A Syntax for Mutual Inductive Families

Ambrus Kaposi
Eötvös Loránd University, Budapest, Hungary
akaposi@inf.elte.hu

Jakob von Raumer
University of Nottingham, UK
jakob@von-raumer.de

Abstract
Inductive families of types are a feature of most languages based on dependent types. They are usually described either by syntactic schemes or by encodings of strictly positive functors such as combinator languages or containers. The former approaches are informal and give only external signatures, the latter approaches suffer from encoding overheads and do not directly represent mutual types.

In this paper we propose a direct method for describing signatures for mutual inductive families using a domain-specific type theory. A signature is a context (roughly speaking, a list of types) in this small type theory. Algebras, displayed algebras and sections are defined by models of this type theory: the standard model, the logical predicate and a logical relation interpretation, respectively. We reduce the existence of initial algebras for these signatures to the existence of the syntax of our domain-specific type theory. As this theory is very simple, its normal syntax can be encoded using indexed W-types. To the best of our knowledge, this is the first formalisation of the folklore fact that mutual inductive types can be reduced to indexed W-types.

The contents of this paper were formalised in the proof assistant Agda.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases type theory, inductive types, mutual inductive types, W-types, Agda

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.23

Supplementary Material All of the results were formalised in the proof assistant Agda, the source code is available online at https://bitbucket.org/javra/inductive-families/src/master/agda.

Funding Ambrus Kaposi: I acknowledge the support of the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme funding scheme, Project no. ED_18-1-2019-0030 (Application-specific highly reliable IT solutions), and of the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunication). Jakob von Raumer: I acknowledge the support of COST Action EUTypes CA15123.

1 Introduction

Programming languages based on type theory rely heavily on easy, flexible and sound ways to define new data types. Usually, type theories allow for the definition of inductive types, which are defined by giving a list of constructors which generate the elements of the type. One prime example for such an inductive type is the type of natural numbers \(N \triangleq \text{Set}\) which is generated by the zero constructor \(0 : \mathbb{N}\) and the successor function \(S : \mathbb{N} \rightarrow \mathbb{N}\). Besides these plain inductive types, dependent type theories often make use of inductive families of types (also called indexed inductive types) where, instead of just a type we define a type family over a previously defined type. This enables us for example to define the type of vectors of a type \(A\) as a family \(\text{Vec} : \mathbb{N} \rightarrow \text{Set}\), by a constructor for the empty vector \(\text{nil} : \text{Vec}0\) and \(\text{cons} : A \rightarrow (n : \mathbb{N}) \rightarrow \text{Vec}n \rightarrow \text{Vec}(S\, n)\). Besides inductive families, another recurring need
is the one for mutual definitions: Often, we want to define more than one inductive type simultaneously with constructors referring to any of these types. For example we might want to obtain the predicates of a natural number being even and odd in reference to each other by defining

\[
\begin{align*}
isEven &: \mathbb{N} \to \text{Set}, \\
isOdd &: \mathbb{N} \to \text{Set}
\end{align*}
\]

by constructors

\[
\begin{align*}
even0 &: \text{isEven } 0, \\
evenS &: (n : \mathbb{N}) \to \text{isOdd } n \to \text{isEven } (Sn), \text{ and} \\
oddS &: (n : \mathbb{N}) \to \text{isEven } n \to \text{isOdd } (Sn).
\end{align*}
\]

Syntaxes of programming languages usually also consist of mutually given inductive types, such as expressions (indexed by their types), commands, blocks, etc. We call these types mutual inductive families.

There is a folklore trick to reduce mutual inductive families to inductive families. For example, \text{isEven}-\text{isOdd} can be simulated by a single family indexed over an extra boolean which says which sort is meant: \text{isEven}? : \text{Bool} \to \mathbb{N} \to \text{Set}. Now \text{isEven} is simulated by \text{isEven}?\text{true} and \text{isOdd} by \text{isEven}?\text{false}. To show that this technique works for every mutual inductive family, we first have to provide a general definition for mutual inductive families.

There is a folklore trick to reduce mutual inductive families to inductive families. For example, \text{isEven}-\text{isOdd} can be simulated by a single family indexed over an extra boolean which says which sort is meant: \text{isEven}? : \text{Bool} \to \mathbb{N} \to \text{Set}. Now \text{isEven} is simulated by \text{isEven}?\text{true} and \text{isOdd} by \text{isEven}?\text{false}. To show that this technique works for every mutual inductive family, we first have to provide a general definition for mutual inductive families.

The description of inductive families was Peter Dybjer’s external scheme [17]. He extended type theory with new derivation rules for inductive families and their constructors, elimination principles and computation rules. His approach does not allow internal manipulation of signatures and it can only be formalised as an extension of a pre-existing syntax of type theory, however it covers mutual inductive families as well.

Another popular method is the functorial approach: strictly positive functors are encoded either using a combinator calculus [15] or using indexed containers [4]. An algebra of such a functor \(F\) is given by a family \(X\) and a morphism \(FX \to X\), the initial algebra is given by the least fixpoint of the functor. The codes for the functors can be expressed internally allowing generic programming with signatures. A powerful application of this method is the automatic derivation of substitution laws for syntaxes with binders [2]. The drawback of the functorial approach is its encoding overhead – mutual types have to be transformed to indexed types, separate constructors have to be given as single families and in uncurried forms. The indexed container encoding, while being very concise, also relies on function extensionality.

E.g. without assuming function extensionality, there are many different, unequal constructors for zero [6, Section 2.1]. These constructors cannot be made definitionally equal even in the presence of function extensionality – they contain definitionally unequal \(\bot \to \mathbb{N}\) functions.

In this paper we aim to formalise mutual inductive families in a direct way, in the spirit of the original Dybjer definition. Drawing inspiration from the syntax of signatures for quotient inductive-inductive types (QIITs) and higher inductive-inductive types (HIITs) given by Kaposi, Kovács and Altenkirch [27, 26], we define signatures for mutual inductive families using the syntax of a small type theory tailor made for this purpose. We call this type theory the theory of signatures. A signature is a context in the theory of signatures, that is, roughly, a list of types. For example, the signature of natural numbers is given by the context \((N : \text{Set}, \theta : N, S : N \to N)\), where \(N, \theta\) and \(S\) are simply variable names. The rules for the theory of signatures enforce that we can only write strictly positive constructors.

This syntax allows us to write down the definition of an inductive family in the same way as it would look like in a theorem prover like Agda [30], Lean [16], or Coq [9].
The syntax for the theory of signatures can be internalised in type theory but it can also be seen as an external type theory in which one can describe signatures. We will present our syntax internally to a type theory, define its semantics and show that all mutual inductive families can be reduced to indexed W-types. All of the results were formalised in the proof assistant Agda, the source code is available online.

Contributions and structure
This paper contributes the following to the literature on inductive types.

- A syntax for mutual inductive families in which signatures can be defined in a direct way, simply by listing the types of sorts and constructors (Section 2). This syntax can be encoded by indexed W-types.
- Semantics for each signature: notions of algebras, displayed algebras and sections (Section 3). These explain what it means that an inductive type specified by a signature exists. The computation rules are specified as propositional (rather than definitional) equalities.
- An extension of the theory of signatures to a full substitution calculus (Section 4.1).
- A proof that each mutual inductive type can be constructed from the theory of signatures (Section 4.2), and as a by-product, a proof that mutual inductive families can be reduced to W-types. The reduction only justifies propositional computation rules.

Related work
As mentioned earlier, schemes for inductive types can be categorised into (1) external schemes, (2) internal combinatorial or (3) internal semantic schemes. Our approach is between (1) and (2). It compares to (2) as lambda-calculus compares to combinatory logic. To illustrate the difference, we list the signature for natural numbers in all approaches. (1) Dybjer [17] defines natural numbers by the formation rule $N : set$ and introduction rules $0 : N$ and $s : (u : N).N$. Our syntax will encode the same information by a sort context $(\cdot ⊢ U)$ and a point context $(\cdot ⊢ \text{El}(\text{var vz}) ⊢ \text{var vz} ⊢_p \text{El}(\text{var vz})$. The difference in encoding is that we use de Bruijn indices instead of variable names and El when decoding an index to a type (but not on the left hand side of the arrow $⇒_p$, see later). (2) In [15, 2], natural numbers are specified by $\sigma \text{Bool}(\lambda b. \text{if } b \text{ then } \square \text{tt else } \square \text{tt})$. The two constructors are encoded as one constructor with a Bool parameter. When this is true (zero case), there are no more parameters (denoted by $\square$), when it is false (successor) there is one recursive argument signified by $X$. The tts are necessary because the type of natural numbers does not have indices. (3) The container representation [4] of natural numbers is given by the type $\text{Bool}$ (expressing that there are two constructors) and a family of sets over $\text{Bool}$, $\lambda b. \text{if } b \text{ then } \bot \text{ else } \top$ expressing that the first constructor has zero and the second constructor has one recursive argument. We list the related work categorised as above.

(1) External syntactic schemes similar to the Dybjer’s were used to describe mutual inductive families of Coq on paper [32] and inside Coq [8], inductive-recursive types [18], subsets of higher inductive types [11, 19, 14], and inductive and coinductive types [10].

