Contents

Chapter 1

[scale=.015,very thin] (0,0) – (100,1) – (200,1) – (300,0) – (200,-1) – (100,-1) – cycle;

.

, , . ,

.

1.1

.

— , ,

.

(IEEE, ANSI)[1]

[2]. ,

.

.

(code review), (, ,),

, , , .

, .

[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

1.2

—
. — , ,
, ( ) ,
.
: 1) ,
, ( ), 2) - ,
, , , ,
(,). .

1.2.1

— Z^3 (Spivey, 1992),
- ,
. , Z.
[4], .
,
, LOTOS[5] (Bolognesi, Brinksma, 1987), .
, . ,
— TLA+[6], CSP (Hoare, 1985), CCS[7] (Milner, 1971), Actor Model, BPMN, etc.

[3] ISO/IEC 13568:2002 — Z formal specification notation
[4] http://mrg.doc.ic.ac.uk/kohei/
[5] ISO 8807:1989 — LOTOS — A formal description technique based on the temporal ordering of observational behaviour
[6] The TLA+ Language and Tools for Hardware and Software Engineers
[7] J.C.M. Baeten. A Brief History of Process Algebra.

1.2.2

’ PDP-6 PDP-10, MATHLAB ( ), : MACSYMA ( ), SCRATCHPAD (, IBM), REDUCE ( ), SAC-I, SACLIB ( ), MUMATH ( ) DERIVE. ’ : AXIOM SCRATCHPAD (NAG), MAGMA (,), MUPAD (,).

1.3

1.3.1

.

.

,

. : 1) VST (CompCert, C); 2) NuPRL (Cornell University, ,); 3) TLA+ (Microsoft Reseach,); 4) Twelf ( ); 5) SystemVerilog ( ).

1.3.2

. .

, , , .

, .

: 1) Coq OCaml - INRIA; 2) Agda Haskell ; 3) Lean C++ Microsoft Research - ; 4) F* – Microsoft Research.

1.3.3

: 1) cubicaltt — [8]CCHM
; 2) yacctt — [9]ABCFHL; 3) Agda –cubical —
; 4) Lean — Lean ; 5) RedPRL
– ABCFHL.

1.3.4

.
, HOL/Isabell, Coq, ACL2, ’
(Satisfiability Modulo Theories, SMT).

30- 20- .
, . 1958,
LISP , .
,
. LISP, Erlang, JavaScript,
Python.
( ).
AUTOMATH (AUT-68 AUT-QE),
’ , 1967.
, - - .
ML/LCF
, ,
, ,
(well-founded) . , ’ (ML),
LCF. LCF HOL88, HOL90, HOL98
HOL/Isabell. (CPL,)
(Charity,).
80-90 , Mizar (, 1989).
PVS (, , , 1995), ACL2 Common Lisp (, , , 1996), Otter
(, 1996).

1.4

: 1)
( ), JIT ; 2)
(
System F).
Erlang (BEAM), JavaScript
(V8), Java (HotSpot), K (Kx), PHP (HHVM), Python (PyPy), LuaJIT
.

[8]Cyril Cohen, Thierry Coquand, Simon Huber, Anders Mörtberg. Cubical Type Theory: a constructive interpretation of the univalence axiom. 2015. https://5ht.co/cubicaltt.pdf

[9]Carlo Angiuli, Brunerie, Coquand, Kuen-Bang Hou (Favonia), Robert Harper, Dan Licata. Cartesian Cubical Type Theory. 2017. https://5ht.co/cctt.pdf

: ML, OCaml, Rust, Haskell, Pony.
LLVM ,
LLVM . Rust MIR LLVM .
.
.

OCaml (, Coq), Rust (,
), Erlang (, -
) Pony (, - CAS-).
.

1.4.1

, $_{CPS}$ – (SSE/AVX)
CAS - .
L4 , HOL/Isabell,
.

1.4.2 -

coq.io, OCaml
Coq.
Coq. .

1.4.3

– Erlang O. PTS
, .

- Erlang. CoC Haskell (Morte).

1.4.4

— Erlang HM
(Erlang, Haskell), ABCFHL (Haskell,OCaml).

1.5

, : i) ; ii) ; iii)
; iv) ,
- , : i)
O_λ — λ- ; ii) O_π — , CCS, CSP π- ;
iii) O_μ — (MatLab, Julia, kx, J); iv) O_Π —
( ); v) O_Σ — ( ); vi) $O_=$ —
- ( ); vii) O_W — ( ); viii) O_I —
( ).

Chapter 2

[scale=.015,very thin] (0,0) – (100,1) – (200,1) – (300,0) – (200,-1) – (100,-1) – cycle;

: i) ,
; ii) , ,
, , .

2.1

,
. : ,
, , 3.
4. ,
2.

1 *(,).* — ’ , П-
- . ’ $x : o$, , ’ ,
$p(o) : U$, $p : concept(o)$.

2 *( ).* ’ $Ob : U$ ’ $Hom : Ob \to Ob \to U$.

3 *( ).* , ’
, .

4 *( ).* — ,
5 , : , -
.

5 *( ).* β η
.

6 *( ).* — , ’ —
maybe- , — (,
 , ,). : O_Π, O_Σ, $O_=$. :
O_{PTS}, O_{MLTT}, O_{HTS}.

7 *( ).* (’) ,
 ( )
 . .

8 *( ).*

$$O_\Pi \to O_\Sigma \to O_= \to O_W \to O_I. \tag{2.1}$$

$$O_{PTS}(O_\Pi) \to O_{CTX}(O_\Pi, O_\Sigma) \to O_{EQU}(.., O_\Sigma, O_=) \to \quad \to O_{ITS}(..., O_=, O_W) \to O_{HTS}(..., O_W, O_I). \tag{2.2}$$

 .
 . . :
$O_{PTS} : O_\Pi \to U$, $O_{CTX} : O_\Pi \to O_\Sigma \to U$, $O_{EQU} : O_\Pi \to O_\Sigma \to O_= \to U$,
$O_{ITS} : O_\Pi \to O_\Sigma \to O_= \to O_W \to U$, $O_{HTS} : O_\Pi \to O_\Sigma \to O_= \to O_W \to O_I \to U$.

 ,
 ,

9 *( ).* , $O_{HTS} = O_{\Pi\Sigma=WI}$,
 , .
 .
:

$$O_{HTS} = O_{\Pi\Sigma=WI} : O_\Pi \to O_\Sigma \to O_= \to O_W \to O_I \to U. \tag{2.3}$$

 , , .
O_{ITS} , .
 .

10 *( ).* *1*
 , .
 . ( ),
 - (, - , , β-, η-).
β-, η- , .
 - .

11 *( ).* , , ,
 - - .
 .

$$O_\infty : O_{CPS} \to O_{PTS} \to O_{MLTT} \to O_{ITS} \to O_{HTS} \to ... \tag{2.4}$$

Table 2.1

O_λ	λ-
O_π	, CCS, CSP π-
O_μ	
O_Π ( )	
O_Σ ( )	
$O_=$ - ( )	
O_W ( )	
O_I ( )	

[scale=0.6]minimal.eps

FIGURE 2.1

12 *( ).*
, , , —
,
. : $O_{I*} \to O_{\Pi=}$, $O_\Pi \to O_{\Pi\Sigma}$, $O_\Pi \to O_{\Pi\Sigma}$, $O_{\Pi*} \to O_\Pi$.

2.2

, , .
.

2.2.1

, .

$$PTS_{CPS} = \{\, O\ b : \{O_{CPS}, O_{PTS}\} Hom : \{1, 2 : 1 \to O_{PTS}, 3 : O_{PTS} \to O_{CPS}\} \tag{2.5}$$

1 2 , 3 ( )
. .

2.2.2

— ,
(,
HTS MLTT,).

$$Total = \{\, O\ b : \{O_{CPS}, O_{PTS}, O_{MLTT}, O_{ITS}, O_{HTS}\} Hom : \{1\ , 2 : 1 \to O_{HTS}, 3 : O_{MLTT} \to O_{ITS} 4 : O_{HTS} \to O_{ITS}, 5 : O_{ITS} \tag{2.6}$$

:

[scale=0.6]full.eps

FIGURE 2.2

2.3

, (
Π) .
— .
.

, ' , —
. —
, .
Either- Σ- .
($N \to U$) ,
, .

2.3.1 O_{PTS}

,
AUTOMATH Henk, Morte, Cedile, Om.

13 *(O_{PTS}).*

$$O_{PTS} = \{\, O\ b : \{\, X : maybe\ PTS, target : maybe\ CPS\ \} Hom : \{\, t\ ype, norm : X \to X, extract : X \to target certify : X \to target = type \circ norm \tag{2.7}$$

14 *(O_{PTS}). O_{PTS} ,* Π- . , . *data PTS = ppure (:ptsPTS)*

Calculus of Construction ( ), Pure Type System (,
).

15 *(O_{Π}). [mathescape=true] data pts (lang: U) = star (n: nat) | var (x: name) (l: nat) | pi (x: name) (l: nat) (f: lang) | lambda (x: name) (l: nat) (f: lang) | app (f a: lang)*

2.3.2 - O_{MLTT}

, , , NuPRL Agda.
ΠΣ $MLTT$: ΠΣ[1], Π∀[2].

[1] https://github.com/zlizta/pisigma-0-2-2
[2] https://github.com/sweirich/pi-forall

16 *(O_{MLTT}).*

$$O_{MLTT} = \{\, O\ b : \{\ maybe\ MLTT\ \} Hom : \{\ type, norm : Ob \to Ob certify : Ob \to Ob = type \circ norm \tag{2.8}$$

17 *(O_{MLTT}). O_{MLTT} - : O_Π, O_Σ, $O_=$. data MLTT = mpure (:ptsMLTT)|msigma(:existsMLTT)|mid(:identityMLTT)*

18 *(O_Σ). Σ- ,*
$O_{MLTT-72}$ $O_{\Pi\Sigma}$: [mathescape=true] data exists (lang: U) = sigma (n: name) (a b: lang) | pair (a b: lang) | fst (p: lang) | snd (p: lang)

19 *($O_=$). ,*
$O_{MLTT-84}$ $O_{\Pi\Sigma=}$: [mathescape=true] data identity (lang: U) = id (t a b: lang) | $id_intro(ab : lang)|id_elim(abcde : lang)$

2.3.3 O_{ITS}

20 *(O_{ITS}).*

$$O_{ITS} = \{\, O\ b : \{\ X : maybe\ ITS, target : maybe\ CPS\ \} Hom : \{\ type, norm, induction : X \to X, extract : X \to target certify \tag{2.9}$$

, ,
: $O_=$, O_Σ, O_Π.

