Elaborating inductive definitions

Pierre-Evariste Dagand, Conor Mcbride

Elaborating Inductive Definitions

Pierre-Évariste Dagand & Conor McBride

Mathematically Structured Programming group,
University of Strathclyde

Abstract

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:

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

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.

$$\boxed{\Gamma \vdash \text{VALID}}$$

$$\frac{}{\vdash \text{VALID}} \qquad \frac{\Gamma \vdash \text{VALID} \quad \Gamma \vdash S : \text{SET}_k}{\Gamma; x : S \vdash \text{VALID}}\ x \notin \Gamma$$

$$\frac{\Gamma \vdash \text{VALID} \quad \Gamma \vdash S : \text{SET} \quad \Gamma \vdash t : S}{\Gamma; x \mapsto t : S \vdash \text{VALID}}\ x \notin \Gamma$$

(a) Context validity

$$\boxed{\Gamma \vdash a \equiv b : T}$$

$$\frac{\Gamma \vdash S : \text{SET} \quad \Gamma; x : S \vdash t : T \quad \Gamma \vdash s : S}{\Gamma \vdash (\lambda_S x.\, t)\ s \equiv t[s/x] : T[s/x]}$$

$$\frac{\Gamma \vdash s : S \quad \Gamma; x : S \vdash T : \text{SET} \quad \Gamma \vdash t : T[s/x]}{\Gamma \vdash \pi_0\ ((s,t)_{x.T}) \equiv s : S}$$

$$\frac{\Gamma \vdash s : S \quad \Gamma; x : S \vdash T : \text{SET} \quad \Gamma \vdash t : T[s/x]}{\Gamma \vdash \pi_1\ ((s,t)_{x.T}) \equiv t : T[s/x]}$$

(b) Judgmental equality

$$\boxed{\Gamma \vdash t : T}$$

$$\frac{\Gamma; x : S; \Delta \vdash \text{VALID}}{\Gamma; x : S; \Delta \vdash x : S} \qquad \frac{\Gamma \vdash s : S \quad \Gamma \vdash S \equiv T : \text{SET}_k}{\Gamma \vdash s : T}$$

$$\frac{\Gamma \vdash \text{VALID}}{\Gamma \vdash \text{SET}_k : \text{SET}_{k+1}} \qquad \frac{\Gamma \vdash \text{VALID}}{\Gamma \vdash \mathbb{1} : \text{SET}_k} \qquad \frac{\Gamma \vdash \text{VALID}}{\Gamma \vdash * : \mathbb{1}}$$

$$\frac{\Gamma \vdash S : \text{SET}_k \quad \Gamma; x : S \vdash T : \text{SET}_k}{\Gamma \vdash (x : S) \times T : \text{SET}_k}$$

$$\frac{\Gamma \vdash s : S \quad \Gamma; x : S \vdash T : \text{SET}_k \quad \Gamma \vdash t : T[s/x]}{\Gamma \vdash (s,t)_{x.T} : (x : S) \times T}$$

$$\frac{\Gamma \vdash p : (x : S) \times T}{\Gamma \vdash \pi_0\, p : S} \qquad \frac{\Gamma \vdash p : (x : S) \times T}{\Gamma \vdash \pi_1\, p : T[\pi_0\, p/x]}$$

$$\frac{\Gamma \vdash S : \text{SET}_k \quad \Gamma; x : S \vdash T : \text{SET}_k}{\Gamma \vdash (x : S) \to T : \text{SET}_k}$$

$$\frac{\Gamma \vdash S : \text{SET}_k \quad \Gamma; x : S \vdash t : T}{\Gamma \vdash \lambda_S x.\, t : (x : S) \to T} \qquad \frac{\Gamma \vdash f : (x : S) \to T \quad \Gamma \vdash s : S}{\Gamma \vdash f\ s : T[s/x]}$$

(c) Typing judgments

Figure 1: Type theory

Scope of this work: This paper aims at *specifying* an elaboration procedure from an inductive definition down to its representation in a universe of inductive types. At the risk of disappointing implementers, we are not describing an implementation. In particular, we shall present the elaboration in a relational style, hence conveniently glancing over the operational details. Our goal is to ease the formal study of inductive definitions, hence the choice of this more abstract style. Nonetheless, this paper is not entirely disconnected from implementation. First, it grew out of our work on the Epigram system (Brady et al.), in which Peter Morris implemented a tactic elaborating an earlier form of inductive definition down to our universe of code. Second, a tutorial implementation of an elaborator for inductive types is currently underway.

1. The Type Theory

For this paper to be self-contained, we shall recall a few definitions from our previous work (Chapman et al., 2010). We shall not dwell on the meta-theoretical properties of this system: Luo (1994) has presented the meta-theoretical properties of Martin-Löf type theory, while Chapman et al. (2010) have studied its extension with a universe of datatypes. We present our core type theory in Figure 1. It is a standard Martin-Löf type theory, with Σ-types and Π-types. We shall write SET_k for the hierarchy of types, implicitly assuming cumulativity of universes. In order to be equality-agnostic, we simply specify our expectations through a judgmental presentation. Our presentation is hopefully not controversial and should adapt easily to regional variations, such as Coq, Agda or an observational type theory (Altenkirch et al., 2007).

A first addition to this core calculus is a universe of enumerations (Fig. 2). The purpose of this universe is to let us define finite collections of labels. Labels are introduced through the Uld type and are then used to define finite sets through the EnumU universe. To index a specific element in such a

$$\frac{\Gamma \vdash \text{VALID}}{\Gamma \vdash \mathsf{Uld} : \textsc{Set}}$$

$$\frac{\Gamma \vdash \text{VALID}}{\Gamma \vdash \text{'}s : \mathsf{Uld}}\ s \text{ a valid identifier}$$

(a) Tags

$$\frac{\Gamma \vdash \text{VALID}}{\Gamma \vdash \mathsf{EnumU} : \textsc{Set}}$$

$$\frac{\Gamma \vdash \text{VALID}}{\Gamma \vdash \mathsf{nilE} : \mathsf{EnumU}}$$

$$\frac{\Gamma \vdash t : \mathsf{Uld} \quad \Gamma \vdash E : \mathsf{EnumU}}{\Gamma \vdash \mathsf{consE}\ t\ E : \mathsf{EnumU}}$$

(b) Enumeration

$$\frac{\Gamma \vdash E : \mathsf{EnumU}}{\Gamma \vdash \mathsf{EnumT}\ E : \textsc{Set}}$$

$$\frac{\Gamma \vdash \text{VALID}}{\Gamma \vdash 0 : \mathsf{EnumT}\ (\mathsf{consE}\ t\ E)}$$

$$\frac{\Gamma \vdash n : \mathsf{EnumT}\ E}{\Gamma \vdash 1\!+\ n : \mathsf{EnumT}\ (\mathsf{consE}\ t\ E)}$$

(c) Index

$$\frac{\Gamma \vdash E : \mathsf{EnumU} \quad \Gamma \vdash P : \mathsf{EnumT}\ E \to \textsc{Set}}{\Gamma \vdash \pi\ E\ P : \textsc{Set}}$$

$$\frac{\Gamma \vdash E : \mathsf{EnumU} \quad \Gamma \vdash P : \mathsf{EnumT}\ E \to \textsc{Set} \quad \Gamma \vdash ps : \pi\ E\ P \quad \Gamma \vdash x : \mathsf{EnumT}\ E}{\Gamma \vdash \mathsf{switch}\ E\ P\ ps\ x : P\ x}$$

(d) Elimination forms

Figure 2: Universe of enumerations

set, we write an EnumT code. To eliminate finite sets, we form a small Π-type π that builds a lookup tuple mapping, for all label e in the enumeration, a value of type $P\ e$. The elimination principle switch performs this lookup, given an EnumT code and a lookup table for it. The equational theory is extended according to this intuition.

