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 $\mathbb{N} : \mathsf{Set}$ which is generated by the zero constructor $0 : \mathbb{N}$ and the successor function $\mathsf{S} : \mathbb{N} \to \mathbb{N}$. 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 $\mathsf{Vec} : \mathbb{N} \to \mathsf{Set}$, by a constructor for the empty vector $\mathsf{nil} : \mathsf{Vec}\,0$ and $\mathsf{cons} : A \to (n : \mathbb{N}) \to \mathsf{Vec}\,n \to \mathsf{Vec}\,(\mathsf{S}\,n)$. 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

$$\mathsf{isEven} : \mathbb{N} \to \mathsf{Set},$$
$$\mathsf{isOdd} \ : \mathbb{N} \to \mathsf{Set}$$

by constructors

$$\mathsf{even0} : \mathsf{isEven}\, 0,$$
$$\mathsf{evenS} : (n : \mathbb{N}) \to \mathsf{isOdd}\, n \to \mathsf{isEven}\, (\mathsf{S}\, n), \text{ and}$$
$$\mathsf{oddS} \ : (n : \mathbb{N}) \to \mathsf{isEven}\, n \to \mathsf{isOdd}\, (\mathsf{S}\, n).$$

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, isEven–isOdd can be simulated by a single family indexed over an extra boolean which says which sort is meant: $\mathsf{isEven?} : \mathsf{Bool} \to \mathbb{N} \to \mathsf{Set}$. Now isEven is simulated by $\mathsf{isEven?}\, \mathsf{true}$ and isOdd by $\mathsf{isEven?}\, \mathsf{false}$. 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 deriviation 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 $F\, X \to X$, 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 $(N : \mathsf{Set},\ 0 : N,\ S : N \to N)$, where N, 0 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[1].

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 $N : \mathit{set}$ and introduction rules $0 : N$ and $s : (u : N)N$. Our syntax will encode the same information by a sort context $(\cdot \triangleright \mathsf{U})$ and a point context $\cdot \triangleright \mathsf{El}\,(\mathsf{var}\,\mathsf{vz}) \triangleright \mathsf{var}\,\mathsf{vz} \Rightarrow_{\mathsf{p}} \mathsf{El}\,(\mathsf{var}\,\mathsf{vz})$. 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 $\Rightarrow_{\mathsf{p}}$, see later). (2) In [15, 2], natural numbers are specified by $`\sigma\,\mathsf{Bool}\,(\lambda b\,.\,\mathsf{if}\,b\,\mathsf{then}\,`\blacksquare\,\mathsf{tt}\,\mathsf{else}\,`\mathsf{X}\,\mathsf{tt}\,(`\blacksquare\,\mathsf{tt}))$. 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 $`\blacksquare$), when it is false (successor) there is one recursive argument signified by $`\mathsf{X}$. 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 Bool (expressing that there are two constructors) and a family of sets over Bool, $\lambda b\,.\,\mathsf{if}\,b\,\mathsf{then}\,\bot\,\mathsf{else}\,\top$ 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 these 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 Synek [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_i (we omit the indices for readability), Π-types, Σ-types, unit type $\mathbf{1}$, propositional equality $- = -$, and indexed W-types [4] (see Appendix A). We write Σ-types as $(x : A) \times B$ and Π-types as $(x : A) \to B$ where B might refer to x. We write implicit arguments in curly braces $\{x : A\} \to B$ 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 ($\hat{\Pi}_\mathsf{s}$, $\hat{\Pi}_\mathsf{p}$). 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 Con_s from point contexts 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.*

$$
\begin{array}{ll}
\mathsf{Ty_s} & : \mathsf{Set} \\
\mathsf{U} & : \mathsf{Ty_s} \\
\hat{\Pi}_\mathsf{s} & : (T : \mathsf{Set}) \to (T \to \mathsf{Ty_s}) \to \mathsf{Ty_s} \\
\mathsf{Con_s} & : \mathsf{Set} \\
\cdot & : \mathsf{Con_s} \\
- \triangleright - & : \mathsf{Con_s} \to \mathsf{Ty_s} \to \mathsf{Con_s} \\
\mathsf{Var_s} & : \mathsf{Con_s} \to \mathsf{Ty_s} \to \mathsf{Set} \\
\mathsf{vz} & : \mathsf{Var_s}\,(\Gamma_s \triangleright A_s)\,A_s \\
\mathsf{vs} & : \mathsf{Var_s}\,\Gamma_s\,A_s \to \mathsf{Var_s}\,(\Gamma_s \triangleright B_s)\,A_s \\
\mathsf{Tm_s} & : \mathsf{Con_s} \to \mathsf{Ty_s} \to \mathsf{Set} \\
\mathsf{var} & : \mathsf{Var_s}\,\Gamma_s\,A_s \to \mathsf{Tm_s}\,\Gamma_s\,A_s \\
- \mathbin{@} - & : \mathsf{Tm_s}\,\Gamma_s\,(\hat{\Pi}_\mathsf{s}\,T\,A_s) \to (\tau : T) \to \mathsf{Tm_s}\,\Gamma_s\,(A_s\,\tau)
\end{array}
\qquad
\begin{array}{ll}
\mathsf{Ty_p} & : \mathsf{Con_s} \to \mathsf{Set} \\
\mathsf{El} & : \mathsf{Tm_s}\,\Gamma_s\,\mathsf{U} \to \mathsf{Ty_p}\,\Gamma_s \\
\hat{\Pi}_\mathsf{p} & : (T : \mathsf{Set}) \to (T \to \mathsf{Ty_p}\,\Gamma_s) \to \mathsf{Ty_p}\,\Gamma_s \\
- \Rightarrow_\mathsf{p} - & : \mathsf{Tm_s}\,\Gamma_s\,\mathsf{U} \to \mathsf{Ty_p}\,\Gamma_s \to \mathsf{Ty_p}\,\Gamma_s \\
\mathsf{Con_p} & : \mathsf{Con_s} \to \mathsf{Set} \\
\cdot & : \mathsf{Con_p}\,\Gamma_s \\
- \triangleright - & : \mathsf{Con_p}\,\Gamma_s \to \mathsf{Ty_p}\,\Gamma_s \to \mathsf{Con_p}\,\Gamma_s
\end{array}
$$

A *sort type* $\mathsf{Ty_s}$ 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 $\hat{\Pi}_\mathsf{s}$. We use the abbreviation $T \hat{\Rightarrow}_\mathsf{s} A_s$ for $\hat{\Pi}_\mathsf{s}\,T\,(\lambda\tau.A_s)$ when $A_s : \mathsf{Ty_s}$. A *sort context* $\mathsf{Con_s}$ 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 $\mathsf{Var_s}$ with zero vz and successor vs constructors. Just as variables, *sort terms* $\mathsf{Tm_s}$ are indexed by a sort context and a sort type. Each variable is a term (var) and we have application $-\mathbin{@}-$ for the function space $\hat{\Pi}_\mathsf{s}$. Note that $t : \mathsf{Tm_s}\,\Gamma_s\,A_s$ 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* $\mathsf{Ty_p}$ over a given sort context. The type formers are the element type for the universe U, a function type with metatheoretic domain $\hat{\Pi}_\mathsf{p}$ and a non-dependent function type $-\Rightarrow_\mathsf{p}-$ 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 $T \hat{\Rightarrow}_\mathsf{p} A_p$ for $\hat{\Pi}_\mathsf{p}\,T\,(\lambda\tau.A_p)$ when $A_p : \mathsf{Ty_p}$. 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.

$$
\begin{array}{ll}
N_s :\equiv (\cdot \triangleright \mathsf{U}) & (N : \mathsf{U}) \\
N :\equiv \left(\cdot \triangleright \mathsf{El}\,(\mathsf{var\,vz}) \triangleright \mathsf{var\,vz} \Rightarrow_\mathsf{p} \mathsf{El}\,(\mathsf{var\,vz})\right) & (\mathit{zero} : N, \mathit{suc} : N \to N)
\end{array}
$$

The only sort is referred to by $\mathsf{var\,vz}$. As shown by the constructor for successor, on the left hand side of the arrow $\Rightarrow_\mathsf{p}$ 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 $A : \mathsf{Set}$. We also assume that we have natural numbers in our metatheory.

$$
\begin{array}{ll}
V_s :\equiv (\cdot \triangleright\ \mathbb{N} \hat{\Rightarrow}_{\mathsf{s}} \mathsf{U}) & (Vec\ : \mathbb{N} \to \mathsf{U}) \\
V\ :\equiv \big(\cdot \triangleright \mathsf{El}\,(\mathsf{var\,vz} \mathbin{@} 0) \triangleright & (nil\ \ : Vec\,0, \\
\quad A \hat{\Rightarrow}_{\mathsf{p}} \hat{\Pi}_{\mathsf{p}}\, \mathbb{N}\, \big(\lambda n.\mathsf{var\,vz} \mathbin{@} n \Rightarrow_{\mathsf{p}} & \quad cons : A \to (n : \mathbb{N}) \to Vec\,n \to \\
\qquad \mathsf{El}\,(\mathsf{var\,vz} \mathbin{@} (n+1))\big)\big) & \qquad Vec\,(n+1))
\end{array}
$$

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 *cons* constructor, we use both kinds of function types: the first two function types are $\hat{\Pi}_{\mathsf{p}}$ as they refer to the parameters of type A and $\mathbb{N}$. The last function type is $\Rightarrow_{\mathsf{p}}$ as it refers to a recursive argument.

