Volume III: Languages

Introduction to Programming Languages

Issue XI: Henk (Barendregt)
Issue XII: Frank (Pfenning)
Issue XIII: Per (Martin-Löf)
Issue XIV: Christine (Paulin-Mohring)
Issue XV: Daniel (Kan)
Issue XVI: Jack (Morava)
Issue XVII: Fabien (Morel)
Issue XVIII: Leslie (Lamport)
Issue XIX: Anders (Mörtberg)
Issue XX: Laurent (Schwartz)
Issue XXI: Urs (Schreiber)
Issue XXII: Paul (Cohen)

Namdak Tonpa

2023 · Groupoid Infinity
III

Зміст

Issue XI: Pure Type System

Maxim Sokhatsky

[1] National Technical University of Ukraine
Igor Sikorsky Kyiv Polytechnical Institute
6 травня 2025 р.

Анотація

This paper introduces the **Henk** language, a novel intermediate representation grounded in a pure type system (PTS) with an infinite hierarchy of universes, ensuring consistency within dependent type theory. Henk serves as a foundational component of a language family designed for formal verification. We present a robust implementation of Henk's type checker and bytecode extractor targeting the Erlang ecosystem, specifically the LING and BEAM virtual machines. The type checker, rooted in Martin-Löf Type Theory (MLTT), supports configurable predicative and impredicative universe hierarchies, enabling a flexible and trusted core for certified applications. Henk's syntax is compatible with the Morte language, extending its base library with support for indexed universes. We demonstrate programming paradigms in Henk, including seamless integration with Erlang's inductive and coinductive data structures. A minimal prelude library accompanies the implementation, supporting infinite I/O operations to facilitate long-running, verified applications. We briefly outline the top-level language **Christine**, which extends Henk's PTS core with general induction, sigma types, and equality, as future work. Empirical results showcase lambda evaluation performance on the BEAM virtual machine, highlighting the efficacy of extracting PTS-based systems to untyped, high-performance lambda interpreters. Drawing on foundational systems like AUTOMATH, MLTT, and the Calculus of Constructions (CoC), this work pioneers a performant approach to certified application development. We propose a layered language stack, with Henk as the critical initial layer, advancing the state of the art in verified software systems.

1 Introduction to Henk

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 [10]'s type theory. Today we have Coq, Agda, Lean, Idris, F* languages which are based on Calculus of Inductive Constructions or CIC [6]. The core of CIC is Calculus of Constructions or CoC [4]. Further development has lead to Lambda Cube [3] and Pure Type Systems by Henk [2] 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 [7], K-rules [8]. 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 [10] 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 is a foundational programming language — the pure type system with the infinite number of universes. All other higher languages like **Per**, **Christine** fully contains (subsumes) **Henk** in its core.

[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

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 **Joe**;
- Level 1 — consistent pure type system **Henk**;
- Level 2 — higher language **Per**.

Табл. 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[4]
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**
O	interpreter	Henk	PTS
K	interpreter	Q	Applicative
PyPy	interpreter/native	N/A	ULC
LuaJIT	interpreter/native	N/A	ULC
JavaScript	interpreter/native	PureScript	System F

2 Pure Type System

The **Henk** language is a dependently typed lambda calculus **Per**, an extension of Coquand' Calculus of Constructions [4] 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 [3]:

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

The **Henk** language is based on languages described first by Erik Meijer and Simon Peyton Jones in 1997 [2]. 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          data pts = star (n: nat)
 V ::= #identifier                | var (n: name)
 S ::= * < #number >              | app (f a: pts)
 O ::= S | V | ( O )              | lambda (x: name) (d c: pts)
     | O O | O → O                | pi (x: name) (d c: pts)
     | λ (I : O ) → O
     | ∀ (I : O ) → O
```

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.

$$\frac{}{\Gamma : Context} \tag{Ctx-formation}$$

$$\frac{\Gamma : Context}{Empty : \Gamma} \tag{Ctx-intro$_1$}$$

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

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 [3]. The equalities of computational semantics presented here as **Path** types in the higher language.

The **Henk** language is the extention of the **Henk** 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 [19]. 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 Henk Tutorial

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 [13]. 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.

```
$ ./henk 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."}]

$ ./henk 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 [11]. Here we will show desugaring to **Henk**.

```
-- 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 Inductive Type System

As was shown by Herman Geuvers [12] 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 [11]. In any case for derivable induction principle in **Henk** we need to have fixpoint somehow in the core.

So-called Calculus of Inductive Constructions [6] 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 **Christine** [5] 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
```