Example 1 (Coding $\{\text{'}a, \text{'}b, \text{'}c\}$)**.** We define this set by merely enumerating its labels, in effect building a list of tags:

$$\{\text{'}a, \text{'}b, \text{'}c\} \triangleq \mathsf{consE}\ \text{'}a\ (\mathsf{consE}\ \text{'}b\ (\mathsf{consE}\ \text{'}c\ \mathsf{nilE})) : \mathsf{EnumU}$$

Example 2 (Coding $\{\text{'}a \mapsto e_a, \text{'}b \mapsto e_b, \text{'}c \mapsto e_c\} : (x : \mathsf{EnumT}\ \{\text{'}a, \text{'}b, \text{'}c\}) \to P x$)**.** We define this function by a straightforward application of the switch eliminator:

$$\{\text{'}a \mapsto e_a, \text{'}b \mapsto e_b, \text{'}c \mapsto e_c\} \triangleq \mathsf{switch}\ (\mathsf{consE}\ \text{'}a\ (\mathsf{consE}\ \text{'}b\ (\mathsf{consE}\ \text{'}c\ \mathsf{nilE})))\ P\ (e_a, (e_b, (e_c, *)))$$

We recall the definition of our universe of inductive types in Figure 3. This universe captures strictly positive types, a generalisation of ML datatypes to dependent types. For pedagogical reason, we choose this simple universe as a first step toward a full-blown universe of inductive families. To define new datatypes, we give their code by *des*cribing their signature functor in Desc. Intuitively, Desc is an algebraic presentation of polynomial functors. The 'Σ code represents sums of monomials, while 'Π represents the exponents. 'var codes the polynomial's variable. To represent finite sums and products, we use, respectively, the 'σ and '$\times$ codes.

The interpretation function $[\![\_]\!]$ turns such a description into the corresponding endofunctor over Set: from a syntactic description of a functor, the interpretation computes the actual semantic object. Its definition is obvious from the codes: a 'Σ is interpreted into a Σ-type, and so on. The notable exception is 'var, which describes the identity functor. Remark that the resulting functor are strictly positive, by construction. Hence, we can construct their least fix-point by defining $\mu\ D \triangleq [\![D]\!]\ (\mu\ D)$ and safely provide a generic elimination principle, induction. The All function computes the inductive hypothesis. The interested reader will find their precise definition elsewhere (Chapman et al., 2010) but the basic intuition we have given here is enough to understand this paper.

Example 3 (Describing natural numbers)**.** Natural numbers can be presented as the least fix-point of the functor $FX = 1 + X$. This is described by:

$$\begin{array}{lcl} \mathsf{NatD} & : & \mathsf{Desc} \\ \mathsf{NatD} & \mapsto & \text{'}\sigma \left\{ \begin{array}{l} \text{'}0, \\ \text{'}\mathsf{suc} \end{array} \right\} \left\{ \begin{array}{l} \text{'}0 \mapsto \text{'}1, \\ \text{'}\mathsf{suc} \mapsto \text{'}\mathsf{var} \times \text{'}1 \end{array} \right\} \end{array} \qquad \begin{array}{lcl} \mathsf{Nat} & : & \textsc{Set} \\ \mathsf{Nat} & \mapsto & \mu\ \mathsf{NatD} \end{array}$$

$$\frac{\Gamma \vdash \text{VALID}}{\Gamma \vdash \mathsf{Desc} : \textsc{Set}_1}$$

$$\frac{\Gamma \vdash \text{VALID}}{\Gamma \vdash \text{'var} : \mathsf{Desc}} \qquad \frac{\Gamma \vdash \text{VALID}}{\Gamma \vdash \text{'1} : \mathsf{Desc}}$$

$$\frac{\Gamma \vdash A : \mathsf{Desc} \quad \Gamma \vdash B : \mathsf{Desc}}{\Gamma \vdash A \text{'}\times B : \mathsf{Desc}}$$

$$\frac{\Gamma \vdash S : \textsc{Set} \quad \Gamma \vdash T : S \to \mathsf{Desc}}{\Gamma \vdash \text{'}\Pi\ S\ T : \mathsf{Desc}}$$

$$\frac{\Gamma \vdash S : \textsc{Set} \quad \Gamma \vdash T : S \to \mathsf{Desc}}{\Gamma \vdash \text{'}\Sigma\ S\ T : \mathsf{Desc}}$$

$$\frac{\Gamma \vdash E : \mathsf{EnumU} \quad \Gamma \vdash T : \mathsf{EnumT}\ E \to \mathsf{Desc}}{\Gamma \vdash \text{'}\sigma\ E\ T : \mathsf{Desc}}$$

(a) Codes

$$\begin{array}{llll}
[\![(D : \mathsf{Desc})]\!]\ (X : \textsc{Set}) & : & \textsc{Set} \\
[\![\text{'var}]\!] & X & \mapsto & X \\
[\![\text{'1}]\!] & X & \mapsto & \mathbb{1} \\
[\![A\ \text{'}\times\ B]\!] & X & \mapsto & [\![A]\!]\ X \times [\![B]\!]\ X \\
[\![\text{'}\Pi\ S\ T]\!] & X & \mapsto & (s : S) \to [\![T\ s]\!]\ X \\
[\![\text{'}\Sigma\ S\ T]\!] & X & \mapsto & (s : S) \times [\![T\ s]\!]\ X \\
[\![\text{'}\sigma\ E\ T]\!] & X & \mapsto & (e : \mathsf{EnumT}\ E) \times [\![T\ e]\!]\ X
\end{array}$$

$$\frac{\Gamma \vdash D : \mathsf{Desc}}{\Gamma \vdash \mu\ D : \textsc{Set}} \qquad \frac{\Gamma \vdash xs : [\![D]\!]\ \mu\ D}{\Gamma \vdash \mathsf{in}\ xs : \mu\ D}$$

(b) Fix-point

$$\mathsf{induction} : (D : \mathsf{Desc})(P : \mu D \to \textsc{Set}) \to ((d : [\![D]\!]\ (\mu D)) \to \mathsf{All}\ D\ P\ d \to P(\mathsf{in}\, d)) \to (x : \mu D) \to P x$$

(c) Elimination principle

Figure 3: Universe of datatypes

Note that we are using a small $\text{'}\sigma$ here: a datatype definition always starts with a finite choice of constructors. This corresponds to the *tagged descriptions* of Chapman et al. (2010): using this structure, we can implement some generic constructions – such as the Zipper or the free monad – and generic theorems – such as in Section 4.1. The interpretation of NatD gives a functor *isomorphic* to F. The presence of a spurious '1 in the description of suc is justified in the next section, when we consider the elaboration of datatype constructors.

Inhabitants of Nat are either $\mathsf{0} \triangleq \mathsf{in}\ (\text{'0}, *)$ or $\mathsf{suc}\ n \triangleq \mathsf{in}\ (\text{'suc}, (n, *))$ for $n : \mathsf{Nat}$. We obtain the minor premises of the induction principle by unfolding the definition of All, which computes the induction hypothesis. In the 0 case, we have $\mathsf{All}\ \mathsf{NatD}\ P\ (\text{'0}, *) = \mathbb{1}$ – i.e. we must prove $P\ \mathsf{0}$ in the base case. In the suc case, we have $\mathsf{All}\ \mathsf{NatD}\ P\ (\text{'suc}, (n, *)) = P\ n$ – i.e. for $n : \mathsf{Nat}$ such that $P\ n$, we must prove $P\ (\mathsf{suc}\ n)$. This corresponds to the standard induction principle of natural numbers.

Example 4 (Describing binary trees)**.** Categorically, binary trees are modeled by the least fix-point of the functor $T_A X = 1 + X \times A \times X$. In our universe, we obtain trees by the following definitions:

$$\begin{array}{l}
\mathsf{TreeD}\ (A : \textsc{Set}) : \mathsf{Desc} \\
\mathsf{TreeD}\ A \mapsto \text{'}\sigma \left\{ \begin{array}{l} \text{'leaf}, \\ \text{'node} \end{array} \right\} \left\{ \begin{array}{l} \text{'leaf} \mapsto \text{'1}, \\ \text{'node} \mapsto \text{'var}\ \text{'}\times\ \text{'}\Sigma\ A\ \lambda_.\ \text{'var}\ \text{'}\times\ \text{'1} \end{array} \right\}
\end{array} \qquad
\begin{array}{l}
\mathsf{Tree}\ (A : \textsc{Set}) : \textsc{Set} \\
\mathsf{Tree}\ A \mapsto \mu\ (\mathsf{TreeD}\ A)
\end{array}$$

We easily check that the interpretation of TreeD is indeed *isomorphic* to T_A.