We revisit the parity example from the introduction.

$$
\begin{array}{ll}
P_s :\equiv (\cdot \triangleright\ \mathbb{N} \hat{\Rightarrow}_{\mathsf{s}} \mathsf{U} \triangleright \mathbb{N} \hat{\Rightarrow}_{\mathsf{s}} \mathsf{U}) & (isEven : \mathbb{N} \to \mathsf{U}, isOdd : \mathbb{N} \to \mathsf{U}) \\
P\ :\equiv \big(\cdot \triangleright \mathsf{El}\,(\mathsf{var}\,(\mathsf{vs\,vz}) \mathbin{@} 0) \triangleright & (even0\ \ : isEven\,0, \\
\quad \hat{\Pi}_{\mathsf{p}}\, \mathbb{N}\, \big(\lambda n.\mathsf{var\,vz} \mathbin{@} n \Rightarrow_{\mathsf{p}} & \quad evenS\ : (n : \mathbb{N}) \to isOdd\,n \to \\
\qquad \mathsf{El}\,(\mathsf{var}\,(\mathsf{vs\,vz}) \mathbin{@} (n+1))\big) \triangleright & \qquad isEven\,(n+1), \\
\quad \hat{\Pi}_{\mathsf{p}}\, \mathbb{N}\, \big(\lambda n.\mathsf{var}\,(\mathsf{vs\,vz}) \mathbin{@} n \Rightarrow_{\mathsf{p}} & \quad oddS\ \ : (n : \mathbb{N}) \to isEven\,n \to \\
\qquad \mathsf{El}\,(\mathsf{var\,vz} \mathbin{@} (n+1))\big)\big) & \qquad isOdd\,(n+1))
\end{array}
$$

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

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 Γ_s and a point context $\Gamma : \mathsf{Con}_{\mathsf{p}}\,\Gamma_s$. 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

$$(N : \mathsf{Set}) \times N \times (N \to N),$$

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

$$(P : N \to \mathsf{Set}) \times P\,z \times ((n : N) \to P\,n \to P\,(s\,n)),$$

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

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

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 (β rules) for the eliminator.

More formally, we will define operations $\_^{\mathsf{A}}$, $\_^{\mathsf{D}}$ and $\_^{\mathsf{S}}$ for computing algebras, displayed algebras and sections. As sort and point contexts are separate, we have to define them separately for both.

$$\begin{array}{lll}
\Gamma_s{}^{\mathsf{A}} : \mathsf{Set} & \Gamma_s{}^{\mathsf{D}} : \Gamma_s{}^{\mathsf{A}} \to \mathsf{Set} & \Gamma_s{}^{\mathsf{S}} : (\gamma_s : \Gamma_s{}^{\mathsf{A}}) \to \Gamma_s{}^{\mathsf{D}}\,\gamma_s \to \mathsf{Set} \\
\Gamma^{\mathsf{A}} : \Gamma_s{}^{\mathsf{A}} \to \mathsf{Set} & \Gamma^{\mathsf{D}} : \Gamma_s{}^{\mathsf{D}}\,\gamma_s \to \Gamma^{\mathsf{A}}\,\gamma_s \to \mathsf{Set} & \Gamma^{\mathsf{S}} : \Gamma_s{}^{\mathsf{S}}\,\gamma_s\,\gamma_s^d \to (\gamma : \Gamma^{\mathsf{A}}\,\gamma_s) \to \\
 & & \quad \Gamma^{\mathsf{D}}\,\gamma_s^d\,\gamma \to \mathsf{Set}
\end{array}$$

Putting them together, we get algebras as $(\gamma_s : \Gamma_s{}^{\mathsf{A}}) \times \Gamma^{\mathsf{A}}\,\gamma_s$, displayed algebras over a (γ_s, γ) by $(\gamma_s^d : \Gamma_s{}^{\mathsf{D}}\,\gamma_s) \times \Gamma^{\mathsf{D}}\,\gamma_s^d\,\gamma$, and sections of (γ_s^d, γ^d) by $(\gamma_s^s : \Gamma_s{}^{\mathsf{S}}\,\gamma_s\,\gamma_s^d) \times \Gamma^{\mathsf{S}}\,\gamma_s^s\,\gamma\,\gamma^d$.

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.*

$$\begin{array}{lll}
_^{\mathsf{A}} : \mathsf{Ty_s} \to \mathsf{Set} & _^{\mathsf{A}} : \mathsf{Var_s}\,\Gamma_s\,A_s \to \Gamma_s{}^{\mathsf{A}} \to A_s{}^{\mathsf{A}} & _^{\mathsf{A}} : \mathsf{Ty_p}\,\Gamma_s \quad \to \Gamma_s{}^{\mathsf{A}} \to \mathsf{Set} \\
_^{\mathsf{A}} : \mathsf{Con_s} \to \mathsf{Set} & _^{\mathsf{A}} : \mathsf{Tm_s}\,\Gamma_s\,A_s \to \Gamma_s{}^{\mathsf{A}} \to A_s{}^{\mathsf{A}} & _^{\mathsf{A}} : \mathsf{Con_p}\,\Gamma_s \to \Gamma_s{}^{\mathsf{A}} \to \mathsf{Set}
\end{array}$$

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).

$$\begin{array}{lll}
\mathsf{U}^{\mathsf{A}} & :\equiv \mathsf{Set} & \cdot^{\mathsf{A}} \quad :\equiv \mathbf{1} \\
(\hat{\Pi}_{\mathsf{s}}\,T\,A_s)^{\mathsf{A}} & :\equiv (\tau : T) \to (A\,\tau)^{\mathsf{A}} & (\Gamma_s \triangleright A_s)^{\mathsf{A}} :\equiv \Gamma_s{}^{\mathsf{A}} \times A_s{}^{\mathsf{A}}
\end{array}$$

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

$$\begin{array}{lll}
\mathsf{vz}^{\mathsf{A}}\,(\gamma_s, \alpha_s) & :\equiv \alpha_s & (\mathsf{var}\,x)^{\mathsf{A}}\,\gamma_s :\equiv x^{\mathsf{A}}\,\gamma_s \\
(\mathsf{vs}\,x)^{\mathsf{A}}\,(\gamma_s, \alpha_s) & :\equiv x^{\mathsf{A}}\,\gamma_s & (t \mathbin{@} \tau)^{\mathsf{A}}\,\gamma_s :\equiv (t^{\mathsf{A}}\,\gamma_s)\,\tau
\end{array}$$

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.

$$\begin{array}{lll}
(\mathsf{El}\,a)^{\mathsf{A}}\,\gamma_s & :\equiv a^{\mathsf{A}}\,\gamma_s & \cdot^{\mathsf{A}}\,\gamma_s \quad :\equiv \mathbf{1} \\
(\hat{\Pi}_{\mathsf{p}}\,T\,A)^{\mathsf{A}}\,\gamma_s & :\equiv (\tau : T) \to (A\,\tau)^{\mathsf{A}}\,\gamma_s & (\Gamma \triangleright A)^{\mathsf{A}}\,\gamma_s :\equiv \Gamma^{\mathsf{A}}\,\gamma_s \times A^{\mathsf{A}}\,\gamma_s \\
(a \Rightarrow_{\mathsf{p}} A)^{\mathsf{A}}\,\gamma_s & :\equiv a^{\mathsf{A}}\,\gamma_s \to A^{\mathsf{A}}\,\gamma_s &
\end{array}$$

▶ **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 $\mathbf{1}$. For natural numbers, we have $N_s{}^{\mathsf{A}} \equiv \mathbf{1} \times \mathsf{Set}$. Given a $(\star, M) : \mathbf{1} \times \mathsf{Set}$, the algebras of its point contexts become $N^{\mathsf{A}}\,(\star, M) \equiv (\mathbf{1} \times M) \times (M \to M)$. For vectors, the sorts in an algebra are elements of $V_s{}^{\mathsf{A}} \equiv \mathbf{1} \times (\mathbb{N} \to \mathsf{Set})$, and given such a $(\star, W)$, the point algebras are given by $V^{\mathsf{A}}\,(\star, W) \equiv \mathbf{1} \times W\,0 \times (A \to (n : \mathbb{N}) \to W\,n \to W\,(n+1))$. For parity, the sorts in an algebra are $P_s{}^{\mathsf{A}} \equiv \mathbf{1} \times (\mathbb{N} \to \mathsf{Set}) \times (\mathbb{N} \to \mathsf{Set})$, and given such a $(\star, E, O)$, point algebras are $P^{\mathsf{A}}\,(\star, E, O) \equiv \mathbf{1} \times E\,0 \times \big((n : \mathbb{N}) \to O\,n \to E\,(n+1)\big) \times \big((n : \mathbb{N}) \to E\,n \to O\,(n+1)\big)$.

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).*

$$\begin{aligned}
&_^{\mathsf{D}} : (A_s : \mathsf{Ty_s}) \quad \to A_s{}^{\mathsf{A}} \to \mathsf{Set} && _^{\mathsf{D}} : (x : \mathsf{Var_s}\ \Gamma_s\ A_s) \to \Gamma_s{}^{\mathsf{D}}\ \gamma_s \to A_s{}^{\mathsf{D}}\ (x^{\mathsf{A}}\ \gamma_s)\\
&_^{\mathsf{D}} : (\Gamma_s : \mathsf{Con_s}) \to \Gamma_s{}^{\mathsf{A}} \to \mathsf{Set} && _^{\mathsf{D}} : (t : \mathsf{Tm_s}\ \Gamma_s\ A_s) \to \Gamma_s{}^{\mathsf{D}}\ \gamma_s \to A_s{}^{\mathsf{D}}\ (t^{\mathsf{A}}\ \gamma_s)
\end{aligned}$$

Point types and contexts become predicates over their corresponding algebra interpretations, but these predicates also depend on witnesses of the predicates for the sort contexts.

$$\begin{aligned}
&_^{\mathsf{D}} : (A : \mathsf{Ty_p}\ \Gamma_s) \quad \to \Gamma_s{}^{\mathsf{D}}\ \gamma_s \to A^{\mathsf{A}}\ \gamma_s \to \mathsf{Set}\\
&_^{\mathsf{D}} : (\Gamma : \mathsf{Con_p}\ \Gamma_s) \to \Gamma_s{}^{\mathsf{D}}\ \gamma_s \to \Gamma^{\mathsf{A}}\ \gamma_s \to \mathsf{Set}
\end{aligned}$$

