Inductive types in the setoid model of type theory

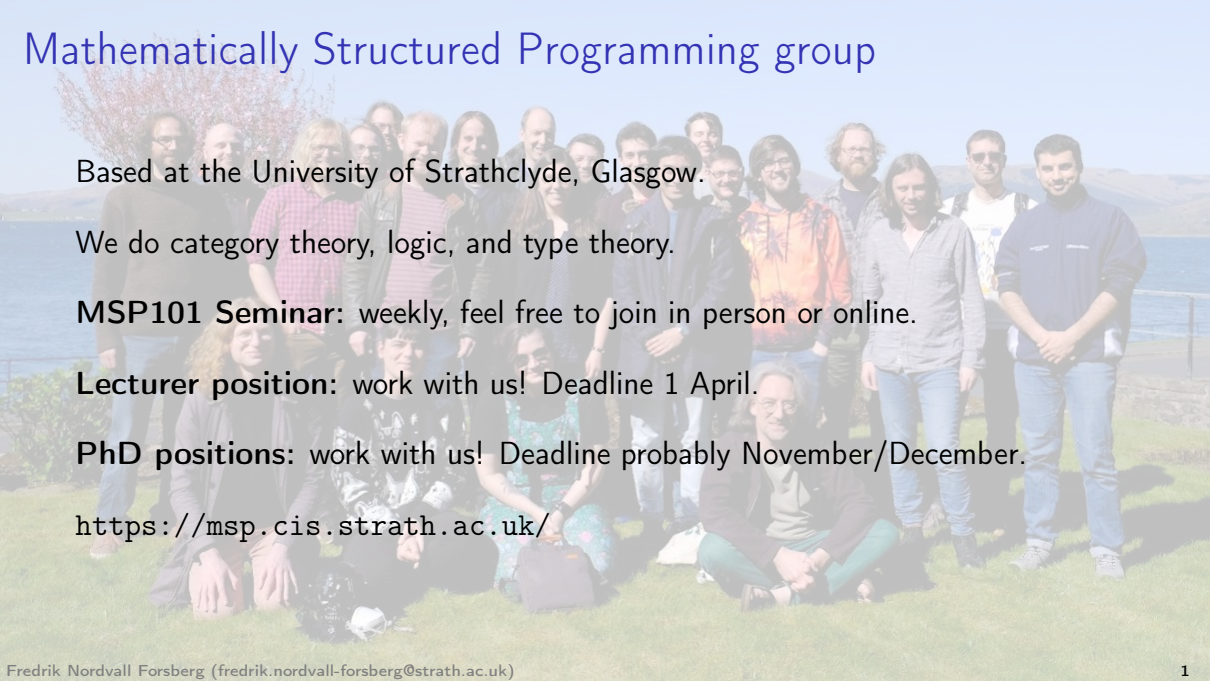
Fredrik Nordvall Forsberg
University of Strathclyde

TRIPLE 2026, Edinburgh
29 March 2026

Joint work with Liang-Ting Chen and Shu-Hung You.

These slides: `https://fredriknf.com/triple2026.pdf`.

Mathematically Structured Programming group



Based at the University of Strathclyde, Glasgow.

We do category theory, logic, and type theory.

MSP101 Seminar: weekly, feel free to join in person or online.

Lecturer position: work with us! Deadline 1 April.

PhD positions: work with us! Deadline probably November/December.

`https://msp.cis.strath.ac.uk/`

Dependently typed programming

Type theory is a functional programming language with a very expressive type system.

Foundation of proof assistants such as Rocq, Lean, and programming languages such as Agda, Idris.

Power comes from data types such as natural numbers, lists, trees, ...

Propositions-as-types paradigm: can encode logic into the type system.

Correspondingly, we must ensure that the type corresponding to $\bot$ really is empty.