Inhabitants of $\mathsf{Tree}\ A$ are either $\mathsf{leaf} \triangleq \mathsf{in}\ (\text{'leaf}, *)$ or $\mathsf{node}\ l\ a\ r \triangleq \mathsf{in}\ (\text{'node}, (l, (a, (r, *))))$ for $l, r : \mathsf{Tree}\ A$ and $a : A$. Again, We obtain the minor premises of the induction principle by unfolding the definition of All. In the leaf case, we have $\mathsf{All}\ (\mathsf{TreeD}\ A)\ P\ (\text{'leaf}, *) = \mathbb{1}$ – i.e. we must prove $P\ \mathsf{leaf}$ in the base case. In the node case, we have $\mathsf{All}\ (\mathsf{TreeD}\ A)\ P\ (\text{'node}, (l, (a, (r, *)))) = P\ l \times P\ r$ – i.e. for $l, r : \mathsf{Tree}\ A$ such that $P\ l$ and $P\ r$ and for any $a : A$, we must prove $P\ (\mathsf{node}\ l\ a\ r)$. This corresponds to the standard induction principle of binary trees.

$$\boxed{\Gamma \vdash t \overset{Inf}{\rightsquigarrow} t' \in T}$$

$$\frac{\Gamma \vdash \textsc{Set}_k \ni T \overset{Chk}{\rightsquigarrow} T' \quad \Gamma \vdash T' \ni t \overset{Chk}{\rightsquigarrow} t'}{\Gamma \vdash (t : T) \overset{Inf}{\rightsquigarrow} t' \in T'}$$

$$\frac{\Gamma ; x : S ; \Delta \vdash \textsc{valid}}{\Gamma ; x : S ; \Delta \vdash x \overset{Inf}{\rightsquigarrow} x \in S}$$

$$\frac{\Gamma \vdash f \overset{Inf}{\rightsquigarrow} f' \in (x : S) \to T \quad \Gamma \vdash S \ni s \overset{Chk}{\rightsquigarrow} s'}{\Gamma \vdash f\ s \overset{Inf}{\rightsquigarrow} f'\ s' \in T[s'/x]}$$

$$\frac{\Gamma \vdash p \overset{Inf}{\rightsquigarrow} p' \in (x : S) \times T}{\Gamma \vdash \pi_0 p \overset{Inf}{\rightsquigarrow} \pi_0 p' \in S}$$

$$\frac{\Gamma \vdash p \overset{Inf}{\rightsquigarrow} p' \in (x : S) \times T}{\Gamma \vdash \pi_1 p \overset{Inf}{\rightsquigarrow} \pi_1 p' \in T[\pi_0 p'/x]}$$

(a) Type synthesis

$$\boxed{\Gamma \vdash T \ni t \overset{Chk}{\rightsquigarrow} t'}$$

$$\frac{\Gamma \vdash s \overset{Inf}{\rightsquigarrow} s' \in S \quad \Gamma \vdash S \equiv T : \textsc{Set}_k}{\Gamma \vdash T \ni s \overset{Chk}{\rightsquigarrow} s'}$$

$$\frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash \textsc{Set}_{k+1} \ni \textsc{Set}_k \overset{Chk}{\rightsquigarrow} \textsc{Set}_k}$$

$$\frac{\Gamma \vdash \textsc{Set}_k \ni S \overset{Chk}{\rightsquigarrow} S' \quad \Gamma ; x : S' \vdash \textsc{Set}_k \ni T \overset{Chk}{\rightsquigarrow} T'}{\Gamma \vdash \textsc{Set}_k \ni (x : S) \to T \overset{Chk}{\rightsquigarrow} (x : S') \to T'}$$

$$\frac{\Gamma ; x : S \vdash T \ni t \overset{Chk}{\rightsquigarrow} t'}{\Gamma \vdash (x : S) \to T \ni \lambda x.\, t \overset{Chk}{\rightsquigarrow} \lambda_S x.\, t'}$$

$$\frac{\Gamma \vdash \textsc{Set}_k \ni S \overset{Chk}{\rightsquigarrow} S' \quad \Gamma ; x : S' \vdash \textsc{Set}_k \ni T \overset{Chk}{\rightsquigarrow} T'}{\Gamma \vdash \textsc{Set}_k \ni (x : S) \times T \overset{Chk}{\rightsquigarrow} (x : S') \times T'}$$

$$\frac{\Gamma \vdash S \ni s \overset{Chk}{\rightsquigarrow} s' \quad \Gamma \vdash T[s'/x] \ni t \overset{Chk}{\rightsquigarrow} t'}{\Gamma \vdash (x : S) \times T \ni (s, t) \overset{Chk}{\rightsquigarrow} (s', t')_{x.T}}$$

$$\frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash \textsc{Set}_k \ni \mathbb{1} \overset{Chk}{\rightsquigarrow} \mathbb{1}} \qquad \frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash \mathbb{1} \ni * \overset{Chk}{\rightsquigarrow} *}$$

(b) Type checking

Figure 4: Bidirectional type checker

2. First Steps in Elaboration

In this section, we shall make our first steps in elaboration. First, we present a bidirectional type system for the calculus introduced in the previous section. Using the flow of information from type synthesis to type checking, we then declutter our term language. We shall see how, with little effort, we can move away from an austere calculus and closer to a proper programming language.

2.1. Bidirectional type checking

The idea of bidirectional type checking (Pierce and Turner, 1998) is to capture, in the specification of the type checker, the local flow of typing information. On the one hand, we will *synthesise* types from variables and functions while, on the other hand, we will *check* terms against these synthesised types. By checking terms against their types, we can use types to structure the term language: for example, we remove the need for writing the domain type in an abstraction, or we can use a tuple notation for telescopes of Σ-types. Following Harper and Stone (2000), we distinguish two categories of terms. The core type theory defines the *internal* language. The bidirectional approach lets us then extend this core language into a more convenient *external* language. We shall hasten to add that this is not a novel idea: for instance, it is the basis of Matita's refinement system (Asperti et al., 2012). Also, we gave a similar presentation in some earlier work (Chapman et al., 2010).

Let us present type synthesis (Fig. 4a) first. The judgment $\Gamma \vdash t \overset{Inf}{\rightsquigarrow} t' \in T$ states that the external term t elaborates to the internal term t' of type T in the context Γ. By convention, we shall keep the inputs of the relation to the left of the $\rightsquigarrow$ symbol, while outputs will be on the right. Note that we

switch from synthesising to checking in only two cases. First, in the application rule, we synthesise the function type, and therefore the argument type, and check the argument against its synthesised type. Second, we can explicitly annotate a checkable term by its type, thus obtaining a (trivially) synthesisable term. We expect the following soundness property to hold:

Theorem 1 (Soundness of type synthesis)**.** *If* $\Gamma \vdash t \overset{Inf}{\rightsquigarrow} t' \in T$*, then* $\Gamma \vdash t' : T$*.*

While type synthesis initiates the flow of typing information, type checking (Fig. 4b) lets us make use of this information to enrich the term language. The interpretation of the judgment $\Gamma \vdash T \ni t \overset{Chk}{\rightsquigarrow} t'$ is that the external term t is checked against the type T in context Γ and elaborates to an internal term t'. The switch from checking to synthesising is made on a purely syntactic basis: we remark that we can segregate the external language in two mutually defined categories. On one hand, the synthesisable terms – consisting of variables, elimination forms and type annotation – and the checkable terms – consisting of the canonical objects. Doing so, we tame the apparent non-determinism: we switch from checking to inferring only when changing of syntactic category. Again, we expect the following soundness property to hold:

Theorem 2 (Soundness of type checking)**.** *If* $\Gamma \vdash T \ni t \overset{Chk}{\rightsquigarrow} t'$*, then* $\Gamma \vdash t' : T$*.*

Proof sketch of Theorem 1 and Theorem 2. The proof is by mutual induction over the synthesis and checking judgments. As we add more rules to the type checking system, we have made sure that these properties are preserved.