The interpretation of U *is predicate space, interpretations of* $\hat{\Pi}_{\mathsf{p}}$ *and sort contexts are pointwise.*

$$\begin{aligned}
&\mathsf{U}^{\mathsf{D}}\ T && :\equiv T \to \mathsf{U} && {}^{\cdot\mathsf{D}}\ \star && :\equiv \mathbf{1}\\
&(\hat{\Pi}_{\mathsf{s}}\ T\ A_s)^{\mathsf{D}}\ f_s && :\equiv (\tau : T) \to (A_s\ \tau)^{\mathsf{D}}\ (f_s\ \tau) && (\Gamma_s \triangleright A_s)^{\mathsf{D}}\ (\gamma_s, \alpha_s) && :\equiv \Gamma_s^{\mathsf{D}}\ \gamma_s \times A_s{}^{\mathsf{D}}\ \alpha_s
\end{aligned}$$

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 El *is again non-interesting, the interpretation of* $\hat{\Pi}_{\mathsf{p}}$ *is pointwise, while the interpretation of* $\Rightarrow_{\mathsf{p}}$ *says that if the predicate holds for the input, then it holds for the output.*

$$\begin{aligned}
&(\mathsf{El}\ a)^{\mathsf{D}}\ \gamma_s^d\ \alpha && :\equiv a^{\mathsf{D}}\ \gamma^d\ \alpha\\
&(\hat{\Pi}_{\mathsf{p}}\ T\ A)^{\mathsf{D}}\ \gamma_s^d\ f && :\equiv (\tau : T) \to (A\ \tau)^{\mathsf{D}}\ \gamma_s^d\ (f\ \tau)\\
&(a \Rightarrow_{\mathsf{p}} A)^{\mathsf{D}}\ \gamma_s^d\ f && :\equiv \{\alpha : a^{\mathsf{A}}\ \gamma_s\} \to a^{\mathsf{D}}\ \gamma_s^d\ \alpha \to A^{\mathsf{D}}\ \gamma_s^d\ (f\ \alpha)
\end{aligned}$$

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

$$\begin{aligned}
&{}^{\cdot\mathsf{D}}\ \gamma_s^d\ \gamma && :\equiv \mathbf{1}\\
&(\Gamma \triangleright A)^{\mathsf{D}}\ \gamma_s^d\ (\gamma, \alpha) && :\equiv \Gamma^{\mathsf{D}}\ \gamma_s^d\ \gamma \times A^{\mathsf{D}}\ \gamma_s^d\ \alpha
\end{aligned}$$

▶ **Example 6** (Revisiting Natural Numbers, Vectors, Parity). Given $(\star, M) : N_s{}^{\mathsf{A}}$ and $(\star, z, s) : N^{\mathsf{A}}\ (\star, M)$, the displayed sort algebra is a predicate on M, concretely $N_s{}^{\mathsf{D}}\ (\star, M) \equiv \mathbf{1} \times (M \to \mathsf{Set})$. This can be seen as the motive of the eliminator if (M, z, s) is the initial algebra. Given such a $(\star, Q)$, the displayed point algebra computes the types of methods of the eliminator, $N^{\mathsf{D}}\ (\star, Q)\ (\star, z, s) \equiv \mathbf{1} \times Q\ z \times \big((n : M) \to Q\ n \to Q\ (s\ n)\big)$ as expected. Given a vector algebra $(\star, W)$, $(\star, \mathit{nil}, \mathit{cons})$, a displayed sort algebra computes to $V_s{}^{\mathsf{D}}\ (\star, W) \equiv \mathbf{1} \times ((n : \mathbb{N}) \to W\ n \to \mathsf{Set})$, and the displayed point algebra is $V^{\mathsf{D}}\ (\star, Q)\ (\star, \mathit{nil}, \mathit{cons}) \equiv \mathbf{1} \times Q\ 0\ \mathit{nil} \times \big((a : A)(x : \mathbb{N})(v : W\ n) \to Q\ n\ v \to Q\ (n+1)\ (\mathit{cons}\ a\ x\ v)\big)$. Finally, given a parity algebra $(\star, E, O)$, $(\star, e0, eS, oS)$, the displayed sort algebra consists of $P_s{}^{\mathsf{D}}\ (\star, E, O) \equiv \mathbf{1} \times ((n : \mathbb{N}) \to E\ n \to \mathsf{Set}) \times ((n : \mathbb{N}) \to O\ n \to \mathsf{Set})$ and given such a $(\star, Q, R)$, displayed point algebras are $P^{\mathsf{D}}\ (\star, Q, R)\ (\star, e0, eS, oS) \equiv \mathbf{1} \times Q\ 0\ e0 \times \big((n : \mathbb{N})(o : O\ n) \to R\ n\ o \to Q\ (n+1)\ (eS\ n\ o)\big) \times \big((n : \mathbb{N})(e : E\ n) \to Q\ n\ e \to R\ (n+1)\ (oS\ n\ e)\big)$. Given a family Q over E and a family R over O, these express that $e0$ witnesses Q, while eS turns witnesses of R into witnesses of Q and oS turns witnesses of Q into witnesses of R.

Sections are dependent binary logical relations, where the interpretation of U, El and $\Rightarrow_{\mathsf{p}}$ 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.*

$$\begin{aligned}
&_^{\mathsf{S}} : (A_s : \mathsf{Ty_s}) \quad \to (\alpha_s : {A_s}^{\mathsf{A}}) \to {A_s}^{\mathsf{D}}\,\alpha_s \to \mathsf{Set}\\
&_^{\mathsf{S}} : (\Gamma_s : \mathsf{Con_s}) \to (\gamma_s : {\Gamma_s}^{\mathsf{A}}) \to {\Gamma_s}^{\mathsf{D}}\,\gamma_s \to \mathsf{Set}
\end{aligned}$$

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.

$$\begin{aligned}
&_^{\mathsf{S}} : (x : \mathsf{Var_s}\,\Gamma_s\,A_s) \to {\Gamma_s}^{\mathsf{S}}\,\gamma_s\,\gamma_s^d \to {A_s}^{\mathsf{S}}\,(x^{\mathsf{A}}\,\gamma_s)\,(x^{\mathsf{D}}\,\gamma_s^d)\\
&_^{\mathsf{S}} : (t : \mathsf{Tm_s}\,\Gamma_s\,A_s) \to {\Gamma_s}^{\mathsf{S}}\,\gamma_s\,\gamma_s^d \to {A_s}^{\mathsf{S}}\,(t^{\mathsf{A}}\,\gamma_s)\,(t^{\mathsf{D}}\,\gamma_s^d)
\end{aligned}$$

The interpretation of point types are dependent binary relations displayed over witnesses of relatedness for the relations for the contexts.

$$\begin{aligned}
&_^{\mathsf{S}} : (A : \mathsf{Ty_p}\,\Gamma_s) \quad \to {\Gamma_s}^{\mathsf{S}}\,\gamma_s\,\gamma_s^d \to (\alpha : A^{\mathsf{A}}\,\gamma_s) \to A^{\mathsf{D}}\,\gamma_s^d\,\alpha \to \mathsf{Set}\\
&_^{\mathsf{S}} : (\Gamma : \mathsf{Con_p}\,\Gamma_s) \to {\Gamma_s}^{\mathsf{S}}\,\gamma_s\,\gamma_s^d \to (\gamma : \Gamma^{\mathsf{A}}\,\gamma_s) \to \Gamma^{\mathsf{D}}\,\gamma_s^d\,\gamma \to \mathsf{Set}
\end{aligned}$$

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

$$\begin{aligned}
&\mathsf{U}^{\mathsf{S}}\,T\,T^d && :\equiv (\tau : T) \to T^d\,\tau\\
&(\hat{\Pi}_{\mathsf{s}}\,T\,A_s)^{\mathsf{S}}\,f_s\,f_s^d && :\equiv (\tau : T) \to (A_s\,\tau)^{\mathsf{S}}\,(f_s\,\tau)\,(f_s^d\,\tau)\\
&\cdot^{\mathsf{S}}\,\star\star && :\equiv \mathbf{1}\\
&(\Gamma_{\mathsf{s}} \triangleright A_s)^{\mathsf{S}}\,(\gamma_s,\alpha_s)\,(\gamma_s^d,\alpha_s^d) && :\equiv {\Gamma_s}^{\mathsf{S}}\,\gamma_s\,\gamma_s^d \times {A_s}^{\mathsf{S}}\,\alpha_s\,\alpha_s
\end{aligned}$$

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

$$\begin{aligned}
&\mathsf{vz}^{\mathsf{S}}\,(\gamma_s^s,\alpha_s^s) && :\equiv \alpha_s^s && \qquad (\mathsf{var}\,x)^{\mathsf{S}}\,\gamma_s^s && :\equiv x^{\mathsf{S}}\,\gamma_s^s\\
&(\mathsf{vs}\,x)^{\mathsf{S}}\,(\gamma_s^s,\alpha_s^s) && :\equiv x^{\mathsf{S}}\,\gamma_s^s && \qquad (t\mathbin{@}\tau)^{\mathsf{S}}\,\gamma_s^s && :\equiv t^{\mathsf{S}}\,\gamma_s^s\,\tau
\end{aligned}$$

Sections on point types express equalities. Each point type ends with an $\mathsf{El}\,a$, and the section says that the function given by a^{S} returns the witness of the predicate α^d. $\hat{\Pi}_{\mathsf{p}}$ is defined pointwise, while $\Rightarrow_{\mathsf{p}}$ 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.

$$\begin{aligned}
&(\mathsf{El}\,a)^{\mathsf{S}}\,\gamma_s^s\,\alpha\,\alpha^d && :\equiv (a^{\mathsf{S}}\,\gamma_s^s\,\alpha = \alpha^d)\\
&(\hat{\Pi}_{\mathsf{p}}\,T\,A)^{\mathsf{S}}\,\gamma_s^s\,f\,f^d && :\equiv (\tau : T) \to (A\,\tau)^{\mathsf{S}}\,\gamma_s^s\,(f\,\tau)\,(f^d\,\tau)\\
&(a \Rightarrow_{\mathsf{p}} A)^{\mathsf{S}}\,\gamma_s^s\,f\,f^d && :\equiv (\alpha : a^{\mathsf{A}}\,\gamma_s) \to A^{\mathsf{S}}\,\gamma_s^s\,(f\,\alpha)\,(f^d\,(a^{\mathsf{S}}\,\gamma_s^s\,\alpha))
\end{aligned}$$