[5] `https://christine.groupoid.space`

```
|  ::= #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 **Christine** 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 BNF from **Per** 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 [14]; 2) Inductive-recursive encoding, algebraic type of algebraic types, inductive family encoding by Dagand [15]; 3) Encoding with motives inductive-inductive definition, also with inductive families, for modeling quotient types by Altenkirch, Kaposi [16]; 4) Henry Ford encoding or encoding with Ran,Lan-extensions by Hamana, Fiore [17]; 5) Church-compatible Bohm-Berarducci encoding Bohm, Berarducci [13]. 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] https://github.com/groupoid/per/tree/main/src/erlang/src

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.

Табл. 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

[7] http://github.com/groupoid/henk

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 **Henk** 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. 2) supporting both predicative and impredicative hierarchies. 3) comparing to other languages, **Henk** is much faster on big terms. 4) **Henk** 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 **Henk** 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 other languages, like Rust and OCaml. 2) Build a theory of compilation and erasing from higher languages to **Henk**. 3) Build a certified interpreter (replace Erlang) in future higher level language. 4) Add General Induction Principle to **Henk** in future language called **Frank**. 5) Add Sigma and Equality to **Frank** in future language called **Christine**.

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.

Література

[1] Saunders MacLane, *Categories for the Working Mathematician*, Graduate Texts in Mathematics, Vol. 5, Springer-Verlag, New York, 1971, ix+262 pp.

[2] Simon Peyton Jones and Erik Meijer, *Henk: A Typed Intermediate Language*, in Proceedings of the First International Workshop on Types in Compilation, 1997.

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

[4] Thierry Coquand and Gerard Huet, *The Calculus of Constructions*, Information and Computation, 1988, pp. 95–120.

[5] Frank Pfenning and Christine Paulin-Mohring, *Inductively Defined Types in the Calculus of Constructions*, in Mathematical Foundations of Programming Semantics, 5th International Conference, Tulane University, New Orleans, USA, March 29–April 1, 1989, pp. 209–228.

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

[7] Christian-Emil Ore, *The Extended Calculus of Constructions (ECC) with Inductive Types*, Information and Computation, Vol. 99, No. 2, 1992, pp. 231–264.

[8] Gilles Barthe, *Extensions of Pure Type Systems*, in Typed Lambda Calculi and Applications: Second International Conference, TLCA '95, Edinburgh, UK, April 10–12, 1995, Proceedings, edited by Mariangiola Dezani-Ciancaglini and Gordon Plotkin, Springer, Berlin, 1995, pp. 16–31.

[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, Vienna, Austria, July 14–17, 2014, pp. 224–239.

[10] P. Martin-Löf and G. Sambin, *Intuitionistic Type Theory*, Studies in Proof Theory, Bibliopolis, 1984.

[11] Aaron Stump, *The Calculus of Dependent Lambda Eliminations*, Journal of Functional Programming, Vol. 27, 2017.

[12] Herman 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 Samson Abramsky, Springer, Berlin, 2001, pp. 166–181.

[13] Corrado Böhm and Alessandro Berarducci, *Automatic Synthesis of Typed Lambda-Programs on Term Algebras*, Theoretical Computer Science, Vol. 39, No. 2–3, 1985, pp. 135–154.

[14] Ralf Hinze and Nicolas Wu, *Histo- and Dynamorphisms Revisited*, in Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming (WGP '13), Boston, Massachusetts, USA, 2013, pp. 1–12.

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

[16] 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), St. Petersburg, FL, USA, 2016, pp. 18–29.

[17] Makoto Hamana and Marcelo P. Fiore, *A Foundation for GADTs and Inductive Families: Dependent Polynomial Functor Approach*, in Proceedings of the 7th ACM SIGPLAN Workshop on Generic Programming (WGP@ICFP 2011), Tokyo, Japan, September 19–21, 2011, pp. 59–70.

[18] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg, *Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom*, CoRR, abs/1611.02108, 2017.

[19] Thierry Coquand, *An Algorithm for Type-Checking Dependent Types*, Science of Computer Programming, Vol. 26, No. 1–3, 1996, pp. 167–177.

Issue XII: Calculus of Inductive Constructions

Namdak Tonpa

May 2025

Анотація

This article develops a specialized framework for proving strong normalization in the Calculus of Constructions (CoC) and the Calculus of Inductive Constructions (CIC). Building on Girard's normalization framework, we adapt neutral terms, elimination contexts, and reducibility candidates to handle dependent types, universes, inductive types, and general induction. The framework is formalized with definitions, lemmas, and a proof of strong normalization, explicitly addressing the complexities of general induction. Applications to Coq's type theory are discussed, emphasizing the framework's modularity and robustness.

7 Introduction to Frank

The Calculus of Constructions (CoC) [15] is a dependently typed lambda calculus with impredicative universes, forming the core of many proof assistants. The Calculus of Inductive Constructions (CIC) [5] extends CoC with inductive types and general induction principles, enabling expressive data structures and proofs, as seen in Coq. Strong normalization, ensuring that every well-typed term reduces to a normal form in finitely many steps, is essential for the consistency of these systems.

The Riba's work *Toward a General Rewriting-Based Framework for Reducibility* [1], provides a unified approach to reducibility proofs using rewriting relations and elimination contexts. This article presents a specialized framework for CoC and CIC, adapting Girard concepts to their dependent types, universes, inductive types, and the general induction principle of CIC. We formalize neutral terms, elimination contexts, and reducibility candidates, proving strong normalization and addressing the complexities of general induction. The framework's modularity makes it suitable for Coq and extensible to other dependently typed systems.

8 Syntax

We define the syntax for CoC and CIC, including the general induction principle. The set of terms $\mathcal{T}$ in CoC and CIC is defined as:

$$t ::= x \mid \text{Sort}\, s \mid \Pi(.x : A).B \mid \lambda x : A.b \mid f\, a$$

$$\mid \text{Ind}(I : A)\{c_1 : C_1, \ldots, c_n : C_n\} \mid \text{Constr}(i, I, t_1, \ldots, t_m) \mid \text{case}(t, I, P, b_1, \ldots, b_n)$$

where: - x is a variable, - $\text{Sort}\, s$ represents universes ($s = \text{Prop}, \text{Type}_i$), - $\Pi(.x : A).B$ is a dependent function type, - $\lambda x : A.b$ is a lambda abstraction, - $f\, a$ is an application, - $\text{Ind}(I : A)\{c_1 : C_1, \ldots, c_n : C_n\}$ defines an inductive type I with constructors $c_i : C_i$, - $\text{Constr}(i, I, t_1, \ldots, t_m)$ is the i-th constructor of I, - $\text{case}(t, I, P, b_1, \ldots, b_n)$ is a dependent case expression for general induction on I.

CoC includes only the first five constructs ($x, \text{Sort}, \Pi, .\lambda, .\ $), while CIC adds inductive types, constructors, and case expressions.

9 Semantics

Here we define typing rules, and rewriting relations for CoC and CIC, including the general induction principle.

10 Properties

10.1 Rewriting Relation

The rewriting relation $\rightarrow \subseteq \mathcal{T} \times \mathcal{T}$ includes: - Beta-reduction: $(\lambda x : A.b)\, a \rightarrow [x \mapsto a]b$. - Inductive reduction (iota-reduction): For an inductive type $\text{Ind}(I : A)\{c_1 : C_1, \ldots, c_n : C_n\}$, if $t = \text{Constr}(i, I, t_1, \ldots, t_m)$, then:

$$\text{case}(t, I, P, b_1, \ldots, b_n) \rightarrow b_i\, t_1 \ldots t_m$$

where b_i is the branch corresponding to constructor c_i.

A term t is *strongly normalizing* if every reduction sequence starting from t is finite. Typing judgments are of the form $\Gamma \vdash t : A$, where $\Gamma = [x_1 : A_1, \ldots, x_n : A_n]$ is a context.

10.2 General Induction in CIC

The general induction principle (dependent elimination) allows reasoning about inductive types with dependent predicates. For an inductive type $\text{Ind}(I : A)\{c_1 : C_1, \ldots, c_n : C_n\}$, the case expression $\text{case}(t, I, P, b_1, \ldots, b_n)$ has type $P\, t$, where: - $P : \Pi(.x : I).\text{Sort}\, s$ is a dependent predicate, - Each branch $b_i : \Pi(.y_1 : T_1). \ldots \Pi(.y_m : T_m).P\, \text{Constr}(i, I, y_1, \ldots, y_m)$ corresponds to constructor c_i.

This principle generalizes simple pattern matching by allowing the result type to depend on the scrutinized term t.

10.3 Neutral Terms

Neutral terms are defined to exclude terms that trigger immediate reductions, accommodating both beta- and iota-reductions in CIC.

Definition 1 (Neutral Terms)**.** A term $t \in \mathcal{T}$ is *neutral*, denoted $t \in \mathcal{N}$, if it is not a lambda abstraction ($\lambda x : A.b$) or a constructor term ($\text{Constr}(i, I, t_1, \ldots, t_m)$). Formally:

$$\mathcal{N} = \{t \in \mathcal{T} \mid t = x \text{ or } t = \text{Sort}\, s \text{ or } t = \Pi(.x : A).B \text{ or } t = f\, a \text{ or}$$

$$t = \text{case}(t', I, P, b_1, \ldots, b_n) \text{ where } t' \notin \text{Constr}\}$$

Case expressions are neutral unless their scrutinee is a constructor, reflecting the iota-reduction rule [1].

10.4 Elimination Contexts

Elimination contexts are extended to handle general induction, capturing the reduction behavior of case expressions.

Definition 2 (Elimination Contexts)**.** An *elimination context* $E \in \mathcal{E}$ is defined inductively:

$$E ::= [] \mid E\, t \mid \text{case}(E, I, P, b_1, \ldots, b_n), \quad t, b_i \in \mathcal{T}$$

The application $E[t]$ is: - $[][t] = t$, - $E\, u[t] = E[t]\, u$, - $\text{case}(E, I, P, b_1, \ldots, b_n)[t] = \text{case}(E[t], I, P, b_1, \ldots, b_n)$.

A set $\mathcal{E}$ is *adequate* if: 1. **Closure under composition**: If $E_1, E_2 \in \mathcal{E}$, then $E_1[E_2] \in \mathcal{E}$. 2. **Stability under reduction**: If $E[t] \to t'$, then either $t' = E'[t]$ for some $E' \in \mathcal{E}$, or $t' \in \mathcal{N}$, or $t' = \text{Constr}(i, I, t_1, \ldots, t_m)$.

The inclusion of dependent case expressions ensures that general induction is modeled correctly [2].

10.5 Reducibility Candidates

Reducibility candidates are defined to ensure strong normalization, accommodating dependent types, universes, and inductive types with general induction.

Definition 3 (Reducibility Candidates)**.** For a type $A \in \mathcal{A}$, a set $\mathcal{R}_A \subseteq \mathcal{T}$ is a *reducibility candidate* if:

1. **Strong normalization**: If $t \in \mathcal{R}_A$, then t is strongly normalizing.
2. **Closure under reduction**: If $t \in \mathcal{R}_A$ and $t \to t'$, then $t' \in \mathcal{R}_A$.
3. **Neutral terms**: If $t \in \mathcal{N}$ and for all $t \to t'$, $t' \in \mathcal{R}_A$, then $t \in \mathcal{R}_A$.
4. **Dependent types**: If $A = \Pi(.x : B).C$, then $t \in \mathcal{R}_A$ if for all $u \in \mathcal{R}_B$, $t\,u \in \mathcal{R}_{[x \mapsto u]C}$.
5. **Universes**: If $A = \text{Sort}\, s$, then $\mathcal{R}_A$ contains all strongly normalizing terms of type s.
6. **Inductive types**: If $A = \text{Ind}(I : A')$, then $\mathcal{R}_A$ contains all terms t such that for any $\text{case}(t, I, P, b_1, \ldots, b_n)$, the result is in $\mathcal{R}_{P\,t}$.

For inductive types, the reducibility candidate ensures that terms behave correctly under general induction, reflecting the dependent nature of case expressions [2].

10.6 Strong Normalization for CoC and CIC

We prove strong normalization using the adapted reducibility framework, explicitly handling general induction.

Theorem 1 (Strong Normalization)**.** For any context Γ and term t, if $\Gamma \vdash t : A$, then $t \in \mathcal{R}_A$, and thus t is strongly normalizing.

Proof. The proof proceeds by induction on the typing derivation $\Gamma \vdash t : A$.

1. Case: $t = x$ If $\Gamma \vdash x : A$, then $(x : A) \in \Gamma$. Since $x \in \mathcal{N}$ and has no reductions, x *Radiolabelled*$\mathcal{R}_A$.

2. Case: $t = \mathrm{Sort}\, s$ If $\Gamma \vdash \mathrm{Sort}\, s : \mathrm{Sort}\, s'$, then $\mathrm{Sort}\, s \in \mathcal{N}$ and is irreducible, so $\mathrm{Sort}\, s \in \mathcal{R}_{\mathrm{Sort}\, s'}$.

3. Case: $t = \Pi(.x : A).B$ If $\Gamma \vdash A : \mathrm{Sort}\, s_1$, $\Gamma, x : A \vdash B : \mathrm{Sort}\, s_2$, then $\Gamma \vdash \Pi(.x : A).B : \mathrm{Sort}\, s$. By induction, $A \in \mathcal{R}_{\mathrm{Sort}\, s_1}$, $B \in \mathcal{R}_{\mathrm{Sort}\, s_2}$. Since $\Pi(.x : A).B \in \mathcal{N}$, it is in $\mathcal{R}_{\mathrm{Sort}\, s}$ if all reducts are, which holds trivially.

4. Case: $t = \lambda x : A.b$ If $\Gamma \vdash A : \mathrm{Sort}\, s$, $\Gamma, x : A \vdash b : B$, then $\Gamma \vdash \lambda x : A.b : \Pi(.x : A).B$. By induction, for all $u \in \mathcal{R}_A$, $[x \mapsto u]b \in \mathcal{R}_{[x \mapsto u]B}$. Thus, $(\lambda x : A.b)\, u \to [x \mapsto u]b \in \mathcal{R}_{[x \mapsto u]B}$, so $\lambda x : A.b \in \mathcal{R}_{\Pi(.x:A).B}$.

5. Case: $t = f\, a$ If $\Gamma \vdash f : \Pi(.x : A).B$, $\Gamma \vdash a : A$, then $\Gamma \vdash f\, a : [x \mapsto a]B$. By induction, $f \in \mathcal{R}_{\Pi(.x:A).B}$, $a \in \mathcal{R}_A$. Thus, $f\, a \in \mathcal{R}_{[x \mapsto a]B}$.

6. Case: $t = \mathrm{Ind}(I : A)\{c_1 : C_1, \ldots, c_n : C_n\}$ If $\Gamma \vdash A : \mathrm{Sort}\, s$, and each constructor $c_i : C_i$ is well-typed, then $\Gamma \vdash I : A$. By induction, $A \in \mathcal{R}_{\mathrm{Sort}\, s}$, and each $C_i \in \mathcal{R}_{\mathrm{Sort}\, s_i}$. Thus, $I \in \mathcal{R}_A$.

7. Case: $t = \mathrm{Constr}(i, I, t_1, \ldots, t_m)$ If $\Gamma \vdash \mathrm{Constr}(i, I, t_1, \ldots, t_m) : I\, u_1 \ldots u_k$, then each $t_j \in \mathcal{R}_{T_j}$ by induction. Although Constr is not neutral, its arguments are reducible, and any case on Constr reduces to a branch in $\mathcal{R}$, ensuring $t \in \mathcal{R}_{I\, u_1 \ldots u_k}$.

8. Case: $t = \mathrm{case}(t', I, P, b_1, \ldots, b_n)$ If $\Gamma \vdash t' : I\, u_1 \ldots u_k$, $\Gamma \vdash P : \Pi(.x : I).\mathrm{Sort}\, s$, and each branch $b_i : \Pi(.y_1 : T_1). \ldots \Pi(.y_m : T_m).P\, \mathrm{Constr}(i, I, y_1, \ldots, y_m)$, then $\Gamma \vdash \mathrm{case}(t', I, P, b_1, \ldots, b_n) : P\, t'$. By induction: - $t' \in \mathcal{R}_{I\, u_1 \ldots u_k}$, - $P \in \mathcal{R}_{\Pi(.x:I).\mathrm{Sort}\, s}$, - Each $b_i \in \mathcal{R}_{\Pi(.y_1:T_1). \ldots \Pi(.y_m:T_m).P\, \mathrm{Constr}(i,I,y_1,\ldots,y_m)}$. If $t' = \mathrm{Constr}(i, I, t_1, \ldots, t_m)$, then:

$$\mathrm{case}(t', I, P, b_1, \ldots, b_n) \to b_i\, t_1 \ldots t_m$$

Since b_i is reducible and each $t_j \in \mathcal{R}_{T_j}$, the result is in $\mathcal{R}_{P\, \mathrm{Constr}(i,I,t_1,\ldots,t_m)}$. If $t' \in \mathcal{N}$, the case expression is neutral, and all its reducts are in $\mathcal{R}_{P\, t'}$ by induction. Thus, $t \in \mathcal{R}_{P\, t'}$.

Since $\mathcal{R}_A$ contains only strongly normalizing terms, $t \in \mathcal{R}_A$ implies t is strongly normalizing [1, 2]. □

Compared to other normalization proofs: - Girard's Candidates: Effective for CoC but less modular for CIC's inductive types and general induction [14]. - Werner's Proof: Specific to CIC, addressing general induction but less general for rewriting [2]. - Normalization by Evaluation (NbE): Semantic and efficient but complex for general induction [3].

11 Conclusion

This specialized framework extends Riba's rewriting-based reducibility approach to prove strong normalization for CoC and CIC, explicitly incorporating the general induction principle of CIC. By formalizing neutral terms, elimination contexts, and reducibility candidates tailored to dependent types, universes, inductive types, and dependent case expressions, it provides a robust tool for Coq's type theory. The framework's modularity supports extensions like universe polymorphism and guarded recursion, making it a versatile foundation for future research in dependently typed systems.

Література

Metatheory

[1] Riba, C. (2008). Toward a General Rewriting-Based Framework for Reducibility. Submitted, available from the author's homepage.

[2] Werner, B. (1994). *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris 7.

[3] Abel, A., & Sattler, C. (2012). Normalization by Evaluation for Call-by-Push-Value and Polarized Lambda Calculus. `https://hal.science/hal-00779623/document`.

[4] Harper, R., & Licata, D. (2007). Mechanizing Metatheory in a Logical Framework. *Journal of Functional Programming*, 17(4-5), 613–673.

CIC

[5] Coquand, T., & Paulin-Mohring, C. (1990). Inductively Defined Types. *Proceedings of the International Conference on Computer Logic (COLOG-88)*, Lecture Notes in Computer Science, 417, 50–66.

[6] Paulin-Mohring, C. (1992). Inductive Definitions in the System Coq: Rules and Properties. *Proceedings of the First International Conference on Typed Lambda Calculi and Applications (TLCA)*, Lecture Notes in Computer Science, 664, 328–345.

[7] Paulin-Mohring, C. (2014). Introduction to the Calculus of Inductive Constructions. *All about Proofs, Proofs for All*, HAL Archives, `https://inria.hal.science/hal-01094195/document`.

[8] Pfenning, F., & Paulin-Mohring, C. (1989). Inductively Defined Types in the Calculus of Constructions. *Proceedings of Mathematical Foundations of Programming Semantics (MFPS)*, Lecture Notes in Computer Science, 442, 209–228. `https://www.cs.cmu.edu/~fp/papers/mfps89.pdf`.

[9] Asperti, A., Ricciotti, W., Sacerdoti Coen, C., & Tassi, E. (2009). A Compact Kernel for the Calculus of Inductive Constructions. *Sadhana*, 34(1), 71–90. `https://www.cs.unibo.it/~sacerdot/PAPERS/sadhana.pdf`.

[10] Dybjer, P. (1997). Inductive Families. *Formal Aspects of Computing*, 9(4), 329–354.

[11] Bezem, M., Coquand, T., Dybjer, P., & Escardó, M. (2024). Type Theory with Explicit Universe Polymorphism. *arXiv preprint*, `https://arxiv.org/pdf/2212.03284`.

PTS

[12] Coquand, T. (1996). An Algorithm for Type-Checking Dependent Types. *Science of Computer Programming*, 26(1-3), 167–177.

[13] de Bruijn, N. G. (1972). Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indagationes Mathematicae*, 34(5), 381–392.

[14] Girard, J.-Y. (1972). *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. PhD thesis, Université Paris 7.

[15] Coquand, T., & Huet, G. (1988). The Calculus of Constructions. *Information and Computation*, 76(2-3), 95–120. `https://core.ac.uk/download/pdf/82038778.pdf`.

Issue XIX: Modal Homotopy Type System

М.Е. Сохацький [1]

[1] Національний технічний університет України
ім. Ігоря Сікорського
26 листопада 2021

Анотація

Here is presented a reincarnation of **cubicaltt** called **Anders**.

12 Introduction to Anders

Anders is a Modal HoTT proof assistant based on: classical MLTT-80 [6] with 0, 1, 2, W types; CCHM [11] in CHM [2] flavour as cubical type system with hcomp/transp operations; HTS [8] strict equality on pretypes; infinitisemal [1] modality primitives for differential geometry purposes. We tend not to touch general recursive higher inductive schemes, instead we will try to express as much HIT as possible through Suspensions, Truncations, Quotients primitives built into type checker core. Anders also aims to support simplicial types Simplex along with Hopf Fibrations built into core for sphere homotopy groups processing. This modification is called **Dan**. Full stack of Groupoid Infinity languages is given at AXIO/1[1] homepage.

The HTS language proposed by Voevodsky exposes two different presheaf models of type theory: the inner one is homotopy type system presheaf that models HoTT and the outer one is traditional Martin-Löf type system presheaf that models set theory with UIP. The motivation behind this doubling is to have an ability to express semisemplicial types. Theoretical work on merging inner and outer languages was continued in 2LTT [9].

Installation. While we are on our road to Lean-like tactic language, currently we are at the stage of regular cubical HTS type checker with CHM-style primitives. You may try it from Github sources: groupoid/anders[2] or install through OPAM package manager. Main commands are **check** (to check a program) and **repl** (to enter the proof shell).

