Indexed Kind Checking for Hierarchical Database Schemas

Anonymous Author(s)

Abstract

Types are an appropriate tool for describing the structure of hierarchical databases. But two common database idioms, *foreign keys* and *secondary indexes*, cannot be expressed using standard type systems. We present a lanaguage in which a database schema is described using a type layered over a dependently sorted index language. The index language consists of keys and relations on keys. Multiple occurrences of the same index-level relation inside of a schema specify a correlation between two distinct portions of a database instance; both secondary indexes and foreign keys can be specified in this manner. We present an algorithmic system of judgments for deciding an indexed kinding relation for schemas and provide it with an intrinsic denotational semantics.

Keywords: Indexed types, Database schemas, Dependent types

ACM Reference Format:
Anonymous Author(s). 2024. Indexed Kind Checking for Hierarchical Database Schemas. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Programmers increasingly choose document database systems due to their high performance and intuitive hierarchical structure. Schemas for such databases are often intentionally omitted under the premise that they inhibit rapid iteration. However, we disagree with this premise. We believe that the arguments in favor of database schemas are analogous to those in favor of static types in programming langauges. By pairing data with its indended meaning, a schema increases programmer comprehension: a programmer may not know what do with a record field called *widgetId*, but a good schema could convey this information, specifying that *widgetId* refers to a structure stored in a specific location of the database.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The type syntax of a standard programming lanaguage such as Typescript is for the most part an appropriate tool to describe the structure of modern hierarchical databases. However, two common features of hierarchical databases lie beyond the reach of standard type systems. First, a database may store *foreign keys*: values, such as the aforementioned *widgetId*, used to refer to entities stored elsewhere in the database. Second, a database may replicate a dataset in multiple configurations (a.k.a. *secondary indexes*) to accomodate multiple access patterns in an efficient, spatially local manner.

The prevalence of foreign keys in hierarchical databases constrasts sharply against the in-memory data structures one typically finds in general purpose lanaguages. A pointer in a language such as C++ is not a foreign key, because statically our concern is the type of referenced data rather than its location. In contrast, a foreign key is drawn from a specific subset of references of a given type. For example, consider an e-commerce database which records purchases performed by customers. Each purchase holds a *creditCardId* foreign key referring to the credit card used for the payment; this card is not drawn from a global pool of all cards, but instead the pool of cards owned by the customer which made the purchase.

Secondary indexes are another common and useful database structure beyond the reach of traditional type systems. We introduce secondary indexes with an example. An e-commerce company wishes to issue a recall on a product. To do so, it must obtain a list of all customers that have purchased an item known to be defective; to compute this list efficiently, the set of all customers which purchased the item must be stored in nearby addresses. At another time, a customer may wish to obtain a list of all items he has purchased from the site; to perform this operation efficiently, we must store the set of all of a customer's purchased items in nearby addresses. These two requirements are at odds if our database merely stores a single collection of all purchases processed, i.e. a "primary index" that maps each purchase ID to a tuple containing the customer making the purchase, the item being purchased, and all other associated information. However, if we redundantly store customer-to-items and item-to-customers maps, we can satisfy both requirements at once. Such redundant data structures are called secondary indexes.

In practice, it is common to find a foreign key that accidentally refers to a deleted entity. Such a foreign key is a time-bomb which will raise an exception the next time the data is accessed. Furthermore, secondary indexes can become out-of-sync due to programmer errors. A formal schema including specifications of foreign keys and indexes could be compiled to a validation routine to detect these errors.

We make the following contributions.

- We design an expressive schema language for hierarchical databases featuring foreign keys and secondary indexes.
- We provide a category-theoretic characterization of the dependently-sorted analog of simple products, which play a central role in our schema language. This characterization is similar to, but more general than, exisiting characterizations e.g. by Palmgren [4]; in particular, we identify a *transpose distribution* property necessary for dependent simple products in models whose total categories, unlike Palmgren's, are not thin.

Since we are working with two distinct concepts which are both typically referred to as *index* – indexes in the sense of database indexes and indices in the sense of indexed type checking – we refer to database secondary indexes as *reconfigurations* in the sequel.