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

$$\begin{aligned}
&\cdot^{\mathsf{S}}\,\gamma_s^s\,\gamma\,\gamma^d && :\equiv \mathbf{1}\\
&(\Gamma \triangleright A)^{\mathsf{S}}\,\gamma_s^s\,(\gamma,\alpha)\,(\gamma^d,\alpha^d) && :\equiv \Gamma^{\mathsf{S}}\,\gamma_s^s\,\gamma\,\gamma^d \times A^{\mathsf{S}}\,\gamma_s^s\,\alpha\,\alpha^d
\end{aligned}$$

▶ **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 $(\star, f)$ having type ${N_s}^{\mathsf{S}}\,(\star, M)\,(\star, Q) \equiv \mathbf{1} \times ((n : M) \to Q\,n)$ together with a witness of $N^{\mathsf{S}}\,(\star, f)\,(\star, z, s)\,(\star, w, h) \equiv \mathbf{1} \times (f\,z = w) \times ((n : N) \to f\,(s\,n) = h\,n\,(f\,n))$. These equalities are the computation rules of the eliminator. For vectors, the section operation computes to ${V_s}^{\mathsf{S}}\,(\star, W)\,(\star, Q) \equiv \mathbf{1} \times \big((n : \mathbb{N})(v : W\,n) \to Q\,n\,v\big)$ and to $V^{\mathsf{S}}\,(\star, f)\,(\star, nil, cons)\,(\star, nil^d, cons^d) \equiv \mathbf{1} \times (f\,0\,nil = nil^d) \times \big((a : A)(n : \mathbb{N})(v : W\,n) \to f\,(n+1)\,(cons\,a\,n\,v) = cons^d\,a\,n\,(f\,n\,v)\big)$. For the parity families, sections are ${P_s}^{\mathsf{S}}\,(\star, E, O)\,(\star, Q, R) \equiv \mathbf{1} \times ((n : \mathbb{N})(e : E\,n) \to Q\,n\,e) \times ((n : \mathbb{N})(o : O\,n) \to R\,n\,o)$ together with $P^{\mathsf{S}}\,(\star, f, g)\,(\star, e0, eS, oS)\,(e0^d, eS^d, oS^d) \equiv \mathbf{1} \times (f\,0\,e0 = e0^d) \times \big((n : \mathbb{N})(o : O\,n) \to f\,(n+1)\,(eS\,n\,o) = eS^d\,n\,(g\,n\,o)\big) \times \big((n : \mathbb{N})(e : E\,n) \to g\,(n+1)\,(oS\,n\,e) = oS^d\,n\,(f\,n\,e)\big)$. A section for parity displayed algebras consists of two functions f, g which map $e0$ to $e0^d$, eS to eS^d and oS to oS^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 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* $\Omega_s : \mathsf{Con_s}$ *and point context* $\Omega : \mathsf{Con_p}\,\Omega_s$*, there are are sort and point* constructors *in the form of*

$$\mathsf{con_s}\,\Omega : {\Omega_s}^{\mathsf{A}} \text{ and}$$
$$\mathsf{con}\,\Omega \;\; : \Omega^{\mathsf{A}}\,(\mathsf{con_s}\,\Omega)$$

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

$$\mathsf{elim_s}\,\Omega\,\omega_s^d : {\Omega_s}^{\mathsf{S}}\,(\mathsf{con_s}\,\Omega)\,\omega_s^d \text{ with}$$
$$\mathsf{elim}\,\Omega\,\omega^d \;\; : \Omega^{\mathsf{S}}\,(\mathsf{elim_s}\,\Omega\omega_s^d)\,(\mathsf{con}\,\Omega)\,\omega^d.$$

Note that this definition of existence only requires the computation rules contained in $\mathsf{elim}\,\Omega\,\omega^d$ 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* $\mathsf{Sub_s}$ *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*

$$\begin{aligned}
&\mathsf{Sub_s} : \mathsf{Con_s} \to \mathsf{Con_s} \to \mathsf{Set}\\
&\epsilon \quad\ : \mathsf{Sub_s}\ \Gamma_s\ \cdot\\
&_,_ \ : \mathsf{Sub_s}\ \Gamma_s\ \Delta_s \to \mathsf{Tm_s}\ \Gamma_s\ A_s \to \mathsf{Sub_s}\ \Gamma_s\ (\Delta_s \rhd A_s)
\end{aligned}$$

Like with the syntax of signatures itself, $\mathsf{Sub_s}$ *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:*

$$\begin{aligned}
&_[_] : \mathsf{Ty_p}\ \Delta_s \quad \to \mathsf{Sub_s}\ \Gamma_s\ \Delta_s \to \mathsf{Ty_p}\ \Gamma_s && _[_] : \mathsf{Var_s}\ \Delta_s\ A_s \to \mathsf{Sub_s}\ \Gamma_s\ \Delta_s \to \mathsf{Tm_s}\ \Gamma_s\ A_s\\
&_[_] : \mathsf{Con_p}\ \Delta_s \to \mathsf{Sub_s}\ \Gamma_s\ \Delta_s \to \mathsf{Con_p}\ \Gamma_s && _[_] : \mathsf{Tm_s}\ \Delta_s\ A_s \to \mathsf{Sub_s}\ \Gamma_s\ \Delta_s \to \mathsf{Tm_s}\ \Gamma_s\ A_s
\end{aligned}$$

given by the defining rules for substitution

$$\begin{aligned}
&\hat{\Pi}_\mathsf{p}\ T\ A[\sigma] && :\equiv \hat{\Pi}_\mathsf{p}\ T\ (\lambda\tau.(A\ \tau)[\sigma]) && \mathsf{vz}[\sigma,t] && :\equiv t\\
&\mathsf{El}\ a[\sigma] && :\equiv \mathsf{El}\ (a[\sigma]) && (\mathsf{vs}\ x)[\sigma,t] && :\equiv x[\sigma]\\
&(a \Rightarrow_\mathsf{p} A)[\sigma] && :\equiv a[\sigma] \Rightarrow_\mathsf{p} A[\sigma] && (\mathsf{var}\ x)[\sigma] && :\equiv x[\sigma]\\
&\cdot\,[\sigma] && :\equiv \cdot && (t \mathbin{@} \tau)[\sigma] && :\equiv t[\sigma] \mathbin{@} \tau\\
&(\Gamma \rhd A)[\sigma] && :\equiv \Gamma[\sigma] \rhd A[\sigma]
\end{aligned}$$

We can derive from this some useful gadgets of the substitutional calculus: We can define the weakening *of a substitution* $\sigma : \mathsf{Sub_s}\ \Gamma_s\ \Delta_s$ *to the substitution* $\mathsf{wk}_\sigma : \mathsf{Sub_s}\ (\Gamma_s \rhd A_s)\ \Delta_s$ *via recursion on* σ *by* $\mathsf{wk}_\epsilon :\equiv \epsilon$ *and* $\mathsf{wk}_{\sigma,t} :\equiv (\mathsf{wk}_\sigma, \mathsf{vs}\,t)$.

Using wk*, we can then recover the categorical structure of the substitutions by defining the identity* $\mathsf{id}_{\Gamma_s} : \mathsf{Sub_s}\ \Gamma_s\ \Gamma_s$ *by recursion of the context* Γ_s*:* $\mathsf{id}. :\equiv \epsilon$ *and* $\mathsf{id}_{\Gamma_s \rhd A_s} :\equiv (\mathsf{wk}_{\mathsf{id}_{\Gamma_s}}, \mathsf{var\,vz})$. *Composition* $\sigma \circ \delta : \mathsf{Sub_s}\ \Gamma_s\ \Delta_s$ *of substitutions* $\sigma : \mathsf{Sub_s}\ \Theta_s\ \Delta_s$ *and* $\delta : \mathsf{Sub_s}\ \Gamma_s\ \Delta_s$ *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 $\pi_1\,\sigma : \mathsf{Sub_s}\ \Gamma_s\ \Delta_s$ *and* $\pi_2\,\sigma : \mathsf{Tm_{s_s}}\ \Gamma_s\ A_s$ *of a substitution* $\sigma : \mathsf{Sub_s}\ \Gamma_s\ (\Delta_s \rhd A_s)$ *are just projections of* $\times$*-types: Any substitution between* Γ_s *and* $\Delta_s \rhd A_s$ *is of the form* σ, t *and we can just set* $\pi_1\,(\sigma,\ t) :\equiv \sigma$ *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:*

$$\_^\mathsf{A} : \mathsf{Sub_s}\ \Gamma_s\ \Delta_s \to {\Gamma_s}^\mathsf{A} \to {\Delta_s}^\mathsf{A}$$

This is done by setting $\epsilon^\mathsf{A} :\equiv \star$ *and* $(\sigma,\ t)^\mathsf{A} :\equiv (\sigma^\mathsf{A}, t^\mathsf{A})$.

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:*

$$\_^{\mathsf{D}} : (\sigma : \mathsf{Sub_s}\, \Gamma_s\, \Delta_s) \to \Gamma_s{}^{\mathsf{D}}\, \gamma_s \to \Delta_s{}^{\mathsf{D}}\, (\sigma^{\mathsf{A}}\, \gamma_s)$$

These are defined, like in the non-displayed case, by setting $\epsilon^{\mathsf{D}}\, \gamma_s^d :\equiv \star$ *and* $(\sigma, t)^{\mathsf{D}}\, \gamma_s^d :\equiv \left(\sigma^{\mathsf{D}}\, \gamma_s^d, t^{\mathsf{D}}\, \gamma_s^d\right)$.

