# Elaborating Inductive Definitions

Pierre-Évariste Dagand & Conor McBride

*Mathematically Structured Programming group,*
*University of Strathclyde*

**Résumé**

We present an elaboration of inductive definitions down to a universe of datatypes. The universe of datatypes is an internal presentation of strictly positive types within type theory. By elaborating an inductive definition – a syntactic artefact – to its code – its semantics – we obtain an internalised account of inductives inside the type theory itself: we claim that reasoning about inductive definitions could be carried in the type theory, not in the meta-theory as it is usually the case. Besides, we give a formal specification of that elaboration process. It is therefore amenable to formal reasoning too. We prove the soundness of our translation and hint at its completeness with respect to Coq's `Inductive` definitions. The practical benefits of this approach are numerous. For the type theorist, this is a small step toward bootstrapping, i.e. implementing the inductive fragment in the type theory itself. For the programmer, this means better support for generic programming: we shall present a lightweight `deriving` mechanism, entirely definable by the programmer and therefore not requiring any extension to the type theory.

In a dependent type theory, inductive types come in various shapes and forms. Unsurprisingly, we can define data-types à la ML, following the *sum-of-product* recipe: we offer a choice of constructors and, for each constructor, comes a product of arguments. An example is the vintage List datatype:

$$\begin{array}{l}\textbf{data}\ \mathsf{List}\ [A : \textsc{Set}] : \textsc{Set}\ \textbf{where}\\ \quad \mathsf{List}\ A\ \ni \mathsf{nil}\\ \quad\qquad\quad\ \mid\ \mathsf{cons}\ (a : A)(as : \mathsf{List}\ A)\end{array}$$

For the working semanticist, this brings fond memory of a golden era: this syntax has a trivial categorical interpretation in term of *signature functor*, here $L_A X = 1 + A \times X$. Without a second thought, we can brush away the syntax, mapping the syntactic representations of sum and product to their categorical counterpart. Handling parameters comes at a minor complexity cost: we merely parameterise the functor itself, for instance with $A$ here.

We ought to make sure that our language of data-type is correct, let alone semantically meaningful. Indeed, if we were to accept the following definition

$$\begin{array}{l}\textbf{data}\ \mathsf{Bad}\ [A : \textsc{Set}] : \textsc{Set}\ \textbf{where}\\ \quad \mathsf{Bad}\ A\ \ni \mathsf{ex}\ (f : \mathsf{Bad}\ A \to A)\end{array}$$

we would make many formal developments a lot easier to prove! To ban these bogus definitions, theorem provers such as Agda [Norell, 2007] or Coq [The Coq Development Team] rely on a positivity checker to ensure that all recursive arguments are in a strictly positive position. The positivity checker is therefore part of the trusted computing base of the theorem prover. Besides, by working on the syntactic representation of datatypes, it is a non negligible piece of software that is a common source of frustration: it either stubbornly prevents perfectly valid definitions – as it sometimes is the case in Coq – or happily accepts obnoxious definitions – as Agda users discover every so often.

While reasoning about datatypes is at a functor away, we seem stuck with these clumsy syntactic presentations: quoting Harper and Stone [2000], "the treatment of datatypes is technically complex, but conceptually straightforward". Following Stone and Harper, most authors [Asperti et al., 2012, Luo, 1994, McBride et al., 2004] have no choice but to throw in the towel and proceed over a "…"-filled skeleton of inductive definition. Proving any property on such definition is a perilous exercise for the author, and a hardship on the reader.

We attribute these difficulties to the formal gap between the syntax of inductive definitions and their semantics. While inductive types have an interpretation in term of initial algebra of strictly positive functors, we are unable to leverage this knowledge. Being stuck with a syntactic artefact, the ghost of the de Bruijn criterion haunts our type theories: inductive definitions elude the type checker and must be enforced by a not-so-small positivity checker. Besides, since the syntax of inductive definitions is hardly amenable to formal reasoning, we are left wondering if its intended semantics is indeed respected. How many inductive skeletons are hidden in the dark closet of your theorem prover?

An alternative to a purely syntactic approach is to reflect inductive types inside the type theory itself. Following Benke et al. [2003], we extend a Martin-Löf type theory with a universe of inductive types, all that for a minor complexity cost [Chapman et al., 2010]. From within the type theory, we are then able to create and manipulate datatypes but also compute over them. However, from a user perspective, these codes are a no-go: manually coding datatypes, for instance, is too cumbersome. Rather than writing low-level codes, we would like to write a honest-to-goodness inductive definition and get the computer to automatically *elaborate* it to a code in the universe. In this paper, we therefore show how we can grow a programming language on top of a type theory with a universe of datatypes.

In this paper, we elaborate upon (pun intended) the syntax of datatypes introduced in an earlier work [Dagand and McBride, 2012]. While this syntax had been informally motivated, this paper is a first step toward a formal specification of its elaboration down to our universe of datatypes. Our contributions are the following:

- In Section 2, we give a crash course in elaboration for dependent types. We will present a bidirectional type checker [Pierce and Turner, 1998] for our type theory. We then extend it to make programming a less cryptic experience. To that purpose, we shall use types as *presentations* of more high-level concepts, such as the notions of finite set or of datatype constructor. While this section does not contain any new result *per se*, we aim at introducing the reader to a coherent collection of techniques that, put together, form a general framework for type-directed elaboration ;
- In Section 3, we specify the elaboration of inductive types down to a simple universe of inductive types. By lack of space, we had to leave out elaboration of inductive families. While the system we present in this paper is restricted to strictly positive types, we take advantage of its simplicity to develop our intuition. In this section, we aim at presenting a general methodology for growing a practical programming language out of a core calculus. The choice of a particular universe of datatypes is in large part irrelevant. In particular, the same ideas are at play in the case of inductive families[1] ;
- In Section 4, we consider two potential extensions of the elaboration machinery. For the proof-assistant implementer, we show how meta-theoretical results on inductives, such as the work of McBride et al. [2004], can be internalised and formally presented in the type theory. For the programmer, we show how a generic `deriving` mechanism à la Haskell can be implemented from within the type theory. This section aims both at demonstrating our universe-based approach, but also at motivating extensions of the basic elaboration machinery.

[1]Inductive families are treated in a companion technical report, available on Dagand's website.