(2) Internal combinatorial schemes are defined by Benke, Dybjer and Jansson [12] for different classes of inductive types for the purpose of generic programming. Their signatures can be seen as uncurried versions of our signatures with some encoding overhead. In addition to our signatures, they separate the cases of parametrised and indexed definitions, while

---

1 https://bitbucket.org/javra/inductive-families/src/master/agda
we only have indexed ones and they also cover infinitary constructors. They also feature
iterated signatures, while we only model those using the function space with metatheoretic
domain. [20, 21] use combinator languages to axiomatise inductive-recursive types and
indexed inductive-recursive types, respectively. The same technique was used to describe
inductive-inductive types [29] and inductive families [15, 2].

(3) Internal semantic schemes: Containers for describing signatures of W-types were
introduced in [1] and extended to indexed W-types (potentially infinitary inductive families)
in [4] and QW-types (allowing equality constructors) [22]. In fact, indexed W-types were
introduced as “tree sets” much earlier, by Petersson and Synke [33]. The more semantic
treatments of higher inductive types [28] and quotient inductive-inductive types [3] don’t
provide schemes for the allowed constructors.

The direct inspiration of our work are the domain-specific type theories for describing
higher inductive-inductive [25] and quotient inductive-inductive signatures [27]. The latter
also derives all QIITs from a theory of QIIT signatures. Note that the analogous result in
our paper is not a consequence of the result of [27]. We use a similar proof, however we
have a much weaker assumption: we derive all mutual inductive families from the theory
of mutual inductive family signatures, instead of the theory of QIIT signatures. Moreover,
we also show how to reduce our weaker theory of signatures to indexed W-types. Such a
reduction is not done in [27], and is probably not possible for the theory of QIIT signatures.

Notation and metatheory

Throughout the paper, we will assume that we are given a type theory with a hierarchy
of universes Set, (we omit the indices for readability), Π-types, Σ-types, unit type 1, pro-
positional equality \(-\equiv-\), and indexed W-types [4] (see Appendix A). We write Σ-types
as \((x : A) \times B\) and Π-types as \((x : A) \rightarrow B\) where \(B\) might refer to \(x\). We write implicit
arguments in curly braces \(\{x : A\} \rightarrow B\) or simply omit them. Definitional equality is denoted
\(-\equiv-\). We presume that the type theory is extensional, that is, given a term \(t : u = v\), we
have \(u \equiv v\). It is expected that all definitions could be translated to intensional type theory
with the necessary coercions and transports following Hofmann’s translation [23, 31, 35]. In
the Agda formalisation we use explicit transports and rewrite rules occasionally as a limited
version of equality reflection. We also assume function extensionality, this is necessary to
handle the Π-types in our syntax with metatheoretic domain (\(\hat{\Pi}_s, \hat{\Pi}_p\)). In the formalisation
we do not use uniqueness of identity proofs and we conjecture that our usages of equality
reflection do not imply it.

2 Signatures for Mutual Inductive Families

In this section we define a syntax for a small type theory for describing signatures of mutual
inductive families. We call this the theory of signatures. The idea is that a signature is a
context in this theory, starting with the declaration of the sorts as functions into the universe
\(U\), then listing the constructors for the sorts in any order. We call these point constructors
following [34]. This theory is much simpler than the full syntax of dependent type theory. For
example, there are no interdependencies between sorts, neither between point constructors,
and no references from sorts to point constructors. We reflect these properties in our syntax
by separating sort contexts \(\text{Con}_s\) from point contexts \(\text{Con}_p\), and the latter will be indexed
over the former. We define an intrinsically typed syntax (in the style of [7, 5]), that is, we
don’t have preterms or typing relations, only well-scoped, well-typed terms, well-formed
contexts and types.
Definition 1 (The Theory of Signatures). The syntax is defined inductively by the following 6 sorts and 13 constructors. These 6 types can be encoded as indexed W-types, the detailed construction of which is provided in Appendix A.

\[
\begin{align*}
\text{Ty}_s & : \text{Set} & \text{Ty}_{p} & : \text{Con}_s \rightarrow \text{Set} \\
U & : \text{Ty}_s & \text{El} & : \text{Tm}_s \Gamma_s U \rightarrow \text{Ty}_p \Gamma_s \\
\hat{\Pi}_s & : (T : \text{Set}) \rightarrow (T \rightarrow \text{Ty}_s) \rightarrow \text{Ty}_s & \hat{\Pi}_p & : (T : \text{Set}) \rightarrow (T \rightarrow \text{Ty}_p \Gamma_s) \rightarrow \text{Ty}_p \Gamma_s \\
\text{Con}_s & : \text{Set} & - \Rightarrow - & : \text{Tm}_s \Gamma_s U \rightarrow \text{Ty}_p \Gamma_s \rightarrow \text{Ty}_p \Gamma_s \\
\cdot & : \text{Con}_s & \text{Con}_p & : \text{Con}_s \rightarrow \text{Set} \\
\rightarrow - & : \text{Con}_s \rightarrow \text{Ty}_s \rightarrow \text{Con}_s & \cdot & : \text{Con}_p \Gamma_s \\
\text{Var}_s & : \text{Con}_s \rightarrow \text{Ty}_s \rightarrow \text{Set} & - \Rightarrow - & : \text{Con}_p \Gamma_s \rightarrow \text{Ty}_p \Gamma_s \rightarrow \text{Con}_p \Gamma_s \\
\text{vs} & : \text{Var}_s \Gamma_s A_s \rightarrow \text{Var}_s (\Gamma_s \triangleright B_s) A_s & \text{Tm}_s & : \text{Con}_s \rightarrow \text{Ty}_s \rightarrow \text{Set} \\
\text{var} & : \text{Var}_s \Gamma_s A_s \rightarrow \text{Tm}_s \Gamma_s A_s & - \circ - & : \text{Tm}_s \Gamma_s (\hat{\Pi}_s T A_s) \rightarrow (\tau : T) \rightarrow \text{Tm}_s \Gamma_s (A_s \tau)
\end{align*}
\]

A sort type $\text{Ty}_s$ is either a universe $U$ or is given by an indexing type $T$ and a sort type for each element of $T$. The latter can be seen as a function space where the domain is metatheoretic, hence the notation $\hat{\Pi}_s$. We use the abbreviation $T \Rightarrow_s A_s$ for $\hat{\Pi}_s T (\lambda \tau.A_s)$ when $A_s : \text{Ty}_s$. A sort context $\text{Con}_s$ is simply a snoc-list of sort types (empty context $\cdot$ and context extension $\triangleright -$). In order to refer to sorts we introduce typed de Bruijn variables $\var{var}$ with zero $\var{vs}$ and successor $\var{vs}$ constructors. Just as variables, sort terms $\text{Tm}_s$ are indexed by a sort context and a sort type. Each variable is a term $\var{var}$ and we have application $- \circ -$ for the function space $\hat{\Pi}_s$. Note that $t : \text{Tm}_s \Gamma_s A_s$ carries similar information to $\Gamma_s \vdash t : A_s$ in a presentation of a syntax with preterms and typing relations, but we do not have preterms, only well-typed terms.

Point constructors are represented by point types $\text{Ty}_p$ over a given sort context. The context formers are the element type for the universe $U$, a function type with metatheoretic domain $\hat{\Pi}_p$ and a non-dependent function type $- \Rightarrow_p -$ where the domain is in $U$. The former function type allows adding parameters to constructors, the latter allows adding recursive arguments. We use the abbreviation $T \Rightarrow_p A_p$ for $\hat{\Pi}_p T (\lambda \tau.A_p)$ when $A_p : \text{Ty}_p$. A point context over a given sort context is a snoc-list of point types all in the same sort context.

Example 2 (Natural Numbers, Vectors, Parity). A common example for inductive types, the natural numbers, with one constructor for zero and one for the successor function, are represented by the following sort and point contexts. On the right hand side, we write the same with an informal notation using variable names.

\[
\begin{align*}
N_s & := (\cdot \triangleright U) & (N : U) \\
N & := (\cdot \Rightarrow \text{El} (\var{var}) \triangleright \var{var} \Rightarrow_p \text{El} (\var{var})) & (\text{zero} : N, \text{suc} : N \rightarrow N)
\end{align*}
\]

The only sort is referred to by $\var{var}$. As shown by the constructor for successor, on the left hand side of the arrow $\Rightarrow_p$ we have to write a sort term of type $U$, and not a point type. This makes sure that all constructors are strictly positive, as the only ways to form sort terms are variables and applications.
An example of a real indexed type is the type family of vectors of a fixed type $A : \text{Set}$. We also assume that we have natural numbers in our metatheory.

$$V_s :\equiv (\vdash N \Rightarrow_s U) \quad \quad (\text{Vec} : N \rightarrow U)$$

$$V :\equiv (\vdash \text{El} (\text{var vz} @ 0) \vdash) \quad \quad (\text{nil} : \text{Vec} 0, \quad \quad \text{cons} : A \rightarrow (n : N) \rightarrow \text{Vec} n \rightarrow$$

$$\text{El} (\text{var vz} a(n + 1))) \quad \quad \text{Vec} (n + 1))$$

As our sort has a function type, whenever we have to refer to it in constructors, we have to use the application $@$ to specify the natural number index. In the $\text{cons}$ constructor, we use both kinds of function types: the first two function types are $\hat{\Pi}_{P} A$ as they refer to the parameters of type $A$ and $N$. The last function type is $\Rightarrow_p$ as it refers to a recursive argument.