21 *(O_{ITS}). data ITS = ipure (:ptsITS)|isigma(:existsITS)|iid(:identityITS)|iITS(:indITS)*

: i) , ; ii) ,
case ; iii) .
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) | com (n: name) (t: tele A) (dim: list name) (s: list (prod (prod name bool) A))

22 *(O_*). ,*
O_*: data ind (lang: U) = datum (n: name) (t: tele lang) (labels: list (label lang)) | case (n: name) (t: lang) (branches: list (branch lang)) | ctor (n: name) (args: list lang)

2.3.4 O_{HTS}

23 *(O_{HTS}).*

$$O_{HTS} = \{\, O\ b : \{\ maybe\ HTS\ \} Hom : \{\ type, norm : Ob \to Ob certify : Ob \to Ob = type \circ norm$$

24 *(O_{HTS}).*
: O_I, O_W, $O_=$, O_Σ, O_Π: data HTS = hpure (:ptsHTS)|hsigma(:existsHTS)|hid(:identityHTS)|hind(:indHTS)|homotopy(:htsHTS)

O_{ITS} Path- , ,
Path- (, ,) ,
.

25 *(O_I). data hts (lang: U) = path (t a b: lang) | plam (n: name) (a: alg) (b: lang) | papp (f: name) (a: lang) (p: alg) | comp$_(ab$: lang)|$fill_(abc$: lang)|$glue_(abc$: lang)|$glue_elem(ab$: lang)|$unglue_elem(ab$: lang)*

, O_{HTS} Id- , $O_=$,
O_I.

2.4 c

(λ-) (,
,).
. ,
O_{CPS}, , *maybe-* .

$$O_{CPS} : O_\lambda \to O_\pi \to O_\mu \to U$$

,
(System F). ,
.
, , ,
- , AVX Rust. ,
- BEAM Erlang.

2.4.1 O_{CPS}

26 *(O_{PS}).*

$$O_{CPS} = \{ O\ b : \{\ maybe\ CPS\ \} Hom : \{\ eval : Ob \to Ob\ \}$$
$$: O_\lambda, O_\pi, O_\mu.$$

27 *(O_{HTS}). data CPS = church (:lambdaCPS)|milner(:piCPS)|tensor(:muCPS)*

28 *(O_λ).* :
( ), : *data lambda = var (x: nat) | lam (l: nat) (d: cps) | app (f a: cps)* ,
- .

29 *(O_π).* ,
O_*: data pi (lang: U) = process (protocol: lang) | spawn (cursors: lang) (core: nat) (program: lang) | snd (cursor: lang) (data: lang) | rcv (cursor: lang) | pub (size: nat) | sub (cursor: lang)

.
, .

2.5 PTS∞

IEEE[3] ESA[4]
. .
AUTOMATH[?] ( ) ,
- [?]. Lean, Coq, F*, Agda
, Calculus of Constructions[?] (CoC) (Calculus
of Inductive Constructions[?] (CiC). ,
- [?].
(Morte[5] , Henk[?]).
Π- (Σ ECC[?],), , PTS∞
[?] ( ), Cedile self- [?][?] (,), - [?] ( ).
– ,
(type checker), ,
, - .
: 1) (Pluto [6]); 2)
; 3) .

2.5.1

- - - -
( ) - [?].
,
( ) .
, JavaScript, Erlang,
PyPy, LuaJIT, K.
++
Rust , - , System F System F_ω.
Erlang .
PTS . .
morte[8], . *cubical*[9].
caramel[10].

$$\{ Sorts = U.\{i\},\ i : NatAxioms = U.\{i\} : U.\{inc\, i\} Rules = U.\{i\} U.\{j\} : U.\{max\, i\, j\} \tag{2.10}$$

– , (Calculus of
Constructions, CoC) . .
CoC, .

[3] IEEE Std 1012-2016 — V&V Software verification and validation
[4] ESA PSS-05-10 1-1 1995 – Guide to software verification and validation
[5] Gabriel Gonzalez. Haskell Morte Library https://github.com/Gabriel439/Haskell-Morte-Library
[6] Rebecca Valentine. Formal Specification of the Plutus Core Language. 2017. https://iohk.io/research/papers/JT5XKNBP
[8] http://github.com/Gabriel439/Haskell-Morte-Library
[9] http://github.com/mortberg/cubicaltt
[10] https://github.com/MaiaVictor/caramel

List of languages, tried as verification targets Table 2.2

Target	Class	Intermediate	Theory
JVM	interpreter/native	Java	F-sub[7]
JVM	interpreter/native	Scala	System F-omega
CLR	interpreter/native	F#	System F-omega
GHC	compiler/native	Haskell	System D
GHC	compiler/native	Morte	CoC
GHC,OCaml	compiler/native	Coq	CiC
O,BEAM	interpreter	Om	PTS$^\infty$
JavaScript	interpreter/native	PureScript	System F

. ,
, , . ( ),
- , , :
, , .

Axioms Sorts
(Rules). ,
:

Henk[**?**],
1997 . Haskell
Henk, .
Π- , λ- , , β- η- .
Henk Morte. (300),
,
, ,
, ( ) ,
.

2.5.2

PTS (CoC) , Morte Henk.
PTS .
BNF

[mathescape=true] I := list nat U := * + * . nat O := U + I + (O) + O O + O → O + λ (I : O) → O + ∀ (I : O) → O

+ — , ;' — , := — BNF- ,
#empty, #nat, #list — BNF- — BNF , *,:,
→, (,), λ, ∀ – . ,
O_{PTS} O_{Π} .

[mathescape=true] data pts (lang: U) = star (n: nat) | var (x: name) (l: nat) | pi (x: name) (l: nat) (d c: lang) | remote (n: name) (n: nat) | lambda (x: name) (l: nat) (d c: lang) | app (f a: lang)

2.5.3

PTS^{∞} – , ,
P_{ω} , ( ) .
[?] - - [11].
PTS Fixpoint,
remote, .

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

U_0 — , U_1 — , U_2 — ,
U_3 — . .

$$So : NatU_o \tag{2.11}$$

S. R
( ). SAR-
. , , .

A_1 i: Nat, j: Nat, i < j$U_i : U_j$

R_1 i : Nat, j : Nat$U_i \to U_j : U_{max(i,j)}$

. PTS .

A_2 i: Nat $U_i : U_{i+1}$(2.12)

R_2 i : Nat, j : Nat $U_i \to U_j : U_j$(2.13)

2.5.4

. list Sigma.
, , , .

$$Ctx - formation\Gamma : Ctx \tag{2.14}$$

[11] $U : U$

$$Ctx - intro$$

$_1$ $\Gamma : Ctx \emptyset : \Gamma$(2.15)

$$Ctx - intro$$

$_2$ A : U_i, $x : A$, $\Gamma : Ctx(x : A) \vdash \Gamma : Ctx$(2.16)

2.5.5

— , β-,η- - .
: 1) - , ( ), ( ); 2)
( ). , O_{PTS}
5 : , - , ( ), β- ,
η- .

Π-formation A:U_i , $x : A \vdash B : U_j \Pi\ (x : A) \to B : U_{p(i,j)}$(2.17)

λ-intro x:A $\vdash b : B \lambda\ (x : A) \to b : \Pi\ (x : A) \to B$(2.18)

App$-elimination f : (\Pi\ (x : A) \to B) \quad a : A f\ a : B\ [a/x]$(2.19)

β-computation x:A $\vdash b : B \quad a : A (\lambda\ (x : A) \to b)\ a = b\ [a/x] : B\ [a/x]$(2.20)

$$subst \pi_1 : A \quad u : A \vdash \pi_2 : B[\pi_1/u]\ \pi_2 : B \tag{2.21}$$

( ) ,
.

[mathescape=true] PTS (A: U): U = (Pi$_F ormer$: $(A- > U)- > U) *$ $(Pi_I ntro : (B : A- > U)- > ((a : A)- > Ba)- > (PiAB)) * (Pi_E lim : (B :$ $A- > U)(a : A)- > (PiAB)- > Ba) * (Pi_C omp1 : (B : A- > U)(a : A)(f :$ $PiAB)- > Equ(Ba)(Pi_E limBa(Pi_I ntroBf))(fa))*((B : A- > U)(a : A)(f :$ $PiAB)- > Equ(PiAB)f(x : A)- > fx))$

3. [?].
( ) Path- , $O_=$ O_I .
Path- . PTS ( ) remote
. .
.
. PTS

CoC[**?**]. : , , ,
.

2.5.6

, .
remote .
. remote self- [**?**][**?**].

[mathescape=true] type (:star,N) D → (:star,N+1) (:var,N,I) D → :true = proplists:is$_d$$efinedNB, om$: $keygetNDI$(: $remote, N)D$→ 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)

2.5.7

N P, .

[mathescape=true] 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)

2.5.8 , ,

.

[mathescape=true] 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)

(-)
.

[mathescape=true] 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

Erlang .

[mathescape=true] 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))

2.5.9

PTS.

$./omhelpme[a,[expr],"toparse.Returns,orerror,.",type,[term],"typechecksandreturnstype.",erase,[term],"tountypedterm.Returns,.",no$

$./omprintfsterasenorma"List/Cons"\ Head->\ Tail->\ Cons->$
$Nil->ConsHead(TailConsNil)ok$

2.5.10

: 1) fixpoint ; 2)
- ; 3) , ; 4) ,
(, ,).

2.5.11

.
λ- λ- .

[mathescape=true] ext (:var,X,N,F) → (:var,X) (:app,A,B,N,F) → (:call,N,ext(F,A,N),[ext(F,B,N)])
(:fn,S,,I,O,N,F)→ (:fun,N,(:clauses,[:clause,N, [(:var,N,S)],[],[ext(F,O,N)]]))
→ []

Erlang PTS$^\infty$. Erlang-
LING BEAM. AST Erlang
(Elixir), Erlang.
Erlang Core, 30 .

2.5.11.1

,
Erlang. Erlang 300 .
O_{PTS} ( ) . ,
.

2.5.11.2 LLVM

— ,
LLVM MIR.
, -
- .

2.5.11.3 FPGA

, , VHDL (, clash).

2.6 ITS

.
( ), PTS ,
.

2.6.1

[mathescape=true] def := data id tele = sum + id tele : exp = exp + id tele : exp where def exp := cotele*exp + cotele → exp + exp → exp + (exp) + app + id + (exp,exp) + cotele → exp + split cobrs + exp .1 + exp .2

0 := empty imp := [import id] brs := 0 + cobrs tele := 0 + cotele app := exp exp cotele := (exp : exp) tele id := [nat] sum := 0 + id tele + id tele | sum ids := [id] br := ids → exp cod := def dec mod := module id where imp def dec := 0 + codec cobrs := | br brs

, , , .

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) | com (n: name) (t: tele A) (dim: list name) (s: list (prod (prod name bool) A))

data ind (lang: U) = datum (n: name) (t: tele lang) (labels: list (label lang)) | case (n: name) (t: lang) (branches: list (branch lang)) | ctor (n: name) (args: list lang)

2.6.2

: 1) ($F_A(X) = 1 + A \times X$ $F_A(X) = A + X \times X$), (1 A).

nil leaf ( ). 2)

, ($F_A(X) = A \times X$) —

, , ( ).

-

.

: $\mu\ X \to 1 + X$

A: $\mu\ X \to 1 + A \times X$

: $\mu\ X \to 1 + X \times X + X$

: $\nu\ X \to A \times X$

A: $\nu\ X \to 1 + A \times X$

: $\mu\ X \to \mu\ Y \to 1 + X \times Y = \mu\ X = List\ X$

,

Π- . - , .

Π- . [?]

Fixpoint .

J Id , ,

funExt, homotopy.