□

2.2. Putting types at work

The type checker we have specified so far only allows untyped abstractions. We extend it with some convenient features.

Tuples: By design of the interpretation of our universes of enumeration and inductive types, we are often going to build inhabitants of Σ-telescope of the form $(a : A) \times (b : B) \times \ldots \times (z : Z) \times \mathbb{1}$. To reduce the syntactic burden of these nested pairs, we elaborate a LISP-inspired tuple notation:

$$\frac{}{\Gamma \vdash \mathbb{1} \ni () \overset{Chk}{\rightsquigarrow} *} \qquad \frac{\Gamma \vdash A \ni x \overset{Chk}{\rightsquigarrow} x' \quad \Gamma \vdash B[x'/a] \ni (xs) \overset{Chk}{\rightsquigarrow} xs'}{\Gamma \vdash (a : A) \times B \ni (x\ xs) \overset{Chk}{\rightsquigarrow} (x', xs')_{a.B}}$$

Finite sets, introduction and elimination: In Section 1, we have used an informal set-like notation for enumerations: we can make that notation formal through elaboration. To do so, we extend the type checker with the following rules:

$$\frac{}{\Gamma \vdash \mathsf{EnumU} \ni \{\} \overset{Chk}{\rightsquigarrow} \mathsf{nilE}} \qquad \frac{\Gamma \vdash \mathsf{EnumU} \ni \{ts\} \overset{Chk}{\rightsquigarrow} E}{\Gamma \vdash \mathsf{EnumU} \ni \{\text{'}a, ts\} \overset{Chk}{\rightsquigarrow} \mathsf{consE}\ \text{'}a\ E}$$

$$\frac{\Gamma \vdash \mathsf{EnumU} \ni \{\text{'}l_0, \ldots, \text{'}l_k\} \overset{Chk}{\rightsquigarrow} E \qquad \Gamma \vdash \pi\ E\ P \ni (e_0 \ldots e_k) \overset{Chk}{\rightsquigarrow} es}{\Gamma \vdash (e : \mathsf{EnumT}\ E) \to P\ e \ni \{\text{'}l_0 \mapsto e_0, \ldots \text{'}l_k \mapsto e_k\} \overset{Chk}{\rightsquigarrow} \mathsf{switch}\ E\ P\ es}$$

Indexing finite sets: Also, rather than indexing into enumerations through the EnumT codes, we would like to be able to write the label and have it elaborate to the corresponding index. This is achieved by the following extension:

$$\frac{}{\Gamma \vdash \mathsf{consE}\ {'t}\ E \ni {'t} \overset{Chk}{\rightsquigarrow} 0} \qquad \frac{\Gamma \vdash E \ni {'t} \overset{Chk}{\rightsquigarrow} n}{\Gamma \vdash \mathsf{consE}\ {'u}\ E \ni {'t} \overset{Chk}{\rightsquigarrow} 1{+}n}$$

Datatype constructors: Finally, while we do not yet have a proper syntax for *declaring* inductive types, we can already extend our term language with constructors. Upon elaborating an external term $c\ a_0 \dots a_k$ against the fix-point of a tagged description, we replace this elaboration problem with the one consisting of elaborating the tuple consisting of the constructor label and the arguments:

$$\frac{\Gamma \vdash [\![{'\sigma}\ E\ T]\!]\ (\mu\ ({'\sigma}\ E\ T)) \ni ({'c}\ a_0 \dots a_k) \overset{Chk}{\rightsquigarrow} \mathsf{in}\ t}{\Gamma \vdash \mu\ ({'\sigma}\ E\ T) \ni c\ a_0 \dots a_k \overset{Chk}{\rightsquigarrow} t}$$

Example 5 (Elaboration of $\mathsf{node}\ l\ a\ r$)**.** In Example 4, we painfully coded the node constructor by writing $\mathsf{in}\ ({'\mathsf{node}}, (l, (a, (r, *))))$. Thanks to the above rule, we can directly write $\mathsf{node}\ l\ a\ r$ and it elaborates as expected:

$$\frac{\Gamma \vdash [\![\mathsf{TreeD}\ A]\!]\ (\mu\ (\mathsf{TreeD}\ A)) \ni ({'\mathsf{node}}\ l\ a\ r) \overset{Chk}{\rightsquigarrow} \mathsf{in}\ ({'\mathsf{node}}, (l, (a, (r, *))))}{\Gamma \vdash \mu\ (\mathsf{TreeD}\ A) \ni \mathsf{node}\ l\ a\ r \overset{Chk}{\rightsquigarrow} \mathsf{in}\ ({'\mathsf{node}}, (l, (a, (r, *))))}$$

3. Elaborating Datatypes

In this section, we specify the elaboration of inductive types down to our Desc universe. While this universe only captures strictly positive types, it is a good exercise to understand the general idea governing the elaboration of inductive definitions. Besides, because the syntax is essentially the same, our presentation should be easy to understand for readers familiar with Coq or Agda.

We adopt the standard sum-of-product high-level notation:

$$\begin{aligned} &\textbf{data}\ \mathsf{D}\ \overrightarrow{[p : P]} : \textsc{Set}\ \textbf{where} \\ &\quad \mathsf{D}\ \overrightarrow{p}\ \ni\ \mathsf{c}_0\ \overrightarrow{(a_0 : T_0)} \\ &\quad \mid \dots \\ &\quad \mid\ \mathsf{c}_k\ \overrightarrow{(a_k : T_k)} \end{aligned}$$

Where the arguments $\vec{p}$ are parameters. A T_i can be recursive, i.e. refer to $\mathsf{D}\,\vec{p}$. Note that it is crucial that the parameters are the same in the definition and the recursive arguments.

Our translation to code follows the structure of the definition. The first level structure consists of the choice of constructors and is translated to a $'\sigma$ code over the finite set of constructors. The second level structure consists of the Σ-telescope of arguments: it translates to right-nested $'\Sigma$ codes. When parsing arguments, we must make sure that the recursive arguments are valid and translate them to the 'var code.

3.1. Description labels

To guide the elaboration of inductive definitions, we extend the type theory with *description labels* (Fig. 5). Their role is akin to programming labels (McBride and McKinna, 2004): they structure

$$\boxed{\Gamma \vdash l\ \mathrm{datatel}}$$

$$\frac{}{\Gamma \vdash \mathbf{D}\ \mathrm{datatel}} \qquad \frac{\Gamma \vdash l\ \mathrm{datatel} \quad \Gamma \vdash t : T}{\Gamma \vdash l\ t\ \mathrm{datatel}}$$

$$\frac{\Gamma \vdash l\ \mathrm{datatel}}{\Gamma \vdash \langle l \rangle : \textsc{Set}_1} \qquad \frac{\Gamma \vdash E : \mathsf{EnumU} \quad \Gamma \vdash T : \mathsf{EnumT}\ E \to \mathsf{Desc}}{\Gamma \vdash \mathsf{return}\ E\ T : \langle l \rangle} \qquad \frac{\Gamma \vdash t : \langle l \rangle}{\Gamma \vdash \mathsf{call}\ \langle l \rangle\ t : \mathsf{Desc}}$$

Figure 5: Description label

the elaboration task and are used to ensure that recursive arguments are correctly elaborated. A description label $\langle l \rangle$ is a list starting with the name of the datatype being defined, followed by the parameters of that datatype. It can be thought of as a phantom type around Desc: it hides a low-level Desc object with an high-level presentation, i.e. the name and parameters of the datatype being defined. Every elaborative step is ran against a description label: thus, we can spot recursive arguments and check that parameters are preserved across a definition.