More expressive data types

In practice we want more expressive data types:

- Indexed inductive types
  - Vectors, well-scoped syntax, invariants, ...
- Inductive-recursive definitions
  - Universes, recursive invariants, ...
- Quotients
  - Cauchy reals, non-free structures, ...
- All of the above (e.g. quotient-inductive-recursive types)
  - non-free universes, well-scoped syntax of dependent types, ...

How do we know that these make sense?

It is easy to criticise

To show something inconsistent, just prove $\bot$.

A single witness is enough.

How to prove that something is consistent?

A priori, we need to show something about all possible proofs/terms.

Das Model

Answer from logic: can use *models* to show unprovability.

Soundness: Provable things are true in the model.

So suffices to construct a single model interpreting the concept of interest, and where $\bot$ is not true.

What is a model of type theory?

There are many variations on the notion of model of type theory.

In general: some kind of mathematical object, with additional structure and properties.

Framework provides interpretation into the model, and guarantees at least soundness (probably also completeness).

Choice here: the framework of Categories with Families.

Examples of models

Topological model Validates "all functions are continuous"

Groupoid model Validates "equality types may be non-trivial"

Logical relations, gluing Gives semantic proofs of normalisation, canonicity

Setoid model Validates "equality can be customised"

Examples of models

Topological model Validates "all functions are continuous"

Groupoid model Validates "equality types may be non-trivial"

Logical relations, gluing Gives semantic proofs of normalisation, canonicity

Setoid model Validates "equality can be customised" ← we will focus on this one

Setoids

A *setoid* $(X, \approx, r, s, t)$ is given by

- A type X
- A relation $\approx : X \to X \to \text{Prop}$
- A proof $r : (\forall x : X).\ x \approx x$
- A proof $s : (\forall x, y : X).\ x \approx y \to y \approx x$
- A proof $t : (\forall x, yz : X).\ x \approx y \to y \approx z \to x \approx z$

A *morphism of setoids* $(X, \approx, r, s, t) \to (X', \approx', r', s', t')$ is given by a function $f : X \to X'$ such that if $x_0 \approx x_1$ then $f(x_0) \approx' f(x_1)$.

See Hofmann [1995] and Altenkirch [1999].

Setoid type formers

Given setoids $(X, \approx_X)$ and $(Y, \approx_Y)$:

Setoid type formers

Given setoids $(X, \approx_X)$ and $(Y, \approx_Y)$:

Products $(X \times Y, \approx)$ where

$$(x_0, y_0) \approx (x_1, y_1) \Leftrightarrow x_0 \approx_X x_1 \text{ and } y_0 \approx_Y y_1.$$

Setoid type formers

Given setoids $(X, \approx_X)$ and $(Y, \approx_Y)$:

Products $(X \times Y, \approx)$ where

$$(x_0, y_0) \approx (x_1, y_1) \Leftrightarrow x_0 \approx_X x_1 \text{ and } y_0 \approx_Y y_1.$$

Function space $(\{f : X \to Y \mid x_0 \approx_X x_1 \implies f(x_0) \approx_Y f(x_1)\}, \approx)$ where

$$f \approx g \Leftrightarrow \big(x_0 \approx_X x_1 \implies f(x_0) \approx_Y g(x_1)\big)$$

Setoid type formers

Given setoids $(X, \approx_X)$ and $(Y, \approx_Y)$:

Products $(X \times Y, \approx)$ where

$$(x_0, y_0) \approx (x_1, y_1) \Leftrightarrow x_0 \approx_X x_1 \text{ and } y_0 \approx_Y y_1.$$

Function space $(\{f : X \to Y \mid x_0 \approx_X x_1 \implies f(x_0) \approx_Y f(x_1)\}, \approx)$ where

$$f \approx g \Leftrightarrow \left(x_0 \approx_X x_1 \implies f(x_0) \approx_Y g(x_1)\right)$$

Natural numbers object $(\mathbb{N}, \approx)$ where $\approx$ is defined inductively by

$$\text{zero} \approx \text{zero}$$

$$n_0 \approx n_1 \implies \text{suc } n_0 \approx \text{suc } n_1$$

Categories with families

A category with families [Dybjer 1995] is given by

- A way to interpret contexts
- A way to interpret substitutions between contexts
- A way to interpret types (for each context)
- An operation for substitutions acting on types
- A way to interpret terms (for each context and type)
- An operation for substitutions acting on terms