```
$ opam install anders
```

Anders is fast, idiomatic and educational (think of optimized Mini-TT). We carefully draw the favourite Lean-compatible syntax to fit 200 LOC in Menhir. The CHM kernel is 1K LOC. Whole Anders compiles under 1 second and checks all the base library under 1/3 of a second [i5-12400]. Anders proof assistant as Homotopy Type System comes with its own Homotopy Library[3].

13 Syntax

The syntax resembles original syntax of the reference CCHM type checker cubicaltt, is slightly compatible with Lean syntax and contains the full set of Cubical Agda [10] primitives (except generic higher inductive schemes).

Here is given the mathematical pseudo-code notation of the language expressions that come immediately after parsing. The core syntax definition of HTS language corresponds to the type defined in OCaml module:

Further Menhir BNF notation will be used to describe the top-level language E parser.

[1] `https://axio.groupoid.space`
[2] `https://github.com/groupoid/anders/`
[3] `https://anders.groupoid.space/lib/`

$$
\begin{array}{rl}
\text{cosmos} := & \mathbf{U}_j \mid \mathbf{V}_k \\
\text{var} := & \mathbf{var}\ \mathit{name} \mid \mathbf{hole} \\
\text{pi} := & \Pi\ \mathit{name}\ E\ E \mid \lambda\ \mathit{name}\ E\ E \mid E\ E \\
\text{sigma} := & \Sigma\ \mathit{name}\ E\ E \mid (E, E) \mid E.1 \mid E.2 \\
0 := & \mathbf{0} \mid \mathbf{ind}_0\ \text{E E E} \\
1 := & \mathbf{1} \mid \star \mid \mathbf{ind}_1\ \text{E E E} \\
2 := & \mathbf{2} \mid 0_2 \mid 1_2 \mid \mathbf{ind}_2\ \text{E E E} \\
\text{W} := & \mathbf{W}\ \text{ident E E} \mid \mathbf{sup}\ \text{E E} \mid \mathbf{ind}_W\ \text{E E} \\
\text{id} := & \mathbf{Id}\ E \mid \mathbf{ref}\ E \mid \mathbf{id}_J\ E \\
\text{path} := & \mathbf{Path}\ E \mid E^i \mid E\ @\ E \\
\text{I} := & \mathbf{I} \mid 0 \mid 1 \mid E\ \bigvee\ E \mid E\ \bigwedge\ E \mid \neg E \\
\text{part} := & \mathbf{Partial}\ E\ E \mid \mathbf{[}\ (E = I) \to E, ...\ \mathbf{]} \\
\text{sub} := & \mathbf{inc}\ E \mid \mathbf{ouc}\ E \mid E\ \mathbf{[}\ I\ \mapsto\ E\ \mathbf{]} \\
\text{kan} := & \mathbf{transp}\ E\ E \mid \mathbf{hcomp}\ E \\
\text{glue} := & \mathbf{Glue}\ E \mid \mathbf{glue}\ E \mid \mathbf{unglue}\ E\ E \\
\text{Im} := & \mathbf{Im}\ \text{E} \mid \mathbf{Inf}\ \text{E} \mid \mathbf{Join}\ \text{E} \mid \mathbf{ind}_{Im}\ \text{E E}
\end{array}
$$

$$
\begin{array}{rl}
\text{E} := & \text{cosmos} \mid \text{var} \mid \text{MLTT} \mid \text{CCHM} \mid \text{Im} \\
\text{CCHM} := & \text{path} \mid \text{I} \mid \text{part} \mid \text{sub} \mid \text{kan} \mid \text{glue} \\
\text{MLTT} := & \text{pi} \mid \text{sigma} \mid \text{id}
\end{array}
$$

Keywords. The words of a top-level language, file or repl, consist of keywords or identifiers. The keywords are following: `module`, `where`, `import`, `option`, `def`, `axiom`, `postulate`, `theorem`, `(`, `)`, `[`, `]`, `<`, `>`, `/`, `.1`, `.2`, Π, Σ, `,`, λ, `V`, ⋁, ⋀, `-`, `+`, `@`, `PathP`, `transp`, `hcomp`, `zero`, `one`, `Partial`, `inc`, ×, →, `:`, `:=`, ↦, `U`, `ouc`, `interval`, `inductive`, `Glue`, `glue`, `unglue`.

Indentifiers. Identifiers support UTF-8. Indentifiers couldn't start with `:`, `-`, →. Sample identifiers: `¬-of-∨`, `1→1`, `is-?`, `=`, `$∼]!005x`, ∞, `x→Nat`.

Modules. Modules represent files with declarations. More accurate, BNF notation of module consists of imports, options and declarations.

menhir

```
start <Module.file> file
start <Module.command> repl
repl : COLON IDENT exp1 EOF | COLON IDENT EOF | exp0 EOF | EOF
file : MODULE IDENT WHERE line* EOF
path : IDENT
line : IMPORT path+ | OPTION IDENT IDENT | declarations
```

Imports. The import construction supports file folder structure (without file extensions) by using reserved symbol / for hierarchy walking.

Options. Each option holds bool value. Language supports following options: 1) girard (enables U : U); 2) pre-eval (normalization cache); 3) impredicative (infinite hierarchy with impredicativity rule); In Anders you can enable or disable language core types, adjust syntaxes or tune inner variables of the type checker.

Declarations. Language supports following top level declarations: 1) axiom (non-computable declaration that breakes normalization); 2) postulate (alter-

native or inverted axiom that can preserve consistency); 3) definition (almost any explicit term or type in type theory); 4) lemma (helper in big game); 5) theorem (something valuable or complex enough).

```
axiom isProp (A : U) : U
def isSet (A : U) : U
 := Π (a b : A) (x y : Path A a b), Path (Path A a b) x y
```

Sample declarations. For example, signature isProp (A : U) of type U could be defined as normalization-blocking axiom without proof-term or by providing proof-term as definition.

In this example (A : U), (a b : A) and (x y : Path A a b) are called telescopes. Each telescope consists of a series of lenses or empty. Each lense provides a set of variables of the same type. Telescope defines parameters of a declaration. Types in a telescope, type of a declaration and a proof-terms are a language expressions exp1.

menhir

```
ident : IRREF | IDENT
lense : LPARENS ident+ COLON exp1 RPARENS
telescope : lense telescope
params : telescope | []
declarations :
    | DEF IDENT params DEFEQ exp1
    | DEF IDENT params COLON exp1 DEFEQ exp1
    | AXIOM IDENT params COLON exp1
```

Expressions. All atomic language expressions are grouped by four categories: exp0 (pair constructions), exp1 (non neutral constructions), exp2 (path and pi applcations), exp3 (neutral constructions).

menhir

```
face : LPARENS IDENT IDENT IDENT RPARENS }
part : face+ ARROW exp1 }
exp0 : exp1 COMMA exp0 | exp1 }
exp1 : LSQ separated(COMMA, part) RSQ }
     | LAM telescope COMMA exp1      | PI telescope COMMA exp1
     | SIGMA telescope COMMA exp1    | LSQ IRREF ARROW exp1 RSQ
     | LT ident+ GT exp1             | exp2 ARROW exp1
     | exp2 PROD exp1                | exp2
```

The LR parsers demand to define exp1 as expressions that cannot be used (without a parens enclosure) as a right part of left-associative application for both Path and Pi lambdas. Universe indicies U_j (inner fibrant), V_k (outer pretypes) and S (outer strict omega) are using unicode subscript letters that are already processed in lexer.

menhir

```
exp2 : exp2 exp3 | exp2 APPFORMULA exp3 | exp3 }
exp3 : LPARENS exp0 RPARENS LSQ exp0 MAP exp0 RSQ }
     | HOLE              | PRE             | KAN             | IDJ exp3
     | exp3 FST          | exp3 SND        | NEGATE exp3     | INC exp3
     | exp3 AND exp3     | exp3 OR exp3    | ID exp3         | REF exp3
     | OUC exp3          | PATHP exp3      | PARTIAL exp3    | IDENT
     | IDENT LSQ exp0 MAP exp0 RSQ }       | HCOMP exp3
     | LPARENS exp0 RPARENS }              | TRANSP exp3 exp3
```

14 Semantics

The idea is to have a unified layered type checker, so you can disbale/enable any MLTT-style inference, assign types to universes and enable/disable hierachies. This will be done by providing linking API for pluggable presheaf modules. We selected 5 levels of type checker awareness from universes and pure type systems up to synthetic language of homotopy type theory. Each layer corresponds to its presheaves with separate configuration for universe hierarchies.

```
def lang : U
 := inductive { UNI: cosmos → lang
              | PI: pure lang → lang
              | SIGMA: total lang → lang
              | ID: strict lang → lang
              | PATH: homotopy lang → lang
              | GLUE: glue lang → lang
              | INDUCTIVE: w012 lang → lang
              }
```

We want to mention here with homage to its authors all categorical models of dependent type theory: Comprehension Categories (Grothendieck, Jacobs), LCCC (Seely), D-Categories and CwA (Cartmell), CwF (Dybjer), C-Systems (Voevodsky), Natural Models (Awodey). While we can build some transports between them, we leave this excercise for our mathematical components library. We will use here the Coquand's notation for Presheaf Type Theories in terms of restriction maps.

14.1 Universe Hierarchies

Language supports Agda-style hierarchy of universes: prop, fibrant (U), interval pretypes (V) and strict omega with explicit level manipulation. All universes are bounded with preorder

$$Fibrant_j \prec Pretypes_k \tag{1}$$

in which j, k are bounded with equation:

$$j < k. \tag{2}$$

Large elimination to upper universes is prohibited. This is extendable to Agda model:

```
def cosmos : U
 := inductive { fibrant: ℕ
              | pretypes: ℕ
              }
```

The **Anders** model contains only fibrant U_j and pretypes V_k universe hierarchies.

14.2 Dependent Types

Definition 4 (Type)**.** A type is interpreted as a presheaf A, a family of sets A_I with restriction maps $u \mapsto u\ f, A_I \to A_J$ for $f : J \to I$. A dependent type B on A is interpreted by a presheaf on category of elements of A: the objects are pairs (I, u) with $u : A_I$ and morphisms $f : (J, v) \to (I, u)$ are maps $f : J \to$ such that $v = u\ f$. A dependent type B is thus given by a family of sets $B(I, u)$ and restriction maps $B(I, u) \to B(J, u\ f)$.

We think of A as a type and B as a family of presheves $B(x)$ varying $x : A$. The operation $\Pi(x : A)B(x)$ generalizes the semantics of implication in a Kripke model.

Definition 5 (Pi)**.** An element $w : [\Pi(x : A)B(x)](I)$ is a family of functions $w_f : \Pi(u : A(J))B(J, u)$ for $f : J \to I$ such that $(w_f u)g = w_{f\ g}(u\ g)$ when $u : A(J)$ and $g : K \to J$.

```
def pure (lang : U) : U
 := inductive { pi: name → nat → lang → lang → pure lang
              | lambda: name → nat → lang → lang
              | app: lang → lang
              }
```

Definition 6 (Sigma)**.** The set $\Sigma(x : A)B(x)$ is the set of pairs (u, v) when $u : A(I), v : B(I, u)$ and restriction map $(u, v)\ f = (u\ f, v\ f)$.