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

$$\_^{\mathsf{S}} : (\sigma : \mathsf{Sub_s}\, \Gamma_s\, \Delta_s) \to \Gamma_s{}^{\mathsf{S}}\, \gamma_s\, \gamma_s^d \to \Delta_s{}^{\mathsf{S}}\, (\sigma^{\mathsf{A}}\, \gamma_s)\, (\sigma^{\mathsf{D}}\, \gamma_s^d)$$

Again, this is happening componentwise by having: $\epsilon^{\mathsf{S}}\, \gamma_s^s :\equiv \star$ *and* $(\sigma, t)^{\mathsf{S}}\, \gamma_s^s :\equiv \left(\sigma^{\mathsf{S}}\, \gamma_s^s, t^{\mathsf{S}}\, \gamma_s^s\right)$.

▶ **Lemma 12.** *It is easy to check that this definition of algebras of a subtitution respects the substitution calculus given in Definition 10 in the following sense:*

$$\begin{aligned}
(A[\sigma])^{\mathsf{A}}\, \gamma_s &= A^{\mathsf{A}}\, (\sigma^{\mathsf{A}}\, \gamma_s), & \mathsf{wk}_\sigma{}^{\mathsf{A}}\, (\gamma_s, \alpha_s) &= \sigma^{\mathsf{A}}\, \gamma_s, \\
(t[\sigma])^{\mathsf{A}}\, \gamma_s &= t^{\mathsf{A}}\, (\sigma^{\mathsf{A}}\, \gamma_s), & (\pi_1\, \sigma)^{\mathsf{A}}\, \gamma_s &= \mathsf{pr}_1\, (\sigma^{\mathsf{A}}\, \gamma_s), \text{ and} \\
\mathsf{id}^{\mathsf{A}}\, \gamma_s &= \gamma_s, & (\pi_2\, \sigma)^{\mathsf{A}}\, \gamma_s &= \mathsf{pr}_2\, (\sigma^{\mathsf{A}}\, \gamma_s). \\
(\sigma \circ \delta)^{\mathsf{A}}\, \gamma_s &= \sigma^{\mathsf{A}}\, (\delta^{\mathsf{A}}\, \gamma_s), &&
\end{aligned}$$

Proof. We can prove the first rule by recursion on the point type $A : \mathsf{Ty_p}\, \Gamma_s$, the second rule by recursing on the term $t : \mathsf{Tm_s}\, \Gamma_s\, A_s$, 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 *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* $\mathsf{El}(a)$ 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:

▶ **Definition 13** (Point Substitution Calculus)**.** *Let us fix a sort context* $\Gamma_s : \mathsf{Con_s}$*. It turns out that there are three ways to construct reasonable terms of point types in* Γ_s*: Via variables to navigate point contexts and application constructors for each of the two kinds of* Π*-types present in the syntax.*

$$\begin{aligned}
&\mathsf{Var_p} : \mathsf{Con_p}\, \Gamma_s \to \mathsf{Ty_p}\, \Gamma_s \to \mathsf{Set} && \mathsf{var} \;\; : \mathsf{Var_p}\, \Gamma\, A \to \mathsf{Tm_p}\, \Gamma\, A \\
&\mathsf{Tm_p} : \mathsf{Con_p}\, \Gamma_s \to \mathsf{Ty_p}\, \Gamma_s \to \mathsf{Set} && _\hat{@}_ : \mathsf{Tm_p}\, \Gamma\, (\hat{\Pi}_{\mathsf{p}}\, T\, A) \to (\tau : T) \to \mathsf{Tm_p}\, \Gamma\, (A\, \tau) \\
&\mathsf{vz} \;\; : \mathsf{Var_p}\, (\Gamma \rhd A)\, A && _@_ : \mathsf{Tm_p}\, \Gamma\, (a \Rightarrow_{\mathsf{p}} A) \to \mathsf{Tm_p}\, \Gamma\, (\mathsf{El}\, a) \to \mathsf{Tm_p}\, \Gamma\, A \\
&\mathsf{vs} \;\; : \mathsf{Var_p}\, \Gamma\, A \to \mathsf{Var_p}\, (\Gamma \rhd B)\, A &&
\end{aligned}$$

Like with the sort substitutions defined in Definition 10, we define substitutions between point contexts over a fixed sort context $\Gamma_s : \mathsf{Con_s}$ *to be lists of point terms:*

$$\begin{aligned}
&\mathsf{Sub_p} : \mathsf{Con_p}\ \Gamma_s \to \mathsf{Con_p}\ \Gamma_s \to \mathsf{Set}\\
&\epsilon \quad\ : \mathsf{Sub_p}\ \Gamma\ \cdot\\
&-,- \ : \mathsf{Sub_p}\ \Gamma\ \Delta \to \mathsf{Tm_p}\ \Gamma\ A \to \mathsf{Sub_p}\ \Gamma\ (\Delta \rhd A)
\end{aligned}$$

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

$$-[-] : \mathsf{Var_p}\ \Delta\ A \to \mathsf{Sub_p}\ \Gamma\ \Delta \to \mathsf{Tm_p}\ \Gamma\ A \qquad -[-] : \mathsf{Tm_p}\ \Delta\ A \to \mathsf{Sub_p}\ \Gamma\ \Delta \to \mathsf{Tm_p}\ \Gamma\ A$$

which are defined recursively by

$$\begin{aligned}
&\mathsf{vz}[\sigma, t] \quad :\equiv t && (\mathsf{var}\ x)[\sigma] :\equiv x[\sigma]\\
&(\mathsf{vs}\ x)[\sigma, t] :\equiv x[\sigma] && (t \mathbin{\hat{@}} \tau)[\sigma] :\equiv t[\sigma] \mathbin{\hat{@}} \tau\\
& && (t @ u)[\sigma] :\equiv t[\sigma] @ u[\sigma]
\end{aligned}$$

Analogously to Definition 10 we can define the weakening $\mathsf{wk}_\sigma : \mathsf{Sub_p}\ (\Gamma \rhd A)\ \Delta$ *of a point substitution* $\sigma : \mathsf{Sub_p}\ \Gamma\ \Delta$ *along a point type* $A : \mathsf{Ty_p}\ \Gamma_s$*, the identity substitution* $\mathsf{id} : \mathsf{Sub_p}\ \Gamma\ \Gamma$ *and the composition* $\sigma \circ \delta : \mathsf{Sub_p}\ \Gamma\ \Delta$ *of substitutions* $\sigma : \mathsf{Sub_p}\ \Theta\ \Delta$ *and* $\delta : \mathsf{Sub_p}\ \Gamma\ \Theta$*.*

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:

▶ **Definition 14** (Semantics of Point Substitutions & Terms)**.** *Let us fix a sort context* $\Gamma_s : \mathsf{Con_s}$ *and an algebra* $\gamma_s : \Gamma_s^{\mathsf{A}}$ *over it. We can give semantic meaning to point types and point substitution by extending the algebra operator with the following components:*

$$\begin{aligned}
&-^{\mathsf{A}} : \mathsf{Var_p}\ \Gamma\ A \to \Gamma^{\mathsf{A}}\ \gamma_s \to A^{\mathsf{A}}\ \gamma_s && -^{\mathsf{A}} : \mathsf{Sub_p}\ \Gamma\ \Delta \to \Gamma^{\mathsf{A}}\ \gamma_s \to \Delta^{\mathsf{A}}\ \gamma_s\\
&-^{\mathsf{A}} : \mathsf{Tm_p}\ \Gamma\ A \to \Gamma^{\mathsf{A}}\ \gamma_s \to A^{\mathsf{A}}\ \gamma_s
\end{aligned}$$

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 $\gamma_s^d : {\Gamma_s}^{\mathsf{D}}\ \gamma_s$*. 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:*

$$\begin{aligned}
&-^{\mathsf{D}} : (x : \mathsf{Var_p}\ \Gamma\ A) \ \to \Gamma^{\mathsf{D}}\ \gamma_s^d\ \gamma \to A^{\mathsf{D}}\ \gamma_s^d\ (x^{\mathsf{A}}\ \gamma)\\
&-^{\mathsf{D}} : (t : \mathsf{Tm_p}\ \Gamma\ A) \ \to \Gamma^{\mathsf{D}}\ \gamma_s^d\ \gamma \to A^{\mathsf{D}}\ \gamma_s^d\ (t^{\mathsf{A}}\ \gamma)\\
&-^{\mathsf{D}} : (\sigma : \mathsf{Sub_p}\ \Gamma\ \Delta) \to \Gamma^{\mathsf{D}}\ \gamma_s^d\ \gamma \to \Delta^{\mathsf{D}}\ \gamma_s^d\ (\sigma^{\mathsf{A}}\ \gamma)
\end{aligned}$$

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 $\mathsf{El}\,N$ in the context $\Delta := (N : \mathsf{U}, z : \mathsf{El}\,N, s : N \Rightarrow_{\mathsf{p}} \mathsf{El}\,N)$ (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 $\big(\mathsf{Tm}\,\Delta\,(\mathsf{El}\,N), z, \lambda t.s \mathbin{@} t\big) : \Delta^{\mathsf{A}}$. Note that given any other algebra $(A, a, f) : \Delta^{\mathsf{A}}$ and natural number $n : \mathsf{Tm}\,\Delta\,(\mathsf{El}\,N)$, we can simply use the algebra interpretation to obtain the result of the non-dependent elimination principle on n: $n^{\mathsf{A}}\,(A, a, f)$ will have type A, moreover $z^{\mathsf{A}}\,(A, a, f) = a$ and $(s \mathbin{@} t)^{\mathsf{A}}\,(A, a, f) = f\,(t^{\mathsf{A}}\,(A, a, f))$ which are the correct computation rules. The same idea works for displayed algebras: we can use the $-^{\mathsf{D}}$ 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 $\Omega_s : \mathsf{Con_s}$ and the point context $\Omega : \mathsf{Con_p}\,\Omega_s$ which we want to construct by giving $\mathsf{con_s}\,\Omega : {\Omega_s}^{\mathsf{A}}$ and $\mathsf{con}\,\Omega : \Omega^{\mathsf{A}}\,(\mathsf{con_s}\,\Omega)$. Our definition of the constructor uses the trick to index several of the constructions by a second sort or point context together with a sort or point substitution from Ω_s or Ω. We can think of this second context as some sort of a "sub-context" of a fixed context.