Categories with families

A category with families [Dybjer 1995] is given by

- A way to interpret contexts
- A way to interpret substitutions between contexts

} category

- A way to interpret types (for each context)
- An operation for substitutions acting on types
- A way to interpret terms (for each context and type)
- An operation for substitutions acting on terms

} functor to families of sets

Categories with families

A category with families [Dybjer 1995] is given by

- A way to interpret contexts
- A way to interpret substitutions between contexts

} category

- A way to interpret types (for each context)
- An operation for substitutions acting on types
- A way to interpret terms (for each context and type)
- An operation for substitutions acting on terms

} functor to families of sets

Theorem Each category with families give a sound interpretation of type theory.

Interpretations of contexts and substitutions

We require a set Ctx, and for each $\Gamma, \Delta \in \mathsf{Ctx}$, a set $\mathsf{Sub}(\Gamma, \Delta)$ with

- An "identity" $\mathsf{id} \in \mathsf{Sub}(\Gamma, \Gamma)$ for each $\Gamma \in \mathsf{Ctx}$;
- A "composition" $\circ : \mathsf{Sub}(\Delta, \Xi) \to \mathsf{Sub}(\Gamma, \Delta) \to \mathsf{Sub}(\Gamma, \Xi)$;
- A "terminal object" $1 \in \mathsf{Ctx}$;
- With a unique $! \in \mathsf{Sub}(\Gamma, 1)$ for each Γ;

such that $\mathsf{id} \circ f = f$, $f \circ \mathsf{id} = f$, $f \circ (g \circ h) = (f \circ g) \circ h$.

Example We can take $\mathsf{Ctx} = \mathsf{Setoid}$, and $\mathsf{Sub}(\Gamma, \Delta) = \mathsf{SetoidMorphism}(\Gamma, \Delta)$. The terminal object is the setoid $(\{*\}, \top)$.

Interpretations of contexts and substitutions

We require a set Ctx, and for each $\Gamma, \Delta \in \mathsf{Ctx}$, a set $\mathsf{Sub}(\Gamma, \Delta)$ with

- An "identity" $\mathsf{id} \in \mathsf{Sub}(\Gamma, \Gamma)$ for each $\Gamma \in \mathsf{Ctx}$;
- A "composition" $\circ : \mathsf{Sub}(\Delta, \Xi) \to \mathsf{Sub}(\Gamma, \Delta) \to \mathsf{Sub}(\Gamma, \Xi)$;
- A "terminal object" $1 \in \mathsf{Ctx}$;
- With a unique $! \in \mathsf{Sub}(\Gamma, 1)$ for each Γ;

such that $\mathsf{id} \circ f = f$, $f \circ \mathsf{id} = f$, $f \circ (g \circ h) = (f \circ g) \circ h$.

Example We can take $\mathsf{Ctx} = \mathsf{Setoid}$, and $\mathsf{Sub}(\Gamma, \Delta) = \mathsf{SetoidMorphism}(\Gamma, \Delta)$. The terminal object is the setoid $(\{*\}, \top)$.

After introducing the interpretations of types, we will also require the interpretation of a "context extension" operation.

