Programming with Higher Inductive Types

Herman Geuvers
(joint work with Niels van der Weide, Henning Basold,
Dan Frumin, Leon Gondelman)

December 16, 2021
FSA Seminar
TU Eindhoven

Overview

- How to define a data type of finite sets?
- Introduction to Dependent Type Theory
- Identity as a type
- Homotopy Type Theory (HoTT)
- A higher inductive type for finite sets

How to define Finite Sets

- Represent a set as a list of elements (with duplicates).
- Operations on sets then become operations on lists.
- But ... not all functions on lists are proper functions on sets (e.g. length)
- In a proper implementation one needs to maintain several invariants.
- What are the proper proof principles and function definition principles for finite sets?

Ingredients of Dependent Type Theory

- Dependent Type Theory (Martin-Löf Type Theory, Calculus of Inductive Constructions, ...) is an integrated system for programming and proving
- Implemented as a Proof Assistant (Coq, Agda, NuPRL, ...)

1. Data types and definition of functions over these
2. Predicate logic via "formula-as-types".
3. Integration of programming and proving
4. Inductive definitions: introduction and elimination rules

Ingredients of DTT: data types and definition of functions

1. Data types are inductive types

```
Inductive List (A : Type) :=
  | nil : List(A)
  | cons : A → List(A) → List(A)
```

2. Functions are defined by pattern matching and well-founded recursion

```
Fixpoint append (A : Type) (ℓ, k : List(A)) :=
  match ℓ with
        | nil        ⇒ k
        | cons a ℓ′ ⇒ cons a (append ℓ′ k)
```

Ingredients of DTT: Predicate logic via "formula-as-types"

1. A proposition is also a type;
   a proposition φ is identified with the type of proofs of φ.
2. $M : A$ is sometimes read as "M is a term of data-type A"
3. $M : A$ is sometimes read as "M is a proof of proposition A"
4. a predicate P on A is a $P : A \to \mathit{Type}$.
5. Dependent function type. Intuitively
   $$\Pi(x : A).B \quad \approx \quad \{f \mid \forall a(a : A \Rightarrow f\,a : B[x := a])\}.$$
6. Example:
   $$\lambda(x : A).\lambda(h : P\,x).h \;:\; \forall(x : A).P\,x \to P\,x$$
   $\forall$ is interpreted as Π.
7. Dependent product type. Intuitively
   $$\Sigma(x : A).B \quad \approx \quad \{\langle a, b\rangle \mid a : A \wedge b : B[x := a])\}.$$
8. Example:
   $$\langle 0, \mathsf{refl}\,0\rangle \;:\; \exists(x : \mathbb{N}).x = x$$
   $\exists$ is interpreted as Σ.

Ingredients of DTT: Integration of programming and proving

Example. Sorting a list of natural numbers.

$$\text{sort} : \text{List}_{\mathbb{N}} \to \text{List}_{\mathbb{N}}$$

More refined:

$$\text{sort} : \text{List}_{\mathbb{N}} \to \exists (y : \text{List}_{\mathbb{N}}), \text{Sorted}(y)$$

$$\text{Sorted}(y) := \forall i < \text{length}(y) - 1 (y[i] \leq y[i+1])$$

Further refined:

$$\text{sort} : \forall (x : \text{List}_{\mathbb{N}}),\ \exists (y : \text{List}_{\mathbb{N}}), (\text{Sorted}(y) \land \text{Perm}(x, y))$$

Ingredients of DTT: Programming with proofs

Example. Sorting a list of natural numbers.

$$\mathsf{sort} : \forall(x : \mathsf{List}_{\mathbb{N}}),\ \exists(y : \mathsf{List}_{\mathbb{N}}), (\mathsf{Sorted}(y) \wedge \mathsf{Perm}(x, y))$$

The proof sort contains a sorting program that can be extracted

$$\mathsf{sort} : \forall(x : \mathsf{List}_{\mathbb{N}}),\ \exists(y : \mathsf{List}_{\mathbb{N}}), \mathsf{Sorted}(y) \wedge \mathsf{Perm}(x, y))$$

$$\widehat{\mathsf{sort}} : \mathsf{List}_{\mathbb{N}} \to \mathsf{List}_{\mathbb{N}}$$

$$\mathsf{correct} : \forall(x : \mathsf{List}_{\mathbb{N}}), (\mathsf{Sorted}(\widehat{\mathsf{sort}}(x)) \wedge \mathsf{Perm}(x, \widehat{\mathsf{sort}}(x)))$$

