Simple Types/Non-dependent types

Type theory ≈ Computational view of types

Categorical judgments

\[ A \text{ type} \]

\[ a : A \]

\[ a_1 = a_2 : A \quad a_1 \neq a_2 \text{ are definitionally equal as elements of type } A \]

\[ \text{sets} \]

\[ \text{not a useful intuition} \]

\[ \text{Hypothesis judgment} \]

\[ \Gamma, x_1 : A_1, \ldots, x_n : A_n \vdash a(x_1, \ldots, x_n) : A \]

\[ \text{abbr} \]

\[ \Gamma \]

\[ \text{variables} \]

\[ \text{open term} \]

One idea: a mapping: \[ a : A_1, \ldots, A_n \rightarrow A \]

The term is parameterized by \( x_1, \ldots, x_n \)

Structural properties ← by experience, it doesn’t work if this is not presented earlier

Identity

\[ \Gamma, x : A, \Gamma' \vdash x : A \]

"I’m writing checks that Frank and Steve have to cash"

\[ \text{Computation} \]

\[ \Gamma, x : A, \Gamma' \vdash b : B \quad \Gamma \vdash a : A \]

\[ \Gamma, \Gamma' \vdash [a/x] b : B \]

"It’s like US algebra" ← variable

A generalized US mathematics

\[ \text{Composition} \]

\[ \Gamma, x : A, \gamma : A, \Gamma' \vdash b : B \]

\[ \Gamma, \Gamma' \vdash [\gamma/x, y] b : B \]

\[ \text{Possibility of various dependence} \]
\[
\frac{\Gamma, x : A, y : B, \Gamma \vdash c : C}{\Gamma, y : B, x : A, \Gamma \vdash c : C} \quad \text{exchange} \quad \text{a little more complicated in dep theory}
\]

---

\[
\text{end structural properties}
\]

\[
\text{“What are the rules for variables? how do assignments behave?”}
\]

---

**Def’al equality**

1) Equivalence relation

2) Congruence (see rules: “you can replace equals w/ equals”)

3) **Functionality** maps respect defl equality

\[
\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash [a_1/x]b = [a_2/x]b : B} \quad \text{equality is delicate & important}
\]

(what I say and don’t say)

---

**Defining types**

1) Formation: how to construct a type

2) Intro

3) Elim

4) Congruence principles

5) Composition rules

(?) 6) Unicity/universality characterize the type

---

it all comes down to that

comes from category theory
Limits
Negative types

Intuitively, either the introduction is "primary" or the elimination is primary (elim-oriented).

Focusing

Unit

\[ \Gamma \vdash \text{unit} \]

\[ \Gamma \vdash * : 1 \]

(no elms)

(no cons)

Product

A type \( \rightarrow \) B type \( \times F \)

\[ \Gamma \vdash \text{a} : A \quad \Gamma \vdash \text{b} : A \]

\[ \Gamma \vdash \langle \text{a}, \text{b} \rangle : A \times B \]

or

\[ \Gamma, \alpha : A, \beta : B \vdash \langle \alpha, \beta \rangle : A \times B \]

will not rewrite

Congruence rules

\[ \Gamma \vdash \text{c} : A \times B \quad \Gamma \vdash \text{c} : A \times B \]

\[ \Gamma \vdash \text{fst } \text{c} : A \quad \Gamma \vdash \text{snd } \text{c} : B \]

\[ \Gamma \vdash \text{a} = \text{a}_2 : A \quad \Gamma \vdash \text{b} = \text{b}_2 : B \]

\[ \Gamma \vdash \langle \text{a}, \text{b} \rangle = \langle \text{a}_2, \text{b}_2 \rangle : A \times B \]

\[ \Gamma \vdash \text{c} = \text{c}_2 : A \times B \]

\[ \Gamma \vdash \text{fst } \text{c} = \text{fst } \text{c}_2 : A \times B \]
\[
\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash \text{fst} \langle a, b \rangle \equiv a : A} \quad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash \text{snd} \langle a, b \rangle \equiv b : B}
\]

Commutation conditions

\[
\begin{array}{c}
\text{uniqueness} \\
\Gamma \vdash c : A \times B \\
\Gamma \vdash \langle \text{fst} \ c, \text{snd} \ c \rangle \equiv c : A \times B
\end{array}
\]

(revisit this later)

is this the right unicity principle?