Interpretations of types

For each $\Gamma \in \mathsf{Ctx}$, we require a set $\mathsf{Ty}(\Gamma)$, and for each $\sigma : \mathsf{Sub}(\Gamma, \Delta)$, a function $\_[\sigma] : \mathsf{Ty}(\Delta) \to \mathsf{Ty}(\Gamma)$, such that

$$A[\mathsf{id}] = A \qquad\qquad A[\tau \circ \sigma] = A[\tau][\sigma]$$

Example Take $\mathsf{Ty}(\Gamma, \approx_\Gamma)$ = "families of setoids over $(\Gamma, \approx_\Gamma)$", consisting of

$$\begin{aligned}
A &: \Gamma \to \mathsf{Set} \\
_ \vdash _ \approx _ &: \gamma_0 \approx_\Gamma \gamma_1 \to A(\gamma_0) \to A(\gamma_1) \to \mathsf{Prop} \\
\mathsf{refl}^* &: \mathsf{refl} \vdash a_0 \approx a_0 \\
\mathsf{sym}^* &: \gamma_{01} \vdash a_0 \approx a_1 \to \mathsf{sym}\,\gamma_{01} \vdash a_1 \approx a_0 \\
\mathsf{trans}^* &: \gamma_{01} \vdash a_0 \approx a_1 \to \gamma_{12} \vdash a_1 \approx a_2 \to \mathsf{trans}\,\gamma_{01}\,\gamma_{12} \vdash a_0 \approx a_2 \\
\mathsf{coe} &: \gamma_0 \approx_\Gamma \gamma_1 \to A(\gamma_0) \to A(\gamma_1) \\
\mathsf{coh} &: (\gamma_{01} : \gamma_0 \approx_\Gamma \gamma_1) \to (a_0 : A(\gamma_0)) \to \gamma_{01} \vdash a_0 \approx \mathsf{coe}\,\gamma_{01}\,a_0
\end{aligned}$$

Interpretations of terms

For each $\Gamma \in \mathsf{Ctx}$ and $A \in \mathsf{Ty}(\Gamma)$, we require a set $\mathsf{Tm}(\Gamma, A)$.

For each $\sigma \in \mathsf{Sub}(\Gamma, \Delta)$, we require $\_[\sigma] : \mathsf{Tm}(\Delta, A) \to \mathsf{Tm}(\Gamma, A[\sigma])$, such that

$$t[\mathsf{id}] = t \qquad\qquad t[\tau \circ \sigma] = t[\tau][\sigma]$$

Example Take
$\mathsf{Tm}((\Gamma, \approx_\Gamma), A) = \{t : (\gamma : \Gamma) \to A(\gamma) \mid (\gamma_{01} : \gamma_0 \approx_\Gamma \gamma_1) \to \gamma_{01} \vdash t\,\gamma_0 \approx t\,\gamma_1\}$

Interpretation of context extensions

Finally, we require an operation

$$\cdot : (\Gamma : \mathsf{Ctx}) \to \mathsf{Ty}(\Gamma) \to \mathsf{Ctx}$$

with

- $\mathsf{wk} \in \mathsf{Sub}(\Gamma \cdot A, \Gamma)$
- $\mathsf{vz} \in \mathsf{Tm}(\Gamma \cdot A, A[\mathsf{wk}])$
- and a universal property characterising

$$\mathsf{Sub}(\Delta, \Gamma \cdot A) \cong \{(\sigma \in \mathsf{Sub}(\Delta, \Gamma), t \in \mathsf{Tm}(\Delta, A[\sigma])\}$$

- satisfying some equations.

Example We can take $(\Gamma, \approx_\Gamma) \cdot A = ((\Sigma \gamma : \Gamma).A(\gamma), \approx_{\Gamma \cdot A})$ where

$$(\gamma_0, a_0) \approx_{\Gamma \cdot A} (\gamma_1, a_1) \Leftrightarrow (\exists \gamma_{01} : \gamma_0 \approx_\Gamma \gamma_1).\ \gamma_{01} \vdash a_0 \approx a_1$$

Interpreting type formers

To interpret type formers, we check that we can implement "semantic" version of the rules for the type former. For example:

A Category with Families supports dependent function types if

Interpreting type formers

To interpret type formers, we check that we can implement "semantic" version of the rules for the type former. For example:

A Category with Families supports dependent function types if

- For $A \in \mathrm{Ty}(\Gamma)$ and $B \in Ty(\Gamma \cdot A)$ there is $\Pi(A, B) \in \mathrm{Ty}(\Gamma)$;