2.6.3

List , $(\mu L_A, in)$ $L_A(X) = 1 + (A \times X)$. $\mu L_A = List(A)$. - $nil : 1 \to List(A)$ $cons : A \times List(A) \to List(A)$ $nil = in \circ inl$ $cons = in \circ inr$, $in = [nil, cons]$. $c : 1 \to C$ $h : A \times C \to C$,

$f = [c, h] : List(A) \to C$ ’ :

$$\{ f \circ nil = c f \circ cons = h \circ (id \times f)$$

$f = foldr(c, h)$. , $\mu(1 + A \times X)$
$[1 \to List(A), A \times List(A) \to List(A)]$. , List-
O_{PTS} :

$\{ f\ oldr = [f \circ nil, h], f \circ cons = h \circ (id \times f) len = [zero, \lambda\, a\, n \to succ\, n](++) = \lambda\, xs\, ys \to [\lambda(x) \to ys, cons](xs) map = \lambda\, f \to [nil, cons \circ (f \times id)]$

[mathescape=true] data list (A: U) = cons (x: A) (cs: list A) | nil

$\{ l\ ist = \lambda\, ctor \to \lambda\, cons \to \lambda\, nil \to ctor cons = \lambda\, x\ \to \lambda\, xs \to \lambda\, list \to \lambda\, cons \to\ \lambda\, nil \to cons\, x\, (xs\, list\, cons\, nil) nil = \lambda\, list \to \lambda\, cons \to \lambda\, nil$

[mathescape=true] module list where map (A B: U) (f: A -> B) : list A -> list B length (A: U): list A -> nat append (A: U): list A -> list A -> list A foldl (A B: U) (f: B -> A -> B) (Z: B): list A -> B filter (A: U) (p: A -> bool) : list A -> list A

$\{ l\ en = foldr\ (\lambda\, x\, n \to succ\, n)\ 0 (++) = \lambda\, ys \to foldr\ cons\ ys map = \lambda\, f \to foldr\ (\lambda x\, xs \to cons\ (f\ x)\ xs)\ nil filter = \lambda\, p \to foldr\ (\lambda x\, xs \to if$

2.7 HTS

2.7.1

[mathescape=true] sys := [sides] side := (id=0)→exp+(id=1)→exp form := formf1+f1+f2 sides := empty+cos+side cos := side,side+side,cos mod := *module* id *where* imps dec f1 := f1/2 f2 := -f2+id+0+1 imp := *import* id brs := empty+cobrs app := exp exp tel := empty+cotel imps := list imp cotel := (exp:exp) tel id := list nat dec := empty+codec u2 := *glue+unglue+Glue* u1 := *fill+comp* ids := list id br := ids→exp+ids@ids→exp codec := def dec cobrs := | br brs sum := empty+id tel+id tel|sum+id tel<ids>sys def := *data* id tel=sum+id tel:exp=exp+id tel:exp *where* def exp := cotel*exp+cotel→exp+exp→exp+(exp)+id (exp,exp)+→exp+*split* cobrs+exp.1+exp.2+ ⟨ids⟩exp+exp@form+app+u2 exp exp sys+u1 exp sys := ( ), + ( ), #empty ( ), #nat ( ), #list ( ) — BNF . , :, *, ⟨, ⟩, (,), =, \, /, -, →, 0, 1, @, [,], **module**, **import**, **data**, **split**, **where**, **comp**, **fill**, **Glue**, **glue**, **unglue**, .**1**, .**2**, , . :
, , . ,
3.

, ,
[0,1] *cubical* .

[mathescape=true] data alg = zero | one | max (a b: alg) | min (a b: alg)

data hts = path (t a b: lang) | plam (n: name) (a: alg) (b: lang) | papp (f: name) (a: lang) (p: alg) | $comp_(ab : lang) | fill_(abc : lang) | glue_(abc : lang) | glue_e lem(ab : lang) | unglue_e lem(ab : lang)$

2.8

O_{CPS} :

data cps = var (x: nat) | lam (l: nat) (d: cps) | app (f a: cps)

, , , ,
, .
, , .
. , CPS- .
, L1 (
64) LuaJIT, V8, HotSpot,
J.
L1 , ,
AVX , , - GPU . ,
, JIT , ,
Erlang, Python, K, LuaJIT.

- ,
, (enum)
Rust. (5)
(3,4).

Rust 0 Java 3 PyPy 8 O-CPS 291 Python 537 K 756 Erlang 10699/1806/436/9 LuaJIT 33856

akkerman$_k$635$ns/iter(+/-73)akkerman_rust8,968ns/iter(+/-322)$
Rust, , ,
$_{CPS}$,
.

2.8.1 Rust

objdump ./target/release/o -d | grep mulpd 223f1: c5 f5 59 0c d3 vmulpd (223f6: c5 dd 59 64 d3 20 vmulpd 0x20(22416: c5 f5 59 4c d3 40 vmulpd 0x40(2241c: c5 dd 59 64 d3 60 vmulpd 0x60(2264d: c5 f5 59 0c d3 vmulpd (22652: c5 e5 59 5c d3 20 vmulpd 0x20(

2.8.2 -

, - O_{CPS}
, : Defer, Continuation, Start ( ),
Return ( ).

data Lazy = Defer (otree: NodeId) (a: AST) (cont: Cont) | Continuation (otree: NodeId) (a: AST) (cont: Cont) | Return (a: AST) | Start

: , , ,
.

data Cont = Expressions (ast: AST) (vec: Option (Iter AST)) (cont: Cont) | Assign (ast: AST) (cont: Cont) | Cond (c,d: AST) (cont: Cont) | Func (a,b,c: AST) (cont: Cont) | List (acc: Vec AST) (vec: Iter AST) (i: Nat) (cont: Cont)

| Call (a: AST) (i: Nat) (cont: Cont) | Return | Intercore (m: Message) (cont: Cont) | Yield (cont: Cont)

2.8.3

O_{CPS} , :
i32, i64.

[mathescape=true] E: V | A | C NC: ";" = [] | ";" m:NL = m FC: ";" = [] | ";" m:FL = m EC: ";" = [] | ";" m:EL = m NL: NAME | o:NAME m:NC = Cons o m FL: E | o:E | m:FC = Cons o m EL: E | EC | o:E m:EC = Cons o m C: N | c:N a:C = Call c a N: NAME | S | HEX | L | F L: "(" ")" = [] | "([" c:NL "]" m:FL ")" = Table c m | "(" l:EL ")" = List l F: "" "" = Lambda [] [] [] | "[" c:NL "]" m:EL "" = Lambda [] c m | "" m:EL "" = Lambda [] [] m

, : AST ( ); Value ; Scalar (, , , , UTF-8 , CPU).

data AST = Atom (a: Scalar) | Vector (a: Vec AST)

data Value = Nil | SymbolInt (a: u16) | SequenceInt (a: u16) | Number (a: i64) | Float (a: f64) | VecNumber (Vec i64) | VecFloat (Vec f64)

data Scalar = Nil | Any | List (a: AST) | Dict (a: AST) | Call (a b: AST) | Assign (a b: AST) | Cond (a b c: AST) | Lambda (otree: Option NodeId) (a b: AST) | Yield (c: Context) | Value (v: Value) | Name (s: String)

2.8.4

(Github[12]).

2.8.5

, , , CAS- ,
.

2.8.5.1

( ) .
- ( ),
.
,
, (
).

: reactor[aux;0;mod[console;network]]; reactor[timercore;1;mod[timer]]; reactor[core1;2;mod[task]]; reactor[core2;3;mod[task]];

[12] https://github.com/voxoz/kernel

2.8.5.2

IP- L1/L2 ’ , , , , ’ L1 ’ AVX .

2.8.5.3

( ). , .

[scale=0.25]pub.eps

FIGURE 2.3 CAS-

( ) , ,
.

[scale=0.25]sub.eps

FIGURE 2.4 CAS-

2.8.5.4

: i) Task- ; ii) Timer- ; iii) IO- . Task- , Timer- — .

[scale=0.25]sys.eps

FIGURE 2.5

( ) ( ). .
InterCore . , ,
, .
InterCore .

Task-

Task- Rust , :
( ), ( ).
— 0- , .
WebSocket .

IO-

IO- ’ Windows, Linux, Mac .

Timer-

(Task, IO, Timer)
(, - , BTree).

2.8.5.5 InterCore

InterCore SPMC , .
, MPSC .
. InterCore poll_bus .
InterCore , pull_bus Yield
, , , , Yield
.

pub [capacity]

CAS , . ,
, . : p: pub[16].

sub [publisher]

CAS , .
. : s: sub[p].

spawn [core ; program ; cursors]

CPS- . Rust
FFI. ,
. : , FFI , . : s-
pawn[0;"etc/proc0";(0;1)].

snd [writer ; data]

. Nil . : snd[p;42].

rcv [reader]

. , Yield.
: rcv[s].

2.8.6

: , ;
, - . : ,
; , . Rust
, O_{CPS} .

2.8.6.1

pub struct Publisher<T> ring: Arc<RingBuffer<T», next: Cell<Sequence>, cursors: UncheckedUnsafeArc<Vec<Cursor»,

2.8.6.2

pub struct Subscriber<T> ring: Arc<RingBuffer<T», token: usize, next: Cell<Sequence>, cursors: UncheckedUnsafeArc<Vec<Cursor»,

: InterCore , Rust,
, CPS- , BSP , Console
WebSocket IO . CPS Rust
, , , Rust.
, IO ,
. .
- .

2.8.6.3

.
InterCore. pub struct Channel publisher: Publisher<Message>, subscribers: Vec<Subscriber<Message»,

2.8.6.4

.
. pub struct Memory<'a> publishers: Vec<Publisher<Value<'a», subscribers: Vec<Subscriber<Value<'a»>,

2.8.6.5

, BSP- (0- , bootstrap) AP
(> 0, application). BSP Console WebSocket IO .
, BSP , AP
(io). InterCore
IO .

pub struct Scheduler<'a> pub tasks: Vec<T3<Job<'a»>, pub bus: Channel, pub queues: Memory<'a>, pub io: IO,

2.8.7 InterCore

InterCore.

pub enum Message Pub(Pub), Sub(Sub), Print(String), Spawn(Spawn), AckSub(AckSub), AckPub(AckPub), AckSpawn(AckSpawn), Exec(usize, String), Select(String, u16), QoS(u8, u8, u8), Halt, Nop,

2.9

, HTS , .
, , , , .
$e_i : O_{n+1} \to O_n$, , $O_{CPS} = O_0$.
, Erlang/OTP.
O_{PTS} Github[13].
O_{CCHM}, Github[14].
.

2.9.1

— , .

2.9.2

— ,
.

[13] https://github.com/groupoid/om
[14] https://github.com/groupoid/cubical

Chapter 3

[scale=.015,very thin] (0,0) – (100,1) – (200,1) – (300,0) – (200,-1) – (100,-1) – cycle;

, CCHM
2017 , .
, ,
CW- , (UIP) ,
- . ,
, (,
).