We revisit the parity example from the introduction.

$$P_s :\equiv (\vdash N \Rightarrow_s U \Rightarrow N \Rightarrow_s U) \quad \quad (\text{isEven} : N \rightarrow U, \text{isOdd} : N \rightarrow U)$$

$$P :\equiv (\vdash \text{El} (\text{var (vs vz) @ 0}) \vdash) \quad \quad (\text{even} 0 : \text{isEven} 0, \quad \quad \text{even} S : (n : N) \rightarrow \text{isOdd} n \rightarrow$$

$$\text{El} (\text{var (vs vz) a(n + 1))) \vdash_{P} \quad \quad \text{isEven} (n + 1), \quad \quad \text{odd} S : (n : N) \rightarrow \text{isEven} n \rightarrow$$

$$\text{El} (\text{var vz a(n + 1))) \vdash_{P} \quad \quad \text{isOdd} (n + 1))$$

The sort context $P_s$ has length two, we refer to the $\text{isEven}$ sort by $\text{var (vs vz)}$, to the $\text{isOdd}$ sort by $\text{var vz}$.

### 3. Algebras, Displayed Algebras, and Sections

In this section we provide semantics for the theory of signatures (Definition 1). A signature is given by a sort context $\Gamma_s$ and a point context $\Gamma : \text{Con}_P \Gamma_s$. For each such signature, we will obtain notions of algebras, displayed algebras and sections of displayed algebras. From the signature for natural numbers given in Example 2 we will derive that a natural number algebra is an element of

$$(N : \text{Set}) \times N \times (N \rightarrow N),$$

a displayed natural number algebra over an algebra $(N, z, s)$ is an element of

$$(P : N \rightarrow \text{Set}) \times P z \times ((n : N) \rightarrow P n \rightarrow P (s n)),$$

and a section of a displayed algebra $(P, w, h)$ over $(N, z, s)$ is an element of

$$(f : (n : N) \rightarrow P n) \times (f z = w) \times ((n : N) \rightarrow f (s n) = h n (f n)).$$

The constructors of the inductive type will be elements of the algebra, the arguments of the eliminator (sometimes called motives and methods) form a displayed algebra over the constructors, while the eliminator itself is a section. The equalities in the section are the computation rules ($\beta$ rules) for the eliminator.
More formally, we will define operations $\overline{A}$, $\overline{D}$ and $\overline{S}$ for computing algebras, displayed algebras and sections. As sort and point contexts are separate, we have to define them separately for both.

\[
\begin{align*}
\Gamma^A_s : \text{Set} & \quad \Gamma^D_s : \Gamma^A_s \rightarrow \text{Set} & \quad \Gamma^S_s : (\gamma_s : \Gamma^A_s) \rightarrow \Gamma^D_s \gamma_s \rightarrow \text{Set} \\
I^A : \Gamma^A_s \rightarrow \text{Set} & \quad I^D : \Gamma^D_s \gamma_s \rightarrow I^A \gamma_s \rightarrow \text{Set} & \quad I^S : \Gamma^S_s \gamma_s \gamma^d_s \rightarrow (\gamma : I^A \gamma_s) \rightarrow \\
& & \quad (\Gamma^D_s \gamma^d_s \gamma_s) \rightarrow \text{Set}
\end{align*}
\]

Putting them together, we get algebras as $(\gamma_s : \Gamma^A_s) \times I^A \gamma_s$, displayed algebras over a $(\gamma_s, \gamma)$ by $(\gamma^d_s : \Gamma^D_s \gamma_s) \times I^D \gamma^d_s \gamma_s$, and sections of $(\gamma^d, \gamma^d)$ by $(\gamma_s : \Gamma^S_s \gamma_s \gamma^d_s) \times I^S \gamma^d_s \gamma^d \gamma^d$.

The algebra operator corresponds to building the standard model (set model, metacircular interpretation [24, 5]) of the theory of signatures.

**Definition 3 (Algebra Operation).** We map sort types and sort contexts to types, variables and terms are mapped to functions from the interpretation of their context to the interpretation of their types, point types and point contexts are mapped to families over the interpretation of the sort contexts.

\[
\begin{align*}
\ast^A : \text{Ty}_A \rightarrow \text{Set} & \quad \ast^A : \text{Var}_s \Gamma^A_s \rightarrow \Gamma^A_s \rightarrow A^A \\
\ast^A : \text{Con}_s \rightarrow \text{Set} & \quad \ast^A : \text{Tm}_s \Gamma^A_s \rightarrow \Gamma^A_s \rightarrow A^A \\
\ast^A : \text{Tm}_p \Gamma^A_s \rightarrow \Gamma^A_s \rightarrow A^A
\end{align*}
\]

We go through each operation in order. First, sort types are interpreted as functions into the universe (left column), and sort contexts become iterated product types (right column).

\[
\begin{align*}
U^A & \quad : \equiv \text{Set} \\
(\ast^A : \text{Tm}_p \Gamma^A_s) & \equiv (\tau : T) \rightarrow (\alpha \tau)^A \\
(\ast^A : \text{Tm}_s \Gamma^A_s) & \equiv d \rightarrow (\alpha \gamma_s)^A
\end{align*}
\]

We use variables and terms to navigate these iterated products via iterated projections, and to apply function sorts to parameters.

\[
\begin{align*}
\text{vz}^A (\gamma_s, \alpha_s) & \equiv \alpha_s \\
(\text{vs} x)^A (\gamma_s, \alpha_s) & \equiv x^A \gamma_s
\end{align*}
\]

For point types, both function types become metatheoretic functions and we erase the element operator, since it does not have any semantic meaning. Just as sort contexts, point contexts are interdependency-free lists of the interpretations of their constituent types.

\[
\begin{align*}
(El a)^A \gamma_s & \equiv a^A \gamma_s \\
(\Pi p T A)^A \gamma_s & \equiv (\tau : T) \rightarrow (\alpha \tau)^A \gamma_s \\
(a \Rightarrow p A)^A \gamma_s & \equiv a^A \gamma_s \rightarrow A^A \gamma_s
\end{align*}
\]

**Example 4 (Revisiting Natural Numbers, Vectors, Parity).** Looking at the signatures in Example 2, we observe that the algebra interpretations are the expected left-nested product types starting with $1$. For natural numbers, we have $N^A_s \equiv 1 \times \text{Set}$. Given a $(\times, M) : 1 \times \text{Set}$, the algebras of its point contexts become $N^A (\times, M) \equiv (1 \times M) \times (M \rightarrow M)$. For vectors, the sorts in an algebra are elements of $V^A_s \equiv 1 \times (\text{Set} \rightarrow \text{Set})$, and given such a $(\times, W)$, the point algebras are given by $V^A (\times, W) \equiv 1 \times W \times (A \rightarrow \text{(n : N)} \rightarrow W n \rightarrow W (n + 1))$. For parity, the sorts in an algebra are $P^A_s \equiv 1 \times (\text{Set} \rightarrow \text{Set}) \times (\text{Set} \rightarrow \text{Set})$, and given such a $(\times, E, O)$, point algebras are $P^A (\times, E, O) \equiv 1 \times E \times (\text{(n : N)} \rightarrow O n \rightarrow E (n + 1)) \times (\text{(n : N)} \rightarrow E n \rightarrow O (n + 1))$.  

FSCD 2020
Displayed algebras can be seen as the logical predicate interpretation [13] of the syntax.

**Definition 5** (Displayed Algebra Operation). Sort contexts and types become predicates over their algebra interpretations, while the displayed algebra interpretation of variables and terms says that they respect the predicates (usually called fundamental lemma).

\[
\cdot D : (A_s : Ty_s) \rightarrow A_s^D \rightarrow \text{Set} \\
\cdot D : (f_s : \text{Con}_s) \rightarrow f_s^D \rightarrow \text{Set} \\
\cdot D : (t : \text{Mm}_s A_s) \rightarrow f_s^D \gamma_s \rightarrow A_s^D (t^A \gamma_s)
\]

The interpretation of \(U\) is predicate space, interpretations of \(\hat{\Pi}_p\) and sort contexts are pointwise.

\[
U^D T := T \rightarrow U \\
\cdot D \ast := \text{1}
\]

\[
(\hat{\Pi}_p T A_s)^D f_s := (\tau : T) \rightarrow (A_s \tau)^D (f_s \tau) \\
(\cdot D \ast A_s)^D (\gamma_s, \alpha_s) := f_s^D \gamma_s \times A_s^D \alpha_s
\]

The interpretation of terms follows the same pattern as for algebras, variables are lookups, application is metatheoretic application, we omit listing them. On point types, the interpretation of \(\Rightarrow_p\) is again non-interesting, the interpretation of \(\hat{\Pi}_p\) is pointwise, while the interpretation of \(\Rightarrow_p\) says that if the predicate holds for the input, then it holds for the output.