Interpreting type formers

To interpret type formers, we check that we can implement "semantic" version of the rules for the type former. For example:

A Category with Families supports dependent function types if

- For $A \in \mathsf{Ty}(\Gamma)$ and $B \in Ty(\Gamma \cdot A)$ there is $\Pi(A, B) \in \mathsf{Ty}(\Gamma)$;
- For $t \in \mathsf{Tm}(\Gamma \cdot A, B)$ there is $\lambda(t) \in \mathsf{Tm}(\Gamma, \Pi(A, B))$;

Interpreting type formers

To interpret type formers, we check that we can implement "semantic" version of the rules for the type former. For example:

A Category with Families supports dependent function types if

- For $A \in \mathsf{Ty}(\Gamma)$ and $B \in Ty(\Gamma \cdot A)$ there is $\Pi(A, B) \in \mathsf{Ty}(\Gamma)$;
- For $t \in \mathsf{Tm}(\Gamma \cdot A, B)$ there is $\lambda(t) \in \mathsf{Tm}(\Gamma, \Pi(A, B))$;
- For $f \in \mathsf{Tm}(\Gamma, \Pi(A, B))$ and $a \in \mathsf{Tm}(\Gamma, A)$ there is $\mathsf{app}(\mathrm{f}, \mathrm{a}) \in \mathsf{Tm}(\Gamma, B\langle a \rangle)$;

where $\langle a \rangle \in \mathsf{Sub}(\Gamma, \Gamma \cdot A)$ is derived from the universal property of $\cdot$.

Interpreting type formers

To interpret type formers, we check that we can implement "semantic" version of the rules for the type former. For example:

A Category with Families supports dependent function types if

- For $A \in \mathsf{Ty}(\Gamma)$ and $B \in Ty(\Gamma \cdot A)$ there is $\Pi(A, B) \in \mathsf{Ty}(\Gamma)$;
- For $t \in \mathsf{Tm}(\Gamma \cdot A, B)$ there is $\lambda(t) \in \mathsf{Tm}(\Gamma, \Pi(A, B))$;
- For $f \in \mathsf{Tm}(\Gamma, \Pi(A, B))$ and $a \in \mathsf{Tm}(\Gamma, A)$ there is $\mathrm{app}(\mathrm{f}, \mathrm{a}) \in \mathsf{Tm}(\Gamma, B\langle a \rangle)$;
- such that $\mathrm{app}(\lambda(t), a) = t\langle a \rangle$ and $\lambda(\mathrm{app}(t[\mathrm{wk}], \mathrm{vz}) = t$ (and substitution equations).

where $\langle a \rangle \in \mathsf{Sub}(\Gamma, \Gamma \cdot A)$ is derived from the universal property of $\cdot$.

Example: The setoid model supports Π-types, Σ-types, Id-types, the empty type, Booleans, quotient types, function extensionality, ...

Taking stock

Categories with families are models with a sound interpretation of basic type theory (e.g. if $\Gamma \vdash t : A$ then we can construct $[\![t]\!] \in \mathsf{Tm}([\![\Gamma]\!], [\![A]\!])$).

We have a particular model in mind: the setoid model.

It should be useful for showing that various inductive types can be combined with quotients in a consistent way.

But what do we mean by an inductive type?

Inductive types, informally

An inductive type X looks like

$$
\begin{aligned}
&\text{data } X \ : \ \text{Type where} \\
&\quad c_0 : X \\
&\quad c_1 : (a : A) \to X \to B(a) \to X \to X \\
&\quad \vdots \\
&\quad c_n : (N \to X) \to X
\end{aligned}
$$

Inductive types, informally

An inductive type X looks like

$$
\begin{aligned}
&\textcolor{orange}{\text{data}}\ X\ :\ \textcolor{blue}{\text{Type}}\ \textcolor{orange}{\text{where}}\\
&\quad c_0 : \mathbb{1} \to X\\
&\quad c_1 : (a : A) \to X \to B(a) \to X \to X\\
&\quad\ \vdots\\
&\quad c_n : (N \to X) \to X
\end{aligned}
$$

