of dependent lambda eliminations. In *Journal of Functional Programming*, volume 27. Cambridge University Press, 2017.