▶ **Definition 15** (The Sort Constructor). *The generalised sort constructor consists of:*

$$\mathsf{con}'_{\mathsf{s}} : \mathsf{Sub_s}\,\Omega_s\,\Gamma_s \to {\Gamma_s}^{\mathsf{A}}$$

We can define this recursively via $\mathsf{con}'_{\mathsf{s}}\,\epsilon :\equiv \star$ *and* $\mathsf{con}'_{\mathsf{s}}\,(\sigma,\, t) :\equiv (\mathsf{con}'_{\mathsf{s}}\,\sigma,\, \mathsf{con}'_{\mathsf{s}}\,t)$ *where on sort terms we will define a constructor operation yielding an algebra of the respective sort type:*

$$\mathsf{con}'_{\mathsf{s}} : \mathsf{Tm_s}\,\Omega_s\,A_s \to {A_s}^{\mathsf{A}}$$

This operation will on universe terms consist of the type of point terms in the point context Ω, while on metatheoretic sort functions, it will return a function with constructor of the applied term:

$$\begin{aligned} &\mathsf{con}'_{\mathsf{s}}\,a :\equiv \mathsf{Tm_p}\,\Omega\,(\mathsf{El}\,a) && \text{for } a : \mathsf{Tm_s}\,\Gamma_s\,\mathsf{U} \text{ and} \\ &\mathsf{con}'_{\mathsf{s}}\,t :\equiv \lambda\tau.\mathsf{con}'_{\mathsf{s}}\,(t \mathbin{\hat{@}} \tau) && \text{for } t : \mathsf{Tm_s}\,\Omega_s\,(\hat{\Pi}_{\mathsf{s}}\,T\,A_s). \end{aligned}$$

This construction is already enough to give the sort constructor required in Theorem 9 by pinning the substitution to be the identity: $\mathsf{con_s}\,\Omega :\equiv \mathsf{con}'_{\mathsf{s}}\,\mathsf{id}_{\Omega_s} : {\Omega_s}^{\mathsf{A}}$.

It is not immediately clear that the operation on substitutions and the operation on sort terms is well-behaved under the pullback along substitutions. We can, however, show that this is indeed the case:

▶ **Lemma 16** (Coherence of the Sort Constructor). *For all substitutions $\sigma : \mathsf{Sub_s}\,\Omega_s\,\Gamma_s$ and $t : \mathsf{Tm_s}\,\Gamma_s\,A_s$, taking a constructor of t pulled back along σ has the same effect as taking the term algebra over the context algebra generated by the constructor on σ, that is, $t^{\mathsf{A}}\,(\mathsf{con}'_{\mathsf{s}}\,\sigma) = \mathsf{con}'_{\mathsf{s}}\,(t[\sigma])$.*

Proof. Let us first do a case distinction on the substitution. If it is ϵ, then $\Gamma_s = \cdot$, 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, lets recurse on the term and see that

$$\begin{aligned}
(\mathsf{var}\,\mathsf{vz})^{\mathsf{A}}(\mathsf{con}'_{\mathsf{s}}\,(\sigma,\, s)) &= \mathsf{vz}^{\mathsf{A}}\,(\mathsf{con}'_{\mathsf{s}}\,\sigma, \mathsf{con}'_{\mathsf{s}}\,s) \\
&= \mathsf{con}'_{\mathsf{s}}\,s \\
&= \mathsf{con}'_{\mathsf{s}}\,(\mathsf{var}\,\mathsf{vz}[\sigma,\, s]),
\end{aligned}$$

$$\begin{aligned}
(\mathsf{var}\,(\mathsf{vs}\,x))^{\mathsf{A}}\,(\mathsf{con}'_{\mathsf{s}}\,(\sigma,\, s)) &= \mathsf{var}\,(\mathsf{vs}\,x)^{\mathsf{A}}(\mathsf{con}'_{\mathsf{s}}\,\sigma, \mathsf{con}'_{\mathsf{s}}\,s) \\
&= (\mathsf{var}\,x)^{\mathsf{A}}\,(\mathsf{con}'_{\mathsf{s}}\,\sigma) \\
&= \mathsf{con}'_{\mathsf{s}}\,(\mathsf{var}\,x[\sigma]) && \text{by induction} \\
&= \mathsf{con}'_{\mathsf{s}}\,(\mathsf{var}\,(\mathsf{vs}\,x)[\sigma,\, s]), && \text{and lastly}
\end{aligned}$$

$$\begin{aligned}
(f \mathbin{\hat{@}} \tau)^{\mathsf{A}}\,(\mathsf{con}'_{\mathsf{s}}\,(\sigma,\, s)) &= f^{\mathsf{A}}\,(\mathsf{con}'_{\mathsf{s}}\,(\sigma,\, s))\,\tau \\
&= \mathsf{con}'_{\mathsf{s}}\,(f[\sigma,\, s])\,\tau && \text{by induction} \\
&= \mathsf{con}'_{\mathsf{s}}\,((f\,\tau)[\sigma,\, s]) && \text{for } f : \hat{\Pi}_{\mathsf{s}}\,T\,B. \qquad \blacktriangleleft
\end{aligned}$$

We can now use this lemma to do a trick with $\mathsf{con}\,\Omega$ similar to the trick we did for $\mathsf{con_s}\,\Omega$: Replace the fixed point context with a variable one, together with a substitution from Ω, 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:*

$$\mathsf{con}' : \mathsf{Sub_p}\,\Omega\,\Gamma \to \Gamma^{\mathsf{A}}\,(\mathsf{con_s}\,\Omega) \qquad\qquad \mathsf{con}' \colon \mathsf{Tm_p}\,\Omega\,A \to A^{\mathsf{A}}\,(\mathsf{con_s}\,\Omega)$$

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

$$t : \mathsf{con}'_{\mathsf{s}}\,a \equiv \mathsf{con}'_{\mathsf{s}}(a[\mathsf{id}]) = a^{\mathsf{A}}\,(\mathsf{con}'_{\mathsf{s}}\,\mathsf{id}_{\Omega_s}) \equiv (\mathsf{El}\,a)^{\mathsf{A}}\,(\mathsf{con_s}\,\Omega),$$

which allows us to define the constructor operator by

$$\begin{aligned}
&\mathsf{con}'\,t :\equiv t && \text{for } t : \mathsf{Tm_p}\,\Omega\,(\mathsf{El}\,a), \\
&\mathsf{con}'t \;:\equiv \lambda\tau.\,\mathsf{con}'\,(t \mathbin{\hat{@}} \tau) && \text{for } t : \mathsf{Tm_p}\,\Omega\,(\hat{\Pi}_{\mathsf{p}}\,T\,A), \text{ and} \\
&\mathsf{con}'t \;:\equiv \lambda u.\,\mathsf{con}'\,(t \mathbin{@} u) && \text{for } t : \mathsf{Tm_p}\,\Omega\,(a \Rightarrow_{\mathsf{p}} A).
\end{aligned}$$

This concludes the definition of the constructors, since we can set, like for the sort constructor, $\mathsf{con}\,\Omega :\equiv \mathsf{con}'\,\mathsf{id}_\Omega : \Omega^{\mathsf{A}}\,(\mathsf{con_s}\,\Omega)$.

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* $\sigma : \mathsf{Sub_p}\,\Omega\,\Gamma$ *and point terms* $t : \mathsf{Tm_p}\,\Gamma\,A$*, pulling back has the same effect as the point constructor as in* $t^{\mathsf{A}}\,(\mathsf{con}'\sigma) = \mathsf{con}'t[\sigma]$.

The proof is by induction on σ 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 $\omega^d_s : \Omega^{\mathsf{D}}_s\,(\mathsf{con_s}\,\Omega)$ and $\omega^d : \Omega^{\mathsf{D}}\,\omega^d_s\,(\mathsf{con}\,\Omega)$. We will proceed in the same order as for the constructors and start by generalizing $\mathsf{elim_s}\,\Omega\,\omega^d$ to arbitrary subcontexts of Ω 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:*

$$\mathsf{elim_s}' : (\sigma : \mathsf{Sub_s}\ \Omega_s\ \Gamma_s) \to \Gamma_s{}^{\mathsf{S}}\ (\sigma^{\mathsf{A}}\ (\mathsf{con_s}\ \Omega))\ (\sigma^{\mathsf{D}}\ \omega_s^d)$$
$$\mathsf{elim_s}' : (t : \mathsf{Tm_s}\ \Omega_s\ A_s) \to A_s{}^{\mathsf{S}}\ (t^{\mathsf{A}}\ (\mathsf{con_s}\ \Omega))\ (t^{\mathsf{D}}\ \omega_s^d)$$

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

$$\mathsf{elim_s}'\ a :\equiv \lambda t.\ t^{\mathsf{D}}\ \omega^d \qquad \text{for } a : \mathsf{Tm_s}\ \Omega_s\ \mathsf{U} \text{ and}$$
$$\mathsf{elim_s}'\ t :\equiv \lambda \tau.\ \mathsf{elim_s}'\ (t \mathbin{\hat{@}} \tau) \qquad \text{for } t : \mathsf{Tm_s}\ \Omega_s\ (\hat{\Pi}_{\mathsf{s}}\ T\ A_s).$$

Now we set $\mathsf{elim_s}\ \Omega\ \omega^d :\equiv \mathsf{elim_s}'\ \mathsf{id}_{\Omega_s}$.

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

▶ **Lemma 20.** *Given a sort substitution* $\sigma : \mathsf{Sub_s}\ \Omega_s\ \Gamma_s$ *and a sort term* $t : \mathsf{Tm_s}\ \Gamma_s\ A_s$, *the eliminator of a pulled back term is the section of the term, evaluated at the eliminator on a substitution:* $\mathsf{elim_s}'\ (t[\sigma]) = t^{\mathsf{S}}\ (\mathsf{elim_s}'\ \sigma)$.