```
def total (lang : U) : U
 := inductive { sigma: name → lang → total lang
              | pair: lang → lang
              | fst: lang
              | snd: lang
              }
```

The presheaf with only Pi and Sigma is called **MLTT-72** [4]. Its internalization in **Anders** is as follows:

```
def MLTT (A : U) : U₁
 := Σ (Π–form   : Π (B : A → U), U)
      (Π–ctor₁ : Π (B : A → U), Pi A B → Pi A B)
      (Π–elim₁ : Π (B : A → U), Pi A B → Pi A B)
      (Π–comp₁ : Π (B : A → U) (a : A) (f : Pi A B),
                 = (B a) (Π–elim₁ B (Π–ctor₁ B f) a) (f a))
      (Π–comp₂ : Π (B : A → U) (a : A) (f : Pi A B),
                 = (Pi A B) f (λ (x : A), f x))
      (Σ–form   : Π (B : A → U), U)
      (Σ–ctor₁ : Π (B : A → U) (a : A) (b : B a) , Sigma A B)
      (Σ–elim₁ : Π (B : A → U) (p : Sigma A B), A)
      (Σ–elim₂ : Π (B : A → U) (p : Sigma A B), B (pr₁ A B p))
      (Σ–comp₁ : Π (B : A → U) (a : A) (b: B a),
                 = A a (Σ–elim₁ B (Σ–ctor₁ B a b)))
      (Σ–comp₂ : Π (B : A → U) (a : A) (b: B a),
                 = (B a) b (Σ–elim₂ B (a, b)))
      (Σ–comp₃ : Π (B : A → U) (p : Sigma A B),
                 = (Sigma A B) p (pr₁ A B p, pr₂ A B p)), Unit
```

14.3 Path Equality

The fundamental development of equality inside MLTT provers led us to the notion of ∞-groupoid as spaces. In this way Path identity type appeared in the core of type checker along with De Morgan algebra on built-in interval type.

```
def CCHM (lang: U) : U
 := inductive { pretype (n: nat)
              | PathP (_: lang) | PLam (_: lang) | PApp (f a: lang)
              | I | 0 | 1 | And (a b: lang) | Or (a b: lang) | Neg (_: lang)
              | Transp (a b: lang) | HComp (a b c d: lang)
              | Partial (_: lang) | PartialP (a b: lang) | System (_: lang)
              | Sub (a b c: lang) | Inc (a b: lang) | Ouc (: lang)
              | Glue (: lang) | GlueElem (a b c: lang) | Unglue (_: lang)
              }
```

Definition 7 (Cubical Presheaf $\mathbb{I}$)**.** The identity types modeled with another presheaf, the presheaf on Lawvere category of distributive lattices (theory of De Morgan algebras) denoted with $\square - \mathbf{I} : \square^{op} \to \text{Set}$.

Definition 8 (Properties of $\mathbf{I}$)**.** The presheaf $\mathbf{I}$: i) has to distinct global elements 0 and 1 (B_1); ii) $\mathbf{I}$(I) has a decidable equality for each I (B_2); iii) $\mathbf{I}$ is tiny so the path functor $X \mapsto X^{\mathbf{I}}$ has right adjoint (B_3).; iv) $\mathbf{I}$ has meet and join (connections).

Interval Pretypes. While having pretypes universe V with interval and associated De Morgan algebra ($\wedge$, $\vee$, -, 0, 1, I) is enough to perform DNF normalization and proving some basic statements about path, including: contractability of singletons, homotopy transport, congruence, functional extensionality; it is not enough for proving β rule for Path type or path composition.

Generalized Transport. Generalized transport transp adresses first problem of deriving the computational β rule for Path types:

```
theorem Path_β (A : U) (a : A) (C : D A) (d: C a a (refl A a))
  : Equ (C a a (refl A a)) d (J A a C d a (refl A a))
 := λ (A : U)
      (a : A)
      (C : Π (x : A) (y : A), PathP (<\_> A) x y → U),
      (d : C a a (<\_> a)),
      <j> transp (<\_> C a a (<\_> a)) −j d
```

Transport is defined on fibrant types (only) and type checker should cover all the cases Note that transp^i (Path^j A v w) φ u_0 case is relying on comp operation which depends on hcomp primitive. Here is given the first part of Simon Huber equations [3] for **transp**:

```
transp^i N φ u₀ =u₀
transp^i U φ A =A
transp^i (Π (x : A), B) φ u₀ v =transp^i B(x/w) φ (u₀ w(i/0))
transp^i (Σ (x : A), B) φ u₀ =(transp^i A φ (u₀.1),transp^i B(x/v) φ (u₀.2))
transp^i (Path^j v w) φ u₀ =<j> comp^i A [φ u₀ j, (j=0) ↦ v, (j=1) ↦ w] (u₀ j)
transp^i (Glue [φ ↦ (T,w)] A) ψ u₀ =glue [φ(i/1) ↦ t'₁] a'₁ : B(i/1)
```

Partial Elements. In order to explicitly define hcomp we need to specify n-cubes where some faces are missing. Partial primitives isOne, 1=1 and UIP on pretypes are derivable in Anders due to landing strict equality Id in V universe. The idea is that (Partial A r) is the type of cubes in A that are only defined when IsOne r holds. (Partial A r) is a special version of the function space IsOne r → A with a more extensional equality: two of its elements are considered judgmentally equal if they represent the same subcube of A. They are equal whenever they reduce to equal terms for all the possible assignment of variables that make r equal to 1.

```
def Partial' (A : U) (i : I) := Partial A i
def isOne : I -> V := Id I 1
def 1=>1 : isOne 1 := ref 1
def UIP (A : V) (a b : A) (p q : Id A a b) : Id (Id A a b) p q := ref p
```

Cubical Subtypes. For (A : U) (i : I) (Partial A i) we can define subtype A [i ↦ u]. A term of this type is a term of type A that is definitionally equal to u when (IsOne i) is satisfied. We have forth and back fusion rules ouc (inc v) = v and inc (outc v) = v. Moreover, ouc v will reduce to u 1=1 when i=1.

```
def sub' (A : U) (i : I) (u : Partial A i) : V := A [i ↦ u ]
def inc' (A : U) (i : I) (a : A) : A [i ↦ [(i =1) → a]] := inc A i a
def ouc' (A : U) (i : I) (u : Partial A i) (a : A [i ↦ u]) : A := ouc a
```

Homogeneous Composition. hcomp is the answer to second problem: with hcomp and transp one can express path composition, groupoid, category of groupoids (groupoid interpretation and internalization in type theory). One of the main roles of homogeneous composition is to be a carrier in [higher] inductive type constructors for calculating of homotopy colimits and direct encoding of CW-complexes. Here is given the second part of Simon Huber equations [3] for **hcomp**:

```
hcomp^i N [φ ↦ 0] 0 =0
hcomp^i N [φ ↦ S u] (S u0) =S (hcomp^i N [φ ↦ u] u0)
hcomp^i U [φ ↦ E] A =Glue [φ ↦ (E(i/1), equiv^i E(i/1−i))] A
hcomp^i (Π (x : A), B) [φ ↦ u] u0 v =hcomp^i B(x/v) [φ ↦ u v] (u0 v)
hcomp^i (Σ (x : A), B) [φ ↦ u] u0 =(v(i/1), comp^i B(x/v) [φ ↦ u.2] u0.2)
hcomp^i (Path^j A v w) [φ ↦ u] u0 =<j> hcomp^i A[φ ↦ u j, (j=0) ↦ v, (j=1) ↦ w] (u0 j)
hcomp^i (Glue [φ ↦ (T,w)] A) [ψ ↦ u] u0 =glue [φ ↦ u(i/1)] (unglue u(i/1))
```

14.4 Strict Equality

To avoid conflicts with path equalities which live in fibrant universes strict equalities live in pretypes universes.

```
def strict (lang : U) : U
 := inductive { Id: name → lang
              | ref: lang → lang
              | idJ: lang → lang → lang
              }
```

We use strict equality in HTS for pretypes and partial elements which live in V. The presheaf configuration with Pi, Sigma and Id is called **MLTT-75** [5]. The

presheaf configuration with Pi, Sigma, Id and Path is called **HTS** (Homotopy Type System).

14.5 Glue Types

The main purpose of Glue types is to construct a cube where some faces have been replaced by equivalent types. This is analogous to how hcomp lets us replace some faces of a cube by composing it with other cubes, but for Glue types you can compose with equivalences instead of paths. This implies the univalence principle and it is what lets us transport along paths built out of equivalences.

```
def glue (lang : U) : U
 := inductive { Glue: lang → lang → lang
              | glue: lang → lang
              | unglue: lang → lang
              }
```

Basic Fibrational HoTT core by Pelayo, Warren, and Voevodsky (2012).

```
def fiber (A B : U) (f: A → B) (y : B): U := Σ (x : A), Path B y (f x)
def isEquiv (A B : U) (f : A → B) : U := Π (y : B), isContr (fiber A B f y)
def equiv (A B : U) : U := Σ (f : A →B), isEquiv A B f
def contrSingl (A : U) (a b : A) (p : Path A a b)
  : Path (Σ (x : A), Path A a x) (a,<i>a) (b,p) := <i> (p @ i, <j> p @ i ∨ j)
def idIsEquiv (A : U) : isEquiv A A (id A)
 := λ (a : A), ((a,<i>a), λ (z : fiber A A (id A) a), contrSingl A a z.1 z.2)
def idEquiv (A : U) : equiv A A := (id A, isContrSingl A)
```

The notion of Univalence was discovered by Vladimir Voevodsky as forth and back transport between fibrational equivalence as contractability of fibers and homotopical multi-dimentional heterogeneous path equality. The Equiv → Path type is called Univalence type, where univalence intro is obtained by Glue type and elim (Path → Equiv) is obtained by sigma transport from constant map.

```
def univ-formation (A B : U) := equiv A B → PathP (<i> U) A B
def univ-intro (A B : U) : univ-formation A B := λ (e : equiv A B),
    <i> Glue B (∂ i) [(i =0) → (A, e), (i = 1) → (B, idEquiv B)]
def univ-elim (A B : U) (p : PathP (<i> U) A B)
  : equiv A B := transp (<i> equiv A (p @ i)) 0 (idEquiv A)
def univ-computation (A B : U) (p : PathP (<i> U) A B)
  : PathP (<i> PathP (<i> U) A B) (univ-intro A B (univ-elim A B p)) p
 := <j i> Glue B (j ∨ ∂ i)
    [ (i = 0) → (A, univ-elim A B p), (i = 1) → (B, idEquiv B),
      (j = 1) → (p @ i, univ-elim (p @ i) B (<k> p @ (i ∨ k)))]
```

Similar to Fibrational Equivalence the notion of Retract/Section based Isomorphism could be introduced as forth-back transport between isomorphism and path equality. This notion is somehow cannonical to all cubical systems and is called Unimorphism here.

```
def iso-Form (A B: U) : U1 := iso A B -> PathP (<i>U) A B
def iso-Intro (A B: U) : iso-Form A B
 := λ (x : iso A B), isoPath A B x.f x.g x.s x.t
```

```
def iso-Elim (A B : U) : PathP (<i> U) A B -> iso A B
 := λ (p : PathP (<i> U) A B),
      (coerce A B p, coerce B A (<i> p @ -i),
      trans^-1-trans A B p, λ (a : A), <k> trans-trans^-1 A B p a @-k, ⋆)
```

Orton-Pitts basis for univalence computability (2017):

```
def ua (A B : U) (p : equiv A B) : PathP (<i> U) A B := univ-intro A B p
def ua-β (A B : U) (e : equiv A B) : Path (A→B) (trans A B (ua A B e)) e.1
 := <i> λ (x : A), (idfun=idfun'' B @ -i)
    (idfun=idfun'' B @ -i) ((idfun=idfun' B @ -i) (e.1 x)) )
```

14.6 de Rham Stack

Stack de Rham or Infinitezemal Shape Modality is a basic primitive for proving theorems from synthetic differential geometry. This type-theoretical framework was developed for the first time by Felix Cherubini under the guidance of Urs Schreiber. The Anders prover implements the computational semantics of the de Rham stack.

```
def ι (A : U) (a : A) : ℑ A := ℑ-unit a
def μ (A : U) (a : ℑ (ℑ A)) := ℑ-join a
def is-coreduced (A : U) : U := isEquiv A (ℑ A) (ι A)
def ℑ-coreduced (A : U) : is-coreduced (ℑ A)
 := isoToEquiv (ℑ A) (ℑ (ℑ A)) (ι (ℑ A)) (μ A)
    (λ (x : ℑ (ℑ A)), <i>x) (λ (y : ℑ A),<i>y)
def ind-ℑβ (A : U) (B : ℑ A→U) (f : Π (a : A), ℑ (B (ι A a))) (a : A)
  : Path (ℑ (B (ι A a))) (ind-ℑ A B f (ι A a)) (f a) := <i> f a
def ind-ℑ-const (A B : U) (b : ℑ B) (x : ℑ A)
  : Path (ℑ B) (ind-ℑ A (λ (i : ℑ A), B) (λ (i : A), b) x) b := <i> b
```

Coreduced induction and its β−quation.