3.1

Each language implementation needs to be checked. The one of possible test cases for type checkers is the direct embedding of type theory model into the language of type checker. As types in Martin-Löf Type Theory (MLTT) are formulated using 5 types of rules (formation, introduction, elimination, computation, uniqueness), we construct aliases for host language primitives and use type checker to prove that it is MLTT. This could be seen as ultimate test sample for type checker as intro-elimination fusion resides in beta-eta rules, so by proving them we prove properties of the host type checker.

Also this issue opens a series of articles dedicated to formalization in cubical type theory the foundations of mathematics. This issue is dedicated to MLTT modeling and its verification. Also as many may not be familiar with Π and Σ types, this issue presents different interpretation of MLTT types.

3.1.0.1

MLTT could be reduced to Π, Σ, Path types, as W-types could be modeled through Σ and Fin/Nat/List/Maybe types could be modeled on W. In this issue Π, Σ, Path are given as a core MLTT and W-types are given as exercise. List, Nat, Fin types are defined in next section.

Interpretations correspond to mathematical theories Table 3.1

Type Theory	Logic	Category Theory	Homotopy Theory
A type	class	object	space
isProp A	proposition	(-1)-truncated object	space
a:A program	proof	generalized element	point
$B(x)$	predicate	indexed object	fibration
$b(x) : B(x)$	conditional proof	indexed elements	section
$\emptyset$	$\bot$ false	terminal object	empty space
1	$\top$ true	initial object	singleton
$A + B$	$A \vee B$ disjunction	coproduct	coproduct space
$A \times B$	$A \wedge B$ conjunction	product	product space
$A \to B$	$A \Rightarrow B$	internal hom	function space
$\sum x : A, B(x)$	$\exists_{x:A} B(x)$	dependent sum	total space
$\prod x : A, B(x)$	$\forall_{x:A} B(x)$	dependent product	space of sections
$\mathbf{Path}_A$	equivalence $=_A$	path space object	path space A^I
quotient	equivalence class	quotient	quotient
W-type	induction	colimit	complex
type of types	universe	object classifier	universe
quantum circuit	proof net	string diagram	

Any new type in MLTT presented with set of 5 rules: i) formation rules, the signature of type; ii) the set of constructors which produce the elements of formation rule signature; iii) the dependent eliminator or induction principle for this type; iv) the beta-equality or computational rule; v) the eta-equality or uniquness principle. Π, Σ, and Path types will be given shortly. This interpretation or rather way of modeling is MLTT specific.

The most interesting are Id types. Id types were added in [1]1984 while original MLTT was introduced in [2]1972. Predicative Universe Hierarchy was added in [3]1975. While original MLTT contains Id types that preserve uniquness of identity proofs (UIP) or eta-rule of Id type, HoTT refutes UIP (eta rule desn't hold) and introduces univalent heterogeneous Path equality ([4]∞-Groupoid interpretation). Path types are essential to prove computation and uniquness rules for all types (needed for building signature and terms), so we will be able to prove all the MLTT rules as a whole.

3.1.0.2

In contexts you can bind to variables (through de Brujin indexes or string names): i) indexed universes; ii) built-in types; iii) user constructed types, and ask questions about type derivability, type checking and code extraction. This system defines the core type checker within its language.

[1] P. Martin-Löf, G. Sambin. Intuitionistic type theory. 1984.
[2] P. Martin-Löf, G. Sambin. The Theory of Types. 1972.
[3] P. Martin-Löf. An intuitionistic theory of types: predicative part. 1975.
[4] M. Hofmann, T. Streicher. The groupoid interpretation of type theory. 1996.

By using this languages it is possible to encode different interpretations of type theory itself and its syntax by construction. Usually the issues will refer to following interpretations: i) type-theoretical; ii) categorical; iii) set-theoretical; iv) homotopical; v) fibrational or geometrical.

-

According to type theoretical interpretation for any type should be provided 5 formal inference rules: i) formation; ii) introduction; iii) dependent elimination principle; iv) beta rule or computational rule; v) eta rule or uniqueness rule. The last one could be exceptional for Path types. The formal representation of all rules of MLTT are given according to type-theoretical interpretation as a final result in this Issue I. It was proven that classical Logic could be embedded into intuitionistic propositional logic (IPL) which is directly embedded into MLTT.

Logical and type-theoretical interpretations could be distincted. Also set-theoretical interpretation is not presented in Table 1.

-

Categorical interpretation is a modeling through categories and functors. First category is defined as objects, morphisms and their properties, then we define functors, etc. In particular, as an example, according to categorical interpretation Π and Σ types of MLTT are presented as adjoint functors, and forms itself a locally closed cartesian category, which will be given a intermediate result in Issue VII: Topos Theory. In some sense we include here topos-theoretical interpretations, with presheaf model of type theory as example (in this case fibrations are constructes as functors, categorically).

-

Set-theoretical interpretations could replace first-order logic, but could not allow higher equalities, as long as inductive types to be embedded directly. Set is modelled in type theory according to homotopical interpretation as n-type.

In classical MLTT uniquness rule of Id type do holds strictly. In Homotopical interpretation of MLTT we need to allow a path space as Path type where uniqueness rule doesn't hold. Groupoid interpretation of Path equality that doesn't hold UIP generally was given in 1996 by Martin Hofmann and Thomas Streicher.

When objects are defined as fibrations, or dependent products, or indexed-objects this leds to fibrational semantics and geometric sheaf interpretation. Several definition of fiber bundles and trivial fiber bindle as direct isomorphisms of Π types is given here as theorem. As fibrations study in homotopical interpretation, geometric interpretation could be treated as homotopical.

3.1.1 Π, Σ, Path

3.1.1.1 Π-

Π is a dependent product type, the generalization of functions. As a function it can serve the wide range of mathematical constructions as its domain and codomain, which are in general: objects, types, or spaces; and could have as its instance: sets, functions, polynomial functors, infinitesimals, ∞-groupoids, topological ∞-groupoid, CW-complexes, categories, languages, etc.

At this light there could be many interpretation of Π types from different areas of mathematics. We give here three: i) logical interpretation of Π as ∀ quantifier from higher order logic that forms a ground of type theory; ii) geomeric intepretation of Π as fiber bundle; iii) categorical interpretation of functions as functors.

-

As a logical system dependent type theory could correspond to higher order logic. However here only type-theoretical model is given completely.

30 *(*Π*-Formation).*

$$(x : A) \to B(x) =_{def} \prod_{x:A} B(x) : U.$$

Pi (A: U) (B: A -> U): U = (x: A) -> B x

31 *(*Π*-Introduction).*

$$\backslash(x : A) \to b(x) =_{def} \prod_{A:U} \prod_{B:A\to U} \prod_{b:\prod_{a:A} B(a)} \lambda x.b(x) : \prod_{y:A} B(y).$$

lambda (A B: U) (b: B): A -> B = x : A)− > blam(A : U)(B : A− > U)(b : (a : A)− > Ba) : PiAB = x : A)− > bx

32 *(Π-Elimination).*

$$f\ a =_{def} \prod_{A:U} \prod_{B:A\to U} \prod_{a:A} \prod_{f:\prod_{x:A} B(a)} f(a) : B(a).$$

apply (A B: U) (f: A -> B) (a: A) : B = f a app (A: U) (B: A -> U) (a: A) (f: Pi A B): B a = f a

1 *(Π-Computation).*

$$f(a) =_{B(a)} (\lambda(x : A) \to f(a))(a).$$

Beta (A:U) (B:A->U) (a: A) (f: Pi A B) : Path (B a) (app A B a (lam A B f)) (f a)

2 *(Π-Uniqueness).*

$$f =_{(x:A)\to B(a)} (\lambda(y : A) \to f(y)).$$

Eta (A:U) (B:A->U) (a:A) (f: Pi A B) : Path (Pi A B) f (x : A)− > fx)

The adjoints Π and Σ is not the only adjoints could be presented in type system. Axiomatic cohesions could contain a set of adjoint pairs as a core type checker operations.

33 *(Dependent Product). The dependent product along morphism $g : B \to A$ in category C is the right adjoint $\Pi_g : C_{/B} \to C_{/A}$ of the base change functor.*

34 *(Space of Sections). Let $\mathbf{H}$ be a $(\infty, 1)$-topos, and let $E \to B : \mathbf{H}_{/B}$ a bundle in $\mathbf{H}$, object in the slice topos. Then the space of sections $\Gamma_\Sigma(E)$ of this bundle is the Dependent Product:*

$$\Gamma_\Sigma(E) = \Pi_\Sigma(E) \in \mathbf{H}.$$

3 *(HomSet). If codomain is set then space of sections is a set. setFun (A B : U) (:isSetB) : isSet(A− > B)*

4 *(Contractability). If domain and codomain is contractible then the space of sections is contractible. piIsContr (A: U) (B: A -> U) (u: isContr A) (q: (x: A) -> isContr (B x)) : isContr (Pi A B)*

35 *(Section). A section of morphism $f : A \to B$ in some category is the morphism $g : B \to A$ such that $f \circ g : BgAfB$ equals the identity morphism on B.*

Geometrically, Π type is a space of sections, while the dependent codomain is a space of fibrations. Lambda functions are sections or points in these spaces, while the function result is a fibration. Π type also represents the cartesian family of sets, generalizing the cartesian product of sets.

36 *(Fiber). The fiber of the map $p : E \to B$ in a point $y : B$ is all points $x : E$ such that $p(x) = y$.*

37 *(Fiber Bundle). The fiber bundle $F \to EpB$ on a total space E with fiber layer F and base B is a structure (F, E, p, B) where $p : E \to B$ is a surjective map with following property: for any point $y : B$ exists a neighborhood U_b for which a homeomorphism $f : p^{-1}(U_b) \to U_b \times F$ making the following diagram commute.*

$$
\begin{array}{c}
(\,m)[matrix of math nodes, row sep = 3em, column sep = \\
3em, minimum width = 2em] p^{-1}(U_b) U_b \times F \\
U_b \\
\text{; [-stealth] (m-1-1) edge node [above] } f \text{ (m-1-2) edge node [left] } p \text{ (m-2-1)} \\
\text{(m-1-2) edge node [right] } pr_1 \text{ (m-2-1);}
\end{array}
$$

38 *(Cartesian Product of Family over B). Is a set F of sections of the bundle with elimination map $app : F \times B \to E$ such that*

$$F \times B \; app \; E \; pr_1 \; B \tag{3.1}$$

pr_1 is a product projection, so pr_1, app are morphisms of slice category $Set_{/B}$. The universal mapping property of F: for all A and morphism $A \times B \to E$ in $Set_{/B}$ exists unique map $A \to F$ such that everything commute. So a category with all dependent products is necessarily a category with all pullbacks.