Ingredients of DTT: Inductive definitions

Example of inductive data types of lists.

```
Inductive List (A : Type) :=
    | nil : List(A)
    | cons : A → List(A) → List(A)
```

This generates

1. constructors
2. a definition scheme for recursive functions on List
3. a principle for proofs by induction over List
4. These are the same (!) elimination principle for List.
   For $P : \mathsf{List}(A) \to \mathit{Type}$:

$$\frac{f_0 : P\ \mathsf{nil} \qquad f_c : \Pi(\ell : \mathsf{List}(A)).P\,\ell \to \Pi(a : A).P\,(\mathsf{cons}\ a\,\ell)}{\mathsf{Rec}\ f_0\,f_c : \Pi(\ell : \mathsf{List}(A)).P\,\ell}$$

Identity is defined inductively

Identity is an inductive type Id (with notation "=")

```
Inductive Id (A : Type) : A → A → Type :=
                  | refl : Π(x : A).x = x
```

The smallest binary relation on A containing $\{(x, x) \mid x : A\}$.

Giving

$\mathrm{refl} : \Pi(A : \mathit{Type})(a : A).a = a$

and the J-rule

$$\frac{P : \Pi(a, b : A).a = b \to \mathit{Type} \qquad r : \Pi(a : A).P\,a\,a\,\mathrm{refl}}{J\,r : \Pi(x, y : A).\Pi(i : x = y).P\,x\,y\,i}$$

with computation rule

$$J\,a\,a\,(\mathrm{refl}\ a) \to r.$$

Properties of the Identity type

The J-rule gives:

- Identity is symmetric: $\mathsf{sym} : a = b \to b = a$
- Identity is transitive: $\mathsf{trans} : a = b \to b = c \to a = c$
- Substitutivity (Leibniz property)

$$\frac{t : Q(a) \qquad r : a = b}{t' : Q(b)}$$

But: t' is not just t. (In fact $t' \equiv J\, a\, b\, r\, t$.)

Properties of the Identity type

The J-rule does not give:

- Function extensionality

$$\frac{f, g : A \to B \qquad r : \forall a : A, f\ a = g\ a}{t : f = g}$$

  for some term t.

- Proof Irrelevance (all proofs are equal).

$$\frac{\text{If } A \text{ is a proposition} \qquad a : A \qquad b : A}{t : a = b}$$

  for some term t.

- Uniqueness of Identity Proofs (UIP).

$$\frac{a, b : A \qquad q_0, q_1 : a = b}{t : q_0 = q_1}$$

  for some term t.

Uniqueness of Identity Proofs (UIP)

Why isn't UIP derivable??

$$\frac{a, b : A \qquad q_0, q_1 : a = b}{t : q_0 = q_1}$$

for some term t.

The intuition of the type $a = b$ is that the only term of this type is refl (and then a and b should be the same).

UIP is equivalent to the K-rule:

$$\frac{a : A \qquad q : a = a}{t : q = \mathrm{refl}\ a\, a}$$

for some term t.

This rule may look even more natural . . . .

There is a countermodel to K (and UIP): M. Hofmann and Th. Streicher, *The groupoid interpretation of type theory*, 1998.

Types are groupoids

A groupoid is defined either as

- A group where the binary operation is a partial function,
- A category in which every arrow is invertible.

For $A : Type$, the proofs of identities between elements of type A form a groupoid:

- For $p : a = b$ and $q : b = c$, we read $p \cdot q$ as composition of p and q (via trans)
- For $p : a = b$, we read p^{-1} as the inverse of a proof (via sym)
- In a groupoid the K rule ($\forall p, p = 1$) obviously does not hold!