We introduce such type using return that takes the (finite) set of constructors and their respective code: doing so, we ensure that we are only accepting tagged descriptions. With call $\langle l \rangle$, we eliminate return by joining constructors and their codes in a $'\sigma$ code, effectively interpreting the choice of constructor:

$$\frac{\Gamma \vdash l\ \mathrm{datatel} \quad \Gamma \vdash E : \mathsf{EnumU} \quad \Gamma \vdash T : \mathsf{EnumT}\ E \to \mathsf{Desc}}{\Gamma \vdash \mathsf{call}\ \langle l \rangle\ (\mathsf{return}\ E\ T) \equiv {}'\sigma\ E\ T : \mathsf{Desc}}$$

3.2. Elaborating inductive types

We shall present our translation in a top-down manner: from a complete definition, we show how the pieces fit together, giving some intuition for the subsequent translations. We then move on to disassemble and interpret each sub-component separately. As we progress, the reader should check that the intuition we gave for the whole is indeed valid. Every elaboration step is backed by a soundness property: proving these properties is inherently bottom-up. After having presented our definitions, we can prove the soundness theorem. The proof is technically unsurprising: we shall briefly sketch it at the end of this section. To further ease the understanding of our machinery, we illustrate each step by elaborating binary trees:

```
data Tree [A : Set] : Set where
  Tree A ∋ leaf
         | node (l : Tree A)(a : A)(r : Tree A)
```

Elaboration of an inductive definition (Fig. 6a): The judgment reads as: in context Γ, the definition **data** $D\overrightarrow{(p : P)} : \textsc{Set}$ **where** *choices* extends the original context to a context Δ in which D has been defined. To obtain this definition, we first elaborate the parameters – via type checking – and move onto elaborating the choice of constructors – via rule (Choices) – introducing a description label in the process.

$$\boxed{\Gamma \vdash \textbf{data}\ D\overrightarrow{(p : P)} : \textsc{Set}\ \textbf{where}\ \mathit{choices} \overset{D}{\rightsquigarrow} \Delta}$$

$$\frac{\begin{array}{c}\Gamma \vdash \textsc{Set}_1 \ni \overrightarrow{(p : P)} \to \textsc{Set} \overset{Chk}{\rightsquigarrow} \overrightarrow{(p : P')} \to \textsc{Set} \\ \Gamma, \overrightarrow{p : P'} \vdash \langle \mathbf{D}\ \vec{p} \rangle \ni \mathit{choices} \overset{Cs}{\rightsquigarrow} \mathit{code}\end{array}}{\Gamma \vdash \textbf{data}\ D\overrightarrow{[p : P]} : \textsc{Set}\ \textbf{where}\ \mathit{choices} \overset{D}{\rightsquigarrow} \Gamma[D \mapsto \lambda\vec{p}.\, \mu\, (\mathsf{call}\ \langle \mathbf{D}\ \vec{p} \rangle\ \mathit{code}) : \overrightarrow{(p : P')} \to \textsc{Set}]}\ (\textsc{Data})$$

(a) Elaboration of definition

$$\boxed{\Gamma \vdash \langle l \rangle \ni \mathit{choices} \overset{Cs}{\rightsquigarrow} \mathit{code}}$$

$$\frac{\begin{array}{cc}T \triangleright l & \Gamma \vdash \mathsf{EnumU} \ni \{t_i\} \overset{Chk}{\rightsquigarrow} E \\ \Gamma \vdash \langle l \rangle \ni c_i \overset{C}{\rightsquigarrow} [t_i \mapsto \mathit{code}_i] & \Gamma \vdash \mathsf{EnumT}\ E \to \mathsf{Desc} \ni \{t_i \mapsto \mathit{code}_i\} \overset{Chk}{\rightsquigarrow} T\end{array}}{\Gamma \vdash \langle l \rangle \ni T\ \mathfrak{z}\ c_0 | \ldots | c_n \overset{Cs}{\rightsquigarrow} \mathsf{return}\ E\ T}\ (\textsc{Choices})$$

(b) Elaboration of constructor choices

$$\boxed{\Gamma \vdash \langle l \rangle \ni c \overset{C}{\rightsquigarrow} [t \mapsto \mathit{code}]}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{UId} \ni \text{'}t \overset{Chk}{\rightsquigarrow} t' \\ \Gamma \vdash \langle l \rangle \ni \mathit{args} \overset{A}{\rightsquigarrow} \mathit{code}\end{array}}{\Gamma \vdash \langle l \rangle \ni t\ \mathit{args} \overset{C}{\rightsquigarrow} [t' \mapsto \mathit{code}]}\ (\textsc{Constructor})$$

(c) Elaboration of constructor

$$\boxed{\Gamma \vdash \langle l \rangle \ni \mathit{args} \overset{A}{\rightsquigarrow} \mathit{code}}$$

$$\frac{\Gamma \vdash \textsc{Set} \ni T \overset{Chk}{\rightsquigarrow} T' \qquad \Gamma, x : T' \vdash \langle l \rangle \ni \Delta \overset{A}{\rightsquigarrow} \mathit{code}_\Delta}{\Gamma \vdash \langle l \rangle \ni (x : T)\Delta \overset{A}{\rightsquigarrow} \text{'}\Sigma\ T'\ \lambda x.\ \mathit{code}_\Delta}\ (\textsc{Arg-Sig})$$

$$\frac{T \triangleright l \qquad \Gamma \vdash \langle l \rangle \ni \Delta \overset{A}{\rightsquigarrow} \mathit{code}_\Delta}{\Gamma \vdash \langle l \rangle \ni (x : T)\Delta \overset{A}{\rightsquigarrow} \text{'}\mathsf{var} \text{'}\times\ \mathit{code}_\Delta}\ (\textsc{Arg-Var})$$

$$\frac{\begin{array}{cc}\Gamma \vdash \textsc{Set} \ni T \overset{Chk}{\rightsquigarrow} T' & \Gamma, t : T' \vdash \langle l \rangle \ni \nabla \overset{A}{\rightsquigarrow} \mathit{code}_\nabla \\ & \Gamma \vdash \langle l \rangle \ni \Delta \overset{A}{\rightsquigarrow} \mathit{code}_\Delta\end{array}}{\Gamma \vdash \langle l \rangle \ni (f : (t : T) \to \nabla)\Delta \overset{A}{\rightsquigarrow} (\text{'}\Pi\ T'\ \lambda t.\ \mathit{code}_\nabla) \text{'}\times\ \mathit{code}_\Delta}\ (\textsc{Arg-Exp})$$

$$\frac{}{\Gamma \vdash \langle l \rangle \ni \epsilon \overset{A}{\rightsquigarrow} \text{'}1}\ (\textsc{Arg-End})$$

(d) Elaboration of arguments

$$\boxed{T \triangleright l}$$

$$\frac{}{D \triangleright \mathbf{D}}\ (\textsc{Match-Name}) \qquad \frac{T \triangleright l}{T\ p \triangleright l\ p}\ (\textsc{Match-Param})$$

(e) Matching label

Figure 6: Elaboration of inductive types

Example 6 (Elaborating Tree)**.** Applied to our example, we obtain:

$$\Gamma \vdash \textbf{data}\ \mathsf{Tree}(A : \textsc{Set}) : \textsc{Set}\ \textbf{where}\ [\mathit{choices}] \overset{D}{\rightsquigarrow} \Gamma[\mathsf{Tree} \mapsto \lambda A.\, \mu\, (\mathsf{call}\ \langle \textbf{Tree}\ A \rangle\ [\mathit{code}]) : A \to \textsc{Set}]$$

where

$$\mathit{choices} \triangleq \mathsf{Tree}\ A \ni \mathsf{leaf} \mid \mathsf{node}\ (l : \mathsf{Tree}\ A)(a : A)(r : \mathsf{Tree}\ A)$$
$$\mathit{code} \triangleq \mathsf{return} \left\{ \begin{array}{l} \text{'leaf}, \\ \text{'node} \end{array} \right\} \left\{ \begin{array}{l} \text{'leaf} \mapsto \text{'1}, \\ \text{'node} \mapsto \text{'var} \text{'}\times \text{'}\Sigma\ A\ \lambda\_.\ \text{'var} \text{'}\times \text{'1} \end{array} \right\}$$