39 *(Trivial Fiber Bundle). When total space E is cartesian product $\Sigma(B, F)$ and $p = pr_1$ then such bundle is called trivial $(F, \Sigma(B, F), pr_1, B)$.*

5 *(Functions Preserve Paths). For a function $f : (x : A) \to B(x)$ there is an $ap_f : x =_A y \to f(x) =_{B(x)} f(y)$. This is called application of f to path or congruence property (for non-dependent case — cong function). This property behaves functoriality as if paths are groupoid morphisms and types are objects.*

6 *(Trivial Fiber equals Family of Sets). Inverse image (fiber) of fiber bundle $(F, B * F, pr_1, B)$ in point $y : B$ equals $F(y)$. FiberPi (B: U) (F: B -> U) (y: B) : Path U (fiber (Sigma B F) B (pi1 B F) y) (F y)*

7 *(Homotopy Equivalence). If fiber space is set for all base, and there are two functions $f, g : (x : A) \to B(x)$ and two homotopies between them, then these homotopies are equal. setPi (A: U) (B: A -> U) (h: (x: A) -> isSet (B x)) (f g: Pi A B) (p q: Path (Pi A B) f g) : Path (Path (Pi A B) f g) p q*

Note that we will not be able to prove this theorem until Issue III: Homotopy Type Theory because bi-invertible iso type will be announced there.

3.1.1.2 Σ-

Σ is a dependent sum type, the generalization of products. Σ type is a total space of fibration. Element of total space is formed as a pair of basepoint and fibration.

3.1.1.3 -

40 *(Σ-Formation). Sigma (A : U) (B : A -> U) : U = (x : A) * B x*

41 *(Σ-Introduction). dpair (A: U) (B: A -> U) (a: A) (b: B a) : Sigma A B = (a,b)*

42 *(Σ-Elimination). pr1 (A: U) (B: A -> U) (x: Sigma A B): A = x.1*
pr2 (A: U) (B: A -> U) (x: Sigma A B): B (pr1 A B x) = x.2
sigInd (A: U) (B: A -> U) (C: Sigma A B -> U) (g: (a: A) (b: B a) -> C (a, b)) (p: Sigma A B) : C p = g p.1 p.2

8 *(Σ-Computation). Beta1 (A: U) (B: A -> U) (a:A) (b: B a) : Equ A a (pr1 A B (a,b))*
Beta2 (A: U) (B: A -> U) (a: A) (b: B a) : Equ (B a) b (pr2 A B (a,b))

9 *(Σ-Uniqueness). Eta2 (A: U) (B: A -> U) (p: Sigma A B) : Equ (Sigma A B) p (pr1 A B p,pr2 A B p)*

3.1.1.4

43 *(Dependent Sum). The dependent sum along the morphism $f : A \to B$ in category C is the left adjoint $\Sigma_f : C_{/A} \to C_{/B}$ of the base change functor.*

3.1.1.5 -

10 *(Axiom of Choice). If for all $x : A$ there is $y : B$ such that $R(x, y)$, then there is a function $f : A \to B$ such that for all $x : A$ there is a witness of $R(x, f(x))$. ac (A B: U) (R: A -> B -> U) : (p: (x:A) -> (y:B)*(R x y)) -> (f:A->B) * ((x:A)->R(x)(f x))*

11 *(Total). If fiber over base implies another fiber over the same base then we can construct total space of section over that base with another fiber. total (A:U) (B C: A -> U) (f: (x:A) -> B x -> C x) (w: Sigma A B) : Sigma A C = (w.1,f (w.1) (w.2))*

12 *(Σ-Contractability). If the fiber is set then the Σ is set. setSig (A:U) (B: A -> U) (sA: isSet A) (sB : (x:A) -> isSet (B x)) : isSet (Sigma A B)*

13 *(Path Between Sigmas). Path between two sigmas $t, u : \Sigma(A, B)$ could be decomposed to sigma of two paths $p : t_1 =_A u_1)$ and $(t_2 =_{B(p@i)} u_2)$. pathSig (A:U) (B : A -> U) (t u : Sigma A B) : Path U (Path (Sigma A B) t u) ((p: Path A t.1 u.1) * PathP (<i>B(p@i)) t.2 u.2)*

3.1.1.6 Path-

The Path identity type defines a Path space with elements and values. Elements of that space are functions from interval $[0, 1]$ to a values of that path space. This ctt file reflects [5]CCHM cubicaltt model with connections. For [6]ABCFHL yacctt model with variables please refer to ytt file. You may also want to read [7]BCH, [8]AFH. There is a [9]PO paper about CCHM axiomatic in a topos.

3.1.1.7

44 *(Path Formation). Hetero (A B: U) (a: A) (b: B) (P: Path U A B) : U = PathP P a b Path (A: U) (a b: A) : U = PathP (<i> A) a b*

45 *(Path Reflexivity). Returns an element of reflexivity path space for a given value of the type. The inhabitant of that path space is the lambda on the homotopy interval* $[0, 1]$ *that returns a constant value a. Written in syntax as <i>a which equals to* $\lambda\ (i : I) \to a$. *refl (A: U) (a: A) : Path A a a*

46 *(Path Application). You can apply face to path. app1 (A: U) (a b: A) (p: Path A a b): A = p @ 0 app2 (A: U) (a b: A) (p: Path A a b): A = p @ 1*

47 *(Path Composition). Composition operation allows to build a new path by given to paths in a connected point.*

$$(m)[matrixofmathnodes, rowsep = 3em, columnsep = 3em, minimumwidth = 3em]ac$$
$$a \quad b$$

; [-stealth] (m-1-1) edge node [above] comp (m-1-2) (m-2-1) edge node [left] $\lambda(i : I) \to a$ *(m-1-1) (m-2-2) edge node [right] q (m-1-2) (m-2-1) edge node [above] p@i (m-2-2);*

composition (A: U) (a b c: A) (p: Path A a b) (q: Path A b c) : Path A a c = comp (<i>Path A a (q@i)) p []

14 *(Path Inversion). inv (A: U) (a b: A) (p: Path A a b): Path A b a = <i> p @ -i*

---

[5]Cyril Cohen, Thierry Coquand, Simon Huber, Anders Mörtberg. Cubical Type Theory: a constructive interpretation of the univalence axiom. 2015. https://5ht.co/cubicaltt.pdf

[6]Carlo Angiuli, Brunerie, Coquand, Kuen-Bang Hou (Favonia), Robert Harper, Dan Licata. Cartesian Cubical Type Theory. 2017. https://5ht.co/cctt.pdf

[7]Marc Bezem, Thierry Coquand, Simon Huber. A model of type theory in cubical sets. 2014. http://www.cse.chalmers.se/ coquand/mod1.pdf

[8]Carlo Angiuli, Kuen-Bang Hou (Favonia), Robert Harper. Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities. 2018. https://www.cs.cmu.edu/ cangiuli/papers/ccctt.pdf

[9]Andrew Pitts, Ian Orton. Axioms for Modelling Cubical Type Theory in a Topos. 2016. https://arxiv.org/pdf/1712.04864.pdf

48 *(Connections). Connections allows you to build square with given only one element of path: i)* $\lambda\ (i,j : I) \to p\ @\ min(i,j)$*; ii)* $\lambda\ (i,j : I) \to p\ @\ max(i,j)$*.*

$$(\ m)[matrixofmathnodes, rowsep = 3em, columnsep = 3em, minimumwidth = 3em]ab$$
$$a \quad a$$

; [-stealth] (m-1-1) edge node [above] p (m-1-2) (m-2-1) edge node [left] $\lambda\ (i : I) \to a$ *(m-1-1) (m-2-2) edge node [right] p (m-1-2) (m-2-1) edge node [above]* $\lambda\ (i : I) \to a$ *(m-2-2);* $(\ m)[matrixofmathnodes, rowsep = 3em, columnsep = 3em, minimumwidth = 3em]bb$

$$a \quad b$$

; [-stealth] (m-1-1) edge node [above] $\lambda\ (i : I) \to b$ *(m-1-2) (m-2-1) edge node [left] p (m-1-1) (m-2-2) edge node [right]* $\lambda\ (i : I) \to b$ *(m-1-2) (m-2-1) edge node [above] p (m-2-2);*

connection1 (A: U) (a b: A) (p: Path A a b) : PathP (<x> Path A (p@x) b) p (<i>b) = <y x> p @ (x y)

connection2 (A: U) (a b: A) (p: Path A a b) : PathP (<x> Path A a (p@x)) (<i>a) p = <x y> p @ (x / y)

15 *(Congruence). Is a map between values of one type to path space of another type by an encode function between types. Implemented as lambda defined on* $[0,1]$ *that returns application of encode function to path application of the given path to lamda argument |*λ *(i:I)* $\to$ *f (p @ i)| for both cases. ap (A B: U) (f: A -> B) (a b: A) (p: Path A a b) : Path B (f a) (f b)*

apd (A: U) (a x:A) (B: A -> U) (f: A -> B a) (b: B a) (p: Path A a x) : Path (B a) (f a) (f x)

16 *(Transport). Transports a value of the domain type to the value of the codomain type by a given path element of the path space between domain and codomain types. Defined as path composition with |[]| of a over a path p — |comp p a []|. trans (A B: U) (p: Path U A B) (a: A) : B*

3.1.1.8 -

49 *(Singleton). singl (A: U) (a: A): U = (x: A) * Path A a x*

17 *(Singleton Instance). eta (A: U) (a: A): singl A a = (a,refl A a)*

18 *(Singleton Contractability). contr (A: U) (a b: A) (p: Path A a b) : Path (singl A a) (eta A a) (b,p) = <i> (p @ i,<j> p @ i/j)*

19 *(Path Elimination, Diagonal). D (A: U) : U = (x y: A) -> Path A x y -> U J (A: U) (x y: A) (C: D A) (d: C x x (refl A x)) (p: Path A x y) : C x y p = subst (singl A x) T (eta A x) (y, p) (contr A x y p) d where T (z: singl A x) : U = C x (z.1) (z.2)*

20 *(Path Elimination, Paulin-Mohring). J is formulated in a form of Paulin-Mohring and implemented using two facts that singleton are contractible and dependent function transport. J (A: U) (a b: A) (P: singl A a -> U) (u: P (a,refl A a)) (p: Path A a b) : P (b,p)*

21 *(Path Elimination, HoTT). J from HoTT book. J (A: U) (a b: A) (C: (x: A) -> Path A a x -> U) (d: C a (refl A a)) (p: Path A a b) : C b p*

22 *(Path Computation).* $trans_comp(A : U)(a : A) : PathAa(transAA(<_> A)a) = fill(< i > A)a[]subst_comp(A : U)(P : A- > U)(a : A)(e : Pa) : Path(Pa)e(substAPaa(reflAa)e) = trans_comp(Pa)eJ_comp(A : U)(a : A)(C : (x : A)- > PathAax- > U)(d : Ca(reflAa)) : Path(Ca(reflAa))d(JAaCda(reflAa)) = subst_comp(singlAa)T(etaAa)dwhereT(z : singlAa) : U = Ca(z.1)(z.2)$

Note that Path type has no Eta rule due to groupoid interpretation.

3.1.1.9

The groupoid interpretation of type theory is well known article by Martin Hoffman and Thomas Streicher, more specific interpretation of identity type as infinity groupoid. The groupoid interpretation of Path equality will be given along with category theory library in Issue VII: Category Theory.

3.1.2

This introduction is a bit wild strives to be simple yet precise. As we defined a language BNF we could define a language AST by using inductive types which is yet to be defined in Issue II: Inductive Types and Models. This SAR notation is due Barendregt.