Homotopy type theory (HoTT)

Fields medal 2002

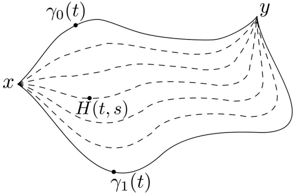
- homotopy theory algebraic varieties
- formulation of motivistic cohomology

Vladimir Voevodsky
2006

mathematics independent of specific definitions

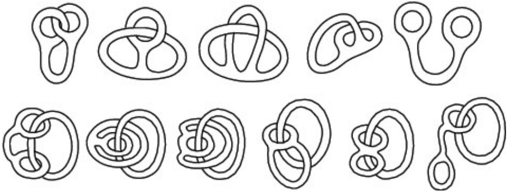
homotopy type theory

- homotopy is the 'proper' notion of equality
- homotopy = continuous transformation

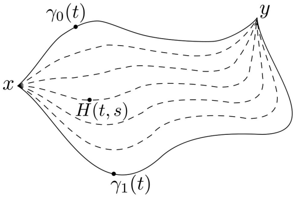


Homotopy Theory

Part of Algebraic Topology dealing with homotopy groups: associating groups to topological spaces to classify them.



- an equality is a path from one object to another (continuous transformation)
- higher equality
  = transformation between paths
  = a path between paths.



Types are topological spaces, equality proofs are paths

Voevodsky: A type A is a topological space and if $a, b : A$ with $p : a = b$, then

p is a continuous path from a to b in A.

If $p, q : a = b$ and $h : p = q$, then

h is a continuous transformation from p to q in A

also called a homotopy.

Equality proofs are paths, path-equalities are higher paths

A property $P : \forall a, b : A, a = b \to \mathit{Type}$ should be closed under continuous transformations of points and paths.

$$\frac{P : \forall a, b : A, a = b \to \mathit{Type} \qquad r : \forall a : A, P\; a\; a\; \mathsf{refl}}{J\, r : \forall x, y : A, \forall i : x = y, P\; x\; y\; i}$$

The following do not hold

$$\frac{a, b : A \qquad q_0, q_1 : a = b}{t : q_0 = q_1}$$

(for some term t)

$$\frac{a : A \qquad q : a = a}{t : q = \mathsf{refl}\; a\; a}$$

(for some term t).

Homotopy Type Theory

Voevodsky's Homotopy Type Theory (HoTT):

- We need to add: Univalence Axiom: for all types A and B:

  $$(A \simeq B) \simeq (A = B)$$

  where $A \simeq B$ denotes that A and B are isomorphic: there are $f : A \to B$ and $g : B \to A$ such that $\forall x : A, g(f\ x) = x$ etc.
- Univalence implies that isomorphic structures can be treated as equal.
- HoTT is the internal language for homotopy theory. All proofs in homotopy theory should be formalised in type theory. (Agda and Coq give support for that.)

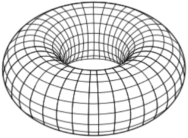
Higher Inductive Types (HITs)

Inductive types + path constructors.

In homotopy theory, one studies the fundamental group of a topological space. Some examples

```
Inductive circle : Type :=
  | base : circle
  | loop : base = base.

Inductive torus : Type :=
| base : torus
| meridian : base = base
| equator : base = base
| surf : meridian · equator = equator · meridian
```



Torus

Questions:

- What are the proper general rules for higher inductive types?
- What are the use cases for higher inductive types in computer science?

Finite Sets according to Kuratowski

A possible definition as an inductive type would be

$$\begin{array}{l} \texttt{Inductive}\ \mathrm{Fin}(\_)\ (A : \mathit{Type}) := \\ \quad \mid\ \emptyset : \mathrm{Fin}(A) \\ \quad \mid\ L : A \to \mathrm{Fin}(A) \\ \quad \mid\ \cup : \mathrm{Fin}(A) \times \mathrm{Fin}(A) \to \mathrm{Fin}(A) \end{array}$$

- Notation: $\{a\}$ for $L\,a$
- Notation: $x \cup y$ for $\cup\, x\, y$
- We require some equations (eg: $\cup$ is commutative, associative, $\emptyset$ is neutral, ...).
- But inductive types are freely generated. We can't simply add extra equations to inductive types.

Possible solutions

1. Data Types with laws (Turner 1980's)
2. Quotient Types
3. Higher Inductive Types

We will look at the last solution.