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* $\mathsf{elim_s}\ \Omega\ \omega^d$ *to be a section of subcontexts of* Ω *and of point terms in* Ω:

$$\mathsf{elim}' : (\sigma : \mathsf{Sub_p}\ \Omega\ \Gamma) \to \Gamma^{\mathsf{S}}\ (\mathsf{elim_s}\ \Omega\ \omega^d)\ (\sigma^{A}\ (\mathsf{con}\ \Omega))\ (\sigma^{\mathsf{D}}\ \omega^d)$$
$$\mathsf{elim}' : (t : \mathsf{Tm_p}\ \Omega\ A) \to A^{\mathsf{S}}\ (\mathsf{elim_s}\ \Omega\ \omega^d)\ (t^{A}\ (\mathsf{con}\ \Omega))\ (t^{\mathsf{D}}\ \omega^d)$$

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

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

$$a^{\mathsf{S}}\ (\mathsf{elim_s}'\ \mathsf{id}_{\Omega_s})\ (t^{\mathsf{A}}\ (\mathsf{con}\ \Omega)) = a^{\mathsf{S}}\ (\mathsf{elim_s}'\ \mathsf{id}_{\Omega_s})\ t = \mathsf{elim_s}'\ a\ t = t^{\mathsf{D}}\ \omega^d.$$

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

$$\mathsf{elim}'t :\equiv \lambda \tau.\ \mathsf{elim}'\ (t \mathbin{\hat{@}} \tau) \text{ for } t : \mathsf{Tm_p}\ \Omega\ (\hat{\Pi}_{\mathsf{p}}\ T\ A), \text{ and}$$
$$\mathsf{elim}'t :\equiv \lambda u.\ \mathsf{elim}'\ (t \mathbin{@} u) \text{ for } t : \mathsf{Tm_p}\ \Omega\ (a \Rightarrow_{\mathsf{p}} A).$$ ◀

Proof of Theorem 9. Lemma 21 completes the construction of the eliminator and setting $\mathsf{elim}\ \Omega\ \omega^d :\equiv \mathsf{elim}'\ \mathsf{id}_{\Omega}$ 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

1 Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers — constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.

2 Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A type and scope safe universe of syntaxes with binding: Their semantics and proofs. *Proc. ACM Program. Lang.*, 2(ICFP):90:1–90:30, July 2018. doi:10.1145/3236785.

3 Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, pages 293–310, Cham, 2018. Springer International Publishing.

4 Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. Indexed containers. *Journal of Functional Programming*, 25, 2015. doi:10.1017/S095679681500009X.

5 Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *ACM SIGPLAN Notices*, volume 51(1), pages 18–29. ACM, 2016.

6 Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification*, PLPV '07, pages 57–68, New York, NY, USA, 2007. ACM. doi:10.1145/1292597.1292608.

7 Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic, 13th International Workshop, CSL '99*, pages 453–468, 1999.

8 Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. Towards certified meta-programming with typed Template-Coq. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*, pages 20–39. Springer, 2018. doi:10.1007/978-3-319-94821-8_2.

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

10 Henning Basold and Herman Geuvers. Type Theory based on Dependent Inductive and Coinductive Types. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of LICS '16*, pages 327–336. ACM, 2016. doi:10.1145/2933575.2934514.

11 Henning Basold, Herman Geuvers, and Niels van der Weide. Higher inductive types in programming. *Journal of Universal Computer Science*, 23:63–88, 2017.

12 Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nord. J. Comput.*, 10(4):265–289, 2003.

13 Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free — parametricity for dependent types. *Journal of Functional Programming*, 22(02):107–152, 2012. doi:10.1017/S0956796812000056.

14 Evan Cavallo and Robert Harper. Higher inductive types in cubical computational type theory. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290314.

15 James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 3–14, New York, NY, USA, 2010. ACM. doi:10.1145/1863543.1863547.

16 Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015. doi:10.1007/978-3-319-21401-6_26.

17 Peter Dybjer. Inductive families. *Formal aspects of computing*, 6(4):440–465, 1994.

18 Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65:525–549, 2000.

19 Peter Dybjer and Hugo Moeneclaey. Finitary higher inductive types in the groupoid model. *Electronic Notes in Theoretical Computer Science*, 336:119–134, 2018. The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII). doi:10.1016/j.entcs.2018.03.019.

20 Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In *Typed Lambda Calculi and Applications, volume 1581 of Lecture Notes in Computer Science*, pages 129–146. Springer, 1999.

21 Peter Dybjer and Anton Setzer. Indexed induction-recursion. *J. Log. Algebr. Program.*, 66(1):1–49, 2006. doi:10.1016/j.jlap.2005.07.001.

22 Marcelo Fiore, Andrew M. Pitts, and S. C. Steenkamp. Constructing Infinitary Quotient-Inductive Types. *arXiv e-prints*, page arXiv:1911.06899, November 2019. arXiv:1911.06899.

23 Martin Hofmann. Conservativity of equality reflection over intensional type theory. In *TYPES 95*, pages 153–164, 1995.

24 Martin Hofmann. Syntax and semantics of dependent types. In *Semantics and Logics of Computation*, pages 79–130. Cambridge University Press, 1997.

25 Ambrus Kaposi and András Kovács. A syntax for higher inductive-inductive types. In *3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018)*, volume 108 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:18, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.FSCD.2018.20.

26 Ambrus Kaposi and András Kovács. Signatures and induction principles for higher inductive-inductive types. *arXiv preprint arXiv:1902.00297*, 2019.

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

28 Peter LeFanu Lumsdaine and Michael Shulman. Semantics of higher inductive types. *Mathematical Proceedings of the Cambridge Philosophical Society*, pages 1–50, 2019. doi:10.1017/S030500411900015X.

29 Fredrik Nordvall Forsberg. *Inductive-inductive definitions*. PhD thesis, Swansea University, 2013.

30 Ulf Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009.

31 Nicolas Oury. *Extensionality in the calculus of constructions*, pages 278–293. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. doi:10.1007/11541868_18.

32 Christine Paulin-Mohring. Inductive definitions in the system Coq - rules and properties. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications (TLCA)*, number 664 in Lecture Notes in Computer Science, 1993.

33 Kent Petersson and Dan Synek. A set constructor for inductive sets in martin-löf's type theory. In David H. Pitt, David E. Rydeheard, Peter Dybjer, Andrew M. Pitts, and Axel Poigné, editors, *Category Theory and Computer Science, Manchester, UK, September 5-8, 1989, Proceedings*, volume 389 of *Lecture Notes in Computer Science*, pages 128–140. Springer, 1989. doi:10.1007/BFb0018349.

34 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

35 Théo Winterhalter, Matthieu Sozeau, and Nicolas Tabareau. Eliminating reflection from type theory. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 91–103. ACM, 2019. doi:10.1145/3293880.3294095.

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* $\mathsf{IW}^{o,r}_{A,B} : I \to \mathsf{Set}$ *for input data*

$$
\begin{aligned}
&I : \mathsf{Set} && \text{(“index type”)}\\
&A : \mathsf{Set} && \text{(“shapes”)}\\
&B : A \to \mathsf{Set} && \text{(“positions”)}\\
&o : A \to I && \text{(“output indices”) and}\\
&r : (a : A) \to B\,a \to I && \text{(“recursive indices”)}
\end{aligned}
$$

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

$$\frac{a : A \qquad c : (b : B\,a) \to \mathsf{IW}^{o,r}_{A,B}\,(r\,a\,b)}{\mathsf{sup}\,a\,b : \mathsf{IW}^{o,r}_{A,B}\,(o\,a)}$$

admitting a dependent eliminator

$$\frac{\begin{array}{ll}C : \{i : I\} \to \mathsf{IW}^{o,r}_{A,B}\,i \to \mathsf{Set} & p : (a : A)\left(c : (b : B\,a) \to \mathsf{IW}^{o,r}_{A,B}\,(r\,a\,b)\right)\\ & \qquad \to \left((b : B\,a) \to C\,(c\,b)\right) \to C\,(\mathsf{sup}\,a\,c)\end{array}}{\mathsf{elim}_{\mathsf{IW}}\,C\,p : (i : I)(w : \mathsf{IW}^{o,r}_{A,B}\,i) \to C\,w}$$

with the reduction rule

$$\mathsf{elim}_{\mathsf{IW}}\,C\,p\,(o\,a)\,(\mathsf{sup}\,a\,c) \equiv p\,a\,c\,(\lambda b.\,\mathsf{elim}_{\mathsf{IW}}\,C\,p\,(r\,a\,b)\,(c\,b)).$$

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.