(Should be a weaker condition then definitional equivalence)

Turned into: "homotopy ceranity"

have a 2-cell (map between maps) which relates the two.

\[
\begin{array}{c}
\text{2-cell (face)} \\
C \leftarrow 2\text{-cell (edge)}
\end{array}
\]

\[
\text{2-cell (face)} \quad \text{2-cell (vertex)}
\]
**Function Space**

"The pointere example"

\[
\begin{align*}
\frac{A \text{ type} \quad B \text{ type}}{A \to B \text{ type}} & \quad \rightarrow F \\
\frac{\Gamma, x:A \vdash b:B}{\Gamma \vdash \lambda x. b : A \to B} & \quad \rightarrow I \\
\frac{\Gamma \vdash b : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash b(a) : B} & \quad \rightarrow E \\
\frac{\Gamma, x:A \vdash \lambda x.b : B \quad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x.b)(a) = [\alpha/x].b : B} & \quad \rightarrow C (\beta) \\
\frac{\Gamma \vdash a : A \to B}{\Gamma \vdash \lambda x.b_1 \equiv \lambda x.b_2 : A \to B} & \quad \rightarrow U (\xi) \chi \to \text{ not function extensionally} \\
\end{align*}
\]

\[
\begin{align*}
\frac{\Gamma \vdash a : A \to B}{\Gamma \vdash a \equiv \lambda x.\alpha(x) : A \to B} & \quad (\eta) \\
\end{align*}
\]

\[
\begin{align*}
\frac{\Gamma, x:A \vdash a_1(x) \equiv a_2(x) : B}{\Gamma \vdash a_1 \equiv a_2 : A \to B} & \quad \text{(ext)} \\
\text{Warning:} \quad \forall x : \mathbb{N}, x + 0 \neq \exists a : \mathbb{N}, a + x \\
\text{Let} \quad x : \mathbb{N} \vdash x + 0 \neq 0 + \alpha : \mathbb{N} \\
\text{Of this algo: definitions equality is simplification, but some equations require proof} \\
\text{to show this requires proof}
\end{align*}
\]
Positive types

\[ \text{Empty} \]

\[ \frac{\text{type}}{\Gamma \vdash a : 0} \]

\[ \text{case for products} \]

\[ \frac{\text{sum}}{A \text{ type } \ B \text{ type } + F} \]

A type \ B type + F

\[ \frac{\text{inl}}{\Gamma \vdash \text{inl}(a) : A + B} \]

\[ \frac{\text{inr}}{\Gamma \vdash \text{inr}(b) : A + B} \]

\[ \frac{\text{case}}{\Gamma \vdash \text{case}(c; x, y; e) : C} \]

\[ \text{case \ (inl(a); x, y; e) \equiv [a/x]e : C} \]

\[ \text{case \ (inr(b); x, y; e) \equiv [b/y]e : C} \]

Nullity?

In Horn, open structure of +/- types is different

Why no diagram for functions?

End express unicity of elimination rule
Props as Types

STN IPL

1 \iff T

A \times B \iff A \land B

\therefore validity is justified by proof theory

A type \iff A prop

M : (A) \iff A true

\text{idee': } x \iff A \leq B

(\exists B)