Example: Finite Sets

```
Inductive Fin (A : Type) :=
```

$\mid\ \emptyset : \mathsf{Fin}(A)$
$\mid\ L : A \to \mathsf{Fin}(A)$
$\mid\ \cup : \mathsf{Fin}(A) \times \mathsf{Fin}(A) \to \mathsf{Fin}(A)$
$\mid\ \mathsf{assoc} : \prod(x, y, z : \mathsf{Fin}(A)), x \cup (y \cup z) = (x \cup y) \cup z$
$\mid\ \mathsf{neut}_1 : \prod(x : \mathsf{Fin}(A)), x \cup \emptyset = x$
$\mid\ \mathsf{neut}_2 : \prod(x : \mathsf{Fin}(A)), \emptyset \cup x = x$
$\mid\ \mathsf{com} : \prod(x, y : \mathsf{Fin}(A)), x \cup y = y \cup x$
$\mid\ \mathsf{idem} : \prod(x : A), \{x\} \cup \{x\} = \{x\}$
$\mid\ \mathsf{trunc} : \prod(x, y : \mathsf{Fin}(A)), \prod(p, q : x = y), p = q$

Elimination Rule for Kuratowski Sets

The non-type dependent variant

$$
\frac{\begin{array}{c}
Y : \mathit{Type} \\
\emptyset_Y : Y \\
L_Y : A \to Y \\
\cup_Y : Y \to Y \to Y \\
a_Y : \prod(a, b, c : Y), a \cup_Y (b \cup_Y c) = (a \cup_Y b) \cup_Y c \\
n_{Y,1} : \prod(a : Y), a \cup_Y \emptyset_Y = a \\
n_{Y,2} : \prod(a : Y), \emptyset_Y \cup_Y a = a \\
c_Y : \prod(a, b : Y), a \cup_Y b = b \cup_Y a \\
i_Y : \prod(a : A), \{a\}_Y \cup_Y \{a\}_Y = \{a\}_Y \\
\mathrm{trunc}_Y : \prod(x, y : Y), \prod(p, q : x = y), p = q
\end{array}}{\mathrm{Fin}(A)\text{-rec}(\emptyset_Y, L_y, \cup_Y, a_Y, n_{Y,1}, n_{Y,2}, c_Y, i_Y) : \mathrm{Fin}(A) \to Y}
$$

Example: membership

We define $\in : A \to \mathrm{Fin}(A) \to Type$.
For $a : A$, $X : \mathrm{Fin}(A)$ we define membership of a in X by recursion over X:

$$\begin{aligned} a \in \emptyset &:= \bot, \\ a \in \{b\} &:= ||a = b||, \\ a \in (x_1 \cup x_2) &:= ||a \in x_1 + a \in x_2|| \end{aligned}$$

Here $||A||$ denotes the truncation of A: the type A where we have identified all elements.

We can prove the following **Theorem** (Set-extensionality):
For all $x, y : \mathrm{Fin}(A)$,
the types $x = y$ and $\prod(a : A), a \in x = a \in y$ are equivalent.

The size of a finite set

The size of a finite set $x : \mathsf{Fin}(A)$ is hard to define.

- We need to decide equality on A. We can only compute size of a finite set if A has decidable equality: we have a term dec with
$$\mathsf{dec} : \Pi(x, y : A)||x = y|| + ||\neg(x = y)||.$$

- The $x \cup y$ is tricky:
$$\#(x \cup y) := \#x + \#y - \#(x \cap y) \quad ??$$
So we first need to define $(x \cap y)$, but then still, the recursive call $\#(x \cap y)$ is not structurally smaller...

Solution: we give an alternative definition of finite sets, $\mathsf{Enum}(A)$, using lists for which we can define the size of a set $\#(x)$ easily. (And we show that $\mathsf{Enum}(A) \simeq \mathsf{Fin}(A)$.)

Alternative definition using lists

We define finite sets using lists.