2 Example

2.1 Foreign Keys

The purpose of a schema is to identify a set of database instances, so before examining our first schema we briefly define some formalisms for describing database instances. We consider *database instances* (also called *instances*) as partial mappings from lists of strings to strings. Given a database instance f and a list of strings σ, we obtain an instance $f|_\sigma$ called *the restriction of f to σ*, defined for lists σ' as $f|_\sigma(\sigma') \overset{def}{=} f(\sigma +\!\!+ \sigma')$, where $\sigma +\!\!+ \sigma'$ is the concatenation of σ and σ'. By an abuse of language, we say "f maps σ to $f|_\sigma$". We write $[s_1, \ldots, s_n]$ for the list containing the n strings $s_1, \ldots, s_n$.

Figure 1 shows a schema fragment for an e-commerce database. It begins by declaring five relations: first four unary relations (predicates) for item ids, customer ids, purchase ids, and card types, and then a ternary relation for card ids. The relation variables and their classifiers have been bolded and colored blue to indicate that they are part of the *index language*. This simple language defines conceptual entities such as predicates and relations, divorced from the physical details of where these entities are stored. The index langauge is layered beneath the schema language, i.e. schemas may depend on indices but not vice versa.

Lines 7-10 use the record type constructor to define a subschema representing credit cards. It denotes the set of instances which map the list ["billingAddr"] to any string and the list ["cardType"] to any string satisfying the **CardType** predicate. We consider the latter a foreign key; rather than taking the naive view that a foreign key is a reference to a specific location in the database, we instead take it to be a string satisfying a predicate in our index-level context (i.e. an element of a conceptual set). We believe our approach is cleaner and more declarative than the naive approach. Because a conceptual set may be replicated at multiple locations in a database, it would not make sense for a foreign key to refer to one of these locations rather than another.

```
1  ⋁ λ (ItemId : str -> prop).
2  ⋁ λ (CustId : str -> prop).
3  ⋁ λ (PurchaseId : str -> prop).
4  ⋁ λ (CardType : str -> prop).
5  ⋁ λ (CardId : (x:str) -> prf (CustId x) -> str -> prop).
6
7  type Card = {
8    billingAddr : str
9    cardType    : { x : str | CardType x }
10 }
11
12 type Purchase = λ(cust : str).λ(prf (CustId cust)). {
13   itemId : { x : str | ItemId x },
14   cardId : { x : str | CardId cust x }
15 }
16
17 type Customer = λ(cust : str).λ(prf (CustId cust)). {
18   purchases : {
19     [p : str] : PurchaseId p > Purchase cust
20   },
21   cards     : {
22     [card : str] : CardId cust card > Card
23   }
24 }
25
26 {
27   cardTypes : { [x : str] : CardType x > "*" }
28   customers : { [x : str] : CustId x > Customer x}
29 }
```

Figure 1. Schema for e-commerce database with foreign keys

Lines 12-15 define an index-to-type operator called *Purchase*, which maps a string **cust** and a proof that **cust** satisfies the **CustId** predicate to a type representing a purchase made by customer **cust**. In the application **CardId cust x**, the second argument to **CardId** is missing. This is because it is a proof; since any two proofs of the same proposition are interchangeable, we apply proofs implicitly to reduce visual clutter.

```
⋁ λ (ItemId : str -> prop).
⋁ λ (CustId : str -> prop).
⋁ λ (Purchased :
      (x : str) -> prf (CustId x) ->
      (y : str) -> prf (ItemId y) -> prop).

{
  custToItem : {
    [c,i : str] : CustId c,ItemId i,Purchased c i > "*"
  }
  itemToCust : {
    [i,c : str] : CustId c,ItemId i,Purchased c i > "*"
  }
}
```

Figure 2. Reconfigurations in an e-commerce database

Lines 17-24 define a index-to-type operator `Customer` mapping a string **cust** and a proof of **CustId cust** to a type representing customers with id **cust**. It denotes the set of instances which

- Map all lists ["purchases", **p**] such that **purchaseId p** holds to instances of type `Purchase` **cust**, and
- Map all lists ["cards", **card**] such that **CardId cust card** holds to instances of type `Card`.

Finally, lines 26-29 define the schema using a record type. It denotes the set of instances which

- Map ["cardTypes"] to a subinstance that stores the set of all card types by mapping each string **x** satisfying the **CardType** predicate to the string `"*"`, and
- Map ["customers"] to a dictionary that maps every **x** satisfying the **CustId** predicate to a subschema of type `Customer` **x**.

