Extensionality in Intensional Type Theory

Or how to compute with funext

Joint work with Nicolas Tabareau, published at POPL22

Part 1 : From Curry-Howard to Martin-Löf

Functional programming for people who don't write actual programs

Functional programming in a nutshell

We can define (higher-order) functions as first-order values

$$\text{double} := \lambda x.\ x + x$$

$$\text{compose} := \lambda f\ g\ x.\ g\ (f\ x)$$

We can apply function to values

$$\text{compose double double}$$

And we can evaluate the programs to get a result (if the computation terminates)

$$\text{double } 2 \quad \longrightarrow \quad 4$$

Type systems in a nutshell

We want to avoid ill formed terms such as double double

We associate a type to every program

$$\text{double} : \mathbb{N} \to \mathbb{N}$$

We may only apply a program to another if the first has the type of the form $A \to B$, and the second has type A.

We can add more types, such as product types (pairs), sum types...

Curry-Howard Correspondence

Parallel between functional programming and constructive propositional logic

Types	Logical formulas
Programs	Proofs
Function type $A \rightarrow B$	Logical implication
Product type $A \times B$	Logical conjunction
Disjoint sum type $A + B$	Logical disjunction

Martin-Löf Type Theory

MLTT goes one step further: It introduces a “type of types” $\mathcal{U}$

(actually a hierarchy $\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 \ldots$)

$$\Gamma \vdash A : \mathcal{U}_i \longrightarrow \Gamma \vdash A \text{ Type}$$

Martin-Löf Type Theory

MLTT goes one step further: It introduces a "type of types" $\mathcal{U}$

(actually a hierarchy $\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 \ldots$)

$$\Gamma \vdash A : \mathcal{U}_i \longrightarrow \Gamma \vdash A \text{ Type}$$

Types may now compute like any regular program.

$$\mathbb{N}_{\leq 2+2} \longrightarrow \mathbb{N}_{\leq 4}$$

Dependent Curry-Howard

As types may now contain variable names, this extends the Curry-Howard correspondence to predicates

$$x : \mathbb{N} \vdash P(x)\ \textsf{Type}$$

Dependent Curry-Howard

As types may now contain variable names, this extends the Curry-Howard correspondence to predicates

$$x : \mathbb{N} \vdash P(x) \ \text{Type}$$

The data-like analogue to predicates is dependent types.
Some instances:

$$x : \mathbb{N} \vdash \mathbb{N}_{\leq x} \ \text{Type}$$

$$x : \mathbb{B} \vdash \text{if } x \text{ then } \top \text{ else } \bot \ \text{Type}$$

Quantifiers

To fully interpret first-order logic, we need quantifiers.

What should be the interpretation of “for all”?

A proof of $\forall n, P(n)$ should associate a proof of P(n) to every integer n.

Quantifiers

To fully interpret first-order logic, we need quantifiers.

What should be the interpretation of “for all”?

A program with type $\forall n, P(n)$ should associate a program with type $P(n)$ to every program with integer type.

Quantifiers

To fully interpret first-order logic, we need quantifiers.

What should be the interpretation of “for all”?

A program with type $\forall n, P(n)$ should associate a program with type $P(n)$ to every program with integer type.

Universal quantifiers should be interpreted as “twisted function types”, whose return type depends on their input value.

$$\prod_{n:\mathbb{N}} P(n)$$

Quantifiers

To fully interpret first-order logic, we need quantifiers.

What should be the interpretation of “there exists”?

A proof of $\exists n, P(n)$ should be an integer n along with a proof of P(n).

Quantifiers

To fully interpret first-order logic, we need quantifiers.

What should be the interpretation of “there exists”?

A program with type $\exists n, P(n)$ should be a program with integer type along with a program with type P(n).