\text{preorder on propositions?!}

\text{HW: 1) show } \leq \text{ is a preorder (RT)}

2) T is greatest \ A \leq T

A \land B \iff A \iff B \iff A \land B \leq A

\text{CSA} \iff \text{CSB}

\text{CSA} \iff \text{C} \leq A \land B

3) T is least \ A \leq A \lor B \iff B \leq A \lor B

A \lor B \iff \text{lub}

\text{A5O} \iff \text{B5C}

\text{A5O} \iff \text{A} \lor \text{B} \leq \text{C}

\text{4) } \implies \text{ is exponential } \ (A \implies B) \land A \leq B

\text{we have a Heyting (pre-)algebra}

(\text{no antisymmetry w/o univalence})

\text{C5A5B}

\text{C5A} \iff \text{B5A5B}

\text{5) distributivity } \ (\land \text{ dist over } \lor)

\text{"Squeezing balloon" : it just pops out somewhere else, but it's the same thing}
Negation

\[ \neg A \equiv A \rightarrow 0 \quad i.e. \ A \Rightarrow \bot \]

easy: \ A \leq \neg \neg A

doubt: \neg \neg A \leq A \quad \text{answering this is a lot of work, but we do NOT have this in general}

\[ \text{aim of proof theory} \]

"I don't not like asparagus"

Irrefutability \neq \text{assert}

\[ \text{trivial: } A \land \neg A \leq \top \]

\[ \text{not: } \top \leq A \lor \neg A \quad \text{in general} \]

Define \( \overline{A} \) (complement of \( A \)) to be smallest \( B \) s.t. \( A \land B \leq \top \)

\[ \text{Note that } A \land \neg A \leq \bot. \]

\[ \text{But if you know something about } A \]

\[ \text{Don't have this in Heyting algebras} \]

If we have \( \overline{A} \), then \( \neg \neg A = A \)

\[ \text{Boolean Algebra: complemented HA} \]

\[ \text{every } A \text{ has a complemented } \overline{A} \]

The part is, \( \neg A \) is \textcolor{red}{\text{not}} \ a complement

(there is no closed world assumption)

Ex.) Show that \( \neg \neg (A \lor \neg A) \) for general \( A \)

\[ \rightarrow \text{ intuitionistic logic is consistent with classical logic} \]
The power of homotopy type theory is the ability to make finer distinctions.
Correction

\( \neg A \) is by def the largest \( B \) inconsistent with \( A \)
1) \( A \land \neg A \leq \bot \)
2) if \( A \land B \leq \bot \) then \( B \leq \neg A \)

\( \bar{A} \) is by def the smallest \( B \) that complements \( A \)
1) \( T \leq A \lor \bar{A} \)
2) if \( T \leq A \lor B \) then \( \bar{A} \leq B \)

So \( \neg A \leq \bar{A} \) but we do not in general have \( \bar{A} \leq \neg A \)

if \( \neg A \leq \bar{A} \), then we have Boolean Algebra in which

\( T \leq A \lor \neg A = A \lor \bar{A} \)
\( \bar{A} \leq A \) (and \( A \leq \bar{A} \))

"real p research"
Key idea: Type-indexed family of types

eg) \( x : \mathbb{N} \vdash \text{Vec}(x) \) type

\[ \{ \text{Vec}(x) \}_{x : \mathbb{N}} \]

Type of vectors of length \( x \).

So if I have \( a : \mathbb{N} \) then Vec\( a \) is a type.

\( a_1 = a_2 : \mathbb{N} \quad \text{Vec}(a_1) \equiv \text{Vec}(a_2) \)

\( \sim \) definitional equality of types

eg) \( \pi, \gamma : A \vdash \text{Id}_A(\pi,\gamma) \) type

Type of identifications of \( \pi, \gamma \)

Proofs of equivalence.