Reconfigurations

Figure 2 shows a schema for a simple database with reconfigurations. An instance satisfies the schema if

- For each customer/item pair **c**, **i** satisfying **Purchased c i**, it maps `["custToItem", c, i]` to the string `"*"` and `["itemToCust", i, c]` to `"*"`.
- It is not defined anywhere else.

Such an instance allows us to efficiently query both the set of all items purchased by a customer (via `custToItem`) and the set of all customers which purchased an item (via `itemToCust`). Importantly, the two fields both refer to the same set of purchase entities via the index-level relation **Purchased**.

3 Syntax

Index-level terms **i**, **j**, **k** consist of string literals, variables, applications, and the proposition constant **true**. An application is annotated with the domain and codomain of the applied

TypeVars	$\stackrel{def}{=}$	the set of all type variables
IndexVars	$\stackrel{def}{=}$	the set of all index variables
x, y, z	$\in$	*TypeVars*
$\mathbf{a}, \mathbf{b}, \mathbf{P}$	$\in$	*IndexVars*
$\mathbf{s}, \mathbf{t}$	$\in$	*Strings*

$\mathbf{i}, \mathbf{j}, \mathbf{k}$ (pre-index)	::=	$\mathbf{s}$	(string literal)
	|	$\mathbf{App}_{[\mathbf{a}:\mathbf{q}],\mathbf{p}}(\mathbf{j}, \mathbf{k})$	(index application)
	|	$\mathbf{a}$	(index variable)
	|	$\mathbf{true}$	(true proposition)
$\mathbf{p}, \mathbf{q}, \mathbf{r}$ (pre-sort)	::=	$\mathbf{str}$	(string sort)
	|	$\mathbf{prop}$	(proposition sort)
	|	$\mathbf{prf\ j}$	(proof sort)
	|	$(\mathbf{a} : \mathbf{q}) \to \mathbf{r}$	(function sort)
τ, σ (pre-type)	::=	$\{[\mathbf{a} : \mathbf{str}] : \tau\}$	(dictionary)
	|	$\{\mathbf{s}_i : \tau_i{}^{i \in 1..n}\}$	(record)
	|	$\{\mathbf{a} : \mathbf{str} \mid \mathbf{i}\}$	(string refinement)
	|	$\lambda \mathbf{a} : \mathbf{q}.\tau$	(index-to-type abstr.)
	|	$\lambda x : \kappa.\tau$	(type-to-type abstr.)
	|	$\bigvee \tau$	(union)
	|	$\tau\ \sigma$	(type app.)
	|	$\tau\ [\ \mathbf{j}\ ]$	(index-to-type app.)
κ, ρ (pre-kind)	::=	$*$	(proper type pre-kind)
	|	$\kappa \to \rho$	(type-to-type op)
	|	$\forall \mathbf{a} : \mathbf{q}.\ \kappa$	(index-to-type op)
Ω, Ψ (pre-sort-context)	::=	$\Omega, \mathbf{a} : \mathbf{q}$	(extension)
	|	$\diamond$	(empty)
Γ (pre-kind-context)	::=	$\Gamma, x : \kappa$	(extension)
	|	$\diamond$	(empty)

Figure 3. Syntax

function; this eases the definition of our denotational semantics, but the types can be inferred and are thus omitted in our examples, which instead use the notation **j k** as shorthand for $\mathbf{App}_{[\mathbf{a}:\mathbf{q}],\mathbf{p}}(\mathbf{j}, \mathbf{k})$.

Following [5], index level classifiers are called *sorts*. Sorts are written using the metavariables **p**, **q**, **r**. We include the sorts **str** for strings, **prop** for propositions, **prf j** proofs of the proposition **j**, and $(\mathbf{a} : \mathbf{q}) \to \mathbf{r}$ for dependent functions.