Existential quantifiers should be interpreted as “twisted product types”, whose second projection type depends on their first projection.

$$\sum_{n:\mathbb{N}} P(n)$$

Datatypes

Finally, MLTT provides a powerful scheme for positive datatypes, possibly involving recursion: inductive types.

$$
\begin{aligned}
&\mathsf{List}\ (A : \mathcal{U}_0) : \mathcal{U}_0 := \\
&\mid \mathsf{nil} : \mathsf{List}\ A \\
&\mid \mathsf{cons} : A \to \mathsf{List}\ A \to \mathsf{List}\ A
\end{aligned}
$$

A list of integers is either the empty list, or the data of an integer and a list of integers

Datatypes

Finally, MLTT provides a powerful scheme for positive datatypes, possibly involving recursion: inductive types.

$$
\begin{aligned}
&\mathsf{List}\ (A : \mathcal{U}_0) : \mathcal{U}_0 := \\
&\mid \mathsf{nil} : \mathsf{List}\ A \\
&\mid \mathsf{cons} : A \to \mathsf{List}\ A \to \mathsf{List}\ A
\end{aligned}
$$

A list of integers is either the empty list, or the data of an integer and a list of integers – and the type of lists is the smallest such type.

Datatypes

Then, to define a function on lists (or to inhabit a predicate), one can reason by pattern-matching:

$$\begin{array}{l}\mathsf{length} : \mathsf{List}\ A \to \mathbb{N} := \\ \mathsf{length}\ \mathsf{nil} = 0 \\ \mathsf{length}\ (\mathsf{cons}\ hd\ tl) = 1 + \mathsf{length}\ tl\end{array}$$

Datatypes

Then, to define a function on lists (or to inhabit a predicate), one can reason by pattern-matching:

$$\begin{aligned}
&\mathsf{length} : \mathsf{List}\ A \to \mathbb{N} := \\
&\mathsf{length}\ \mathsf{nil} = 0 \\
&\mathsf{length}\ (\mathsf{cons}\ hd\ tl) = 1 + \mathsf{length}\ tl
\end{aligned}$$

Of course, there are constraints on pattern matching and inductive definitions to avoid loops and paradoxes.

Datatypes

Inductive definitions are versatile enough to define equality:

$$\mathsf{Eq}\ (A : \mathcal{U}_0)\ (a : A) : A \to \mathcal{U}_0 :=$$
$$\mid \mathsf{refl} : \mathsf{Eq}\ A\ a\ a$$

Datatypes

Inductive definitions are versatile enough to define equality:

$$\begin{aligned}&\mathsf{Eq}\ (A : \mathcal{U}_0)\ (a : A) : A \to \mathcal{U}_0 := \\ &\mid \mathsf{refl} : \mathsf{Eq}\ A\ a\ a\end{aligned}$$

We can then prove Leibniz' property by pattern-matching

$$\begin{aligned}&\mathsf{transp}\ (P : A \to \mathcal{U}_0)\ (t : P\ a)\ (e : \mathsf{Eq}\ A\ a\ b) : P\ b \\ &\mathsf{transp}\ P\ t\ \mathsf{refl} = t\end{aligned}$$

Putting the "constructive" in constructive mathematics

With all these ingredients, MLTT is

- a fully-fledged constructive framework – as was used by Bishop
- a versatile programming language

Putting the “constructive” in constructive mathematics

With all these ingredients, MLTT is

- a fully-fledged constructive framework – as was used by Bishop
- a versatile programming language

You can use it for most of mathematics, from number theory and combinatorics to constructive analysis.

Putting the "constructive" in constructive mathematics

With all these ingredients, MLTT is

- a fully-fledged constructive framework – as was used by Bishop
- a versatile programming language

You can use it for most of mathematics, from number theory and combinatorics to constructive analysis.

Moreover, everything you define in MLTT is a program that you can evaluate → computational content for proofs

A bit of meta-theory

Normalization theorem : every well-typed program in MLTT eventually terminates on a canonical normal form.

A bit of meta-theory

Normalization theorem : every well-typed program in MLTT eventually terminates on a canonical normal form.

Every integer eventually computes to an actual integer

Every proof of $\exists n, P(n)$ provides a concrete integer n along with a proof of P(n)

Every definable function is computable

Every type computes to a useable, type-like normal form

A bit of meta-theory

Normalization theorem : every well-typed program in MLTT eventually terminates on a canonical normal form.