\[
(E \alpha)^D \gamma_s^d \alpha := \alpha^D \gamma_s^d \alpha \\
(\hat{\Pi}_p T A)^D \gamma_s^d f := (\tau : T) \rightarrow (A \tau)^D \gamma_s^d (f \tau) \\
(a \Rightarrow_p A)^D \gamma_s^d f := (\alpha : a^D \gamma_s) \rightarrow a^D \gamma_s^d \alpha \rightarrow A^D \gamma_s^d (f \alpha)
\]

Finally, point contexts are interpreted as iterated products again, they contain witnesses that the predicates hold for everything in the algebra.

\[
(\cdot D \ast A)^D \gamma_s^d (\gamma, \alpha) := f^D \gamma_s^d \gamma \times A^D \gamma_s^d \alpha
\]

**Example 6** (Revisiting Natural Numbers, Vectors, Parity). Given \((\ast, M) : N_s^A\) and \((\ast, z, s) : N^A (\ast, M)\), the displayed sort algebra is a predicate on \(M\), concretely \(N_s^D (\ast, M) \equiv 1 \times (M \rightarrow \text{Set})\). This can be seen as the motive of the eliminator if \((M, z, s)\) is the initial algebra. Given such a \((\ast, Q)\), the displayed point algebra computes the types of methods of the eliminator, \(N^D (\ast, Q) (\ast, z, s) \equiv 1 \times Q z \times ((n : M) \rightarrow Q n \rightarrow Q (s n))\) as expected. Given a vector algebra \((\ast, W)\), \((\ast, \text{nil, cons})\), a displayed sort algebra computes to \(V_s^D (\ast, W) \equiv 1 \times (n : \text{N}) \rightarrow W n \rightarrow \text{Set}\), and the displayed point algebra is \(V^D (\ast, Q) (\ast, \text{nil, cons}) \equiv 1 \times Q 0 \text{nil} \times ((a : A)(x : \text{N})(v : W n) \rightarrow Q n v \rightarrow Q (n + 1) (\text{cons} a x v))\). Finally, given a parity algebra \((\ast, E, O)\), \((\ast, e\theta, e S, o S)\), the displayed sort algebra consists of \(P_s^D (\ast, E, O) \equiv 1 \times ((n : \text{N}) \rightarrow E n \rightarrow \text{Set}) \times ((n : \text{N}) \rightarrow O n \rightarrow \text{Set})\) and given such a \((\ast, Q, R)\), displayed point algebras are \(P^D (\ast, Q, R) (\ast, e\theta, e S, o S) \equiv 1 \times Q 0 e\theta \times ((n : \text{N})(o : O n) \rightarrow R n o \rightarrow Q (n + 1) (e S n o)) \times ((n : \text{N})(e : E n) \rightarrow Q n e \rightarrow R (n + 1) (o S n e))\). Given a family \(Q\) over \(E\) and a family \(R\) over \(O\), these express that \(e\theta\) witnesses \(Q\), while \(e S\) turns witnesses of \(R\) into witnesses of \(Q\) and \(o S\) turns witnesses of \(Q\) into witnesses of \(R\).
Sections are dependent binary logical relations, where the interpretation of $U$, $\text{El}$ and $\Rightarrow_p$ are non-standard.

**Definition 7 (Section Operation).** For sorts and sort contexts, sections are dependent binary relations over the corresponding algebra and displayed algebra. “Dependent” here means the type of the second argument of the relation depends on the first.

$$^S : (A_s : T \equiv_s) \rightarrow (\alpha_s : A^\alpha_s) \rightarrow A^D_s \alpha_s \rightarrow \text{Set}$$

$$^S : (\Gamma_s : \text{Con}_s) \rightarrow (\gamma_s : \Gamma^\alpha_s) \rightarrow \Gamma^D_s \gamma_s \rightarrow \text{Set}$$

The interpretation of variables expresses that if the relation holds at the context, then it also holds at the type for the algebra and displayed algebra interpretation of the variable. We have the same for terms.

$$^S : (x : \text{Var}_s \Gamma_s) \rightarrow \Gamma^S_s \gamma_s \alpha \rightarrow (\alpha : A^\alpha_s) \rightarrow A^D_s \gamma_s \alpha \rightarrow \text{Set}$$

$$^S : (t : \text{TM}_s \Gamma_s) \rightarrow \Gamma^S_s \gamma_s \alpha \rightarrow (\gamma : \Gamma^\alpha_s) \rightarrow \Gamma^D_s \gamma_s \gamma \rightarrow \text{Set}$$

Sections of the universe are given as dependent functions (instead of dependent relation space as is usual for logical relations). The interpretation of $\hat{\Pi}_s$ is pointwise, and so is that of sort contexts.

$$\hat{\Pi}_s^T T \equiv (\tau : T) \rightarrow T^d \tau$$

$$(\hat{\Pi}_s T A)^S f_s f^d_s \equiv (\tau : T) \rightarrow (A_s \tau^S_s (f_s \tau) (f^d_s \tau))$$

$$(\hat{\Pi}_s V) \Rightarrow (A_s \Rightarrow (\gamma_s, \alpha_s) (\gamma^d_s, \alpha^d_s)) \equiv I^S_s \gamma_s \alpha^d \times A^S_s \alpha \alpha_s$$

Sort terms follow the usual pattern of variables selecting sort interpretations via projections of products and interpreting the application by metatheoretic application:

$$^S \text{var} \equiv (\text{var} x)^S \gamma_s^x \equiv x^S \gamma_s^x$$

$$^S (\text{var} x)^S \equiv x^S \gamma_s^x$$

$$^S (t \Rightarrow \alpha)^S \equiv I^S \gamma_s^x \alpha$$

Sections on point types express equalities. Each point type ends with an $\text{El}_a$, and the section says that the function given by $a^S$ returns the witness of the predicate $a^d$. $\hat{\Pi}_s$ is defined pointwise, while $\Rightarrow_p$ says that for any input, the outputs of $f$ and $f^d$ are related by $A^S$, where we use $a^S$ again to produce a witness of the predicate on the right hand side.

$$(\text{El}_a)^S \gamma^x_s \alpha \alpha \equiv (a^S \gamma^x_s \alpha = a^d)$$

$$(\hat{\Pi}_s T A)^S \gamma_s^x f f^d \equiv (\tau : T) \rightarrow (A \tau)^S \gamma_s^x (f \tau) (f^d \tau)$$

$$(a \Rightarrow_p A)^S \gamma_s^x f f^d \equiv (\alpha : a^\alpha_s) \rightarrow A^S \gamma_s^x (f \alpha) (f^d (a^S \gamma^x_s \alpha))$$

The definition of sections of point contexts is, again, just an iteration of products.

$$^S \gamma^x_s \gamma^d \equiv 1$$

$$(\Gamma \Rightarrow A)^S \gamma^x_s (\gamma, \alpha) (\gamma^d, \alpha^d) \equiv I^S \gamma^x_s \gamma^d \times A^S \gamma^x_s \alpha \alpha^d$$
Example 8 (Revisiting Natural Numbers, Vectors, Parity). Using the same notation for algebras and displayed algebras as in Example 6, a section of a natural number displayed algebra is a \((\ast, f)\) having type \(N^5_\ast (\ast, M) (\ast, Q) \equiv 1 \times ((n : M) \rightarrow Q n)\) together with a witness of \(N^5_\ast (\ast, f) (\ast, z, s) (\ast, w, h) \equiv 1 \times (f z = w) \times ((n : N) \rightarrow f (s n) = h n (f n))\). These equalities are the computation rules of the eliminator. For vectors, the section operation computes to \(V^5_\ast (\ast, W) (\ast, Q) \equiv 1 \times ((n : N)(v : W n) \rightarrow Q n v)\) and to \(V^5_\ast (\ast, f) (\ast, \text{nil}, \text{cons}) (\ast, \text{nil}^d, \text{cons}^d) \equiv 1 \times (f \text{nil} = \text{nil}^d) \times ((a : A)(n : N)(v : W n) \rightarrow f (n + 1)(\text{cons} a n v) = \text{cons}^d a n (f n v))\). For the parity families, sections are \(P^5_\ast (\ast, E, O) (\ast, Q, R) \equiv 1 \times ((n : \mathbb{N})(e : E n) \rightarrow Q n e) \times ((n : \mathbb{N})(o : O n) \rightarrow R n o)\) together with \(P^5_\ast (\ast, f, g) (\ast, e \theta, c S, o S) (e \theta^d, c S^d, o S^d) \equiv 1 \times (f e \theta = e \theta^d) \times ((n : \mathbb{N})(o : O n) \rightarrow f (n + 1)(c S n o) = c S^d n (g n o)) \times ((n : \mathbb{N})(e : E n) \rightarrow g (n + 1)(o S n e) = o S^d n (f n e))\). A section for parity displayed algebras consists of two functions \(f, g\) which map \(e \theta\) to \(e \theta^d\), \(c S\) to \(c S^d\) and \(o S\) to \(o S^d\).