Cells/paths (Hott)

eg) \( \pi, \gamma : A, \rho, \eta : \text{Id}_A(\pi,\gamma) \vdash \text{Id}_A(\text{Id}_A(\pi,\gamma), (\rho, \eta)) \)

\( \sim \) higher dimensional type theory

Eventually: a type is an \( \infty \)-groupoid

\[
\begin{array}{c}
\Gamma \vdash A \text{ type} \\
\Gamma \vdash A_1 \equiv A_2 \\
\Gamma \vdash a_1 = a_2 : A \\
\Gamma \vdash \text{ot} x \\
\Gamma' \vdash x : \Gamma \\
\end{array}
\]

\[ \{ (A(a_1,-), A(a_2,-))_{a_1, a_2 : A} \}_{a_1, a_2 : A} \]

no longer just a mapping from \( \Gamma \) to \( A \)

(\text{So you need fibrations})
Structural Properties

\[
\Gamma, x : A, \Gamma' \vdash x : A \quad \forall \mathcal{R} \quad \text{can be } b : B \text{ or } B \text{ type}
\]

\[
\Gamma, x : A, \Gamma' \vdash J \quad \Gamma \vdash a : A \quad \text{s/t}
\]

\[
\Gamma[a/x] \Gamma' \vdash [a/x] J
\]

sequential dependencies

\[
\begin{align*}
\Gamma, x : A, \Gamma' \vdash B \text{ type} & \quad \Gamma \vdash a_1 \equiv a_2 : A \\
\Gamma[a/x] \Gamma' \vdash [a_1/x] B \equiv [a_2/x] B & \quad \text{or} 2 \\
\end{align*}
\]

\[
\begin{align*}
\Gamma, x : A, \Gamma' \vdash b B & \quad \Gamma \vdash a_1 \equiv a_2 : A \\
\Gamma[a/x] \Gamma' \vdash [a_1/x] B \equiv [a_2/x] B & \quad \text{or} 2 \\
\end{align*}
\]

functionality

\[
\begin{align*}
\Gamma \vdash a : A & \quad \Gamma \vdash A \equiv B \\
\Gamma \vdash a : B & \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash a : A & \quad \Gamma \equiv \Gamma' \\
\Gamma' \vdash a : A & \quad a_1 \equiv a_2 : A \\
\text{A type} & \quad A_1 \equiv A_2
\end{align*}
\]

\[
\begin{align*}
\text{respect} & \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash J & \quad \Gamma \vdash A \text{ type} \\
\Gamma, x : A \vdash J & \quad \text{workbeing} \\
\end{align*}
\]

\[
\Gamma, x : A, \psi : B, \Gamma' \vdash J \\
\Gamma, \psi : B, \psi, x : A, \Gamma' \vdash J
\]

exchange

\[
\Gamma, \psi : A, \psi : A, \Gamma' \vdash J \\
\Gamma, \psi, \psi : [\psi, x : y] \Gamma' \vdash [\psi, x : y] J
\]

\[
\begin{align*}
\text{unification} & \\
\end{align*}
\]

\[
\begin{align*}
\Gamma, \psi : A & \quad \psi : A, \Gamma' \vdash J
\end{align*}
\]

Tilcon: \( \Gamma \) is a DAG
Natural numbers

\[ \Gamma \vdash a : N \quad \Gamma \vdash \text{succ}(a) : N \]

\[ \Gamma \vdash \text{rec}[b; x, y, c](a) : D \]

\[ \text{plus} = \lambda x y. \text{rec}[x; u, v, \text{succ}(v)](y) \]

\[ \text{vec}[b; x, y, c](0) = b : D \]

\[ \text{vec}[b; x, y, c](\text{succ}(a)) = [a, \text{vec}[b; x, y, c](a)/x, y]c : D \]

Check:

\[ a + 0 = a \]

\[ a + \text{succ}b = \text{succ}(a + b) \]

\[ 0 + a \neq a \]

\[ \text{succ}a + b \neq \text{succ}(a + b) \]

\[ \text{for all } m, n, \quad \text{vec}[m, n] + \text{vec}[m, n] = \text{vec}[m + n] \]

\[ \alpha : N, \gamma : N \vdash \alpha + \gamma \neq \gamma + \alpha : N \]

\[ \text{vec}(0 + \alpha) \neq \text{vec} \alpha \]

\[ \text{vec} \alpha = \text{vec}(\alpha + 0) \]