→ Typing is decidable

A great foundation for proof assistants (Agda, Coq, Lean…)

Part 2 : Intensionality versus Extensionality

The Inductive Equality is not so Useful

```
Inductive eq (A : Type) (a : A) : A -> Type :=
| eq_refl : eq A a a
```

The equality supplied by MLTT encodes equality of programs, not equality of behaviours.

> In the empty context, the only equality proof is **`eq_refl`**, which means the terms have to be convertible.

> Equality in the empty context is decidable.

> No hope for function extensionality or quotient types.

The Inductive Equality is not so Useful

```
Inductive eq (A : Type) (a : A) : A -> Type :=
| eq_refl : eq A a a
```

The equality supplied by MLTT encodes equality of programs, not equality of behaviours.

No way to prove $\lambda x.x+1 = \lambda x.1+x$ (same functions, different programs)

No way to prove that two equivalent propositions are equal

Equality on coinductive types is not interesting

You can't have your cake and eat it too

Possible workarounds:

> Use axioms : just postulate function extensionality, etc

You can't have your cake and eat it too

Possible workarounds:

> Use axioms : just postulate function extensionality, etc

> Use setoids : equip every type with an equivalence relation, and ensure that functions preserve them.

You can't have your cake and eat it too

Possible workarounds:

> Use axioms : just postulate function extensionality, etc

> Use setoids : equip every type with an equivalence relation, and ensure that functions preserve them.

> Add the reflection rule for equality (extensional type theory)

You can't have your cake and eat it too

Possible workarounds:

> Use axioms : just postulate function extensionality, etc

> Use setoids : equip every type with an equivalence relation, and ensure that functions preserve them.

> Add the reflection rule for equality (extensional type theory)

> Use cubical type theory

Observational Type Theory

Altenkirch and McBride designed OTT to fix the inductive equality.

Main insight: instead of being an inductive data structure, equality is defined by recursion on the types

$$S\,(S\,0) \sim_{\mathbb{N}} S\,(S\,0)$$

Altenkirch et al, *Towards observational type theory*, 2006
Altenkirch et al, *Observational Equality, Now!*, 2007

Observational Type Theory

Altenkirch and McBride designed OTT to fix the inductive equality.

Main insight: instead of being an inductive data structure, equality is defined by recursion on the types

$$S\,(S\,0) \sim_{\boxed{\mathbb{N}}} S\,(S\,0)$$

Altenkirch et al, *Towards observational type theory*, 2006
Altenkirch et al, *Observational Equality, Now!*, 2007

Observational Type Theory

Altenkirch and McBride designed OTT to fix the inductive equality.

Main insight: instead of being an inductive data structure, equality is defined by recursion on the types

$$S\,(S\,0) \sim_{\mathbb{N}} S\,(S\,0) \quad \longrightarrow \quad S\,0 \sim_{\mathbb{N}} S\,0$$

Altenkirch et al, *Towards observational type theory*, 2006
Altenkirch et al, *Observational Equality, Now!*, 2007

Observational Type Theory

Altenkirch and McBride designed OTT to fix the inductive equality.

Main insight: instead of being an inductive data structure, equality is defined by recursion on the types

$$S\,(S\,0) \sim_{\mathbb{N}} S\,(S\,0) \longrightarrow S\,0 \sim_{\mathbb{N}} S\,0$$

$$\longrightarrow 0 \sim_{\mathbb{N}} 0$$

Altenkirch et al, *Towards observational type theory*, 2006
Altenkirch et al, *Observational Equality, Now!*, 2007

Observational Type Theory

Altenkirch and McBride designed OTT to fix the inductive equality.

Main insight: instead of being an inductive data structure, equality is defined by recursion on the types

$$S\,(S\,0) \sim_{\mathbb{N}} S\,(S\,0) \longrightarrow S\,0 \sim_{\mathbb{N}} S\,0$$

$$\longrightarrow 0 \sim_{\mathbb{N}} 0$$

$$\longrightarrow \top$$

Altenkirch et al, *Towards observational type theory*, 2006
Altenkirch et al, *Observational Equality, Now!*, 2007

Observational Type Theory

Altenkirch and McBride designed OTT to fix the inductive equality.