Elaboration of constructor choices (Fig. 6b): The judgment reads as: in a context Γ, the sum of constructors *choices* defining the datatype l elaborates to a description *code*. To elaborate the choice of constructors, we elaborate each individual constructor – via rule (Constructor) – hence obtaining their respective constructor name and code. We then return the finite collection of constructor names and their corresponding codes. This elaboration step is subject to the soundness property:

Lemma 1. *If* $\left\{ \begin{array}{l} \Gamma \vdash l\ \text{datatel} \\ \Gamma \vdash \langle l \rangle \ni \mathit{choices} \overset{Cs}{\rightsquigarrow} \mathit{code} \end{array} \right.$ *, then* $\Gamma \vdash \mathit{code} : \langle l \rangle$

Example 7 (Elaborating Tree)**.** Applied to our example, we obtain:

$$\Gamma, A : \textsc{Set} \vdash \langle \textbf{Tree}\ A \rangle \ni [\mathit{choices}] \overset{Cs}{\rightsquigarrow} [\mathit{code}]$$

Where *choices* and *code* have been defined above.

Elaboration of constructor (Fig. 6c): The judgment reads as: in a context Γ, the constructor c defining a datatype l elaborates to a label t, the constructor name, and a description *code*. The role of this elaboration step is twofold. First, we extract the constructor name and elaborate it into a label – via type checking against Ulid. Second, we elaborate the arguments of that constructor – via any of the rules (Arg-Sig), (Arg-Var), (Arg-Exp), or (Arg-End)– hence obtaining a Desc code. We return the pair of the label and the arguments' code, subject to the following soundness property:

Lemma 2. *If* $\left\{ \begin{array}{l} \Gamma \vdash l\ \text{datatel} \\ \Gamma \vdash \langle l \rangle \ni c \overset{C}{\rightsquigarrow} [t \mapsto \mathit{code}] \end{array} \right.$, *then* $\left\{ \begin{array}{l} \Gamma \vdash t : \mathit{Ulid} \\ \Gamma \vdash \mathit{code} : \mathit{Desc} \end{array} \right.$

Example 8 (Elaborating Tree)**.** Since our datatype definition has two constructors, there are two instances of constructor elaboration:

$$\Gamma, A : \textsc{Set} \vdash \langle \textbf{Tree}\ A \rangle \ni \mathsf{leaf} \overset{C}{\rightsquigarrow} [\text{'leaf} \mapsto \text{'1}]$$
$$\Gamma, A : \textsc{Set} \vdash \langle \textbf{Tree}\ A \rangle \ni \mathsf{node}\ (l : \mathsf{Tree}\ A)(a : A)(r : \mathsf{Tree}\ A) \overset{C}{\rightsquigarrow} [\text{'node} \mapsto \text{'var} \text{'}\times \text{'}\Sigma\ A\ \lambda\_.\ \text{'var} \text{'}\times \text{'1}]$$

Elaboration of arguments (Fig. 6d): The judgment reads as: in a context Γ, a constructor's arguments *args* defining the datatype l elaborate to a description *code*. Intuitively, the arguments form a telescope of Σ-types, hence our translation to 'Σ and '× codes. The rules (Arg-Sig) and (Arg-Var) are non-deterministic: T could either be a proper type or a recursive call. In the first case, this maps to a standard 'Σ code, while in the second case, we must make sure that the recursive call is valid – via the judgment $T \triangleright l$ – and, if so, we generate a 'var code. We also support exponentials in the definitions, mapping them to the 'Π code – via rule (Arg-Exp). Once all arguments have been processed, we conclude by generating the '1 code – via rule (Arg-End). This translation is subject to the following soundness property:

Lemma 3. *If* $\begin{cases} \Gamma \vdash l \text{ datatel} \\ \Gamma \vdash \langle l \rangle \ni \mathit{args} \overset{A}{\rightsquigarrow} \mathit{code} \end{cases}$ *, then* $\Gamma \vdash \mathit{code} : \mathsf{Desc}$

Example 9 (Elaborating Tree)**.** Elaborating the arguments of the leaf constructor is trivial:

$$\frac{}{\Gamma, A : \textsc{Set} \vdash \langle \mathbf{Tree}\ A \rangle \ni \epsilon \overset{A}{\rightsquigarrow} \text{'1}}$$

As for the node constructor, we obtain its code through the following sequence of elaborations:

$$\dfrac{\mathsf{Tree}\ A \triangleright \mathbf{Tree}\ A \qquad \dfrac{\dfrac{\dfrac{}{\Gamma, A : \textsc{Set}, a : A \vdash \langle \mathbf{Tree}\ A \rangle \ni \epsilon \overset{A}{\rightsquigarrow} \text{'1}}}{\Gamma, A : \textsc{Set}, a : A \vdash \langle \mathbf{Tree}\ A \rangle \ni (r : \mathsf{Tree}\ A) \overset{A}{\rightsquigarrow} \text{'var} \times \text{'1}}}{\Gamma, A : \textsc{Set} \vdash \langle \mathbf{Tree}\ A \rangle \ni (a : A)(r : \mathsf{Tree}\ A) \overset{A}{\rightsquigarrow} \text{'}\Sigma\ A\ \lambda\_.\ \text{'var} \times \text{'1}}}{\Gamma, A : \textsc{Set} \vdash \langle \mathbf{Tree}\ A \rangle \ni (l : \mathsf{Tree}\ A)(a : A)(r : \mathsf{Tree}\ A) \overset{A}{\rightsquigarrow} \text{'var} \times \text{'}\Sigma\ A\ \lambda\_.\ \text{'var} \times \text{'1}}$$

Matching label (Fig. 6e): The judgment reads as: the type T matches the description label l. In rule (Choices) and rule (Arg-Var), we match the label l against a type T using the judgment $T \triangleright l$. Through this judgment, we spot the recursive arguments in the datatype definition, enforcing that the parameters are unchanged. Whilst this judgment has no impact on the soundness and completeness of the elaborator – we could have used a special symbol to denote recursive arguments – its role is crucial from an usability perspective: this definition style is more natural.

We can now prove the soundness of the whole translation: the elaboration of a datatype in a valid context Γ returns an extended context Δ that is valid. We formulate soundness as follow:

Theorem 3 (Soundness of elaboration)**.** *If* $\begin{cases} \Gamma \vdash \textsc{valid} \\ \Gamma \vdash \mathbf{data}\ D\overrightarrow{(p : P)} : \textsc{Set}\ \mathbf{where}\ \mathit{choices} \overset{D}{\rightsquigarrow} \Delta \end{cases}$*,*
then $\Delta \vdash \textsc{valid}$*.*

Proof. First, we prove Lemma 3 by induction on the list of arguments. We then obtain Lemma 2. By applying this lemma to all constructors, we obtain Lemma 1. The soundness theorem follows.

□

While our soundness theorem gives some hint as to the correctness of our specification, we could obtain a stronger result by proving an equivalence between Coq's `Inductive` definitions and the corresponding datatype declaration in our system. This equivalence amounts to proving the equivalence of the associated elimination forms, i.e. `Fixpoint` in Coq and induction in our system. However, since we do not know of any formal description of elimination principles generated from an `Inductive` definition, we shall use the simpler presentation given by Giménez (1995).