4 Existence of Inductive Families

When does a type theory “support” types of our specification of mutual inductive families and how does this compare to well-established notions of inductive types? The intended meaning of the signatures is clear from the definition of their algebras as seen in Section 3, the types of their eliminators and computation rules are specified in the definitions of displayed algebras and sections. This means that we can formally say what it means for inductive types of their eliminators and computation rules are specified in the definitions of displayed algebras and sections. This means that we can formally say what it means for inductive families to exist in a type theory. In this section, we will prove that any metatheory with indexed W-types supports our notion of mutual inductive families or, in other words, mutual inductive families can be reduced to indexed W-types:

Theorem 9 (Existence of Inductive Families). For every signature of inductive families given by a sort context \(\Omega_s : \text{Con}_s\) and point context \(\Omega : \text{Con}_p \Omega_s\), there are are sort and point constructors in the form of

\[
\text{con}_s \Omega : \Omega_s^\Lambda \quad \text{and} \quad \text{con} \Omega : \Omega^\Lambda (\text{con}_s \Omega)
\]

such that for each displayed algebra given by motives \(\omega_s^d : \Omega_s^D (\text{con}_s \Omega)\) and methods \(\omega^d : \Omega^D \omega_s^d (\text{con} \Omega)\) we have an eliminator given by sections

\[
\text{elim}_s \Omega \omega_s^d : \Omega_s^S (\text{con}_s \Omega) \omega_s^d \quad \text{with} \quad \text{elim} \Omega \omega^d : \Omega^S (\text{elim}_s \Omega \omega_s^d) (\text{con} \Omega) \omega^d.
\]

Note that this definition of existence only requires the computation rules contained in \(\text{elim} \Omega \omega^d\) to hold propositionally. One might also wish for strict reduction rules instead to enable better computational behaviour.

Our strategy to prove this theorem is to first extend our syntax to a full substitution calculus including sort and point substitutions and point types (Section 4.1). Then we construct a term model using the extended syntax, which we can then show to be the initial algebra (Section 4.2).
4.1 A Substitution Calculus for the Syntax

The syntax for usual type theories includes substitutions. We did not have to mention them in the theory of signatures because of the simplicity of mutual inductive definitions. In other words, our syntax only contains normal forms (there are also no conversion rules in our syntax). However when doing constructions on the syntax, it is sometimes useful to have a full syntax, this includes a category of substitutions. We will make use of them in Section 4.2.

**Definition 10 (Sort Substitutions).** A calculus of substitutions \( \text{Sub}_s \) of sort contexts is useful to compare sort contexts themselves as well as to relate point contexts over different sort contexts. We define them to be inductively generated by

\[
\begin{align*}
\text{Sub}_s : \text{Con}_s &\rightarrow \text{Con}_s \rightarrow \text{Set} \\
\epsilon &\quad : \text{Sub}_s \, \Gamma_s, \\
\varpi, \pi &\quad : \text{Sub}_s \, \Gamma_s \, \Delta_s \rightarrow \text{Tm}_s \, \Gamma_s \, A_s \rightarrow \text{Sub}_s \, \Gamma_s \, (\Delta_s \triangleright A_s)
\end{align*}
\]

Like with the syntax of signatures itself, \( \text{Sub}_s \) can be encoded as an indexed W-type as shown in Appendix A. These substitutions allow us to substitute point types, point contexts, and sort terms via the following “pullback” operations:

\[
\begin{align*}
\text{[-]} & : \text{Tm}_s \, \Delta_s \rightarrow \text{Sub}_s \, \Gamma_s \, \Delta_s \rightarrow \text{Tm}_s \, \Gamma_s, \\
\text{[-]} & : \text{Con}_s \, \Delta_s \rightarrow \text{Sub}_s \, \Gamma_s \, \Delta_s, \\
\text{[-]} & : \text{Tm}_s \, \Delta_s \rightarrow \text{Sub}_s \, \Gamma_s \, \Delta_s \rightarrow \text{Tm}_s \, \Gamma_s, A_s
\end{align*}
\]

given by the defining rules for substitution

\[
\begin{align*}
\text{El}_A [\sigma] &\quad : \hat{\Pi}_p \, A[\sigma] \equiv \hat{\Pi}_p \, (\lambda x. (A \, \tau)[\sigma]) \\
\vz [\sigma, t] &\quad : \equiv t \\
\text{El}[a[\sigma]] &\quad : \equiv \text{El}(a[\sigma]) \\
(\text{a} \Rightarrow A)[\sigma] &\quad : \equiv a[\sigma] \Rightarrow A[\sigma] \\
(x[\sigma]) &\quad : \equiv x[\sigma] \\
(\text{var}[\sigma]) &\quad : \equiv (x \, \tau)[\sigma] \equiv t[\sigma] \, \alpha \, \tau \\
(I^r \triangleright A)[\sigma] &\quad : \equiv I[\sigma] \triangleright A[\sigma]
\end{align*}
\]

We can derive from this some useful gadgets of the substitutional calculus: We can define the weakening of a substitution \( \sigma : \text{Sub}_s \, \Gamma_s \, \Delta_s \rightarrow \text{Sub}_s \, (\Gamma_s \triangleright A_s) \, \Delta_s \) via recursion on \( \sigma \) by \( \text{wk}_s : \text{Sub}_s \, (\Gamma_s \triangleright A_s) \, \Delta_s \rightarrow \text{Sub}_s \, \Gamma_s \, \Delta_s \) via recursion on \( \sigma \) by \( \text{wk}_s : \equiv \epsilon \) and \( \text{wk}_{s, t} : \equiv (\text{wk}_s, \vz) \).

Using \( \text{wk}_s \), we can then recover the categorical structure of the substitutions by defining the identity \( \text{id}_{\Gamma_s} : \text{Sub}_s \, \Gamma_s \rightarrow \text{Sub}_s \, \Gamma_s \) by recursion of the context \( \Gamma_s \) : \( \text{id} \equiv \epsilon \) and \( \text{id}_{\Gamma_s \triangleright A_s} : \equiv (\text{wk}_{s, t}, \vz) \). Composition \( \sigma \circ \delta : \text{Sub}_s \, \Gamma_s \, \Delta_s \) of substitutions \( \sigma : \text{Sub}_s \, \Theta_s \, \Delta_s \) and \( \delta : \text{Sub}_s \, \Gamma_s \, \Delta_s \) is defined by recursion on the first substitution: \( \epsilon \circ \delta : \equiv \epsilon \), \( (\sigma, t) \circ \delta : \equiv (\sigma \circ \delta, t[\delta]) \).

The projections \( \pi_1 \sigma : \text{Sub}_s \, \Gamma_s \, \Delta_s \) and \( \pi_2 \sigma : \text{Tm}_s \, \Gamma_s \, A_s \) of a substitution \( \sigma : \text{Sub}_s \, \Gamma_s \, (\Delta_s \triangleright A_s) \) are just projections of \( \times \)-types: Any substitution between \( \Gamma_s \) and \( \Delta_s \triangleright A_s \) is of the form \( \sigma, t \) and we can just set \( \pi_1 \sigma : \equiv \sigma \) and \( \pi_2 \sigma, t : \equiv t \).

Obviously, we might also want to consider algebras, displayed algebras, and their sections over these substitutions.

**Definition 11 (Semantics of Sort Substitutions).** We can extend the algebra operator by defining it on substitutions by functions between the interpretations of sort contexts:

\[
A^s : \text{Sub}_s \, \Gamma_s \, \Delta_s \rightarrow \text{Sub}_s \, \Gamma_s \, \Delta_s^A
\]

This is done by setting \( \epsilon^A : \equiv * \) and \( (\sigma, t)^A : \equiv (\sigma^A, t^A) \).
The type of displayed algebras over a sort substitution should be the type of function between the displayed algebras of its domain and codomain, where in the latter we have to apply the function which we get from the algebra over the substitution:

\[-D: (\sigma : \text{Sub}_s \Gamma_s \Delta_s) \rightarrow \Gamma_s^D \gamma_s \rightarrow \Delta_s^D (\sigma^A \gamma_s)\]

These are defined, like in the non-displayed case, by setting \(e^D \gamma_s^d := \star \) and \((\sigma, t)^D \gamma_s^d := (\sigma^D \gamma_s^d, t^D \gamma_s^d)\).

A section of a displayed algebra of a sort substitution is supposed to map sections of its domain to sections of its codomain:

\[-S: (\sigma : \text{Sub}_s \Gamma_s \Delta_s) \rightarrow \Gamma_s^S \gamma_s \gamma_s^d \rightarrow \Delta_s^S (\sigma^A \gamma_s) (\sigma^D \gamma_s^d)\]

Again, this is happening componentwise by having: \(e^S \gamma_s^s := \star \) and \((\sigma, t)^S \gamma_s^s := (\sigma^S \gamma_s^s, t^S \gamma_s^s)\).

\[\blacktriangleright \text{Lemma 12.} \quad \text{It is easy to check that this definition of algebras of a substitution respects the substitution calculus given in Definition 10 in the following sense:}\]

\[
(\Lambda[s])^A \gamma_s = \Lambda^A (\sigma^A \gamma_s), \\
(t[s])^A \gamma_s = t^A (\sigma^A \gamma_s), \\
\text{id}^A \gamma_s = \gamma_s, \\
(\sigma \circ \delta)^A \gamma_s = \sigma^A (\delta^A \gamma_s),
\]

\[
\text{wk}^A_\sigma (\gamma_s, \alpha_s) = \sigma^A \gamma_s, \\
(\pi_1 \sigma)^A \gamma_s = \text{pr}_1 (\sigma^A \gamma_s), \text{ and} \\
(\pi_2 \sigma)^A \gamma_s = \text{pr}_2 (\sigma^A \gamma_s).
\]

\[\blacktriangleright \text{Proof.} \quad \text{We can prove the first rule by recursion on the point type } A : Ty_p \Gamma_s, \text{ the second rule by recursing on the term } t : Tm_p \Gamma_s A, \text{ the third by induction on the context, and all other by induction by the substitution. Analogous rules hold for displayed algebras over substitutions.}\]