i	$I_i : \mathsf{Set}$	$A_i : \mathsf{Set}$	$B_i : A_i \to \mathsf{Set}$	$o_i : A_i \to I_i$	$r_i : (a : A_i) \to B_i\, a \to I_i$
$\mathsf{Ty_s}$	$\mathbf{1}$	$\mathbf{1}$ $+\mathsf{Set}$	$\mathsf{inl}\,\star \mapsto \mathbf{0}$ $\mathsf{inr}\,T \mapsto T$	$- \mapsto \star$	$- \mapsto \star$
$\mathsf{Con_s}$	$\mathbf{1}$	$\mathbf{1}$ $+\mathsf{Ty_s}$	$\mathsf{inl}\,\star \mapsto \mathbf{0}$ $\mathsf{inr}\,B \mapsto \mathbf{1}$	$- \mapsto \star$	$- \mapsto \star$
$\mathsf{Var_s}$	$\mathsf{Con_s}$	$\mathsf{Con_s}$ $+\mathsf{Con_s} \times \mathsf{Ty_s}$	$\mathsf{inl}\,\Gamma_s \mapsto \mathbf{0}$ $\mathsf{inr}\,(\Gamma_s, B') \mapsto \mathbf{1}$	$\mathsf{inl}\,\Gamma_s \mapsto (\Gamma_s,\, B)$ $\mathsf{inr}\,(\Gamma_s, B') \mapsto (\Gamma_s,\, B')$	$-$ $\mathsf{inr}\,(\Gamma_s, B') \star \mapsto \Gamma_s$
$\mathsf{Tm_s}\,\Gamma_s$	$\mathsf{Ty_s}$	$\mathsf{Ty_s}$ $+(T : \mathsf{Set}) \times (T \to \mathsf{Ty_s}) \times T$	$\mathsf{inl}\,B \mapsto \mathbf{0}$ $\mathsf{inr}\,- \mapsto \mathbf{1}$	$\mathsf{inl}\,B \mapsto \mathbf{0}$ $\mathsf{inr}\,(T, B, \tau) \mapsto B\,\tau$	$-$ $\mathsf{inr}\,(T, B, \tau) \star \mapsto \hat{\Pi}_\mathsf{s}\, T\, B$
$\mathsf{Sub_s}\,\Gamma_s$	$\mathbf{1}$	$\mathbf{1}$ $+(B : \mathsf{Ty_s}) \times \mathsf{Tm_s}\,\Gamma_s\,B$	$\mathsf{inl}\,\star \mapsto \mathbf{0}$ $\mathsf{inr}\,(B, t) \mapsto \mathbf{1}$	$- \mapsto \star$	$- \mapsto \star$
$\mathsf{Ty_p}\,\Gamma_s$	$\mathbf{1}$	$\mathsf{Tm_s}\,\Gamma\,\mathsf{U}$ $+\mathsf{Set}$ $+\mathsf{Tm_s}\Gamma\,\mathsf{U}$	$\mathsf{inl}\,a \mapsto \mathbf{0}$ $\mathsf{inr}\,(\mathsf{inl}\,\tau) \mapsto T$ $\mathsf{inr}\,(\mathsf{inr}\,a) \mapsto \mathbf{1}$	$- \mapsto \star$	$- \mapsto \star$
$\mathsf{Con_p}\,\Gamma_s$	$\mathbf{1}$	$\mathbf{1}$ $+\mathsf{Ty_s}\,\Gamma_s$	$\mathsf{inl}\,\star \mapsto \mathbf{0}$ $\mathsf{inr}\,A \mapsto \mathbf{1}$	$- \mapsto \star$	$- \mapsto \star$
$\mathsf{Var_p}\,-A$	$\mathsf{Con_p}\,\Gamma_s$	$\mathsf{Con_p}\,\Gamma_s$ $+\mathsf{Con_p}\,\Gamma_s \times \mathsf{Ty_p}\,\Gamma_s$	$\mathsf{inl}\,\Gamma \mapsto \mathbf{0}$ $\mathsf{inr}\,(\Gamma, A') \mapsto \mathbf{1}$	$\mathsf{inl}\,\Gamma \mapsto (\Gamma,\, A)$ $\mathsf{inr}\,(\Gamma, A') \mapsto (\Gamma,\, A')$	$-$ $\mathsf{inr}\,(\Gamma, A') \star \mapsto \Gamma$
$\mathsf{Tm_p}\,\Gamma$	$\mathsf{Ty_p}\,\Gamma_s$	$(A : \mathsf{Ty_p}\,\Gamma_s) \times \mathsf{Var_p}\,\Gamma\,A$	$\mathsf{inl}\,(A, v) \mapsto \mathbf{0}$ $\mathsf{inr}\,(\mathsf{inl}\,-) \mapsto \mathbf{2}$ $\mathsf{inr}\,(\mathsf{inr}\,-) \mapsto \mathbf{1}$	$\mathsf{inl}\,(A, v) \mapsto A$ $\mathsf{inr}\,(\mathsf{inl}\,(A, a)) \mapsto A$ $\mathsf{inr}\,(\mathsf{inr}\,(T, A, \tau)) \mapsto A\,\tau$	$-$ $\mathsf{inr}\,(\mathsf{inl}\,(A, a))\,0 \mapsto (a \Rightarrow_\mathsf{p} A)$ $\mathsf{inr}\,(\mathsf{inl}\,(A, a))\,1 \mapsto \mathsf{El}\,a$ $\mathsf{inr}\,(\mathsf{inr}\,(T, A, \tau)) \star \mapsto \hat{\Pi}_\mathsf{p}\, T\, A$
$\mathsf{Sub_p}\,\Gamma$	$\mathbf{1}$	$\mathbf{1}$ $+(A : \mathsf{Ty_p}\,\Gamma_s) \times \mathsf{Tm_p}\,\Gamma\,A$	$\mathsf{inl}\,\star \mapsto \mathbf{0}$ $\mathsf{inr}\,(A, t) \mapsto \mathbf{1}$	$- \mapsto \star$	$- \mapsto \star$

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

$$
\begin{aligned}
\mathsf{Ty_s} &:\equiv \mathsf{IW}^{o_{\mathsf{Ty_s}}, r_{\mathsf{Ty_s}}}_{A_{\mathsf{Ty_s}}, B_{\mathsf{Ty_s}}}\ \star\,,\\
\mathsf{Con_s} &:\equiv \mathsf{IW}^{o_{\mathsf{Con_s}}, r_{\mathsf{Con_s}}}_{A_{\mathsf{Con_s}}, B_{\mathsf{Con_s}}}\ \star\,,\\
\mathsf{Var_s}\ -B &:\equiv \mathsf{IW}^{o_{\mathsf{Var_s}}\ B, r_{\mathsf{Var_s}}\ B}_{A_{\mathsf{Var_s}}\ B, B_{\mathsf{Var_s}}\ B},\\
\mathsf{Tm_s}\ \Gamma_s &:\equiv \mathsf{IW}^{o_{\mathsf{Tm_s}}\ \Gamma_s, r_{\mathsf{Tm_s}}\ \Gamma_s}_{A_{\mathsf{Tm_s}}\ \Gamma_s, B_{\mathsf{Tm_s}}\ \Gamma_s},\\
\mathsf{Sub_s}\ \Gamma_s &:\equiv \mathsf{IW}^{o_{\mathsf{Sub_s}}\ \Gamma_s, r_{\mathsf{Sub_s}}\ \Gamma_s}_{A_{\mathsf{Sub_s}}\ \Gamma_s, B_{\mathsf{Sub_s}}\ \Gamma_s},\\
\mathsf{Ty_p}\ \Gamma_s &:\equiv \mathsf{IW}^{o_{\mathsf{Ty_p}}, r_{\mathsf{Ty_p}}}_{A_{\mathsf{Ty_p}}, B_{\mathsf{Ty_p}}}\ \star\,,\\
\mathsf{Con_p}\ \Gamma_s &:\equiv \mathsf{IW}^{o_{\mathsf{Con_p}}, r_{\mathsf{Con_p}}}_{A_{\mathsf{Con_p}}, B_{\mathsf{Con_p}}}\ \star\,,\\
\mathsf{Var_p}\ -A &:\equiv \mathsf{IW}^{o_{\mathsf{Var_p}}\ A, r_{\mathsf{Var_p}}\ A}_{A_{\mathsf{Var_p}}\ A, B_{\mathsf{Var_p}}\ A},\\
\mathsf{Tm_p}\ \Gamma &:\equiv \mathsf{IW}^{o_{\mathsf{Tm_p}}\ \Gamma, r_{\mathsf{Tm_p}}\ \Gamma}_{A_{\mathsf{Tm_p}}\ \Gamma, B_{\mathsf{Tm_p}}\ \Gamma},\\
\mathsf{Sub_p}\ \Gamma &:\equiv \mathsf{IW}^{o_{\mathsf{Sub_p}}\ \Gamma, r_{\mathsf{Sub_p}}\ \Gamma}_{A_{\mathsf{Sub_p}}\ \Gamma, B_{\mathsf{Sub_p}}\ \Gamma},
\end{aligned}
$$

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 (σ, s), since there are no point terms in the empty point context. Now, by recursion on the term we see that

$$\begin{aligned}
(\mathsf{var}\,\mathsf{vz})^{\mathsf{A}}\,(\mathsf{con}'\,(\sigma,\,s)) &= (\mathsf{var}\,\mathsf{vz})^{\mathsf{A}}\,(\mathsf{con}'\,\sigma, \mathsf{con}'\,s)\\
&= \mathsf{con}'\,s\\
&= \mathsf{con}'\,(\mathsf{var}\,\mathsf{vz}[\sigma,\,s]),\\
(\mathsf{var}(\mathsf{vs}\,x))^{\mathsf{A}}\,(\mathsf{con}'\,(\sigma,\,s)) &= (\mathsf{var}\,(\mathsf{vs}\,x))^{\mathsf{A}}\,(\mathsf{con}'\,\sigma, \mathsf{con}'\,s)\\
&= (\mathsf{var}\,x)^{\mathsf{A}}\,(\mathsf{con}'\,\sigma)\\
&= \mathsf{con}'\,(\mathsf{var}\,x[\sigma]) && \text{by induction}\\
&= \mathsf{con}'\,(\mathsf{var}\,(\mathsf{vs}\,x)[\sigma,\,s]),\\
(t\mathbin{@}u)^{\mathsf{A}}\,(\mathsf{con}'\,\sigma) &= t^{\mathsf{A}}\,(\mathsf{con}'\,\sigma)\;\left(u^{\mathsf{A}}\,(\mathsf{con}'\,\sigma)\right)\\
&= \mathsf{con}'\,(t[\sigma])\,(\mathsf{con}'\,(u[\sigma])) && \text{by induction}\\
&= \mathsf{con}'\,((t\mathbin{@}u)[\sigma]),\ \text{and}\\
(t\mathbin{\hat{@}}\tau)^{\mathsf{A}}\,(\mathsf{con}'\,\sigma) &= t^{\mathsf{A}}\,(\mathsf{con}'\,\sigma)\,\tau\\
&= \mathsf{con}'\,(t[\sigma])\,\tau && \text{by induction}\\
&= \mathsf{con}'\,((t\mathbin{\hat{@}}\tau)[\sigma]).
\end{aligned}$$

◀