Our types τ, σ are also called *schemas*. The type $\{[\mathbf{a} : \mathbf{str}] : \tau\}$ represents string-keyed dictionaries of value type τ. The type τ depends on the index variable **a**, and $\{[\mathbf{a} : \mathbf{str}] : \tau\}$ denotes the set of database instances which map the key **s** to a database instance in the set denoted by $\tau[\mathbf{s}/\mathbf{a}]$ if it is non-empty and otherwise do not contain the key s. Record types, type-type applications, and type-type abstractions in

$$\begin{array}{rcl} \theta & ::= & \{[\mathbf{a} : \mathbf{str}] : \theta\} \\ & | & \{s_i : \theta_i{}^{i\in 1..n}\} \\ & | & \lambda \mathbf{a} : \mathbf{q}.\theta \\ & | & \lambda \mathbf{x} : \kappa.\theta \\ & | & \bigvee \theta \end{array}$$

Figure 4. Type values

Red-Ind-App

$$\frac{}{(\lambda(\mathbf{a} : \mathbf{q}).\tau)\ [\ \mathbf{j}\ ] \hookrightarrow \tau[\mathbf{j}/\mathbf{a}]}$$

Red-Ty-App

$$\frac{}{(\lambda(x : \kappa).\tau)\ \sigma \hookrightarrow \tau[\sigma/x]}$$

Figure 5. Top-level Reduction Rules

$$\begin{array}{rcl} E & ::= & [] \\ & | & \{[\mathbf{a} : \mathbf{str}] : E\} \\ & | & \{s_i : \theta_i{}^{i\in 1..(j-1)},\ s_j : E,\ s_i : \tau_i{}^{i\in (j+1)..n}\} \\ & | & \lambda \mathbf{a} : \mathbf{q}.E \\ & | & \lambda x : \kappa.E \\ & | & \bigvee E \\ & | & E\ \tau \\ & | & \theta\ E \\ & | & E\ [\ \mathbf{i}\ ] \end{array}$$

Figure 6. Evaluation Contexts

our type syntax are standard. A string refinement type of the form $\{\mathbf{a} : \mathbf{str} \mid \mathbf{i}\}$ represents the set of all strings $\mathbf{s}$ such that $\mathbf{i}[\mathbf{s}/\mathbf{a}]$ holds.

Finally, our type syntax includes index-type abstractions $\lambda(\mathbf{a} : \mathbf{q}).\tau$, index-type applications $\tau\ [\ \mathbf{j}\ ]$, and unions $\bigvee \tau$. In a well-kinded union type $\bigvee \tau$, the type τ is an index-type function whose codomain is $*$, the kind of proper types. τ then denotes a function which maps a semantic index to a set of database instances; $\bigvee \tau$ denotes the union over the applications of τ to all semantic indices in its domain.

In figures 1 and 2, a dictionary type of the form $\{[\mathbf{a} : \mathbf{str}] : \mathbf{i_1}, \dots, \mathbf{i_n} > \tau\}$ is sugar for

$$\{[\mathbf{a} : \mathbf{str}] : \bigvee \lambda(\mathbf{a_1} : \mathbf{prf}\ \mathbf{i_1}) \dots \bigvee \lambda(\mathbf{a_n} : \mathbf{prf}\ \mathbf{i_n}).\tau\}$$

4 Operational Semantics

Unlike a traditional lambda calculus, the purpose of a λ_{schema} expression is to describe a set of database instances rather than a computation. Nonetheless, λ_{schema} features abstraction and application and therefore has operational semantics.

The purpose of λ_{schema}'s operational semantics is to reduce a type τ to a semantically equivalent canonical form θ called a "type value". Figure 4 gives the syntax of type values. The binary relation $\hookrightarrow$ on types is defined in figure 5; it represents the substitution of a type or index into an abstraction. An evaluation context E, defined in Figure 6, is a type containing a single hole $[]$. We write $E[\tau]$ for the substitution of τ for the hole of E. We write $\tau \longrightarrow \tau'$ to mean there exists E, τ_0, and τ_0' such that $\tau = E[\tau_0]$, $\tau' = E[\tau_0']$, and $\tau_0 \hookrightarrow \tau_0'$. We write $\longrightarrow^*$ for the reflexive transitive closure of $\longrightarrow$. We write $\tau \not\longrightarrow$ if there exists no τ' with $\tau \longrightarrow \tau'$. We write $\tau \Downarrow \tau'$ if $\tau \longrightarrow^* \tau'$ and $\tau' \not\longrightarrow$.

