Constructing Inductive-Inductive Types via Type Erasure

Thorsten Altenkirch[1], Ambrus Kaposi[2], András Kovács[2], and Jakob von Raumer[1]

[1] University of Nottingham, United Kingdom
thorsten.altenkirch@nott.ac.uk, jakob@von-raumer.de
[2] Eötvös Loránd University, Budapest, Hungary
{akaposi, kovacsandras}@inf.elte.hu

Inductive-inductive types [6, 4] allow the mutual definition of a type and, for example, a type family indexed over that type. This can be used to encode collections of types which are intricately inter-related such as the syntax of type theory itself, where we define a type $\mathsf{Con} : \mathcal{U}$ of contexts at the same time as a type of types over a context: $\mathsf{Ty} : \mathsf{Con} \to \mathcal{U}$. Thes types could be populated by an empty context $\mathsf{nil} : \mathsf{Con}$, a function for context extension $\mathsf{ext} : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}(\Gamma) \to \mathsf{Con}$, a unit type former $\mathsf{unit} : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}(\Gamma)$ and a former for Π-types of the form $\mathsf{pi} : (\Gamma : \mathsf{Con})(A : \mathsf{Ty}(\Gamma)) \to \mathsf{Ty}(\mathsf{ext}(\Gamma, A)) \to \mathsf{Ty}(\Gamma)$.

Many theorem provers like Coq [2] or Lean [3], which are based on foundations similar to the calculus of constructions (CoC), do not provide native support for inductive-inductive types. This raises the question about whether each example of an inductive-inductive type can be reduced to a (mutual) inductive family, which is supported by these kinds of system.

In the above example of contexts and types, we can achieve this goal by first stripping away the type dependencies ("type erasure") and then defining a predicate which states whether an instance of the erased types is well-formed according to the dependencies we erased (cf. last year's talk [1]). The recursor is obtained by defining a relation between erased types and the types of an arbitrary algebra `M`, which then can be shown to be functional. In Lean, the main definitions of such a construction would look like this, with `CT` being the erased types of contexts and types, `CTw` being their well-formedness predicate, and `rel` being the recursor relation:

```
inductive CT : bool → Type
| nil : CT ⊥
| ext : CT ⊥ → CT ⊤ → CT ⊥
| unit : CT ⊥ → CT ⊤
| pi : CT ⊥ → CT ⊤
    → CT ⊤ → CT ⊤
```

```
def CTw_arg (b : bool) : Type := if b then unit else CT ⊥

inductive CTw : Π b, CT b → CTw_arg b → Prop
| nil : CTw ⊥ CT.nil ()
| ext : Π Γ A xΓ, CTw ⊥ Γ xΓ → CTw ⊤ A Γ → CTw ⊥ (CT.ext Γ A) ()
| unit : Π Γ xΓ, CTw ⊥ Γ xΓ → CTw ⊤ (CT.unit Γ) Γ
| pi : Π Γ A B xΓ, CTw ⊥ Γ xΓ → CTw ⊤ A Γ → CTw ⊤ B (CT.ext Γ A)
    → CTw ⊤ (CT.pi Γ A B) Γ
```

```
def rel_arg (b : bool) : Type := if b then M.C else Σ γ, M.T γ

inductive rel : Π b, CT b → rel_arg b → Prop
| nil : rel ⊥ CT.nil M.nil
| ext : Π Γ A γ a, rel ⊥ Γ γ → rel ⊤ A ⟨γ, a⟩ → rel ⊥ (CT.ext Γ A) (M.ext γ a)
| unit : Π Γ γ, rel ⊥ Γ γ → rel ⊤ (CT.unit Γ) ⟨γ, M.unit γ⟩
| pi : Π Γ A B γ a b, rel ⊥ Γ γ → rel ⊤ A ⟨γ, a⟩ → rel ⊤ B ⟨M.ext γ a, b⟩
    → rel ⊤ (CT.pi Γ A B) ⟨γ, M.pi γ a b⟩
```

This raises the question about how to prove that this strategy works for *every possible inductive-inductive type* instead of specific examples. Compared to last year's contribution [1], considerable progress on this generalization has been made.

Dissecting the question, we provide answers for the following follow-up questions:

What are inductive-inductive types? We modify the approach of Kaposi and Kovács [5] to use the contexts of a domain-specific type theory to generate the signatures of inductive-inductive types. The syntax differentiates between *sort constructors* and *point constructors*.

What are inductive families? We use a similar approach to specify inductive families: We define a syntax of *sort contexts* and of *contexts over a sort context* to capture mutual inductive families which may be parameterized over metatheoretic types.

What assumptions do we postulate? We define notions of *displayed algebras and their sections* over contexts describing inductive families, such that we can formally denote the prerequesite of our reduction: for every such context, there exists an algebra (the constructor) which is initial in the sense that every displayed algebra over it has a section (the dependent eliminator).

How to define type erasure and well-formedness. We define two *syntactical translations*: One for the type erasure and, depending on an algebra over it, one for the inductively defined well-formedness predicate, like the one contained in the above code snippet.

How to define the initial algebra. Assuming algebras over these transformed contexts, we generate an algebra over the original inductive-inductive context of which we are confident that we will be able to show its initiality.

The above approach for constructing initial algebras differs from the term model construction in [5] in that we give a factorization of algebras into erased presyntax and well-formedness predicates. This can be seen as a first step towards a generic method of initiality proofs for various type theories presented as quotient inductive-inductive types. This would require to generalize the above steps to allow equality constructors in the syntax of signatures. We have formalized all results in Agda: `https://github.com/javra/indind-agda`

References

[1] Thorsten Altenkirch, Ambrus Kaposi, András Kovács, et al. Reducing inductive-inductive types to indexed inductive types. *TYPES 2018*, 2018.

[2] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The Coq proof assistant reference manual: Version 6.1. 1997.

[3] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 378–388, 2015.

[4] Gabe Dijkstra. *Quotient inductive-inductive definitions.* PhD thesis, University of Nottingham, 2017.

[5] Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. Constructing quotient inductive-inductive types. *Proceedings of the ACM on Programming Languages*, 3(POPL):2, 2019.

[6] Fredrik Nordvall Forsberg. *Inductive-inductive definitions.* PhD thesis, Swansea University, 2013.