The model which is initial in the category of all models is usually called the \textit{term model}. This is because in this model, a type gets interpreted as the set of all of its terms. Since our signatures form – or are at least strongly inspired by – a type theoretic syntax as well, we might hope to deploy the same strategy for inductive families. In the core of this interpretation is the issue of how to find an interpretation for a given sort term \(a\) of the universe token \(U\). The interpretation of this ought to be the terms of the point type \(\text{El}(a)\) associated with this sort term. But our syntax does not mention terms of point types at all, since point constructors are not interdependent! So our solution is to retrofit the theory with terms, as well as substitutions for the point contexts:

\[\blacktriangleright \text{Definition 13 (Point Substitution Calculus). Let us fix a sort context } \Gamma_s : \text{Con}_s. \text{ It turns out that there are three ways to construct reasonable terms of point types in } \Gamma_s: \text{ Via variables to navigate point contexts and application constructors for each of the two kinds of } \Pi\text{-types present in the syntax.}\]

\[
\text{Var}_p : \text{Con}_p \Gamma_s \rightarrow Tm_p \Gamma_s \rightarrow \text{Set} \\
\text{Var}_p : \text{Var}_p \Gamma A \rightarrow Tm_p \Gamma A \\
\text{Var}_p : \text{Var}_p (\Gamma \triangleright A) A \\
\text{Var}_p : \text{Var}_p \Gamma A \rightarrow \text{Var}_p (\Gamma \triangleright B) A
\]
Like with the sort substitutions defined in Definition 10, we define substitutions between point contexts over a fixed sort context $Γ_s : \text{Con}_s$ to be lists of point terms:

$\text{Sub}_p : \text{Con}_s \to \text{Con}_s \to \text{Set}$

$\epsilon : \text{Sub}_p Γ \cdot –, – : \text{Sub}_p Γ \Delta \to \text{Tm}_p Γ A \to \text{Sub}_p Γ (Δ \triangleright A)$

All of these can again be encoded as indexed W-types (cf. Appendix A). We can again define a pullback operations for variables and terms – this time for point terms – along substitutions in the form of

$\cdot : \text{Var}_p Δ A \to \text{Sub}_p Γ Δ \to \text{Tm}_p Γ A$

$\cdot : \text{Tm}_p Δ A \to \text{Sub}_p Γ Δ \to \text{Tm}_p Γ A$

which are defined recursively by

$vz[σ,t] :≡ t$

$(\text{var } x)[σ] :≡ x[σ]$

$(t \hat{=} τ)[σ] :≡ t[σ] \hat{=} τ$

$(t \circ u)[σ] :≡ t[σ] \circ u[σ]$

Analogously to Definition 10 we can define the weakening $\text{wk}_σ : \text{Sub}_p (Γ \triangleright A) Δ$ of a point substitution $σ : \text{Sub}_p Γ Δ$ along a point type $A : \text{Tm}_p Γ_s$, the identity substitution $\text{id} : \text{Sub}_p Γ Γ$ and the composition $σ \circ δ : \text{Sub}_p Γ Δ$ of substitutions $σ : \text{Sub}_p Θ Δ$ and $δ : \text{Sub}_p Γ Θ$.

As an auxiliary construction for our existence proof we will furthermore need notions of algebras, displayed algebras, and sections for the point terms and point substitutions:

$\triangleright \text{Definition 14 (Semantics of Point Substitutions & Terms).}$

Let us fix a sort context $Γ_s : \text{Con}_s$ and an algebra $γ_s : Γ^A$ over it. We can give semantic meaning to point types and point substitution by extending the algebra operator with the following components:

$\triangleright A : \text{Var}_p Γ A \to Γ^A γ_s \to A^A γ_s$

$\triangleright A : \text{Tm}_p Γ A \to Γ^A γ_s \to A^A γ_s$

$\triangleright A : \text{Sub}_p Γ Δ \to Γ^A γ_s \to A^A γ_s$

These components are, in essence, defined the same way as their respective parts on sorts. Of course, apart from these defining equations, this definition of algebras is also well-behaved under the other components of substitutional calculus, analogous to sort substitutions (cf. Lemma 12).

Let us now also fix a displayed algebra $γ^d_s : Γ^D γ_s$. For the displayed version of these algebras, the interpretation of point terms and of point substitutions needs to depend on these and, additionally, on an algebra and displayed algebra of the underlying point context. This leads to the following interpretations:

$\triangleright D : (x : \text{Var}_p Γ A) \to Γ^D γ^d_s \to A^D γ^d_s (x^A γ)$

$\triangleright D : (t : \text{Tm}_p Γ A) \to Γ^D γ^d_s \to A^D γ^d_s (t^A γ)$

$\triangleright D : (σ : \text{Sub}_p Γ Δ) \to Γ^D γ^d_s \to A^D γ^d_s (σ^A γ)$

Again, we define them by equations resembling the ones for sort substitutions, and again, substitution rules analogous to the ones in Lemma 12 hold.
4.2 Constructing all Inductive Families from the Syntax

In this section, assuming our type theory supports the theory of signatures (including the extensions of Section 4.1), we show that all mutual inductive families described by signatures exist. To give an intuition for this construction, consider the example of natural numbers: In its initial algebra, we want the interpretation of the $U$ sort to contain exactly the elements $z$, $s \, z$, $s \,(s \, z)$, ..., where $z$ and $s$ are point terms, pointing to the zero and successor constructor, respectively. But we observe that these are just the point terms of the type $\text{El}\, N$ in the context $\Delta := (N : U, z : \text{El}\, N, s : N \Rightarrow \text{El}\, N)$ (for the sake of the example, we use variable names and don’t separate sort and point contexts). So we define the initial algebra as $\text{tm}\, \Delta (\text{El}\, N), z, \lambda t. s \, t) : \Delta^A$. Note that given any other algebra $(A, a, f) : \Delta^A$ and natural number $n : \text{tm}\, \Delta (\text{El}\, N)$, we can simply use the algebra interpretation to obtain the result of the non-dependent elimination principle on $n$: $n^A (A, a, f)$ will have type $A$, moreover $z^A (A, a, f) = a$ and $(s \, @ t)^A (A, a, f) = f (t^A (A, a, f))$ which are the correct computation rules. The same idea works for displayed algebras: we can use the $\text{inl}$ operation on a natural number (given as a term) to obtain the result of the dependent elimination principle. In the following we will give the general description of this approach and prove its initiality by giving the dependent eliminator.

For the remainder of this section, let us fix the sort context $\Omega_s : \text{Con}_s$ and the point context $\Omega : \text{Con}_p\, \Omega_s$ which we want to construct by giving $\text{con}_s \, \Omega : \Omega_s^A$ and $\Omega : \Omega^A (\text{con}_s \, \Omega)$. Our definition of the constructor uses the trick to index several of the constructions by a second number (given as a term) to obtain the result of the dependent elimination principle. In this section, assuming our type theory supports the theory of signatures (including the extensions of Section 4.1), we show that all mutual inductive families described by signatures exist. To give an intuition for this construction, consider the example of natural numbers: In its initial algebra, we want the interpretation of the $U$ sort to contain exactly the elements $z$, $s \, z$, $s \,(s \, z)$, ..., where $z$ and $s$ are point terms, pointing to the zero and successor constructor, respectively. But we observe that these are just the point terms of the type $\text{El}\, N$ in the context $\Delta := (N : U, z : \text{El}\, N, s : N \Rightarrow \text{El}\, N)$ (for the sake of the example, we use variable names and don’t separate sort and point contexts). So we define the initial algebra as $\text{tm}\, \Delta (\text{El}\, N), z, \lambda t. s \, t) : \Delta^A$. Note that given any other algebra $(A, a, f) : \Delta^A$ and natural number $n : \text{tm}\, \Delta (\text{El}\, N)$, we can simply use the algebra interpretation to obtain the result of the non-dependent elimination principle on $n$: $n^A (A, a, f)$ will have type $A$, moreover $z^A (A, a, f) = a$ and $(s \, @ t)^A (A, a, f) = f (t^A (A, a, f))$ which are the correct computation rules. The same idea works for displayed algebras: we can use the $\text{inl}$ operation on a natural number (given as a term) to obtain the result of the dependent elimination principle. In the following we will give the general description of this approach and prove its initiality by giving the dependent eliminator.

For the remainder of this section, let us fix the sort context $\Omega_s : \text{Con}_s$ and the point context $\Omega : \text{Con}_p\, \Omega_s$ which we want to construct by giving $\text{con}_s \, \Omega : \Omega_s^A$ and $\Omega : \Omega^A (\text{con}_s \, \Omega)$. Our definition of the constructor uses the trick to index several of the constructions by a second number (given as a term) to obtain the result of the dependent elimination principle. In the following we will give the general description of this approach and prove its initiality by giving the dependent eliminator.