```
def ℑ-ind (A : U) (B : ℑ A→U) (c : Π (a : ℑ A),
    is-coreduced (B a)) (f : Π (a : A), B (ι A a)) (a : ℑ A) : B a
 := (c a (ind-ℑ A B (λ (x : A), ι (B (ι A x)) (f x)) a)).1.1
def ℑ-indβ (A : U) (B : ℑ A→U) (c : Π (a : ℑ A),
    is-coreduced (B a)) (f : Π (a : A), B (ι A a)) (a : A)
  : Path (B (ι A a)) (f a) ((ℑ-ind A B c f) (ι A a))
 := <i> sec-equiv (B (ι A a)) (ℑ (B (ι A a)))
    (ι (B (ι A a)), c (ι A a)) (f a) @-i
```

Geometric Modal HoTT Framework: Infinitesimal Proximity, Formal Disk, Formal Disk Bundle, Differential.

```
def ∼ (X : U) (a x' : X) : U := Path (ℑ X) (ι X a) (ι X x')
def 𝔻 (X : U) (a : X) : U := Σ (x' : X), ∼ X a x'
def inf-prox-ap (X Y : U) (f : X → Y) (x x' : X) (p : ∼ X x x')
  : ∼ Y (f x) (f x') := <i> ℑ-app X Y f (p @ i)
def T∞ (A : U) : U := Σ (a : A), 𝔻 A a
def inf-prox-ap (X Y : U) (f : X → Y) (x x' : X) (p : ∼ X x x')
  : ∼ Y (f x) (f x') := <i> ℑ-app X Y f (p @ i)
def d (X Y : U) (f : X → Y) (x : X) (ε : 𝔻 X x)
  : 𝔻 Y (f x) := (f ε.1, inf-prox-ap X Y f x ε.1 ε.2)
def T∞-map (X Y : U) (f : X→Y) (τ : T∞ X) : T∞ Y := (f τ.1, d X Y f τ.1 τ.2)
```

14.7 Inductive Types

Anders currently don't support Lean-compatible generic inductive schemes definition. So instead of generic inductive schemes Anders supports well-founded trees (W-types). Basic data types like List, Nat, Fin, Vec are implemented as W-types in base library.

- W, 0, 1, 2 basis of MLTT-80 (Martin-Löf)
- General Schemes of Inductive Types (Paulin-Mohring)

14.8 Higher Inductive Types

As for higher inductive types Anders has Three-HIT foundation (Coequalizer, Path Coequalizer and Colimit) to express other HITs. Also there are other foundations to consider motivated by typical tasks in homotopy (type) theory:

- Coequalizer, Path Coequalizer and Colimit (van der Weide)
- Suspension, Truncation, Quotient (Groupoid Infinity)
- General Schemes of Higher Inductive Types (Cubical Agda)

14.9 Simplicial Types

Modification of Anders with Simplicial types and Hopf Fibrations built intro the core of type checker is called **Dan** with following recursive syntax (having f as Simplecies and *coh* as Path-coherence functions):

```
simplex n [v_0 .. v_n] { f_0, f_1, ..., f_n | coh i_1 i_2 ... i_n } : Simplex
```

and instantiation example:

```
def s_∞ : Simplicial
 := Π (v e : Simplex),
      δ_10 =v, δ_11 =v, s_0 <v,
      δ_20 =e ∘ e, s_10 <δ_20
      ⊢ ∞ (v, e, δ_20 | δ_10 δ_11, s_0, δ_20, s_10)
```

15 Properties

Soundness and completeness link syntax to semantics. Canonicity, normalization, and totality ensure computational adequacy. Consistency and decidability guarantee logical and practical usability. Conservativity and initiality support extensibility and universality.

15.1 Soundness and Completeness

Soundness is proven via cubical sets [11, 12, 13].

15.2 Canonicity, Normalization and Totality

Canonicity and normalization hold constructively [14, 15].

15.3 Consistency and Decidability

Consistency follows from the model [16]. Decidability is achieved for type checking [13].

15.4 Conservativity and Initiality

Conservativity and initiality is discussed bu Shulman[18, 17]. Initiality is implicit in the syntactic construction [12].

16 Conclusion

This paper presents Anders, a proof assistant that reimplements cubicaltt within a Modal Homotopy Type System framework, based on MLTT-80 and CCHM/CHM. It integrates HTS strict equality, infinitesimal modalities, and primitives like suspensions or quotients, with the extension adding simplicial types and Hopf fibrations. Anders offers an efficient, idiomatic system — compiling in under one second — using a syntax of Lean and semantics of cubicaltt and Cubical Agda. As a practical refinement of cubicaltt, Anders serves as an accessible tool for homotopy type theory, with potential for incremental enhancements like a tactic language.

Література

[1] Felix Cherubini. *Cartan Geometry in Modal Homotopy Type Theory*. 2019. https://arxiv.org/pdf/1806.05966.pdf.

[2] Thierry Coquand, Simon Huber, and Anders Mörtberg. *On Higher Inductive Types in Cubical Type Theory*. 2017. https://staff.math.su.se/anders.mortberg/papers/cubicalhits.pdf.

[3] Simon Huber. *On Higher Inductive Types in Cubical Type Theory*. 2017. http://www.cse.chalmers.se/~simonhu/misc/hcomp.pdf.

[4] Per Martin-Löf. *An Intuitionistic Theory of Types*. 1972.

[5] Per Martin-Löf. *An Intuitionistic Theory of Types: Predicative Part*. 1975.

[6] Per Martin-Löf. *Intuitionistic Type Theory*. 1980. https://raw.githubusercontent.com/michaelt/martin-lof/master/pdfs/Bibliopolis-Book-retypeset-1984.pdf.

[7] Christine Paulin-Mohring. *Introduction to the Calculus of Inductive Constructions*. 2015. https://hal.inria.fr/hal-01094195/document.

[8] Vladimir Voevodsky. *A simple type system with two identity types*. 2013. https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/HTS.pdf.

[9] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. *Two-Level Type Theory and Applications*. 2019. https://arxiv.org/pdf/1705.03307.pdf.

[10] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. *Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types*. 2019. https://staff.math.su.se/anders.mortberg/papers/cubicalagda.pdf.

[11] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. *Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom*. 2018. https://staff.math.su.se/anders.mortberg/papers/cubicalagda.pdf.

[12] Steve Awodey. *Type Theory and Homotopy*. In *Epistemology versus Ontology: Essays on the Philosophy and Foundations of Mathematics in Honour of Per Martin-Löf*, pages 183–201. Springer, 2012.

[13] Thierry Coquand. *A Survey of Constructive Models of Univalence*. 2018. Preprint or lecture notes.

[14] Simon Huber. *Canonicity for Cubical Type Theory*. *Journal of Automated Reasoning*, 61(1–4):173–205, 2017.

[15] Thomas Streicher. *Semantics of Type Theory: Correctness, Completeness and Independence Results*. Birkhäuser, Basel, 1991.

[16] Marc Bezem, Thierry Coquand, and Simon Huber. *A Model of Type Theory in Cubical Sets*. arXiv preprint arXiv:1406.1731, 2014.

[17] Michael Shulman. *Univalence for Inverse Diagrams and Homotopy Canonicity. Mathematical Structures in Computer Science*, 25(5):1203–1277, 2015.

[18] Martin Hofmann. *Syntax and Semantics of Dependent Types*. In *Semantics and Logics of Computation*, pages 79–130. Cambridge University Press, 1997.

Issue XX: Analytical Type System

М.Е. Сохацький [1]

[1] Національний технічний університет України
ім. Ігоря Сікорського
26 листопада 2017

Анотація

The formalization of mathematical analysis in proof assistants has advanced significantly with systems like Lean and Coq, which have mechanized key results in functional analysis, such as Bochner integration, L^2 spaces, and the theory of distributions. This article introduces Laurent, a novel proof assistant built on MLTT-72, a minimal Martin-Löf Type Theory with Pi and Sigma types, omitting identity types (e.g., `Id`, `J`) in favor of `Prop` predicates and truncated Sigma types. Laurent embeds explicit primitives for calculus, measure theory, and set theory with open sets and topology directly into its core, complemented by a tactics language inspired by Lean, Coq, and recent near tactics. Designed to unify classical and constructive analysis, it targets the mechanization of Laurent Schwartz's Théorie des Distributions and Analyse Mathématique alongside Errett Bishop's Foundations of Constructive Analysis. We present its foundational constructs and demonstrate its application to theorems in sequences, Lebesgue integration, L^2 spaces, and distributions, arguing that its design offers an intuitive yet rigorous approach to analysis, appealing to classical analysts while preserving constructive precision. Laurent emerges as a specialized tool for computational mathematics, advancing the mechanization of functional analysis.

17 Introduction to Laurent

The mechanization of mathematical theorems has transformed modern mathematics, enabling rigorous verification of proofs through computational tools known as proof assistants. Systems like Lean and Coq have emerged as leaders in this field, leveraging dependent type theory to formalize a wide range of mathematical domains.

Despite their successes, Lean and Coq often rely on extensive libraries (e.g., Lean's `mathlib` or Coq's Mathematical Components) and general-purpose tactics—such as `ring`, `field`, or `linearith`—that, while effective, can feel detached from the intuitive reasoning of classical analysis. This gap has inspired the development of Laurent, a proof assistant tailored for mathematical analysis, functional analysis, and distribution theory. Laurent integrates explicit primitives for sets, measures, and calculus into its core, paired with a tactics language akin to Lean and Coq, augmented by recent innovations like `near` tactics [1]. This design aims to reflect the spirit of classical mathematics while enabling constructive theorem-proving, offering a specialized tool for researchers in functional analysis.

This article outlines Laurent's architecture and demonstrates its mechanization of classical and constructive theorems, drawing on examples from sequences, Lebesgue integration, and L^2 spaces. We target formal mathematics audience emphasizing computational mathematics and frontier research in functional analysis.

$$
\begin{gathered}
\text{Laurent} := \text{MLTT} \mid \text{CALC} \\
\text{MLTT} := \text{Cosmos} \mid \text{Var} \mid \text{Forall} \mid \text{Exists} \\
\text{CALC} := \text{Base} \mid \text{Set} \mid \text{Q} \mid \text{Mu} \mid \text{Lim} \\
\text{Cosmos} := \mathbf{Prop} \ : \ \mathbf{U_0} \ : \ \mathbf{U_1} \\
\text{Var} := \mathbf{var}\ \text{ident} \mid \mathbf{hole} \\
\text{Forall} := \forall\ \text{ident E E} \mid \lambda\ \text{ident E E} \mid \text{E E} \\
\text{Exists} := \exists\ \text{ident E E} \mid (\text{E}, \text{E}) \mid \text{E}.\mathbf{1} \mid \text{E}.\mathbf{2} \\
\text{Base} := \mathbb{N} \mid \mathbb{Z} \mid \mathbb{Q} \mid \mathbb{R} \mid \mathbb{C} \mid \mathbb{H} \mid \mathbb{O} \mid \mathbb{V}^n \\
\text{Set} := \mathbf{Set} \mid \mathbf{SeqEq} \mid \mathbf{And} \mid \mathbf{Or} \mid \mathbf{Complement} \mid \mathbf{Intersect} \\
\mid \mathbf{Power} \mid \mathbf{Closure} \mid \mathbf{Cardinal} \\
\text{Q} := -/\!\sim \mid \mathbf{Quot} \mid \mathbf{Lift_Q} \mid \mathbf{Ind_Q} \\
\text{Mu} := \mathbf{mu} \mid \mathbf{Measure} \mid \mathbf{Lebesgue} \mid \mathbf{Bochner} \\
\text{Lim} := \mathbf{Seq} \mid \mathbf{Sup} \mid \mathbf{Inf} \mid \mathbf{Limit} \mid \mathbf{Sum} \mid \mathbf{Union}
\end{gathered}
$$

18 Lean and Coq in Functional Analysis

Lean, developed by Leonardo de Moura, is built on a dependent type theory variant of the Calculus of Inductive Constructions (CIC), with a small inference kernel and strong automation. Its mathematical library, `mathlib`, includes for-

malizations of Lebesgue measure, Bochner integration, and L^2 spaces, upporting proofs up to research-level mathematics. Tactics like `norm_num` and `continuity` automate routine steps, though their generality can obscure domain-specific insights.

Both systems, while powerful, prioritize generality over domain-specific efficiency [2]. Laurent addresses this by embedding analysis primitives directly into its core, inspired by recent advancements in near tactics, which enhance proof search with contextual awareness.

19 The Laurent Theorem Prover

Laurent is designed to mechanize theorems in classical and constructive analysis with a focus on functional analysis. Its core is built on dependent types—Pi (functions) and Sigma (pairs)—augmented by explicit primitives for sets, measures, and calculus operations. Unlike Lean and Coq, where such notions are library-defined, Laurent's primitives are native, reducing abstraction overhead and aligning with classical mathematical notation.

19.1 Basic Constructs and Set Theory

Laurent's syntax begins with fundamental types: natural numbers ($\mathbb{N}$), integers ($\mathbb{Z}$), rationals ($\mathbb{Q}$), reals ($\mathbb{R}$), complex numbers ($\mathbb{C}$), quaternions ($\mathbb{H}$), octanions ($\mathbb{O}$) and n-vectors ($\mathbb{V}^n$) all embedded in the core. Sets are first-class objects, defined using lambda abstractions. For example:

```
let set_a : exp =
  Set (Lam ("x", Real,
    RealIneq (Gt, Var "x", Zero)))
```

represents the set $\{x : \mathbb{R} \mid x > 0\}$. Operations like supremum and infimum are built-in:

$$\sup\{x > 0\} = +\infty,$$
$$\inf\{x > 0\} = 0,$$

computed via `Sup set_a` and `Inf set_a`, reflecting the unbounded and bounded-below nature of the positive reals.

19.2 Measure Theory and Integration

Measure theory is central to functional analysis, and Laurent embeds Lebesgue measure as a primitive:

```
let interval_a_b (a : exp) (b : exp) : exp =
  Set (Lam ("x", Real,
   And (RealIneq (Lte, a, Var "x"),
        RealIneq (Lte, Var "x", b))))

let lebesgue_measure (a : exp) (b : exp) : exp =
```

```
  Mu (Real, Power (Set Real), Lam ("A", Set Real,
    If (RealIneq (Lte, a, b),
        RealOps (Minus, b, a),
        Infinity)))
```

This defines $\mu([a,b]) = b - a$ for $a \leq b$, otherwise ∞. The Lebesgue integral is then constructed:

```
let integral_term : exp =
  Lam ("f", Forall ("x", Real, Real), Lam ("a", Real, Lam ("b", Real,
    Lebesgue (Var "f", Mu (Real, Power (Set Real), Lam ("A", Set Real,
      If (And (RealIneq (Lte, Var "a", Var "b"),
               SetEq (Var "A", interval_a_b (Var "a") (Var "b"))),
          RealOps (Minus, Var "b", Var "a"), Zero))),
      interval_a_b (Var "a") (Var "b")))))
```

representing $\int_{[a,b]} f \, d\mu$, with type signature $f, a, b : \mathbb{R} \to \mathbb{R}$.

19.3 L^2 Spaces

The L^2 space, critical in functional analysis, is defined as:

```
let l2_space : exp =
  Lam ("f", Forall ("x", Real, Real),
    RealIneq (Lt,
      Lebesgue (Lam ("x", Real,
        RealOps (Pow, RealOps (Abs, App (Var "f", Var "x"), Zero),
        RealOps (Plus, One, One))),
        lebesgue_measure Zero Infinity, interval_a_b Zero Infinity),
      Infinity))
```

This encodes $\{f : \mathbb{R} \to \mathbb{R} \mid \int_0^\infty |f(x)|^2 \, d\mu < \infty\}$, leveraging Laurent's measure and integration primitives.

19.4 Sequences and Limits

Laurent mechanizes classical convergence proofs efficiently. Consider the sequence $a_n = \frac{1}{n}$:

```
let sequence_a : exp =
  Lam ("n", Nat, RealOps (Div, One, NatToReal (Var "n")))

let limit_a : exp =
  Limit (Seq sequence_a, Infinity, Zero,
    Lam ("ε", Real, Lam ("p", RealIneq (Gt, Var "ε", Zero),
      Pair (RealOps (Div, One, Var "ε"),
        Lam ("n", Nat, Lam ("q", RealIneq (Gt, Var "n", Var "N"),
          RealIneq (Lt, RealOps (Abs,
          RealOps (Minus, App (sequence_a, Var "n"), Zero), Zero),
            Var "ε")))))))
```

This proves $\lim_{n \to \infty} \frac{1}{n} = 0$, with $\forall \varepsilon > 0$, $\exists N = \frac{1}{\varepsilon}$ such that $n > N$ implies $\left|\frac{1}{n}\right| < \varepsilon$.

20 Examples of Theorem Mechanization

Laurent's design excels in mechanizing foundational theorems across differential calculus, integral calculus, and functional analysis. Below, we present a selection of classical results formalized in Laurent, showcasing its explicit primitives and constructive capabilities.

20.1 Taylor's Theorem with Remainder

Taylor's Theorem provides an approximation of a function near a point using its derivatives. If $f : \mathbb{R} \to \mathbb{R}$ is n-times differentiable at a, then:

$$f(x) = \sum_{k=0}^{n-1} \frac{f^{(k)}(a)}{k!}(x-a)^k + R_n(x),$$

where $R_n(x) = o((x-a)^{n-1})$ as $x \to a$.

In Laurent this encodes the theorem's structure, with diff_k representing the k-th derivative and 'remainder' satisfying the little-o condition, verifiable via Laurent's limit primitives.

20.2 Fundamental Theorem of Calculus

The Fundamental Theorem of Calculus links differentiation and integration. If f is continuous on $[a, b]$, then $F(x) = \int_a^x f(t)\,dt$ is differentiable, and $F'(x) = f(x)$:

Laurent's 'Lebesgue' primitive and 'diff' operator directly capture the integral and derivative, aligning with classical intuition.

20.3 Lebesgue Dominated Convergence Theorem

In functional analysis, the Dominated Convergence Theorem ensures integral convergence under domination. If $f_n \to f$ almost everywhere, $|f_n| \leq g$, and $\int g < \infty$, then $\int f_n \to \int f$: This leverages Laurent's sequence and measure primitives, with 'Limit' automating convergence proofs via near tactics.

20.4 Schwartz Kernel Theorem

For distributions, the Schwartz Kernel Theorem states that every continuous bilinear form $B : \mathcal{D}(\mathbb{R}^n) \times \mathcal{D}(\mathbb{R}^m) \to \mathbb{R}$ is represented by a distribution $K \in \mathcal{D}'(\mathbb{R}^n \times \mathbb{R}^m)$ such that $B(\phi, \psi) = \langle K, \phi \otimes \psi \rangle$: This uses Sigma types to pair the kernel K with its defining property, reflecting Laurent's support for advanced functional analysis.

20.5 Banach Space Duality

In Banach spaces, there's a bijection between closed subspaces of X and X^* via annihilators: $A \mapsto A^\perp$, $B \mapsto {}^\perp B$. Laurent formalizes this as:

```
let bijection_theorem = Π (Set Real, ("X",
  If (banach_space (Var "X"),
    And (
      Π (Set (Var "X"), ("A",
        If (closed_subspace (Var "X", Var "A"),
            Id (Set (Var "X"), Var "A", pre_annihilator (Var "X",
                annihilator (Var "X", Var "A"))), Bool))),
      Π (Set (dual_space (Var "X")), ("B",
        If (closed_subspace (dual_space (Var "X"), Var "B"),
            Id (Set (dual_space (Var "X")), Var "B", annihilator (Var "X",
                pre_annihilator (Var "X", Var "B"))), Bool))))), Bool)))
```

This showcases Laurent's ability to handle normed spaces and duality, critical in functional analysis.

20.6 Banach-Steinhaus Theorem

The Banach-Steinhaus Theorem ensures uniform boundedness of operators.

If $\sup_{\alpha \in A} \|T_\alpha x\|_Y < \infty$ for all $x \in X$, then there exists M such that $\|T_\alpha\|_{X \to Y} \leq M$:

This uses Laurent's norm and operator primitives, with near tactics simplifying boundedness proofs.

20.7 de Rham Theorem

The de Rham Theorem relates differential forms and integrals over loops. For an open $\Omega \subset \mathbb{R}^n$ and a C^1 1-form ω, if $\int_\gamma \omega = 0$ for all loops γ, there exists f such that $\omega = df$:

```
let de_rham_theorem =
  Π (Nat, ("n",
    Π (Set (Vec (n, Real, RealOps RPlus, RealOps RMult)), ("Omega",
      Π (one_form Omega n, ("omega",
        And (c1_form Omega n (Var "omega"),
          And (Π (loop Omega n, ("gamma",
              Id (Real, integral (Var "omega", Var "gamma"), zero))),
            Σ (zero_form Omega, ("f", And (
                Id (one_form Omega n, Var "omega", differential (Var "f")),
                Π (Nat, ("m", If (cm_form Omega n (Var "m") (Var "omega"),
                  cm_form Omega n (Var "m") (Var "f"), Bool))))))))))))))
```

This demonstrates Laurent's capacity for topology and differential geometry, integrating forms and limits.

These examples highlight Laurent's versatility, from basic calculus to advanced functional analysis, leveraging its native primitives and tactics for intuitive yet rigorous mechanization.

21 Core Tactics of General Proof Assistant

Laurent's proof assistant leverages a rich tactics language to mechanize theorems in functional analysis, blending classical intuition with constructive rigor. Unlike

general-purpose systems like Lean and Coq, Laurent's tactics are tailored to the domain-specific needs of analysis, incorporating explicit primitives for limits, measures, and algebraic structures. This section outlines key tactics used in Laurent, including specialized solvers for rings, fields, and linear arithmetic, and demonstrates their application to functional analysis proofs.

These tactics form the backbone of proof construction, mirroring Coq's logical framework but optimized for Laurent's syntax.

21.1 Intro

Introduces variables from universal quantifiers. For a goal $\forall x : \mathbb{R}, P(x)$, `intro x` yields a new goal $P(x)$ with x in the context.

21.2 Elim

Eliminates existential quantifiers or applies induction (not fully implemented in the current prototype).

21.3 Apply

Applies a lemma or hypothesis to the current goal (pending full implementation).

21.4 Exists

Provides a witness for an existential quantifier. For $\exists x : \mathbb{R}, P(x)$, `exists 0` substitutes $x = 0$ into $P(x)$.

21.5 Assumption

Closes a goal if it matches a hypothesis or simplifies to a trivial truth (e.g., $0 < \varepsilon$ when $\varepsilon > 0$ is in context).

21.6 Auto

Attempts to resolve goals using context hypotheses, ideal for trivial cases.

21.7 Split

Splits conjunctive goals ($P \wedge Q$) into subgoals P and Q.

22 Analysis-Specific Tactics of Laurent

For functional analysis, Laurent introduces tactics that exploit its calculus and measure primitives. These tactics leverage Laurent's `Limit`, `Lebesgue`, and

`RealIneq` primitives, reducing manual effort in limit and integration proofs compared to Lean's library-based approach.

22.1 Limit

Expands limit definitions. For a goal $\lim_{n\to\infty} a_n = L$, it generates:

$$\forall \varepsilon > 0, \exists N : \mathbb{N}, \forall n > N, |a_n - L| < \varepsilon,$$

enabling step-by-step convergence proofs. This is crucial for sequences like $\frac{1}{n} \to 0$.

22.2 Continuity

Unfolds continuity definitions at a point. For a goal `continuous_at (f, a)`, it generates:

$$\forall \varepsilon > 0, \exists \delta > 0, \forall x, |x - a| < \delta \implies |f(x) - f(a)| < \varepsilon,$$

transforming the target into an ε-δ formulation using Laurent's `RealIneq` primitives for inequalities and `RealOps` for arithmetic operations (e.g., absolute value, subtraction). This facilitates step-by-step proofs of continuity, such as for the Fundamental Theorem of Calculus, by exposing the logical structure directly in the prover's core, contrasting with Lean's reliance on library theorems.

22.3 Near

Introduces a neighborhood assumption. Given a goal involving a point a, `near x a` adds $x_{\text{near}} : \mathbb{R}$ and $\delta_x > 0$ with $|x_{\text{near}} - a| < \delta_x$, facilitating local analysis as in Taylor's Theorem.

22.4 ApplyLocally

Applies a local property (e.g., from a `near` assumption) to simplify the goal, automating steps in proofs like the Schwartz Kernel Theorem.

To handle the algebraic manipulations ubiquitous in functional analysis (e.g., norms, integrals), Laurent incorporates solvers inspired by Lean and Coq:

23 Algebraic Solvers

To handle the algebraic manipulations ubiquitous in functional analysis (e.g., norms, integrals), Laurent incorporates solvers inspired by Lean and Coq.

Lean's `ring` and `linarith` rely on `mathlib`, while Coq's `field` uses library-defined fields. Laurent embeds these solvers in its core, alongside analysis tactics, reducing dependency on external definitions. This design accelerates proofs in L^2 spaces, Banach duality, and distribution theory, aligning with the needs of a mathematical audience exploring frontier research in computational analysis.

23.1 Ring

Solves equalities in commutative rings. For example, it verifies:

$$(f(x) + g(x))^2 = f(x)^2 + 2f(x)g(x) + g(x)^2,$$

using $\mathbb{R}$'s ring structure. This is implemented via normalization and equality checking in Laurent's core.

23.2 Field

Resolves field equalities and inequalities involving division. For $\int_0^\infty |f(x)|^2 \, d\mu < \infty$, `field` simplifies expressions like:

$$\frac{f(x)^2}{g(x)^2} = \left(\frac{f(x)}{g(x)}\right)^2 \quad (g(x) \neq 0),$$

crucial for quotient manipulations in Banach spaces.

23.3 Big Number Normalization

Automates numerical simplification and equality checking for expressions involving rational numbers and basic functions. For a goal like $2 + 3 = 5$ or $|\sin(0)| = 0$, it evaluates:

$$\texttt{norm\_num} : e \mapsto r,$$

where e is an expression (e.g., $2/3+1/2$, $\ln(1)$), and r is either a rational number (via OCaml's `Num` library) or an unevaluated symbolic form. It supports operations including addition, subtraction, multiplication, division, exponentiation, absolute value, logarithms, and trigonometric functions, approximating transcendental results to high precision (e.g., 20 decimal places for sin, cos). This tactic is essential for verifying norm computations, such as $\|f\|_2^2 = \int |f(x)|^2 \, dx$, by reducing concrete numerical subgoals in Banach space proofs.

23.4 Inequality Set Predicates

Handles linear arithmetic inequalities. In the Banach-Steinhaus Theorem, it proves:

$$\|T_\alpha x\|_Y \leq M \|x\|_X,$$

by resolving linear constraints over $\mathbb{R}$, integrating seamlessly with `RealIneq` backed by Z3 SMT solver (morally correct for inequalities).

24 Discussion and Future Directions

Laurent has built-in primitives for streamline proofs in measure theory, integration, and L^2 spaces, while its tactics language ensures flexibility. Compared to Lean's library-heavy approach or Coq's constructive focus, Laurent balances

classical intuition with formal precision, making it accessible to analysts accustomed to paper-based reasoning. Future work includes expanding Laurent's tactics repertoire, formalizing advanced theorems (e.g., dominated convergence, distribution theory).

Hosted at [1], Laurent invites community contributions to refine its role in computational mathematics.

25 Conclusion

Laurent represents a specialized advancement in theorem mechanization, tailored for classical and constructive analysis. By embedding analysis primitives and leveraging topological tactics and algebraic solvers, it offers a unique tool for functional analysts, complementing the broader capabilities of Lean and Coq. This work underscores the potential of domain-specific proof assistants in advancing computational mathematics.

Література

[1] Affeldt R., Cohen C., Mahboubi A., Rouhling D., Strub P-Y. *Classical Analysis with Coq*, Coq Workshop 2018, Oxford, UK doi:

[2] Boldo S., Lelay C., Melquiond G. *Formalization of Real Analysis: A Survey of Proof Assistants and Libraries*, Mathematical Structures in Computer Science, 2016, 26 (7), pp.1196-1233. doi:10.1017/S0960129514000437

[3] Schwartz, L. *Analyse Mathématique*, Hermann, Paris, 1967.

[4] Bishop, E. *Foundations of Constructive Analysis*, McGraw-Hill, New York, 1967.

[5] Bridges, D. *Constructive Mathematics: A Foundation for Computable Analysis*, Theoretical Computer Science, 1999, 219 (1-2), pp.95–109.

[6] Booij, A. *Analysis in Univalent Type Theory*, PhD thesis, University of Birmingham, 2020. Available at: `https://etheses.bham.ac.uk/id/eprint/10411/7/Booij2020PhD.pdf`

[8] Ziemer, W. P., Torres, M. *Modern Real Analysis*, Springer, New York, 2017. Available at: `https://www.math.purdue.edu/~torresm/pubs/Modern-real-analysis.pdf`

[1] `https://github.com/groupoid/laurent`

Issue XXI: Super Type System

М.Е. Сохацький [1]

[1] Національний технічний університет України
ім. Ігоря Сікорського
26 листопада 2025

Анотація

Here is presented Groupoid Infinity language for TED-K.

26 Introduction to Urs

27 Super Type System

27.1 Bosonic Modality

The ◯ modality in cohesive type theory projects a type to bosonic parity ($g = 0$). For a type $A : \mathbf{U}_{i,g}$, $\bigcirc A$ forces the type to be bosonic, aligning with supergeometry and quantum physics.

In Urs, ◯ operates on types in graded universes from **Graded**, with applications in bosonic quantum fields **qubit** and supergeometry **SmthSet**.

27.2 Bose

Definition 9 (Bosonic Modality Formation)**.** The ◯ modality is a type operator on graded universes, mapping to bosonic parity:

$$\bigcirc : \prod_{i:\mathbb{N}} \prod_{g:\mathbf{Grade}} \mathbf{U}_{i,g} \to \mathbf{U}_{i,0}.$$