Inductive types, informally

An inductive type X looks like

$$
\begin{aligned}
&\textsf{data } X \ : \ \textsf{Type where} \\
&\quad c_0 : \mathbb{1} \to X \\
&\quad c_1 : (a : A) \times X \times B(a) \times X \to X \\
&\quad \vdots \\
&\quad c_n : (N \to X) \to X
\end{aligned}
$$

Inductive types, informally

An inductive type X looks like

$$
\begin{aligned}
&\textsf{data } X \ : \ \textsf{Type where} \\
&\quad c_0 : \mathbb{1} \to X \\
&\quad c_1 : (a : A) \times (\mathbb{1} \to X) \times B(a) \times (\mathbb{1} \to X) \to X \\
&\quad \vdots \\
&\quad c_n : (N \to X) \to X
\end{aligned}
$$

Inductive types, informally

An inductive type X looks like

$$\begin{aligned}&\textsf{data } X \ : \ \textsf{Type where}\\ &\quad c_0 : \mathbb{1} \to X\\ &\quad c_1 : (a : A) \times (\mathbb{1} \to X) \times B(a) \times (\mathbb{1} \to X) \to X\\ &\quad \vdots\\ &\quad c_n : (N \to X) \to X\end{aligned}$$

or equivalently

$$\begin{aligned}&\textsf{data } X \ : \ \textsf{Type where}\\ &\quad c : \mathbb{1} + \big((a : A) \times (\mathbb{1} \to X) \times B(a) \times (\mathbb{1} \to X)\big) + \ldots + (N \to X) \to X\end{aligned}$$

Inductive types, formally

Inductive types in a Category with families are described by the following inductive definition:

$$
\begin{aligned}
&\mathsf{data}\ \mathsf{Desc}\ (\Gamma : \mathsf{Ctx})\ : \mathsf{Type}\ \mathsf{where}\\
&\quad \mathsf{end} : \mathsf{Desc}\ \Gamma\\
&\quad \mathsf{non} : (A : \mathsf{Ty}\ \Gamma) \to \mathsf{Desc}\ (\Gamma \cdot A) \to \mathsf{Desc}\ \Gamma\\
&\quad \mathsf{arg} : (A : \mathsf{Ty}\ \Gamma) \to \mathsf{Desc}\ \Gamma \to \mathsf{Desc}\ \Gamma\\
&\quad \mathsf{alt} : \mathsf{Desc}\ \Gamma \to \mathsf{Desc}\ \Gamma \to \mathsf{Desc}\ \Gamma
\end{aligned}
$$

This is similar to a schema for inductive-recursive definitions by Dybjer and Setzer [1999].

Constructor arguments and other rules

If the CwF supports a unit type, Σ-types, function types, and sum types, we can compute what argument each description represents:

$$
\begin{aligned}
&\mathsf{Arg} : (D : \mathsf{Desc}\ \Gamma) \to (X : \mathsf{Ty}\ \Gamma) \to \mathsf{Ty}\ \Gamma \\
&\mathsf{Arg}\ \mathsf{end}\ X = \mathbb{1} \\
&\mathsf{Arg}\ (\mathsf{non}\ \mathsf{A}\ \mathsf{D})\ X = \Sigma(A, \mathsf{Arg}\ D\ (X\ [\mathsf{wk}])) \\
&\mathsf{Arg}\ (\mathsf{arg}\ \mathsf{A}\ \mathsf{D})\ X = (A \Rightarrow X) \times \mathsf{Arg}\ D\ X \\
&\mathsf{Arg}\ (\mathsf{alt}\ \mathsf{D}\ \mathsf{E})\ X = \mathsf{Arg}\ D\ X + \mathsf{Arg}\ E\ X
\end{aligned}
$$

Goal: Given $D : \mathsf{Desc}\,1$, we want to have $\mu D \in \mathsf{Ty}(1)$ together with $c \in \mathsf{Tm}(1 \cdot \mathsf{Arg}\ D\ \mu D, \mu D)$, and implementation of the elimination rules.