For the remainder of this section, let us fix the sort context $\Omega_s : \text{Con}_s$ and the point context $\Omega : \text{Con}_p\, \Omega_s$ which we want to construct by giving $\text{con}_s \, \Omega : \Omega_s^A$ and $\Omega : \Omega^A (\text{con}_s \, \Omega)$. Our definition of the constructor uses the trick to index several of the constructions by a second number (given as a term) to obtain the result of the dependent elimination principle. In the following we will give the general description of this approach and prove its initiality by giving the dependent eliminator.

For the remainder of this section, let us fix the sort context $\Omega_s : \text{Con}_s$ and the point context $\Omega : \text{Con}_p\, \Omega_s$ which we want to construct by giving $\text{con}_s \, \Omega : \Omega_s^A$ and $\Omega : \Omega^A (\text{con}_s \, \Omega)$. Our definition of the constructor uses the trick to index several of the constructions by a second number (given as a term) to obtain the result of the dependent elimination principle. In the following we will give the general description of this approach and prove its initiality by giving the dependent eliminator.

For the remainder of this section, let us fix the sort context $\Omega_s : \text{Con}_s$ and the point context $\Omega : \text{Con}_p\, \Omega_s$ which we want to construct by giving $\text{con}_s \, \Omega : \Omega_s^A$ and $\Omega : \Omega^A (\text{con}_s \, \Omega)$. Our definition of the constructor uses the trick to index several of the constructions by a second number (given as a term) to obtain the result of the dependent elimination principle. In the following we will give the general description of this approach and prove its initiality by giving the dependent eliminator.

For the remainder of this section, let us fix the sort context $\Omega_s : \text{Con}_s$ and the point context $\Omega : \text{Con}_p\, \Omega_s$ which we want to construct by giving $\text{con}_s \, \Omega : \Omega_s^A$ and $\Omega : \Omega^A (\text{con}_s \, \Omega)$. Our definition of the constructor uses the trick to index several of the constructions by a second number (given as a term) to obtain the result of the dependent elimination principle. In the following we will give the general description of this approach and prove its initiality by giving the dependent eliminator.
Proof. Let us first do a case distinction on the substitution. If it is $\epsilon$, then $I_\epsilon = \cdot$, and it is easy to see that there are no terms in the empty sort context. Thus, we can assume the substitution to be of the form $(\sigma, s)$. In this case, let us recurse on the term and see that

$$
(var \, vz)^A(\con'_s(\sigma, s)) = vz^A(\con'_s, \con'_s(\sigma, s)) \\
= \con'_s \tau \\
= \con'_s(\var \, vz[\sigma, s]),
$$

$$(var \, (v \, x))^A(\con'_s(\sigma, s)) = var \, (v \, x)^A(\con'_s(\sigma, s)) \\
= \con'_s(\var \, x[\sigma]) \\
= \con'_s(\var \, (v \, x)[\sigma, s]),$$

and define the constructor recursively on point types.

We can now use this lemma to do a trick with $\con \, \Omega$ similar to the trick we did for $\con \, \Omega$: Replace the fixed point context with a variable one, together with a substitution from $\Omega$, and define the constructor recursively on point types.

Definition 17 (The Point Constructor). We define operations on point contexts and point terms, resulting in algebras, in the following form:

$$\con' : \Subp \, \Omega \, \Gamma \rightarrow \Gamma^A(\con \, \Omega) \hspace{1cm} \con' : \Tm_p \, \Omega \, A \rightarrow A^A(\con \, \Omega)$$

The operation on point substitutions is defined recursively by $\con' \epsilon = \cdot$ and $\con' (\sigma, t) \equiv (\con' \sigma, \con' t)$, whereas for point terms, note that if $t : \Tm_p \, \Omega \, (El a)$, then by Lemma 16

$$t : \con'_s a \equiv \con'_s(a[id]) = a^A(\con'_s id_{\Omega}) \equiv (El a)^A(\con \, \Omega),$$

which allows us to define the constructor operator by

$$\con' t \equiv t \hspace{1cm} \text{for } t : \Tm_p \, \Omega \, (El a),$$

$$\con' t \equiv \lambda \tau. \con' (t \hat{\circ} \tau) \hspace{1cm} \text{for } t : \Tm_p \, \Omega \, (\hat{\Pi}_p \, T \, A), \text{ and}$$

$$\con' t \equiv \lambda u. \con' (t \hat{\circ} u) \hspace{1cm} \text{for } t : \Tm_p \, \Omega \, (a \Rightarrow_p A).$$

This concludes the definition of the constructors, since we can set, like for the sort constructor,

$$\con \, \Omega \equiv \con' \id_{\Omega} : A^A(\con \, \Omega).$$

Again, the construction comes with a property that makes it coherent under pulled back point terms. Analogously to Lemma 16, this coherence looks as follows:

Lemma 18 (Coherence of the Point Constructor). For all point substitutions $\sigma : \Subp \, \Omega \, \Gamma$ and point terms $t : \Tm_p \, \Gamma \, A$, pulling back has the same effect as the point constructor as in $\Gamma^A(\con' \sigma) = \con' \sigma[t]$. The proof is by induction on $\sigma$ and $t$, and analogous to the one of Lemma 16, see Appendix B.

With the constructors defined, let us move on to the construction of the eliminator. Let us from now on fix displayed algebras $\omega^A : \Omega^A(\con \, \Omega)$ and $\omega^d : \Omega^A(\con \, \Omega)$. We will proceed in the same order as for the constructors and start by generalizing $\elim \omega : \Omega^A \omega^d$ to arbitrary subcontexts of $\Omega$ by giving constructions on sort substitutions and sort terms.
Definition 19 (The Eliminator). The generalized eliminator will take substitutions or sort terms to give sections of sort types or sort contexts, respectively:

\[ \text{elim}_s' : (\sigma : \text{Sub}_s \Omega_s \Gamma_s) \rightarrow \Gamma_s^S (\sigma^A (\text{con}_s \Omega)) (\sigma^D \omega_s^d) \]
\[ \text{elim}_s : (t : \text{Tm}_s \Omega_s A_s) \rightarrow A_s^S (t^A (\text{con}_s \Omega)) (t^D \omega_s^d) \]

The first rule is defined by recursion using the second construction as usual: \( \text{elim}_s' \epsilon \equiv \star \) and \( \text{elim}_s' (\sigma, t) \equiv (\text{elim}_s' \sigma, \text{elim}_s' t) \). For the sort terms, we observe that, by Lemmas 16 and 18, for \( a : \text{Tm}_s \Omega_s U \) and \( t : a^A (\text{con}_s \Omega) \) we have \( U^S (a^D \omega_s^d (t^A (\text{con}_s \Omega))) = a^D \omega_s^d t \), and thus we can set, disregarding transports,

\[ \text{elim}_s' a : \equiv \lambda t. t^D \omega^d \quad \text{for } a : \text{Tm}_s \Omega_s U \]
\[ \text{elim}_s' t : \equiv \lambda t. \text{elim}_s' (t \hat{\cdot} \tau) \quad \text{for } t : \text{Tm}_s \Omega_s (\Pi_s T A_s). \]

Now we set \( \text{elim}_s \Omega_s \omega_s^d : \equiv \text{elim}_s' \text{id}_{\Omega_s} \).

Similar to Lemma 16, these definitions are coherent in the following form:

Lemma 20. Given a sort substitution \( \sigma : \text{Sub}_s \Omega_s \Gamma_s \) and a sort term \( t : \text{Tm}_s \Gamma_s A_s \), the eliminator of a pulled back term is the section of the term, evaluated at the eliminator on a substitution: \( \text{elim}_s' (t[\sigma]) = t^S (\text{elim}_s' \sigma) \).

Proof. The proof strategy is exactly the same as for Lemma 16.

As a last step, we still need to prove the computation rules for the eliminator, consisting of a section a displayed algebra over a given point context. Consistent with Definition 15, we generalize them to arbitrary point substitutions and point terms.

Lemma 21 (Computation Rules). We prove the computation rules for our eliminator \( \text{elim}_s \Omega_s \omega_s^d \) to be a section of subcontexts of \( \Omega \) and of point terms in \( \Omega \):

\[ \text{elim}' : (\sigma : \text{Sub}_p \Omega \Gamma) \rightarrow \Gamma^S (\text{elim}_s \Omega_s \omega_s^d) (\sigma^A (\text{con}_s \Omega)) (\sigma^D \omega_s^d) \]
\[ \text{elim}' : (t : \text{Tm}_p \Omega A) \rightarrow A^S (\text{elim}_s \Omega_s \omega_s^d) (t^A (\text{con}_s \Omega)) (t^D \omega_s^d) \]

Proof. Using the \( \text{elim}' \) for terms, the one for substitutions can be implemented in a straightforward way by recursion on the point substitution: \( \text{elim}' \epsilon \equiv \star \) and \( \text{elim}' (\sigma, t) \equiv (\text{elim}' \sigma, \text{elim}' t) \).