50 *(Terms). Point in initial object of language AST inductive definition is called a term. If type theory or language is defined as an inductive type (AST) then the term is defined as its instance.*

51 *(Sorts). N-indexed set of universes* $\mathrm{U}_{n\in\mathrm{N}}$*. Could have any number of elements which defines different type systems. All built-in types as long as user defined types are landed usually by default in* U_0 *universe. Sorts represented in type checker as a separate constructor.*

52 *(Axioms). The inclusion rules* $\mathrm{U_i} : \mathrm{U_j}, i, j \in \mathrm{N}$*, that define which universe is element of another given universe. You may attach any rules that joins* i, j *in some way. Axioms with sorts define universe hierarchy.*

53 *(Rules). The set of landings* $\mathrm{U_i} \to \mathrm{U_j} : \mathrm{U}_{\lambda(\mathrm{i,j}),\mathrm{i,j}\in\mathrm{N}}$*, where* $\lambda : \mathrm{N} \times \mathrm{N} \to \mathrm{N}$*. These rules define term dependence or how we land (in which universe) formation rules in definitions.*

54 *(Predicative hierarchy). If* λ *in Rules is an uncurried function* $\max : \mathrm{N} \times \mathrm{N} \to \mathrm{N}$ *then such universe hierarchy is called predicative.*

55 *(Impredicative hierarchy). If* λ *in Rules is a second projection of a tuple* $\mathrm{snd} : \mathrm{N} \times \mathrm{N} \to \mathrm{N}$ *then such universe hierarchy is called impredicative.*

56 *(Definitional Equality). For any* $\mathrm{U}_i, i \in \mathrm{N}$ *there is defined an equality between its members and between its instances. For all x,y* $\in A$*, there is defined a x=y. Definitional equality compares normalized term instances.*

57 *(SAR). The universum space is configured with a triple of: i) sorts, a set of universes* $\mathrm{U}_{n \in \mathrm{N}}$ *indexed over set N; ii) axioms, a set of inclusions* $\mathrm{U_i} : \mathrm{U_j}, i, j \in \mathrm{N}$*; iii) rules of term dependence universe landing, a set of landings* $\mathrm{U_i} \to \mathrm{U_j} : \mathrm{U}_{\lambda(\mathrm{i,j}),\mathrm{i,j} \in \mathrm{N}}$*, where* λ *could be function max (predicative) or snd (impredicative).*

1 *(CoC). SAR* $= \{\{\star,\}, \{\star :\}, \{i \to j : j; i, j \in \{\star,\}\}$*. Terms live in universe* $\star$*, and types live in universe . In CoC* $\lambda = \mathrm{snd}$*.*

2 *(*PTS^∞*).* $SAR = \{\mathrm{U}_{i \in \mathrm{N}}, \mathrm{U_i} : \mathrm{U}_{\mathrm{j;i<j;i,j} \in \mathrm{N}}, \mathrm{U_i} \to \mathrm{U_j} : \mathrm{U}_{\lambda(\mathrm{i,j});\mathrm{i,j} \in \mathrm{N}}\}$*. Where* U_i *is a universe of i-level or i-category in categorical interpretation. The working prototype of* PTS^∞ *is given in Addendum I: Pure Type System for Erlang*[10]*.*

3.1.3

Speaking of type checker execution, we introduce context or dictionary with types and terms, from which we can derive typed variables. This chain could be implemented as nested sigma types (due to R.A.G.Seely) or list types (due to Voevodsky). Categorically dependent type theory is built upon categories of contexts.

58 *(Empty Context).*

$$\gamma_0 : \Gamma =_{def} \star.$$

59 *(Context Comprehension).*

$$\Gamma\ ; A =_{def} \sum_{\gamma:\Gamma} A(\gamma).$$

60 *(Context Derivability).*

$$\Gamma \vdash A =_{def} \prod_{\gamma:\Gamma} A(\gamma).$$

[10] M.Sokhatsky,P.Maslianko. The Systems Engineering of Consistent Pure Language with Effect Type System for Certified Applications and Higher Languages. AIP Conference Proceedings. 2018. doi:10.1063/1.5045439

3.1.4

Here is given formal model of type-theoretical interpretation of Martin-Löf Type Theory. It combines 4 Path rules (no eta), 5 Π rules, and 6 Σ rules (two elims). The proof is provided by direct embedding (internalizing) the model intro the model of type checker which is even more powerful.

61 *(MLTT). The MLTT as a Type is defined by taking all rules for* Π, Σ *and Path types into one* Σ *telescope or context. MLTT (A: U): U =* $(Pi_Former : (A- > U)- > U) * (Pi_Intro : (B : A- > U)- > ((a : A)- > Ba)- > (PiAB)) * (Pi_Elim : (B : A- > U)(a : A)- > (PiAB)- > Ba) * (Pi_Comp1 : (B : A- > U)(a : A)(f : PiAB)- > Equ(Ba)(Pi_ElimBa(Pi_IntroBf))(fa)) * (Pi_Comp2 : (B : A- > U)(a : A)(f : PiAB)- > Equ(PiAB)f(x : A)- > fx)) * (Sigma_Former : (A- > U)- > U) * (Sigma_Intro : (B : A- > U)(a : A)- > (b : Ba)- > SigmaAB) * (Sigma_Elim1 : (B : A- > U)(_ : SigmaAB)- > A) * (Sigma_Elim2 : (B : A- > U)(x : SigmaAB)- > B(pr1ABx)) * (Sigma_Comp1 : (B : A- > U)(a : A)(b : Ba)- > PathAa(Sigma_Elim1B(Sigma_IntroBab))) * (Sigma_Comp2 : (B : A- > U)(a : A)(b : Ba)- > Path(Ba)b(Sigma_Elim2B(a, b))) * (Sigma_Comp3 : (B : A- > U)(p : SigmaAB)- > Path(SigmaAB)p(pr1ABp, pr2ABp)) * (Id_Former : A- > A- > U) * (Id_Intro : (a : A)- > PathAaa) * (Id_Elim : (x : A)(C : DA)(d : Cxx(Id_Introx))(y : A)(p : PathAxy)- > Cxyp) * (Id_Comp : (a : A)(C : DA)(d : Caa(Id_Introa))- > Path(Caa(Id_Introa))d(Id_ElimaCda(Id_Introa))) * U$

23 *(Model Check). There is an instance of MLTT. instance (A: U): MLTT A = (Pi A, lam A, app A, Beta A, Eta A, Sigma A, dpair A, pr1 A, pr2 A, Beta1 A, Beta2 A, Eta2 A, Path A, refl A, J A,* J_compA, A)

The result of the work is a mltt.ctt file which can be runned using cubicaltt. Note that computation rules take a seconds to type check.

$ time cubical -b mltt.ctt Checking: MLTT Checking: instance File loaded.

real 0m6.308s user 0m6.278s sys 0m0.014s

3.2

3.2.1 Empty

empty type lacks both introduction rules and eliminators. However, it has recursor and induction.

data empty = emptyRec (C: U): empty -> C = split emptyInd (C: empty -> U): (z: empty) -> C z = split

3.2.2 Unit

data unit = star unitRec (C: U) (x: C): unit -> C = split tt -> x unitInd (C: unit -> U) (x: C tt): (z: unit) -> C z = split tt -> x

3.2.3 Bool

62 *(Bool). bool is a run-time version of the boolean logic you may use in your general purpose applications. bool is isomorphic to 1+1: either unit unit. data bool = false | true b1: U = bool -> bool b2: U = bool -> bool -> bool negation: b1 = split false -> true; true -> false or: b2 = split false -> idfun bool; true -> lambda bool bool true and: b2 = split false -> lambda bool bool false; true -> idfun boo boolEq: b2 = lamb bool (bool -> bool) negation boolRec (C: U) (f t: C): bool -> C = split false -> f ; true -> t boolInd (C: bool -> U) (f: A false) (t: A true): (n:bool) -> A n = split false -> f ; true -> t*

3.2.4 Maybe

63 *(Maybe). Maybe has representing functor $M_A(X) = 1 + A$. It is used for wrapping values with optional nothing constructor. In ML-family languages this type is called Option (Miranda, ML). There is an isomorphims between (fix maybe) and nat. data maybe (A: U) = nothing | just (x: A) maybeRec (A P: U) (n: P) (j: A -> P): maybe A -> P = split nothing -> n; just a -> j a*

maybeInd (A: U) (P: maybe A -> U) (n: P nothing) (j: (a: A) -> P (just a)): (a: maybe A) -> P a = split nothing -> n ; just x -> j x

3.2.5 Either

either is a representation for sum types or disjunction. data either (A B: U) = left (x: A) | right (y: B) eitherRec (A B C: U) (b: A -> C) (c: B -> C): either A B -> C = split inl x -> b(x) ; inr y -> c(y)

eitherInd (A B: U) (C: either A B -> U) (x: (a: A) -> C (inl a)) (y: (b: B) -> C (inr b)) : (x: either A B) -> C x = split inl i -> x i ; inr j -> y j

3.2.6 Tuple

tuple is a representation for non-dependent product types or conjunction. data tuple (A B: U) = pair (x: A) (y: B) prod (A B: U) (x: A) (y: B): $(.A)*B = (x, y) tupleRec(ABC : U)(c : (x : A)(y : B) -> C) : (x : tupleAB) -> C = splitpairab -> cabtupleInd(AB : U)(C : tupleAB -> U)(c : (x : A)(y : B) -> C(pairxy)) : (x : tupleAB) -> Cx = splitpairab -> cab$

3.2.7 Nat

Pointed Unary System is a category nat with the terminal object and a carrier nat having morphism [zero: 1nat → nat, succ: nat → nat]. The initial object of nat is called Natural Number Object and models Peano axiom set.

data nat = zero | succ (n: nat) natEq: nat -> nat -> bool natCase (C:U) (a b: C): nat -> C natRec (C:U) (z: C) (s: nat->C->C) : (n:nat) -> C

natElim (C:nat->U) (z: C zero) (s: (n:nat)->C(succ n)): (n:nat) -> C(n) natInd (C:nat->U) (z: C zero) (s: (n:nat)->C(n)->C(succ n)): (n:nat) -> C(n)

3.2.8 List

64 *(List). The data type of list L over a given set A can be represented as the initial algebra* $(\mu L_A, in)$ *of the functor* $L_A(X) = 1 + (AX)$*. 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* $in = [nil, cons]$*. data list (A: U) = nil | cons (x:A) (xs: list A) listCase (A C:U) (a b: C): list A -> C listRec (A C:U) (z: C) (s: A->list A->C->C): (n:list A) -> C listElim (A: U) (C:list A->U) (z: C nil) (s: (x:A)(xs:list A)->C(cons x xs)): (n:list A) -> C(n) listInd (A: U) (C:list A->U) (z: C nil) (s: (x:A)(xs:list A)->C(xs)->C(cons x xs)): (n:list A) -> C(n) null (A:U): list A −> bool head (A:U): list A −> maybe A tail (A:U): list A −> maybe (list A) nth (A:U): nat −> list A −> maybeA append (A: U): list A −> list A −> list A reverse (A: U): list A −> list A map (A B: U): (A −> B) −> list A −> list B zip (AB: U): list A −> list B −> list (tuple A B) foldr (AB: U): (A −> B −> B) −> B −> list A −> B foldl (AB: U): (B −> A −> B) −> B −> list A −> B switch (A: U): (Unit −> list A) −> bool −> list A filter (A: U): (A −> bool) −> list A −> list A length (A: U): list A −> nat listEq (A: eq): list A.1 −> list A.1 −> bool*