Inductive types in the setoid model

For setoids, our plan is to first construct the underlying set inductively (in the metatheory), and then its equivalence relation.

Unfortunately, this does not work directly: infinitary constructors use a function space, which should refer to the equivalence relation we have not yet constructed.

Following Emmenegger [2021] for the special case of W-types, we over-approximate and later single out the well-behaved elements.

We define

$$\begin{array}{c}\mathsf{data}\ \mu_0(D : \mathsf{Desc}\ 1) : \mathsf{Type}\ \mathsf{where}\\ \mathsf{c} : \mathsf{Arg}_0\ D\ (\mu_0\ D) \to \mu_0\ D\end{array}$$

and a predicate $\mathsf{isExtensional} : \mu_0\ D \to \mathsf{Prop}$ by induction on D.

Interpreting the inductive type

We then let

$$\mu D = (\{x : \mu_0 D \mid \mathsf{isExtensional}\ x\}, \approx, \ldots)$$

where $\approx$ is inductively defined to make c a setoid morphism, plus the reflexive-transitive closure.

To define the eliminator, we first have to construct the type IH D X P of induction hypothesis for a given motive $P \in \mathsf{Ty}(\Gamma \cdot X)$. We then reuse the eliminator in the metatheory.

Validating the rules

Given $D : \mathsf{Desc}\,1$, we construct the following in the setoid model:

$$\mu D \in \mathsf{Ty}(1) \qquad\qquad c \in \mathsf{Tm}(1 \cdot \mathsf{Arg}\ D\ \mu D, \mu D)$$

$$\begin{aligned}\mathsf{elim} : (P : \mathsf{Ty}\ (\Gamma \cdot \mu\ D)) \\ \to \mathsf{Tm}(\Gamma \cdot \mathsf{Arg}\ D\ (\mu\ D) \cdot \mathsf{IH}\ D\ (\mu\ D), P[\,\mathsf{wk} \cdot \mathsf{con}\,]) \\ \to \mathsf{Tm}(\Gamma \cdot \mu\ D, P)\end{aligned}$$

and prove the substitution and computation rules.

This shows that inductive types are consistent with e.g. function extensionality (no surprise!).

Next steps

We have done indexed inductive types already. The development is ripe for extension to more advanced data types:

- Quotient-inductive types
- Inductive-recursive types
- Quotient-inductive-recursive types

The foundational status of the latter is currently unknown.

Summary

An example of what research in type theory might look like:

- Understand existing concepts (Category with Families, setoids)
- Extend with a new concept (adding inductive types to setoid model)
- Evidence we got it right: prove a theorem (we can validate all rules)

If we are lucky, this can all happen in a proof assistant (no mistakes, and fun!).

Summary

An example of what research in type theory might look like:

- Understand existing concepts (Category with Families, setoids)
- Extend with a new concept (adding inductive types to setoid model)
- Evidence we got it right: prove a theorem (we can validate all rules)

If we are lucky, this can all happen in a proof assistant (no mistakes, and fun!).

Thanks!

References

- Thorsten Altenkirch, 1999. "Extensional equality in intensional type theory". *LICS '99*. doi:10.1109/LICS.1999.782636.
- Peter Dybjer, 1995. "Internal type theory". *TYPES '95*. doi:10.1007/3-540-61780-9_66.
- Peter Dybjer and Anton Setzer, 1999. "A Finite Axiomatization of Inductive-Recursive Definitions". *TLCA '99*. doi:10.1007/3-540-48959-2_11.
- Jacopo Emmenegger, 2021. "W-types in setoids". *Logical Methods in Computer Science* 17(3:28). doi:10.46298/LMCS-17(3:28)2021.
- Martin Hofmann, 1995. "Extensional concepts in intensional type theory". PhD thesis, University of Edinburgh. ECS-LFCS-95-327.