$\texttt{Inductive}$ Enum $(A : \mathit{Type}) :=$
$|\ \mathsf{nil} : \mathsf{Enum}(A)$
$|\ \mathsf{cons} : A \to \mathsf{Enum}(A) \to \mathsf{Enum}(A)$
$|\ \mathsf{dupl} : \prod(a : A) \prod(x : \mathsf{Enum}(A)), \mathsf{cons}\ a\,(\mathsf{cons}\ a\,x) = \mathsf{cons}\ a\,x$
$|\ \mathsf{comm} : \prod(a, b : A) \prod(x : \mathsf{Enum}(A)), \mathsf{cons}\ a\,(cons\,b\,x) = \mathsf{cons}\ b\,(cons\,a\,x)$
$|\ \mathsf{trunc} : \prod(x, y : \mathsf{Enum}(A)), \prod(p, q : x = y), p = q$

It can be proven that

$$\mathsf{Enum}(A) \simeq \mathsf{Fin}(A)$$

The size of a finite set

Using the alternative definition $\mathrm{Enum}(A)$ we can define the size of a set $\#(x)$, for types A with a decidable equality.

$$\begin{aligned} \#(\mathrm{nil}) &:= 0, \\ \#(\mathrm{cons}\ a\, k) &:= \#\, k \text{ if } a \in k \\ \#(\mathrm{cons}\ a\, k) &:= 1 + \#\, k \text{ if } a \notin k \end{aligned}$$

Note: a simple length function of the underlying list is just not well-defined: If we set

$$\mathrm{length}(\mathrm{cons}\ a\, k) := 1 + \#\, k$$

then we can't prove the equality

$$\mathrm{length}(\mathrm{cons}\ a\, (\mathrm{cons}\ a\, k)) = \mathrm{length}(\mathrm{cons}\ a\, k)$$

so we cannot type such a function with

$$\mathrm{length} : \mathrm{Enum}(A) \to \mathbb{N}.$$

Interface for Finite Sets

A type operator $T : \mathit{Type} \to \mathit{Type}$ is an implementation of finite sets if for each A the type $T(A)$ has

- $\emptyset_{T(A)} : T(A)$,
- an operation $\cup_{T(A)} : T(A) \to T(A) \to T(A)$,
- for each $a : A$ there is $\{a\}_{T(A)} : T(A)$,
- a predicate $a \in_{T(A)} \_\_ : T(A) \to \mathit{Type}$

and there is a homomorphism $f : T(A) \to \mathrm{Fin}(A)$:

$$f\,\emptyset_{T(A)} = \emptyset \qquad f(x \cup_{T(A)} y) = f\,x \cup f\,y$$
$$f\,\{a\}_{T(A)} = \{a\} \qquad a \in_{T(A)} x = a \in f\,x$$

Such a homomorphism is always surjective, and therefore:

- functions on $\mathrm{Fin}(A)$ can be carried over to any implementation of finites sets
- all properties of these functions carry over.

Conclusion and Further Work

- Higher inductive types can be used to capture "data types with additional equalities":
  - HiTs closely represent the specification,
  - the type enforces the programs to obey the additional equalities,
  - the proof principle (elimination rule) takes the equations into account.
- Isomorphic representations allow to transfer recursive functions and proof principles.
- Univalence implies the "Structure Identity Principle": if $A \simeq B$, then $A \equiv B$.
- The "precision" of HiTs can be used e.g. to represent type theory in type theory.
- Another potential application of HoTT: use dihomotopy (directed paths) to model concurrent processes abstractly (Goubault et al.)

Concurrency via dihomotopies

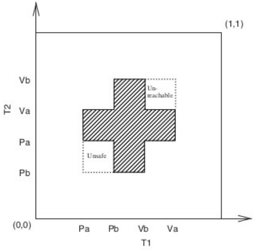
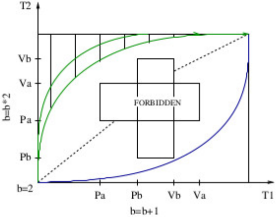


Fig. 2. The Swiss flag example—two processes sharing two resources.



T2 gets a and b before T1 does: b=5!

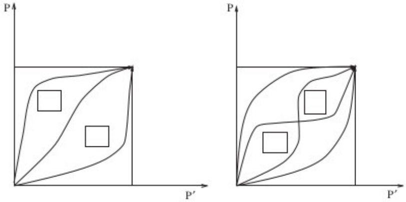


Fig. 6. The two possible relative configurations of holes.