We implement \( \text{elim}' \) for a term \( t : \text{Tm}_p \Omega A \) by case distinction on its type \( A \). If \( A = \text{El} a \), we prove the following equality with the help of Lemmas 18 and 20:

\[ a^S (\text{elim}_s' \text{id}_{\Omega_s}) (t^A (\text{con}_s \Omega)) = a^S (\text{elim}_s' \text{id}_{\Omega_s}) t = \text{elim}_s' a t = t^D \omega_s^d. \]

For the other two cases, we use the induction hypotheses:

\( \text{elim}' t : \equiv \lambda \tau. \text{elim}' (t \hat{\cdot} \tau) \) for \( t : \text{Tm}_p \Omega (\Pi_p T A) \), and
\( \text{elim}' t : \equiv \lambda u. \text{elim}' (t \circ u) \) for \( t : \text{Tm}_p \Omega (a \Rightarrow_p A) \).

Proof of Theorem 9. Lemma 21 completes the construction of the eliminator and setting \( \text{elim} \Omega_s \omega_s^d : \equiv \text{elim}' \text{id}_{\Omega} \) completes the existence proofs for of inductive families.
5 Conclusions and further work

We defined a syntax of signatures for mutual inductive families which is very close to the usual way of specifying such types in proof assistants: by a list of sorts and then a list of constructors. We defined semantics for these signatures and showed how to derive the initial algebra for any signature just by using the syntax of signatures. The syntax of signatures was only given by normal forms, hence we could encode them as indexed W-types. Thus we obtained a formalisation of the reduction of mutual inductive families to indexed W-types. The lack of such a proof in the literature might be due to the absence of direct, convenient descriptions of mutual inductive types.

In the future, we would like to investigate how to integrate the theory of signatures into the core language of a proof assistant and how generic programming can be performed by induction on signatures, e.g. proving injectivity, disjointness of constructors, or decidability of equality. Also, we would like to extend the theory of signatures and its semantics with infinitary constructors. Currently, infinitely branching trees cannot be described as a signature, and as a consequence, the theory of signatures itself cannot be described as a signature.

References


A Deriving the Syntax from Indexed W-Types

We recall the notion of an indexed W-type.

Definition 22 (Indexed W-Types, [4]). The indexed W-type $IW_{A,B}^{o,r} : I \rightarrow Set$ for input data

\[
I : \text{Set} \\
A : \text{Set} \quad (\text{"index type"}) \\
B : A \rightarrow \text{Set} \quad (\text{"shapes"}) \\
o : A \rightarrow I \quad (\text{"output indices"}) \text{ and} \\
r : (a : A) \rightarrow B a \rightarrow I \quad (\text{"recursive indices"})
\]

is the inductive type on the constructor of the following form:

\[
a : A \quad c : (b : B a) \rightarrow IW_{A,B}^{o,r} (r a b) \\
sup a b : IW_{A,B}^{o,r} (o a)
\]

admitting a dependent eliminator

\[
C : \{i : I\} \rightarrow IW_{A,B}^{o,r} i \rightarrow Set \\
p : (a : A) \left(c : (b : B a) \rightarrow IW_{A,B}^{o,r} (r a b)\right) \rightarrow \left((b : B a) \rightarrow C (c b)\right) \rightarrow C (sup a c)
\]

with the reduction rule

\[
\text{elim}_{IW} C p (o a) (sup a c) \equiv p a c (\lambda b. \text{elim}_{IW} C p (r a b) (c b)).
\]

Using this definition of indexed W-types we now want to represent our extended syntax as such:
Table 1: The input data for the indexed W-types representing the internalized syntax for inductive families.

<table>
<thead>
<tr>
<th>i</th>
<th>(i) : Set</th>
<th>(A_i : \text{Set} )</th>
<th>(B_i : A_i \rightarrow \text{Set} )</th>
<th>(\alpha_i : A_i \rightarrow I_i )</th>
<th>(\tau_i : (a : A_i) \rightarrow B_i a \rightarrow I_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ty</td>
<td>1</td>
<td>+Set</td>
<td>(\text{inl} \rightarrow \mathbf{0} )</td>
<td>(\text{inr} T \rightarrow T )</td>
<td>(- \rightarrow * )</td>
</tr>
<tr>
<td>Con</td>
<td>1</td>
<td>+Ty</td>
<td>(\text{inl} \rightarrow 0 )</td>
<td>(\text{inr} B \rightarrow 1 )</td>
<td>(- \rightarrow * )</td>
</tr>
<tr>
<td>Var</td>
<td>Con</td>
<td></td>
<td>(\text{inl} \rightarrow 0 )</td>
<td>(\text{inr} (\Gamma', B') \rightarrow 1 )</td>
<td>(\text{inr} (\Gamma', B') \rightarrow \Gamma' )</td>
</tr>
<tr>
<td>Tm</td>
<td>(T_i )</td>
<td>(\Gamma ' )</td>
<td>(\text{inl} B \rightarrow 0 )</td>
<td>(\text{inr} B \rightarrow 0 )</td>
<td>(- \rightarrow * )</td>
</tr>
<tr>
<td>Sub</td>
<td>(\Gamma_i )</td>
<td>1</td>
<td>(+(B : T_i) \times Tm_i \Gamma_i B )</td>
<td>(\text{inr} (B, \ell) \rightarrow 1 )</td>
<td>(- \rightarrow * )</td>
</tr>
<tr>
<td>Var - (A )</td>
<td>Con</td>
<td>(\Gamma_i )</td>
<td>(\text{inl} \rightarrow 0 )</td>
<td>(\text{inr} (\ell) \rightarrow 1 )</td>
<td>(- \rightarrow * )</td>
</tr>
<tr>
<td>Tm</td>
<td>(T_i )</td>
<td>(\Gamma ' )</td>
<td>(\text{inl} \rightarrow 0 )</td>
<td>(\text{inr} \rightarrow 1 )</td>
<td>(- \rightarrow * )</td>
</tr>
<tr>
<td>Sub</td>
<td>(\Gamma_i )</td>
<td>1</td>
<td>(+(A : T_i) \times Tm_i \Gamma_i A )</td>
<td>(\text{inr} (\ell) \rightarrow 1 )</td>
<td>(- \rightarrow * )</td>
</tr>
</tbody>
</table>

Definition 23 (IF-Syntax as W-Types). We define the types defined in Definition 1, Definition 10, and Definition 13 as follows:

\[
\begin{align*}
\text{Ty}_i & \equiv \text{IW}^{\text{Ty}_i}_{\text{Ty}_i} \\
\text{Con}_i & \equiv \text{IW}^{\text{Con}_i}_{\text{Con}_i} \\
\text{Var}_i - B & \equiv \text{IW}^{\text{Var}_i}_B \times \text{Var}_B \quad B \\
\text{Tm}_i \Gamma_i & \equiv \text{IW}^{\text{Tm}_i \Gamma_i}_{\text{Tm}_i \Gamma_i} \\
\text{Sub}_i \Gamma_i & \equiv \text{IW}^{\text{Sub}_i \Gamma_i}_{\text{Sub}_i \Gamma_i} \\
\text{Ty}_i \Gamma_i & \equiv \text{IW}^{\text{Ty}_i \Gamma_i}_{\text{Ty}_i \Gamma_i} \\
\text{Con}_i \Gamma_i & \equiv \text{IW}^{\text{Con}_i \Gamma_i}_{\text{Con}_i \Gamma_i} \\
\text{Var}_i - A & \equiv \text{IW}^{\text{Var}_i}_A \times \text{Var}_A \quad A \\
\text{Tm}_i \Gamma & \equiv \text{IW}^{\text{Tm}_i \Gamma}_{\text{Tm}_i \Gamma} \\
\text{Sub}_i \Gamma & \equiv \text{IW}^{\text{Sub}_i \Gamma}_{\text{Sub}_i \Gamma}
\end{align*}
\]

where the respective input data for the indexed W-types is given in Table 1.
B Proof of Lemma 18

Proof. Repeating the strategy of the proof of Lemma 16, we again see that we can assume the substitution to be of an extended form \((\sigma, s)\), since there are no point terms in the empty point context. Now, by recursion on the term we see that

\[
\begin{align*}
(\text{var } v z)^A (\text{con'} (\sigma, s)) &= (\text{var } v z)^A (\text{con'} \sigma, \text{con'} s) \\
&= \text{con'} s \\
&= \text{con'} (\text{var } v z[\sigma, s]),
\end{align*}
\]

\[
\begin{align*}
(\text{var}(v s x))^A (\text{con'} (\sigma, s)) &= (\text{var}(v s x))^A (\text{con'} \sigma, \text{con'} s) \\
&= (\text{var} x)^A (\text{con'} \sigma) \\
&= \text{con'} (\text{var} x[\sigma]) & \text{by induction} \\
&= \text{con'} (\text{var} (v s x)[\sigma, s]),
\end{align*}
\]

\[
\begin{align*}
(t \circ u)^A (\text{con'} \sigma) &= t^A (\text{con'} \sigma) (u^A (\text{con'} \sigma)) \\
&= \text{con'} (t[\sigma]) (\text{con'}(u[\sigma])) & \text{by induction} \\
&= \text{con'} ((t \circ u)[\sigma]), \text{ and}
\end{align*}
\]

\[
\begin{align*}
(t \circ \tau)^A (\text{con'} \sigma) &= t^A (\text{con'} \sigma) \tau \\
&= \text{con'} (t[\sigma]) \tau & \text{by induction} \\
&= \text{con'} ((t \circ \tau)[\sigma]).
\end{align*}
\]