3.2.9 Stream

stream is a record form of the list's cons constructor. It models the infinity list that has no terminal element.

data stream (A: U) = cons (x: A) (xs: stream A)

3.2.10 Fin

fin is the inductive defintion of set with finite elements.

data fin (n: nat) = fzero | fsucc (. $fin(predn)$)

fz (n: nat): fin (succ n) = fzero fs (n: nat): fin n -> fin (succ n) = $x : finn) - > fsuccx$

3.2.11 Vector

vector is the inductive defintion of limited length list.

data vector (A: U) (n: nat) = nil | cons (_:A)(_:$vectorA(predn)$)

seq — abstract compositional sequences.

data seq (A: U) (B: A -> A -> U) (X Y: A) = seqNil (_:A)|$seqCons(XYZ : A)$(_:BXY)(_:$SeqABYZ$)

3.2.12

You know Church encoding which also has its dependent alanolgue in CoC, however in Coq it is imposible to detive Inductive Principle as type system lacks fixpoint and functional extensionality. The example of working compiler of PTS languages are Om and Morte. Assume we have Church encoded NAT:

[mathescape=true] nat = (X:U) -> (X -> X) -> X -> X

where first parameter ($X- > X$) is a *succ*, the second parameter X is *zero*, and the result of encoding is landed in X. Even if we encode the parameter

[mathescape=true] list (A: U) = (X:U) -> X -> (A -> X) -> X

and paremeter A let's say live in 42 universe and X live in 2 universe, then by the signature of encoding the term will be landed in X, thus 2 universe. In other words such dependency is called impredicative displaying that landed term is not a predicate over parameters. This means that Church encoding is incompatible with predicative type checkers with predicative of predicative-cumulative hierarchies.

In HoTT n-types is encoded as n-groupoids, thus we need to add a predicate in which n-type we would like to land the encoding:

[mathescape=true] NAT (A: U) = (X:U) -> isSet X -> X -> (A -> X) -> X

Here we added isSet predicate. With this motto we can implement propositional truncation by landing term in isProp or even HIT by langing in isGroupoid:

[mathescape=true] TRUN (A:U) type = (X: U) -> isProp X -> (A -> X) -> X S1 = (X:U) -> isGroupoid X -> ((x:X) -> Path X x x) -> X MONOPLE (A:U) = (X:U) -> isSet X -> (A -> X) -> X NAT = (X:U) -> isSet X -> X -> (A -> X) -> X

The main publication on this topic could be found at [**?**] and [**?**]. Here we have the implementation of Unit impredicative encoding in HoTT.

[mathescape=true] upPath (X Y:U)(f:X->Y)(a:X->X): X -> Y = o X X Y f a downPath (X Y:U)(f:X->Y)(b:Y->Y): X -> Y = o X Y Y b f naturality (X Y:U)(f:X->Y)(a:X->X)(b:Y->Y): U = Path (X->Y)(upPath X Y f a)(downPath X Y f b)

unitEnc': U = (X: U) -> isSet X -> X -> X isUnitEnc (one: unitEnc'): U = (X Y:U)(x:isSet X)(y:isSet Y)(f:X->Y) -> naturality X Y f (one X x)(one Y y)

unitEnc: U = (x: unitEnc') * isUnitEnc x unitEncStar: unitEnc = $(X : U)$(_:$isSetX$)$- > idfunX, XY : U)$(_:$isSetX$)$(_:$isSetY$)$- > refl(X- > Y))unitEncRec(C : U)(s : isSetC)(c : C) : unitEnc- > C = z : unitEnc)- > z.1CscunitEncBeta(C : U)(s : isSetC)(c : C) : PathC(unitEncRecCscunitEncStar)c = reflCcunitEncEta(z : unitEnc)$:

$PathunitEncunitEncStarz = undefinedunitEncInd(P : unitEnc- > U)(a : unitEnc) : PunitEncStar- > Pa = substunitEncPunitEncStara(unitEncEtaa)unitEncCondition(n : unitEnc') : isProp(isUnitEncn) = fg : isUnitEncn)- >< h > xy : U)- > X : isSetx)- > Y : isSety)- > F : x- > y)- >< i > R : x)- > Y(F(nxXR))(nyY(FR))(< j > fxyXYF@jR)(< j > gxyXYF@jR)@h@i$

3.3

Homotypy Type Theory takes its origins in 1996 from groupoid interpretation by Hofmann and Streicher's, and later (in 10 years) was formalized by Awodey, Warren and Voevodsky. Voevodsky constrtucted Kan simplicial sets interpretation of type theory and discovered the property of this model, that was named univalence. This property allows to identify isomorphic structures in terms of type theory.

Homotopy type theory to classical homotopy theory is like Euclidian syntethic geometry (points, lines, axioms and deduction rules) to analytical geometry with cartesian coordinates on R^n (geometric and algebraic) [11]

In the same way as inductive types extends MLTT for inductive programming, the higher inductive types (HIT) extend homotopy type theory for geometry programming. You can directly encode CW-complexes by using HIT. The definition of HIT syntax will be given in the next Issue IV: Higher Inductive Types.

3.3.1

The first higher equality we meet in homotopy theory is a notion of homotopy, where we compare two functions or two path spaces (which is sort of dependent families). The homotopy interval I = $[0, 1]$ is the perfect foundation for definition of homotopy.

65 *(Interval). Compact interval. data I = i0 | i1 | seg <i> [(i=0) -> i0, (i=1) -> i1]*

You can think of I as isomorphism of equality type, disregarding carriers on the edges. By mapping $i0, i1 : I$ to $x, y : A$ one can obtain identity or equality type from classic type theory.

66 *(Interval Split). The convertion function from* I *to a type of comparison is a direct eliminator of interval. The interval is also known as one of primitive higher inductive types which will be given in the next Issue IV: Higher Inductive Types. pathToHtpy (A: U) (x y: A) (p: Path A x y): I -> A = split i0 -> x; i1 -> y; seg @ i -> p @ i*

[11] We will denote geometric, type theoretical and homotopy constants bold font R while analitical will be denoted with double lined letters R.

Equality	Homotopy	∞-Groupoid
reflexivity	constant path	identity morphism
symmetry	inversion of path	inverse morphism
transitivity	concatenation of paths	composition of mopphisms

67 *(Homotopy). The homotopy between two function $f, g : X \to Y$ is a continuous map of cylinder $H : X \times I \to Y$ such that*

$$\{\, H\ (x, 0) = f(x), H(x, 1) = g(x).$$

homotopy (X Y: U) (f g: X -> Y) (p: (x: X) -> Path Y (f x) (g x)) (x: X): I -> Y = pathToHtpy Y (f x) (g x) (p x)

3.3.2

The first text about groupoid interpretation of type theory can be found in Francois Lamarche: A proposal about Foundations[12]. Then Martin Hofmann and Thomas Streicher wrote the initial document on groupoid interpretation of type theory[13].

There is a deep connection between higher-dimentinal groupoids in category theory and spaces in homotopy theory, equipped with some topology. The category or groupoid could be built where the objects are particular spaces or types, and morphisms are path types between these types, composition operation is a path concatenation. We can write this groupoid here recalling that it should be category with inverted morphisms.

cat: U = (A: U) * (A -> A -> U) groupoid: U = (X: cat) * isCatGroupoid X PathCat (X: U): cat = (X,$xy : X) -> PathXxy$)

isCatGroupoid (C: cat): U = (id: (x: C.1) -> C.2 x x) * (c: (x y z:C.1) -> C.2 x y -> C.2 y z -> C.2 x z) * (inv: (x y: C.1) -> C.2 x y -> C.2 y x) * (inv$_l eft$: $(xy : C.1)(p : C.2xy) -> Path(C.2xx)(cxyxp(invxyp))(idx)) *$ ($inv_r ight$: $(xy : C.1)(p : C.2xy) -> Path(C.2yy)(cyxy(invxyp)p)(idy)) *$ ($left$: $(xy : C.1)(f : C.2xy) -> Path(C.2xy)(cxxy(idx)f)f) * (right : (xy :$ $C.1)(f : C.2xy) -> Path(C.2xy)(cxyyf(idy))f) * ((xyzw : C.1)(f : C.2xy)(g :$ $C.2yz)(h : C.2zw) -> Path(C.2xw)(cxzw(cxyzfg)h)(cxywf(cyzwgh)))$

PathGrpd (X: U) : groupoid = ((Ob,Hom),id,c,sym X,compPathInv X,compInvPath X,L,R,Q) where Ob: U = X Hom (A B: Ob): U = Path X A B id (A: Ob): Path X A A = refl X A c (A B C: Ob) (f: Hom A B) (g: Hom B C): Hom A C = comp (<i> Path X A (g@i)) f [] From here should be clear what it meant to be groupoid interpretation of path type in type theory. In the same way we can construct categories of $\prod$ and $\sum$ types. In Issue VIII: Topos Theory such categories will be given.

[12] http://www.cse.chalmers.se/ coquand/Proposal.pdf

[13] Martin Hofmann and Thomas Streicher. The Groupoid Interpretation of Type Theory. 1996.

3.3.3

68 *(funExt-Formation)* $funext_form(AB : U)(fg : A- > B) : U = Path(A- > B)fg$

69 *(funExt-Introduction) funext (A B: U) (f g: A -> B) (p: (x:A) -> Path B (f x) (g x)) :* $funext_formABfg =< i > a : A)- > pa@i$

70 *(funExt-Elimination) happly (A B: U) (f g: A -> B) (p:* $funext_formABfg)(x : A) : PathB(fx)(gx) = cong(A- > B)B(h : A- > B)- > applyABhx)fgp$

71 *(funExt-Computation)* $funext_Beta(AB : U)(fg : A- > B)(p : (x : A)- > PathB(fx)(gx)) : (x : A)- > PathB(fx)(gx) = x : A)- > happlyABfg(funextABfgp)x$

72 *(funExt-Uniqueness)* $funext_Eta(AB : U)(fg : A- > B)(p : Path(A- > B)fg) : Path(Path(A- > B)fg)(funextABfg(happlyABfgp))p = refl(Path(A- > B)fg)p$

3.3.4

73 *( ). pullback (A B C:U) (f: A -> C) (g: B -> C): U = (a: A) * (b: B) * Path C (f a) (g b)*

pb1 (A B C: U) (f: A -> C) (g: B -> C) : pullback A B C f g -> A = $x : pullbackABCfg)- > x.1$

pb2 (A B C: U) (f: A -> C) (g: B -> C) : pullback A B C f g -> B = $x : pullbackABCfg)- > x.2.1$

pb3 (A B C: U) (f: A -> C) (g: B -> C) : (x: pullback A B C f g) -> Path C (f x.1) (g x.2.1) = $x : pullbackABCfg)- > x.2.2$

74 *( ). kernel (A B: U) (f: A -> B): U = pullback A A B f f*

75 *( ). hofiber (A B: U) (f: A -> B) (y: B): U =* $pullback\ A\ unit\ B\ f\ (x : unit)- > y)$