Lemma 4. *For any inductive definition* $\mathbf{Ind}(X : \textsc{Set})\langle \mathcal{C}_0 | \ldots | \mathcal{C}_n \rangle$ *in Coq, the corresponding inductive definition* **data** $X : \textsc{Set}\ \mathbf{where}\ X \ni \mathcal{C}_0 \mid \cdots \mid \mathcal{C}_n$ *in our system elaborates to a code* D *having an extensionally equal elimination principle.*

Proof. To prove this result, we compute the elaboration of a constructor form $\mathcal{C}$ (Definition 2.2, (Giménez, 1995)). This merely consists in applying the rule (Constructor): we denote $\lfloor \_ \rfloor$ the result of this elaboration step. We proceed by induction over the syntax of a constructor form and obtain:

$$\begin{aligned} \lfloor X \rfloor &\mapsto \text{'1} \\ \lfloor (x : M) \to \mathcal{C} \rfloor &\mapsto \text{'}\Sigma\ M\ \lambda x.\ \lfloor \mathcal{C} \rfloor \\ \lfloor (x : M \to X) \to \mathcal{C} \rfloor &\mapsto (\text{'}\Pi\ M\ \lambda\_.\ \text{'var}) \times \lfloor \mathcal{C} \rfloor \end{aligned}$$

We thus get a translation from Giménez's recursive type declarations to a code in our universe:

$$\lfloor \mathbf{Ind}(X : \textsc{Set})\langle \mathcal{C}_0 \mid \ldots \mid \mathcal{C}_n \rangle \rfloor \mapsto {}'\sigma\ n\ \{i \mapsto \lfloor \mathcal{C}_i \rfloor\}$$

Having done that, it is then a straightforward symbol-pushing exercise to prove that Coq's elimination rules (Section 3.1.1, Giménez (1995)) can be reduced to our generic elimination principle. The crux of the matter consists in showing that the minor premises – defined by $\mathcal{E}_1$ in that paper – are extensionally equivalent to the induction hypothesis $((d : [\![D]\!]\ (\mu D)) \to \mathsf{All}\ D\ (\mu D)\ P\ d \to P(\mathsf{in} d))$ in our system.

□

A corollary of this lemma amounts to the completeness of our syntax of datatypes, i.e. if we consider that inductive types are characterised by their elimination principle, our presentation is as expressive as Coq's presentation:

Theorem 4 (Completeness of elaboration)**.** *For an inductive type* $\mathbf{Ind}(X : \textsc{Set})\langle \mathcal{C}_0 \mid \ldots \mid \mathcal{C}_n \rangle$ *in Coq, any function introduced by a* `Fixpoint` *definition over* X *admits an extensionally equivalent definition in our system. Conversely, our generic elimination principle is accepted by Coq.*

Proof. The fact that our induction principle is accepted by Coq is a known result (Chapman et al., 2010). The other direction consists in proving that any `Fixpoint` definition can be implemented using our induction principle. To this end, we use Giménez reduction of `Fixpoint` definitions down to elimination rules. By Lemma 4, we have that Coq's elimination rules are equivalent to ours.

□

4. Reflections on Inductives

Having described our infrastructure to elaborate inductive definitions down to descriptions, we would like to give an overview of the possibilities offered by such a system. Indeed, in a purely syntactic presentation of inductives, we are stuck at the meta-level of the type theory: if we want to provide support to manipulate inductive types, it must be implemented as part of the theorem prover, out of the type theory. Alternatively, a quoting/unquoting mechanism could be provided but guaranteeing the safety of such an extension is likely to be tricky.

Because our type theory reflects inductives in itself, the meta-theory of inductive types is no more than a universe. What used to be meta-theoretical constructions can now be implemented from within the type theory, benefiting from the various amenities offered by a dependently-typed programming language. By adequately extending the elaboration machinery, the user would be given a convenient and high-level syntax to refer to these type-theoretic constructions. In this section, we present two examples of such "reflection on inductives". Our first example, reflecting constructions on constructors (McBride et al., 2004), will appeal to the implementers: we hint at the possibility of implementing key features of the type theory within itself, a baby step toward bootstrapping. Our second example, providing a user defined `deriving` mechanism, should appeal to programmers: we illustrate how programmers could provide generic operations over datatypes and see them automatically integrated in their development.

4.1. A few constructions on constructors, internalized

McBride et al. (2004) describe a collection of lemmas that theorem prover's implementer would like to export with every inductive type. In that paper, the authors first show how one can reduce case analysis and course-of-value recursion to standard induction. Then, they describe two lemmas over

datatype constructors: *no confusion* – constructors are injective and disjoint – and *acyclicity* – we can automatically disprove equalities of the form $x = t$ where x appears constructor-guarded in t.

However, since this paper works on the syntactic form of datatype definitions, it is rife with "..." definitions. For instance, the authors reduce case analysis to induction with no less than ten ellipsis in the construction. In our system, we generically derive case analysis by a mere definition *within the type theory*. To do so, we simply ignore the induction hypothesis from the generic induction principle:

$$\begin{aligned}&\mathsf{case}\ (D : \mathsf{Desc})(P : \mu D \to \textsc{Set})(\mathit{cases} : ((d : [\![D]\!]\ (\mu D)) \to P(\mathsf{in}\, d)))(x : \mu D)\ :\ P x\\ &\mathsf{case}\ D\ P\ \mathit{cases}\ x\ \mapsto\ \mathsf{induction}\ D\ P\ (\lambda d.\, \lambda\_.\ \mathit{cases}\ d)\ x\end{aligned}$$

Similarly, the authors specify and prove the no confusion lemma over the skeleton of an inductive definition. In our system, this result is internalised through two definitions. In the following, we will assume that D is a tagged description, i.e. $D = {'\sigma}\ E\ T$ where E is the finite sets of constructor labels. An inhabitant of $\mu\, D$ is therefore a pair $\mathsf{in}\,(c, a)$ with c representing the constructor name and a the tuple of arguments. The NoConfusion lemma states that two (equal) terms $x, y : \mu\, D$ must be the same constructor (second case, asking the impossible) and their arguments must be equal (first case):

$$\begin{aligned}&\mathsf{NoConfusion}\quad (x : \mu D)\quad (y : \mu D)\quad :\ \textsc{Set}_1\\ &\mathsf{NoConfusion}\ (\mathsf{in}\,(c_x, a_x))\ (\mathsf{in}\,(c_y, a_y))\ \mid\ \mathsf{decideEq\text{-}EnumT}\ c_x\ c_y\\ &\mathsf{NoConfusion}\ (\mathsf{in}\,(c_x, a_x))\ (\mathsf{in}\,(c_x, a_y))\ \mid\ \quad \mathsf{equal}\ \mathsf{refl}\quad \mapsto\ (P : \textsc{Set}) \to (a_x \equiv a_y \to P) \to P\\ &\mathsf{NoConfusion}\ (\mathsf{in}\,(c_x, a_x))\ (\mathsf{in}\,(c_y, a_y))\ \mid\ \quad \mathsf{not\text{-}equal}\ q\quad \mapsto\ (P : \textsc{Set}) \to P\end{aligned}$$

The proof of this lemma consists simply in deciding whether the constructor tag are equal or not, hence discriminating the constructors and deconstructing the equality proof:

$$\begin{aligned}&\mathsf{noConfusion}\quad (x : \mu D)\quad (y : \mu D)\quad (q : x \equiv y)\ :\ \mathsf{NoConfusion}\ x\ y\\ &\mathsf{noConfusion}\ (\mathsf{in}\,(c_x, a_x))\ (\mathsf{in}\,(c_y, a_y))\quad q\quad \mid\ \mathsf{decideEq\text{-}EnumT}\ c_x\ c_y\\ &\mathsf{noConfusion}\ (\mathsf{in}\,(c_x, a_x))\ (\mathsf{in}\,(c_x, a_y))\quad q\quad \mid\ \quad \mathsf{equal}\ \mathsf{refl}\quad \mapsto\ \lambda P.\, \lambda \mathit{rec}.\ \mathit{rec}\ q\\ &\mathsf{noConfusion}\ (\mathsf{in}\,(c_x, a_x))\ (\mathsf{in}\,(c_y, a_y))\quad q\quad \mid\ \quad \mathsf{not\text{-}equal}\ \mathit{neq}\quad \mapsto\ \lambda P.\, \mathbb{0}\text{-}\mathsf{elim}\ (\mathit{neq}\ q)\end{aligned}$$

At this stage, we have proved this lemma generically, for all tagged descriptions. Hence, after defining a new datatype, a user can directly use this lemma on her definition. For convenience, a subsequent elaboration phase should specialise this lemma to the datatype being defined.