Definitional equality \(~\text{calculational}~\)

Propositional equality \(~\text{proof}~\)
Universe: cumulative hierarchy of universes (simple types)

1. \( U_0 : U_1 : U_2 : \ldots \)
2. \( U_0 \subseteq U_1 \subseteq U_2 \subseteq \ldots \)

replace a type by \( A \in U \) i.e. \( A \in U_i \) for some \( i \)

\[
\begin{align*}
\Gamma \vdash U_i : U_{i+1} & \quad (U_i - P) \\
\Gamma \vdash A : U_i & \quad (U_{i+1} - I)
\end{align*}
\]

Idea: 1) elements of universes are types
2) every type is the element of some universe
Simple types → Dependent types

\[ \Gamma \vdash A : U \quad \Gamma \vdash B : U \]
\[ \Gamma \vdash A \to B : U \]

\[ \Gamma \vdash A : U \quad \Gamma \vdash B : U \]
\[ \Gamma \vdash A + B : U \]

system solves constraints (e.g. Coq)

it's like an inaccessible cardinal, where all the power sets are contained in it

\[ \Gamma, x : A \vdash a(x) : [\text{inl}(x)/x] D \]
\[ \Gamma, y : B \vdash b(y) : [\text{inr}(y)/y] D \]
\[ \Gamma \vdash \text{case} \ [a, a, y, b] (c) : [c/2] D \]

the join of the case does not have to be constant!

Define \( 2 := 1 + 1 \)
\[ \text{tt} := \text{inl}(\text{tt}) \]
\[ \text{ff} := \text{inr}(\text{ff}) \]
\[ \text{if} (a, b, c) := \text{case} [\_ b, \_ c] (a) \]

not a special form, the blender is just degenerate

Notice: \( a : 2 \quad z : 2 \vdash D : U \)
\[ b : [\text{tt/2}] D \]
\[ c : [\text{ff/2}] D \]
\[ \text{if} (a, b, c) : [a/2] D \]

\( a : \text{Vec}(10) \)
\( b : \text{Vec}(20) \)
\( z : 2 \)

beginning of expressiveness of type theory

by analogy: a "unique role" in a "type"

but it's not proper to speak of it that way

as long as B-only, no problems

unicity causes problems

exercise: ind. principle for booleans
Remark: In the setting I'm describing, propositions (things we state in math) are types. You're taught that propositions are booleans; in geometry class, you learned truth tables; the rules of boolean logic, and then you went to Euclid writing chart form proofs, where lines were justified with principles. But I never really did understand what the truth tables had to do with the charts. It's not so clear; the thing they wanted to teach you was the proof objects, but there was this crazy story that there were only two propositions, true and false. I want you to keep in mind: booleans are not propositions, and the if I'm writing is not implication.

All conventional programming languages screw this up; a program that yields a boolean when it's run is a predicate, when it is not. A myriad of messes in PL stem from this misunderstanding.

\[
\Gamma \vdash a : \text{N} \quad \Gamma, z : \text{N} \vdash \text{D} : \text{U} \\
\Gamma \vdash b : [a/z] \text{D} \quad \Gamma, z : \text{N}, y : [a/z] \text{D} \vdash c : \text{true(a/z)D} \\
\Gamma \vdash \text{rec}[b; x, y, c](a) : [a/z] \text{D}
\]

We need languages which are suitable for human discourse but are also executable. That is what type theory is promising us. The math is the code. What the hell are DSLs about? What is a domain? The domain is all of math and science. I don't see how you draw boundaries around domains. You want a language where you can express mathematics, and that should be executable. Mathematics is the language of science. If we have this, then we have a complete unification, and reasoning and programming are the same. The whole point of doing type theory is that grand unified theory.

So a smart person would just go home at this point. But the program verification folks would then write this imperative program with messages and objects and mutable state, and then prove it. But the math is the program. Why not just run the spec? It's like when you dialed phone numbers by moving your finger as long as the number you wanted to dial. It's crazy!