Main insight: instead of being an inductive data structure, equality is defined by recursion on the types

$$f \sim_{A \to B} g$$

Altenkirch et al, *Towards observational type theory*, 2006
Altenkirch et al, *Observational Equality, Now!*, 2007

Observational Type Theory

Altenkirch and McBride designed OTT to fix the inductive equality.

Main insight: instead of being an inductive data structure, equality is defined by recursion on the types

$$f \sim_{\boxed{A \to B}} g$$

Altenkirch et al, *Towards observational type theory,* 2006
Altenkirch et al, *Observational Equality, Now!*, 2007

Observational Type Theory

Altenkirch and McBride designed OTT to fix the inductive equality.

Main insight: instead of being an inductive data structure, equality is defined by recursion on the types

$$f \sim_{A \to B} g \quad \longrightarrow \quad \prod_{x:A} f\,x \sim_B g\,x$$

Altenkirch et al, *Towards observational type theory,* 2006
Altenkirch et al, *Observational Equality, Now!*, 2007

Part 3

TT^{obs} : *Yet Another Flavor of OTT*

Eliminating observational equality

$$\frac{P : \mathbb{N} \to \mathsf{Type} \qquad n, m : \mathbb{N} \qquad e : m \sim_{\mathbb{N}} n \qquad t : P\, m}{P\, n}$$

Eliminating observational equality

$$\frac{P : \mathbb{N} \to \mathsf{Type} \qquad n, m : \mathbb{N} \qquad e : m \sim_{\mathbb{N}} n \qquad t : P\, m}{P\, n}$$

$$\frac{A, B : \mathsf{Type} \qquad e : A \sim_{\mathsf{Type}} B \qquad x : A}{\mathsf{cast}(A, B, e, x) : B}$$

Eliminating observational equality

$$\frac{P : \mathbb{N} \to \mathsf{Type} \qquad n, m : \mathbb{N} \qquad e : m \sim_{\mathbb{N}} n \qquad t : P\, m}{\mathsf{cast}(P\, m, P\, n, \mathsf{ap}_f\, e, t) : P\, n}$$

$$\frac{A, B : \mathsf{Type} \qquad e : A \sim_{\mathsf{Type}} B \qquad x : A}{\mathsf{cast}(A, B, e, x) : B}$$

Eliminating observational equality

Cast computes by recursion on types and terms:

$$\mathsf{cast}(A \to B, A' \to B', e, f)$$

Eliminating observational equality

Cast computes by recursion on types and terms:

$$\mathsf{cast}(A \to B, A' \to B', e, f)$$

compatible

Eliminating observational equality

Cast computes by recursion on types and terms:

$$\mathsf{cast}(A \to B, A' \to B', e, f) \longrightarrow$$
$$\lambda(x : A').\, \mathsf{cast}(B, B', \pi_2\, e, f\, \mathsf{cast}(A', A, \pi_1\, e^{-1}, x))$$

Eliminating observational equality

Cast computes by recursion on types and terms:

$$\mathsf{cast}(A \to B, A' \to B', e, f) \longrightarrow$$

$$\lambda(x : A').\,\mathsf{cast}(B, B', \pi_2\, e, f\,\mathsf{cast}(A', A, \pi_1\, e^{-1}, x))$$

$$e : (A \to B) \sim_{\mathsf{Type}} (A' \to B')$$

Eliminating observational equality

Cast computes by recursion on types and terms:

$$\mathsf{cast}(A \to B, A' \to B', e, f) \longrightarrow$$

$$\lambda(x : A').\,\mathsf{cast}(B, B', \pi_2\, e, f\,\mathsf{cast}(A', A, \pi_1\, e^{-1}, x))$$

$$e : (A \to B) \sim_{\mathsf{Type}} (A' \to B') \longrightarrow$$

$$e : (A \sim_{\mathsf{Type}} A') \times (B \sim_{\mathsf{Type}} B')$$

Definitional Proof-Irrelevance

How do we prove reflexivity or transitivity of the equality with cast?