4.2. Deriving operations on datatypes

Another possible extension of our system is a generic `deriving` mechanism. In the Haskell language, we can write a definition such as

$$\begin{aligned}&\textbf{data}\ \mathsf{Nat} : \textsc{Set}\ \textbf{where}\\ &\quad \mathsf{Nat}\ \ni\ 0\\ &\qquad \mid\ \mathsf{suc}\,(n : \mathsf{Nat})\\ &\quad \textbf{deriving}\ \mathsf{Eq}\end{aligned}$$

that automatically generates an equality test for the given datatype. Again, since datatypes are a meta-theoretical entity, this deriving mechanism has to be provided by the implementer and, template programming aside, they cannot be implemented by the programmers themselves.

In our framework, we could extend the elaborator for datatypes with a `deriving` mechanism. However, for such a mechanism to work, we must restrict ourselves to decidable properties: for example, if the user asks to derive equality on a datatype that does not admit a decidable equality (e.g. Brouwer ordinals), this should fail immediately. To solve this issue, we add one level of indirection: while we cannot decide equality for *any* datatype, we can decide whether a datatype belongs to a sub-universe *for which* equality is decidable. Hence, to introduce a derivable property P in the type theory, the programmer would populate the following record structure:

$$\begin{aligned}&\mathsf{Derivable}\,(P : \mathsf{Desc} \to \textsc{Set})\ :\ \textsc{Set}_1\\ &\mathsf{Derivable}\ P\ \mapsto \begin{cases} \mathsf{subDesc} & : \mathsf{Desc} \to \textsc{Set}_1\\ \mathsf{membership} & : (D : \mathsf{Desc}) \to \mathsf{Decidable}\,(\mathsf{subDesc}\ D)\\ \mathsf{derive} & : \mathsf{subDesc}\ D \to P\ D \end{cases}\end{aligned}$$

For example, in the case of equality, the programmer has first to provide a function eqDesc : Desc $\to$ Set$_1$. One possible (perhaps simplistic, but valid) sub-universe consists only of products, finite sums, recursive call, and unit: it is enough to describe natural numbers and variants thereof. She then implements a procedure membershipEq : $(D : \mathsf{Desc}) \to \mathsf{Decidable}\,(\mathsf{eqDesc}\ D)$ deciding whether a given Desc code fits into this sub-universe or not. Recall that Decidable A corresponds to $A + \neg A$. It should be clear that the membership of a Desc code to our sub-universe of finite products and sums is decidable. Finally, she implements the key operation deriveEq : eqDesc $D \to (x\ y : \mu\ D) \to \mathsf{Decidable}\,(x \equiv y)$ that decides equality of two objects, assuming that they belong to the sub-universe. This implements the structure Eq : Derivable $(\lambda D.\,(x\ y : \mu\ D) \to \mathsf{Decidable}\ x \equiv y)$.

While elaborating a datatype, it is then straightforward – and automatic – for us to generate its derivable property, or reject it immediately: we simply compute membership on the specific code. If we obtain a negative response, we report an error. If we obtain a positive witness, we pass that witness to derive and instantiate the property. For example, since natural numbers fit into the eqDesc universe, the elaboration machinery would automatically generate the following decision procedure

$$\begin{array}{lcl} \mathsf{Nat\text{-}eq}\ (x\ y : \mathsf{Nat}) & : & \mathsf{Decidable}\ x \equiv y \\ \mathsf{Nat\text{-}eq} \quad x\ y & \mapsto & \mathsf{deriveEq}\ (\mathsf{witness}\ (\mathsf{membershipEq}\ \mathsf{NatD})\ *) \end{array}$$

without any input from the user but the `deriving` Eq clause. Here, witness is a library function that extracts the witness from a true decidable property: applied to NatD, the function membershipEq computes a positive witness that we can simply extract. Again, this calls for an extension of elaboration, supporting such a `deriving` mechanism.

5. Conclusion

In this paper, we have striven to give a coherent framework for elaboration in type theory. We have organised our system around two structuring ideas. First, by using the flow of typing information, we obtain a richer and more flexible term language. Second, by using types as presentation of high-level concepts, such as inductive definition, we can effectively guide the elaboration process. This technique is conceptually simple and therefore amenable to formal reasoning. This simplicity together with the soundness proofs should convince the reader of its validity.

From there, we believe that reasoning on inductive definitions can be liberated from the elusive ellipsis: proofs and constructions on inductives ought to happen within the type theory itself. After Harper and Stone (2000), we claim that if the treatment of datatypes is conceptually straightforward then it ought to be technically straightforward and implemented as a generic program in the type theory. For non-straightforward properties, our results should be reusable across calculi – such as the Calculus of Inductive Constructions – and not too rigidly tied to our universe of datatypes. Besides, we were careful to present elaboration as a relation rather than a mere program, making it more amenable to abstract reasoning.

We also had a glimpse at two possible extensions of the elaboration process. While we did not formalise these examples, our expectations seem reasonable and our experience modeling them in Agda further supports this impression. We have seen how generic theorems on inductive types can be internalised as generic programs: besides the benefit of reducing the trusted computing base, their validity is guaranteed by type checking. Also, we have presented a generic `deriving` mechanism: with no extension to the type theory, we are able to let the user define sub-universes that support certain operations. These operations could then be automatically specialised to the datatypes that support them, without any user intervention.

Future work Due to space restriction, we have focused our presentation on inductive types: our elaboration machinery can nonetheless be extended to deal with inductive families. An interesting extension of the elaborator would be to support functions, such as in the definition of μ that calls into $[\![\_]\!]$ ($\mu\, D$): this kind of definition is rejected by Coq and has to be manually coded in our system. Another challenge would be to internalise the elaboration process *itself* in type theory, hence obtaining a correct-by-construction translation.

Acknowledgements We are very grateful to the anonymous reviewers, their comments were extremely valuable. We thank Pierre Boutillier for pointing us to the relevant literature on Coq's treatment of inductives. We also thank our colleagues Guillaume Allais and Stevan Andjelkovic for many stimulating discussions and for their input on this paper. The authors are supported by the Engineering and Physical Sciences Research Council, Grant EP/G034699/1.

References

T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In *PLPV*, 2007.

A. Asperti, W. Ricciotti, C. S. Coen, and E. Tassi. A Bi-Directional refinement algorithm for the calculus of (Co)Inductive constructions. 2012. URL `http://arxiv.org/abs/1202.4905`.

M. Benke, P. Dybjer, and P. Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 2003.

E. Brady, J. Chapman, P.-E. Dagand, A. Gundry, C. McBride, P. Morris, and U. Norell. An Epigram implementation. URL `http://www.e-pig.org/`.

J. Chapman, P.-E. Dagand, C. McBride, and P. Morris. The gentle art of levitation. In *ICFP*, pages 3–14, 2010.

P.-E. Dagand and C. McBride. Transporting functions across ornaments. In *ICFP*, pages 103–114, 2012.

E. Giménez. Codifying guarded definitions with recursive schemes. In *Types for Proofs and Programs*, volume 996, chapter 3, pages 39–59. 1995.

R. Harper and C. Stone. A Type-Theoretic interpretation of standard ML. In *Proof, Language, and Interaction: essays in honour of Robin Milner*, 2000.

Z. Luo. *Computation and Reasoning*. Oxford University Press, 1994.

C. McBride and J. McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.

C. McBride, H. Goguen, and J. McKinna. A few constructions on constructors. In *Types for Proofs and Programs*, pages 186–200, 2004.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

B. C. Pierce and D. N. Turner. Local type inference. In *POPL*, 1998.

The Coq Development Team. *The Coq Proof Assistant Reference Manual*.