```
def bosonic (i : Nat) (g : Grade) (A : U i g) : U i 0
```

Definition 10 (Bosonic Modality Introduction)**.** Applying ◯ to a type A produces $\bigcirc A$ with bosonic parity:

$$\Gamma \vdash A : \mathbf{U}_{i,g} \quad \to \quad \Gamma \vdash \bigcirc A : \mathbf{U}_{i,0}.$$

Definition 11 (Bosonic Modality Elimination)**.** The eliminator for $\bigcirc A$ maps bosonic types to properties in $\mathbf{U_0}$:

$$\mathbf{Ind}_{\bigcirc} : \prod_{i:\mathbb{N}} \prod_{g:\mathbf{Grade}} \prod_{A:\mathbf{U}_{i,g}} \prod_{\phi:(\bigcirc A)\to\mathbf{U_0}} \left(\prod_{a:\bigcirc A} \phi\ a \right) \to \prod_{a:\bigcirc A} \phi\ a.$$

```
def bosonic_ind (i : Nat) (g : Grade) (A : U i g)
    (phi : (bosonic i g A) -> U_0)
    (h : Π (a : bosonic i g A), phi a)
  : Π (a : bosonic i g A), phi a
```

Theorem 2 (Idempotence of Bosonic)**.** The $\bigcirc$ modality is idempotent, as it always projects to bosonic parity:

$$\bigcirc\textbf{-idem} : \prod_{i:\mathbb{N}} \prod_{g:\mathbf{Grade}} \prod_{A:\mathbf{U}_{i,g}} (\bigcirc(\bigcirc A)) = (\bigcirc A).$$

```
def bosonic_idem (i : Nat) (g : Grade) (A : U i g)
  : (bosonic i 0 (bosonic i g A)) = (bosonic i g A)
```

Theorem 3 (Bosonic Qubits)**.** For $C, H : \mathbf{U_0}$, the type $\bigcirc\mathbf{Qubit}(C, H)$ models bosonic quantum states:

$$\bigcirc\textbf{-qubit} : \prod_{i:\mathbb{N}} \prod_{g:\mathbf{Grade}} \prod_{C,H:\mathbf{U_0}} (\bigcirc\mathbf{Qubit}(C, H)) : \mathbf{U}_{i,0}.$$

```
def bosonic_qubit (i : Nat) (g : Grade) (C H : U_0) : U i 0
 := bosonic i g (Qubit C H)
```

27.3 Braid

The $\mathbf{Braid}_n(X)$ type models the braid group $B_n(X)$ on n strands over a smooth set $X : \mathbf{SmthSet}$, the fundamental group of the configuration space $\mathbf{Conf}^n(X)$, used in knot theory, quantum computing, and smooth geometry.

In Urs, $\mathbf{Braid}_n(X)$ is a type in $\mathbf{U_0}$, parameterized by $n : \mathbf{Nat}$ and $X : \mathbf{SmthSet}$, supporting braid generators σ_i and relations, with applications to anyonic quantum gates and knot invariants.

Definition 12 (Braid Formation)**.** The type $\mathbf{Braid}_n(X)$ is formed for each $n : \mathbf{Nat}$ and $X : \mathbf{SmthSet}$:

$$\mathbf{Braid} : \prod_{n:\mathbf{Nat}} \prod_{X:\mathbf{SmthSet}} \mathbf{U_0}.$$

```
def Braid (n : Nat) (X : SmthSet) : U_0
(* Braid group type *)
```

Definition 13 (Braid Introduction)**.** Terms of type $\mathbf{Braid}_n(X)$ are introduced via the **braid** constructor, representing generators σ_i for $i : \mathbf{Fin}\ (n-1)$:

$$\mathbf{braid} : \prod_{n:\mathbf{Nat}} \prod_{X:\mathbf{SmthSet}} \prod_{i:\mathbf{Fin}\ (n-1)} \mathbf{Braid}_n(X).$$

```
def braid (n : Nat) (X : SmthSet) (i : Fin (n-1)) : Braid n X
(* Braid generator sigma_i *)
```

Definition 14 (Braid Elimination)**.** The eliminator for $\mathbf{Braid}_n(X)$ maps braid elements to properties in $\mathbf{U_0}$:

$$\mathbf{BraidInd} : \prod_{n:\mathbf{Nat}} \prod_{X:\mathbf{SmthSet}} \prod_{\beta:\mathbf{Braid}_n(X)\to\mathbf{U_0}} \left(\prod_{b:\mathbf{Braid}_n(X)} \beta\ b \right) \to \prod_{b:\mathbf{Braid}_n(X)} \beta\ b.$$

```
def braid_ind (n : Nat) (X : SmthSet)
    (beta : Braid n X -> U_0)
    (h : Π (b : Braid n X), beta b)
  : Π (b : Braid n X), beta b
```

Theorem 4 (Braid Relations)**.** For $n : \mathbf{Nat}$, $X : \mathbf{SmthSet}$, $\mathbf{Braid}_n(X)$ satisfies the braid group relations (Commutation and Yang-Baxter):

$$\prod_{n:\mathbf{Nat}} \prod_{X:\mathbf{SmthSet}} \prod_{i,j:\mathbf{Fin}\ (n-1),\ |i-j|\geq 2} \sigma_i \cdot \sigma_j = \sigma_j \cdot \sigma_i,$$

$$\prod_{n:\mathbf{Nat}} \prod_{X:\mathbf{SmthSet}} \prod_{i:\mathbf{Fin}\ (n-2)} \sigma_i \cdot \sigma_{i+1} \cdot \sigma_i = \sigma_{i+1} \cdot \sigma_i \cdot \sigma_{i+1}.$$

```
def braid_rel_comm (n : Nat) (X : SmthSet) (i j : Fin (n-1))
  : Path (braid i · braid j) (braid j · braid i)

def braid_rel_yang (n : Nat) (X : SmthSet) (i : Fin (n-2))
  : Path (braid i · braid (i+1) · braid i) (braid (i+1) · braid i ·
braid (i+1))
```

Theorem 5 (Configuration Space Link)**.** For $n : \mathbf{Nat}$, $X : \mathbf{SmthSet}$, $\mathbf{Braid}_n(X)$ is the fundamental groupoid of $\mathbf{Conf}^n(X)$:

$$\prod_{n:\mathbf{Nat}} \prod_{X:\mathbf{SmthSet}} \mathbf{Braid}_n(X) \cong \pi_1(\mathbf{Conf}^n(X)).$$

```
def braid_conf (n : Nat) (X : SmthSet)
  : Path (Braid n X) (pi_1 (Conf n X))
```

Theorem 6 (Quantum Braiding)**.** For $C, H : \mathbf{U_0}$, $\mathbf{Braid}_n(X)$ acts on $\mathbf{Qubit}(C,H)^{\otimes n}$ as braiding operators:

$$\mathbf{braid\_qubit} : \prod_{n:\mathbf{Nat}} \prod_{C,H:\mathbf{U_0}} \prod_{X:\mathbf{SmthSet}} \mathbf{Braid}_n(X) \to \left(\mathbf{Qubit}(C,H)^{\otimes n} \to \mathbf{Qubit}(C,H)^{\otimes n}\right).$$

```
def braid_qubit (n : Nat) (C H : U_0) (X : SmthSet)
  : Braid n X -> (Qubit C H)^n -> (Qubit C H)^n
```

Theorem 7 (Braid Group Delooping)**.** For $n : \mathbf{Nat}$, the delooping $\mathbf{BB}_n$ of the braid group B_n is a 1-groupoid:

$$\mathbf{BB}_n : \mathbf{Grpd}\ 1 \equiv \Im(\mathbf{Conf}^n(\mathbb{R}^2)).$$

```
def BB_n (n : Nat) : Grpd 1 := ℑ (Conf n ℝ^2)
```

27.4 Graded Universes

Graded Universes. The $\mathbf{U}_\alpha$ type represents a graded universe indexed by a monoid $\mathcal{G} = \mathbb{N} \times \mathbb{Z}/2\mathbb{Z}$, where $\alpha \in \mathcal{G}$ encodes a level ($\mathbb{N}$) and parity ($\mathbb{Z}/2\mathbb{Z}$: 0 = bosonic, 1 = fermionic). Graded universes support type hierarchies with cumulativity, graded tensor products, and coherence rules, used in supergeometry (e.g., bosonic/fermionic types), quantum systems (e.g., graded qubits), and cohesive type theory.

In Urs, $\mathbf{U}_\alpha$ is a type indexed by $\alpha : \mathcal{G}$, with operations like lifting, product formation, and graded tensor products, extending standard universe hierarchies to include parity, building on **Tensor**.

Definition 15 (Grading Monoid)**.** The grading monoid $\mathcal{G}$ is defined as $\mathbb{N}\times\mathbb{Z}/2\mathbb{Z}$, with operation $\oplus$ and neutral element $\mathbf{0}$, encoding level and parity.

$$\begin{aligned} \mathcal{G} &: \mathbf{Type} \equiv \mathbb{N} \times \mathbb{Z}/2\mathbb{Z}, \\ \oplus &: \mathcal{G} \to \mathcal{G} \to \mathcal{G}, \\ (\alpha, \beta) &\mapsto (\text{fst } \alpha + \text{fst } \beta,\ (\text{snd } \alpha + \text{snd } \beta) \mod 2), \\ \mathbf{0} &: \mathcal{G} \equiv (0, 0). \end{aligned}$$