76 *( ). pullbackSq (Z A B C: U) (f: A -> C) (g: B -> C) (z1: Z -> A) (z2: Z -> B): U = (h: (z:Z) -> Path C ((o Z A C f z1) z) (((o Z B C g z2)) z)) * isEquiv Z (pullback A B C f g) (induced Z A B C f g z1 z2 h)*

24 *( ). completePullback (A B C: U) (f: A -> C) (g: B -> C) : pullbackSq (pullback A B C f g) A B C f g (pb1 A B C f g) (pb2 A B C f g)*

3.3.5

77 *(Fibration-1) Dependent fiber bundle derived from Path contractability.* *isFBundle1 (B: U) (p: B -> U) (F: U): U = (*$:(b : B) -> isContr(PathU(pb)F)) * ((x : SigmaBp) -> B)$

78 *(Fibration-2). Dependent fiber bundle derived from surjective function.* *isFBundle2 (B: U) (p: B -> U) (F: U): U = (V: U) * (v: surjective V B) * ((x: V) -> Path U (p (v.1 x)) F)*

79 *(Fibration-3). Non-dependent fiber bundle derived from fiber truncation.* *im1 (A B: U) (f: A -> B): U = (b: B) * pTrunc ((a:A) * Path B (f a) b) BAut (F: U): U =* $im1\ unit\ U\ (x : unit) -> F) unitIm1(AB : U)(f : A -> B) : im1ABf -> B = x : im1ABf) -> x.1 unitBAut(F : U) : BAutF -> U = unitIm1 unitU(x : unit) -> F)$

*isFBundle3 (E B: U) (p: E -> B) (F: U): U = (X: B -> BAut F) * (classify B (BAut F) (b :* $B) -> fiberEBpb)(unitBAutF)X) where classify(A' A : U)(E' : A' -> U)(E : A -> U)(f : A' -> A) : U = (x : A') -> PathU(E'(x))(E(f(x)))$

80 *(Fibration-4). Non-dependen fiber bundle derived as pullback square.* *isFBundle4 (E B: U) (p: E -> B) (F: U): U = (V: U) * (v: surjective V B) * (v': prod V F -> E) * pullbackSq (prod V F) E V B p v.1 v' (x : prodVF) -> x.1)*

3.3.6

81 *(Equivalence).* *fiber (A B: U) (f: A -> B) (y: B): U = (x: A) * Path B y (f x) isSingleton (X:U): U = (c:X) * ((x:X) -> Path X c x) isEquiv (A B: U) (f: A -> B): U = (y: B) -> isContr (fiber A B f y) equiv (A B: U): U = (f: A -> B) * isEquiv A B f*

82 *(Surjective).* *isSurjective (A B: U) (f: A -> B): U = (b: B) * pTrunc (fiber A B f b)*

*surjective (A B: U): U = (f: A -> B) * isSurjective A B f*

83 *(Injective).* *isInjective' (A B: U) (f: A -> B): U = (b: B) -> isProp (fiber A B f b)*

*injective (A B: U): U = (f: A -> B) * isInjective A B f*

84 *(Embedding).* *isEmbedding (A B: U) (f: A -> B) : U = (x y: A) -> isEquiv (Path A x y) (Path B (f x) (f y)) (cong A B f x y)*

*embedding (A B: U): U = (f: A -> B) * isEmbedding A B f*

85 *(Half-adjoint Equivalence).* *isHae (A B: U) (f: A -> B): U = (g: B -> A) * (eta*$:Path(idA)(oABAgf)(idfunA)) * (eps$*:Path(idB)(oBABfg)(idfunB))*$((x : A) -> PathB(f((eta_@0)x))((eps_@0)(fx)))$

*hae (A B: U): U = (f: A -> B) * isHae A B f*

3.3.7

86 *(iso-Formation)* $iso_Form(AB : U) : U = isIsoAB -> PathUAB$

87 *(iso-Introduction)* $iso_Intro(AB : U) : iso_FormAB$

88 *(iso-Elimination)* $iso_Elim(AB : U) : PathUAB -> isIsoAB$

89 *(iso-Computation)* $iso_Comp(AB : U)(p : PathUAB) : Path(PathUAB)(iso_IntroAB(iso_ElimABp))p$

90 *(iso-Uniqueness)* $iso_Uniq(AB : U)(p : isIsoAB) : Path(isIsoAB)(iso_ElimAB(iso_IntroABp))p$

3.3.8

91 *(uni-Formation)* $univ_Formation(AB : U) : U = equivAB -> PathUAB$

92 *(uni-Introduction) equivToPath (A B: U): univ*$_F$*ormationAB* $= p : equivAB) -><i>GlueB[(i = 0) -> (A, p), (i = 1) -> (B, substU(equivB)BB(<_> B)(idEquivB))]$

93 *(uni-Elimination) pathToEquiv (A B: U) (p: Path U A B) : equiv A B = subst U (equiv A) A B p (idEquiv A)*

94 *(uni-Computation) eqToEq (A B : U) (p : Path U A B) : Path (Path U A B) (equivToPath A B (pathToEquiv A B p)) p = <j i> let Ai: U = p@i in Glue B [(i=0) -> (A,pathToEquiv A B p), (i=1) -> (B,pathToEquiv B B (<k> B)), (j=1) -> (p@i,pathToEquiv Ai B (<k> p @ (i k)))]*

95 *(uni-Uniqueness) transPathFun (A B : U) (w: equiv A B) : Path (A -> B) w.1 (pathToEquiv A B (equivToPath A B w)).1*

3.4

CW-complexes are fundamental objects in homotopy type theory and even included inside cubical type checker in a form of higher (co)-inductive types (HITs). Just like regular (co)-inductive types could be described as recur- sive terminating (well-founded) or non-terminating trees, higher inductive types could be described as CW-complexes. Defining HIT means to define some CW- complex directly using cubical homogeneous composition structure as an element of initial algebra inside cubical model.

3.4.1

96 *(Interval). Compact interval. data I = i0 | i1 | seg <i> [(i=0) -> i0, (i=1) -> i1]*

You can think of I as isomorphism of equality type, disregarding carriers on the edges. By mapping $i0, i1 : I$ to $x, y : A$ one can obtain identity or equality type from classic type theory.

3.4.2 n-

97 *(Shperes and Disks). Here are some example of using dimensions to construct spherical shapes. data S1 = base | loop <i> [(i = 0) -> base, (i = 1) -> base] data S2 = point | surf <i j> [(i = 0) -> point, (i = 1) -> point, (j = 0) -> point, (j = 1) -> point] (j = 0) -> point, (j = 1) -> point]*

3.4.3

98 *(Suspension). data susp (A: U) = north | south | merid (a: A) <i> [(i = 0) -> north , (i = 1) -> south]*

3.4.4

99 *(Truncation).*

data pTrunc (A: U) – (-1)-trunc, mere proposition truncation = pinc (a: A) | pline (x y: pTrunc A) <i> [(i = 0) -> x, (i = 1) -> y]

data sTrunc (A: U) – (0)-trunc, set truncation = sinc (a: A) | sline (a b: sTrunc A) (p q: Path (sTrunc A) a b) lt;i jgt; [(i = 0) -> p @ j, (i = 1) -> q @ j, (j = 0) -> a, (j = 1) -> b]

data gTrunc (A: U) – (1)-trunc, groupoid truncation = ginc (a: A) | gline (a b: gTrunc A) (p q: Path (gTrunc A) a b) (r s: Path (Path (gTrunc A) a b) p q) lt;i j kgt; [(i = 0) -> r @ j @ k, (i = 1) -> s @ j @ k, (j = 0) -> p @ k, (j = 1) -> q @ k, (k = 0) -> a, (k = 1) -> b]

3.4.5

100 *(Quotient). data quot (A: U) (R: A -> A -> U) = inj (a: A) | quoteq (a b: A) (r: R a b) lt;igt; [(i = 0) -> inj a, (i = 1) -> inj b] data setquot (A: U) (R: A -> A -> U) = quotient (a: A) | identification (a b: A) (r: R a b) lt;igt; [(i = 0) -> quotient a, (i = 1) -> quotient b] | setTruncation (a b: setquot A R) (p q: Path (setquot A R) a b) lt;i jgt; [(i = 0) -> p @ j, (i = 1) -> q @ j, (j = 0) -> a, (j = 1) -> b]*

3.4.6

101 *(Pushout). One of the notable examples is pushout as it's used to define the cell attachment formally, as others cofibrant objects. data pushout (A B C: U) (f: C -> A) (g: C -> B) = po1 (:A)|po2(:B)|po3(c : C) < i > [(i = 0)− > po1(fc), (i = 1)− > po2(gc)]*

3.5

3.5.1

Process Calculus defines formal business process engine that could be mapped onto Synrc/BPE Erlang/OTP application or OCaml Lwt library with Coq.io

front-end. Here we will describe an Erlang approach for modeling processes. We will describe process calculus as a formal model of two types: 1) the general abstract MLTT interface of process modality that can be used as a formal binding to low-level programming or as a top-level interface; 2) the low-level formal model of Erlang/OTP generic server.

102 *(Storage). The secure storage based on verified cryptography. NOTE: For simplicity let it be a compatible list. storage: U -> U = list*

103 *(Process). The type formation rule of the process is a* Σ *telescope that contains: i) protocol type; ii) state type; iii) in-memory current state of process in the form of cartesian product of protocol and state which is called signature of the process; iv) monoidal action on signature; v) persistent storage for process trace. process : U = (protocol state: U) * (current: prod protocol state) * (act: id (prod protocol state)) * (storage (prod protocol state))*

104 *(Spawn). The sole introduction rule, process constructor is a tuple with filled process type information. Spawn is a modal arrow representing the fact that process instance is created at some scheduler of CPU core. spawn (protocol state: U) (init: prod protocol state) (action: id (prod protocol state)) : process = (protocol,state,init,action,nil)*

105 *(Accessors). Process type defines following accessors (projections, this eliminators) to its structure: i) protocol type; ii) state type; iii) signature of the process; iv) current state of the process; v) action projection; vi) trace projection. protocol (p: process): U = p.1 state (p: process): U = p.2.1 signature (p: process): U = prod p.1 p.2.1 current (p: process): signature p = p.2.2.1 action (p: process): id (signature p) = p.2.2.2.1 trace (p: process): storage (signature p) = p.2.2.2.2*

NOTE: there are two kinds of approaches to process design: 1) Semigroup: $P \times S \to S$; and 2) Monoidal: $P \times S \to P \times S$, where P is protocol and S is state of the process.

106 *(Receive). The modal arrow that represents sleep of the process until protocol message arrived. receive (p: process) : protocol p = axiom*

107 *(Send). The response free function that represents sending a message to a particular process in the run-time. The Send nature is async and invisible but is a part of process modality as it's effectfull. send (p: process) (message: protocol p) : unit = axiom*

108 *(Execute). The Execute function is an eliminator of process stream performing addition of a single entry to the secured storage of the process. Execute is a transactional or synchronized version of asynchronous Send. execute (p: process) (message: protocol p) : process = let step: signature p = (action p) (message, (current p).2) in (protocol p, state p, step, action p, cons step (trace p))*