We can’t!

Definitional Proof-Irrelevance

How do we prove reflexivity or transitivity of the equality with cast?

We can't!

Second insight of OTT: we need a layer of *proof-irrelevant* types that will contain the observational equality.

Now any two proofs of the same equality are undistinguishable
→ definitional K/UIP

Technical point: J on refl

So, can we use pattern-matching on the observational equality, as with the inductive equality?

Not quite: since it is proof-irrelevant, one cannot analyze the equality witness.

Technical point: J on refl

So, can we use pattern-matching on the observational equality, as with the inductive equality?

Not quite: since it is proof-irrelevant, one cannot analyze the equality witness.

No worries though: with cast and proof-irrelevance, we can define J – but it won't compute on reflexivity without adding a controversial rule:

$$\mathsf{cast}(X, Y, e, t) \xrightarrow[\text{X and Y convertible}]{} t$$

Inductive Types

Regular inductive types work just fine.

However, indexed inductive types need a new constructor to handle cast values, which might not have a canonical form.

Inductive Types

Regular inductive types work just fine.

However, indexed inductive types need a new constructor to handle cast values, which might not have a canonical form.

For instance, the inductive equality becomes:

```
Inductive eq (A : Type) (a : A) : A -> Type :=
| eq_refl : eq A a a
| eq_cast : forall b, a ~A b -> eq A a b
```

This is the OTT analogue to Swan's encoding of equality types.
It implies that canonicity is weakened for indexed inductive types.

But wait, there's more!

TT^{obs} is a proper extension of MLTT (all MLTT proofs remain valid!) that adds extensionality principles:

> Function extensionality

> Equality of coinductives is bisimulation

> Proposition extensionality for the proof-irrelevant propositions

> Axiom K/UIP (no univalence!)

But wait, there's more!

But we can add more:

> Irrelevant squash types and relevant box types

> Subset types, such as $\{n : \mathbb{N} \mid n \leq 10\}$

> Quotients of a type by a *proof-irrelevant* equivalence relation

Quotient types

$$\frac{A : \mathsf{Type} \qquad R : A \to A \to \mathsf{Prop} \qquad \mathsf{equiv}(R)}{A/R : \mathsf{Type}}$$

$$\pi_{A/R} : A \to A/R$$

$$\pi_{A/R}\, x \sim_{A/R} \pi_{A/R}\, y \longrightarrow R\ x\ y$$

Meta-Theory

So far, we have presented a re-cast of OTT as an extension of MLTT.

Main contribution: a proper development of the meta-theory of TT^{obs}

> Consistency

> Normalization

> Canonicity

> Decidability of type-checking.

Consistency

Consistency can be proved by constructing a model.

This can be done in a constructive set theory (or a type theory) that is strong enough to do induction-recursion, or plain ZF set theory.

From there, we obtain that

> there are no inhabitants of $\perp$ in the empty context

> there are no proofs of anti-diagonal equalities between types

Normalization and canonicity

Normalization, canonicity and decidability of conversion can be proved using logical relations.

We used the induction-recursion based framework of Abel, Öhman and Vezzosi to formally prove these three properties in Agda.

Abel et al, *Decidability of conversion for type theory in type theory*

Normalization and canonicity

Interesting points of the proof:

> No computation in Prop

> This makes canonicity reliant on consistency

> Reducibility of cast relies on having an inductive description of the inhabitants of Type. Incompatible with reducibility candidates?

Semantics

Observational equality computes on types

But this doesn't mean semantical universes are not restricted to syntactical types

TT^{obs} enjoys a wide range of models, such as sheaf toposes. It could be a very good language for toposes once extended with proof-irrelevant impredicativity.

Implementation is not too Difficult

All in all, we only need three ingredients:

> Definitionally proof-irrelevant types

Already featured in Coq, Agda and Lean

> Two primitives cast and ~, along with rewriting rules

> A new constructor for indexed inductive types

We used Jesper Cockx's rewrite rules to implement TT^{obs} in Agda.

There are plans to add it as an option to the Coq kernel

Thank you