```
def 𝒢 : Type := ℕ × ℤ/2ℤ
def ⊕ (α β : 𝒢) : 𝒢 := (fst α + fst β, (snd α + snd β) mod 2)
def ↾ : 𝒢 := (0, 0)
```

Definition 16 (Graded Universe Formation)**.** The universe $\mathbf{U}_\alpha$ is a type indexed by $\alpha : \mathcal{G}$, containing types of grade α. A shorthand notation $\mathbf{U}_{i,g}$ is used for $\mathbf{U}\ (i, g)$.

$$\begin{aligned} \mathbf{U} &: \mathcal{G} \to \mathbf{Type}, \\ \mathbf{Grade} &: \mathbf{Set} \equiv \{0, 1\}, \\ \mathbf{U}_{i,g} &: \mathbf{Type} \equiv \mathbf{U}(i, g) : \mathbf{U}_{i+1}. \end{aligned}$$

```
def U (α : 𝒢) : Type := Universe α
def Grade : Set := {0, 1}
def U (i : Nat) (g : Grade) : Type := U (i, g)
def U₀ (g : Grade) : U (1, g) := U (0, g)
def U₀₀ : Type := U (0, 0)
def U₁₀ : Type := U (1, 0)
def U₀₁ : Type := U (0, 1)
```

Definition 17 (Graded Universe Coherence Rules)**.** Graded universes support coherence rules for lifting, product formation, and substitution, ensuring type-

theoretic consistency.

$$\begin{aligned}
\textbf{lift} &: \prod_{\alpha,\beta:\mathcal{G}} \prod_{\delta:\mathcal{G}} \mathbf{U}\ \alpha \to (\beta = \alpha \oplus \delta) \to \mathbf{U}\ \beta, \\
\textbf{univ} &: \prod_{\alpha:\mathcal{G}} \mathbf{U}\ (\alpha \oplus (1,0)), \\
\textbf{cumul} &: \prod_{\alpha,\beta:\mathcal{G}} \prod_{A:\mathbf{U}\ \alpha} \prod_{\delta:\mathcal{G}} (\beta = \alpha \oplus \delta) \to \mathbf{U}\ \beta, \\
\textbf{prod} &: \prod_{\alpha,\beta:\mathcal{G}} \prod_{A:\mathbf{U}\ \alpha} \prod_{B:A\to\mathbf{U}\ \beta} \mathbf{U}\ (\alpha \oplus \beta), \\
\textbf{subst} &: \prod_{\alpha,\beta:\mathcal{G}} \prod_{A:\mathbf{U}\ \alpha} \prod_{B:A\to\mathbf{U}\ \beta} \prod_{t:A} \mathbf{U}\ \beta, \\
\textbf{shift} &: \prod_{\alpha,\delta:\mathcal{G}} \prod_{A:\mathbf{U}\ \alpha} \mathbf{U}\ (\alpha \oplus \delta).
\end{aligned}$$

```
def lift (α β : 𝒢) (δ : 𝒢) (e : U α) : β = α ⊕ δ → U β :=
  λ eq : β = α ⊕ δ, transport (λ x : 𝒢, U x) eq e

def univ-type (α : 𝒢) : U (α ⊕ (1, 0)) :=
  lift α (α ⊕ (1, 0)) (1, 0) (U α) refl

def cumul (α β : 𝒢) (A : U α) (δ : 𝒢) : β = α ⊕ δ → U β :=
  lift α β δ A

def prod-rule (α β : 𝒢) (A : U α) (B : A → U β) : U (α ⊕ β) :=
  Π (x : A), B x

def subst-rule (α β : 𝒢) (A : U α) (B : A → U β) (t : A) : U β :=
  B t

def shift (α δ : 𝒢) (A : U α) : U (α ⊕ δ) :=
  lift α (α ⊕ δ) δ A refl
```

Definition 18 (Graded Tensor Introduction)**.** Graded tensor products combine types with matching levels, combining parities.

$$\begin{aligned}
\textbf{tensor} &: \prod_{i:\mathbb{N}} \prod_{g_1,g_2:\mathbf{Grade}} \mathbf{U}_{i,g_1} \to \mathbf{U}_{i,g_2} \to \mathbf{U}_{i,(g_1+g_2 \bmod 2)}, \\
\textbf{pair-tensor} &: \prod_{i:\mathbb{N}} \prod_{g_1,g_2:\mathbf{Grade}} \prod_{A:\mathbf{U}_{i,g_1}} \prod_{B:\mathbf{U}_{i,g_2}} \prod_{a:A} \prod_{b:B} \textbf{tensor}(i,g_1,g_2,A,B).
\end{aligned}$$

```
def tensor (i : Nat) (g₁ g₂ : Grade)
    (A : U i g₁) (B : U i g₂) : U i (g₁ + g₂ mod 2)
 := A ⊗ B

def pair-tensor (i : Nat) (g₁ g₂ : Grade) (A : U i g₁)
    (B : U i g₂) (a : A) (b : B) : tensor i g₁ g₂ A B
 := a ⊗ b
```

Definition 19 (Graded Tensor Eliminators)**.** Eliminators for graded tensor products project to their components.

$$\otimes\text{-}\mathbf{prj}_1 : (A \otimes B) \to A,$$
$$\otimes\text{-}\mathbf{prj}_2 : (A \otimes B) \to B.$$

```
def pr₁ (i : Nat) (g₁ g₂ : Grade)
    (A : U i g₁) (B : U i g₂) (p : A ⊗ B) : A := p.1

def pr₂ (i : Nat) (g₁ g₂ : Grade)
    (A : U i g₁) (B : U i g₂) (p : A ⊗ B) : B := p.2
```

Theorem 8 (Monoid Properties)**.** The grading monoid $\mathcal{G}$ satisfies associativity and identity laws.

$$\mathbf{assoc} : ((\alpha \oplus \beta) \oplus \gamma) = (\alpha \oplus (\beta \oplus \gamma)),$$
$$\mathbf{id\text{-}left} : (\alpha \oplus \mathbf{0}) = \alpha,$$
$$\mathbf{id\text{-}right} : (\mathbf{0} \oplus \alpha) = \alpha.$$

```
def assoc (α β γ : 𝒢) : (α ⊕ β) ⊕ γ = α ⊕ (β ⊕ γ) := refl
def ident-left (α : 𝒢) : α ⊕ ⊬ = α := refl
def ident-right (α : 𝒢) : ⊬ ⊕ α = α := refl
```

27.5 KU

The $\mathbf{KU^G}$ type represents generalized K-theory, a topological invariant used to classify vector bundles or operator algebras over a space, twisted by a groupoid. It is a cornerstone of algebraic topology and mathematical physics, with applications in quantum field theory, string theory, and index theory.

In the cohesive type system, $\mathbf{KU^G}$ operates on smooth sets **SmthSet** and groupoids $\mathbf{Grpd_1}$, producing a type in the universe $\mathbf{U}_{(\mathbf{0},\mathbf{0})}$. It incorporates a twist to account for non-trivial topological structures, making it versatile for modeling complex physical systems.

Definition 20 (KU^G-Formation)**.** The generalized K-theory type $\mathbf{KU^G}$ is formed over a term $X : \mathbf{U}_{(\mathbf{0},\mathbf{0})}$, a groupoid $G : \mathbf{U}_{(\mathbf{0},\mathbf{0})}$, and a twist $\tau : \prod_{x:X} \mathbf{U}_{(\mathbf{0},\mathbf{0})}$, yielding a type in the universe $\mathbf{U}_{(\mathbf{0},\mathbf{0})}$:

$$\mathbf{KU^G} : \prod_{X:\mathbf{U}_{(\mathbf{0},\mathbf{0})}} \prod_{G:\mathbf{U}_{(\mathbf{0},\mathbf{0})}} \prod_{\tau:\prod_{x:X} \mathbf{U}_{(\mathbf{0},\mathbf{0})}} \mathbf{U}_{(\mathbf{0},\mathbf{0})}.$$

```
type exp =
  | KU^G of exp * exp * exp
```

Definition 21 (KU^G-Introduction)**.** A term of type $\mathbf{KU}^{\mathbf{G}}(X, G, \tau)$ is introduced by constructing a generalized K-theory class, representing a stable equivalence class of vector bundles or operators over X, twisted by G and τ:

$$\mathbf{KU}^{\mathbf{G}} : \prod_{X:\mathbf{U}_{(\mathbf{0},\mathbf{0})}} \prod_{G:\mathbf{U}_{(\mathbf{0},\mathbf{0})}} \prod_{\tau:\prod_{x:X} \mathbf{U}_{(\mathbf{0},\mathbf{0})}} \mathbf{KU}^{\mathbf{G}}(X, G, \tau).$$

```
let KU^G_intro (x : exp) (g : exp) (tau : exp) : exp =
    KU^G (x, g, tau)
```

Definition 22 (KU^G-Elimination)**.** The eliminator for $\mathbf{KU}^{\mathbf{G}}$ allows reasoning about generalized K-theory classes by mapping them to properties or types dependent on $\mathbf{KU}^{\mathbf{G}}(X, G, \tau)$, typically by analyzing the underlying bundle or operator structure over X:

$$\mathbf{KU}^{\mathbf{G}}\mathbf{Ind} : \prod_{X:\mathbf{U}_{(\mathbf{0},\mathbf{0})}} \prod_{G:\mathbf{U}_{(\mathbf{0},\mathbf{0})}} \prod_{\tau:\prod_{x:X} \mathbf{U}_{(\mathbf{0},\mathbf{0})}} \prod_{\beta:\mathbf{KU}^{\mathbf{G}}(X,G,\tau)\to\mathbf{U}_{(\mathbf{0},\mathbf{0})}} \left(\prod_{k:\mathbf{KU}^{\mathbf{G}}(X,G,\tau)} \beta\, k \right) \to \prod_{k:\mathbf{KU}^{\mathbf{G}}(X,G,\tau)} \beta\, k.$$

```
let KU^G_ind (x : exp) (g : exp) (tau : exp) (beta : exp) (h : exp) : exp =
    (* Hypothetical eliminator *)
    App (Var "KU^GInd", KU^G (x, g, tau))
```

Theorem 9 (K-Theory Stability)**.** The type $\mathbf{KU}^{\mathbf{G}}(X, G, \tau)$ is stable under suspension, meaning it is invariant under the suspension operation in the spectrum, reflecting its role in stable homotopy theory:

$$\mathbf{stability} : \prod_{X:\mathbf{U}_{(\mathbf{0},\mathbf{0})}} \prod_{G:\mathbf{U}_{(\mathbf{0},\mathbf{0})}} \prod_{\tau:\prod_{x:X} \mathbf{U}_{(\mathbf{0},\mathbf{0})}} \mathbf{KU}^{\mathbf{G}}(X, G, \tau) =_{\mathbf{U}_{(\mathbf{0},\mathbf{0})}} \mathbf{KU}^{\mathbf{G}}(\mathbf{Susp}\, X, G, \tau).$$

```
let KU^G_stability (x : exp) (g : exp) (tau : exp) : exp =
    Path (Universe (0, Bose), KU^G (x, g, tau), KU^G (Susp x, g, tau))
```

Theorem 10 (Refinement to Differential K-Theory, Theorem 3.4.5)**.** The type $\mathbf{KU}^{\mathbf{G}}(X, G, \tau)$ can be refined to differential K-theory by incorporating a connection, as provided by $\mathbf{KU}^{\mathbf{G}}_\flat(X, G, \tau, \mathit{conn})$:

$$\mathbf{refine}_{\mathbf{KU}^{\mathbf{G}}_\flat} : \prod_{X:\mathbf{U}_{(\mathbf{0},\mathbf{0})}} \prod_{G:\mathbf{U}_{(\mathbf{0},\mathbf{0})}} \prod_{\tau:\prod_{x:X} \mathbf{U}_{(\mathbf{0},\mathbf{0})}} \prod_{\mathit{conn}:\Omega^1(X)} \mathbf{KU}^{\mathbf{G}}(X, G, \tau) \to \mathbf{KU}^{\mathbf{G}}_\flat(X, G, \tau, \mathit{conn}).$$

```
let KU^G_to_KU_\flat^G (x : exp) (g : exp) (tau : exp) (conn : exp) : exp =
    KU_\flat^G (x, g, tau, conn)
```