$$\frac{}{\Omega \vdash \mathbf{s} : \mathbf{str}} \qquad \frac{}{\Omega, \mathbf{a} : \mathbf{q}, \Omega' \vdash \mathbf{a} : \mathbf{q}}$$

$$\frac{\Omega \vdash \mathbf{j} : (\mathbf{a} : \mathbf{q}) \to \mathbf{r} \qquad \Omega \vdash \mathbf{k} : \mathbf{q}}{\Omega \vdash \mathbf{App}_{[\mathbf{a}:\mathbf{q}],\mathbf{r}}(\mathbf{j}, \mathbf{k}) : \mathbf{r}[\mathbf{k}/\mathbf{a}]} \qquad \frac{}{\Omega \vdash \mathbf{true} : \mathbf{prop}}$$

$$\frac{}{\Omega \vdash \mathbf{str}} \qquad \frac{\Omega \vdash \mathbf{q} \qquad \Omega, \mathbf{a} : \mathbf{q} \vdash \mathbf{r}}{\Omega \vdash (\mathbf{a} : \mathbf{q}) \Rightarrow \mathbf{r}} \qquad \frac{\Omega \vdash \mathbf{j} : \mathbf{prop}}{\Omega \vdash \mathbf{prf}\ \mathbf{j}}$$

$$\frac{}{\diamond \vdash} \qquad \frac{\Omega \vdash \qquad \Omega \vdash \mathbf{p}}{\Omega, \mathbf{a} : \mathbf{p} \vdash}$$

Figure 7. Sorting, sort formation, and sort context formation

5 Static semantics

5.1 Index-level static semantics

Index-level judgment rules are shown in Figure 22. A judgment of the form $\Omega \vdash \mathbf{q}$ means that the pre-sort $\mathbf{q}$ is a well-formed sort under the context Ω. A judgment of the form $\Omega \vdash$ means that the sort pre-context Ω is a well-formed sort context, i.e. for each binding $\mathbf{a} : \mathbf{q}$ such that $\Omega = \Omega_1, \mathbf{a} : \mathbf{q}, \Omega_2$ we have $\Omega_1 \vdash \mathbf{q}$. Finally, a judgment $\Omega \vdash \mathbf{j} : \mathbf{q}$ means that the index $\mathbf{j}$ has sort $\mathbf{q}$ under context Ω.

5.2 Type-level static semantics

Type-level judgment rules are shown in Figure 23. A judgment of the form $\Omega \vdash \kappa$ means that the pre-kind κ is a well-formed kind under context Ω. A judgment of the form $\Omega \vdash \Gamma$ means that Γ is a well-formed kind context under sort context Ω. Finally, a judgment of the form $\Omega \mid \Gamma \vdash \tau : \kappa$ means that the type τ has kind κ under sort context Ω and kind context Γ.

6 Denotational Semantics

6.1 Index Language

We interpret our index langauage using Dybjer's *Categories-with-Families* (CwF) framework for semantically modeling dependently typed languages [1]. We first review the definition of CwF, then review the standard set-theoretic CwF, and finally provide an interpretation of the index language with respect to a CwF. The entirety of this section is standard. However, we rename some of the standard terminology to fit into the present setting, e.g. instead of dependent types we have dependent sorts.

$$\frac{\Omega, \mathbf{a} : \mathbf{str} \mid \Gamma \vdash \tau : *}{\Omega \mid \Gamma \vdash \{[\mathbf{a} : \mathbf{str}] : \tau\} : *} \qquad \frac{\Omega \mid \Gamma \vdash \tau_i : *^{\; i \in 1..n}}{\Omega \mid \Gamma \vdash \{\mathbf{s}_i : \tau_i{}^{i \in 1..n}\} : *}$$

$$\frac{\Omega, \mathbf{a} : \mathbf{str} \vdash \mathbf{i} : \mathbf{prop}}{\Omega \mid \Gamma \vdash \{\mathbf{a} : \mathbf{str} \mid \mathbf{i}\} : *}$$

K-IndAbs
$$\frac{\Omega \vdash \mathbf{q} \qquad \Omega, \mathbf{a} : \mathbf{q} \mid \Gamma \vdash \tau : \kappa \qquad a \notin FV(\Gamma)}{\Omega \mid \Gamma \vdash \lambda \mathbf{a} : \mathbf{q}.\tau : \forall \mathbf{a} : \mathbf{q}.\kappa}$$

$$\frac{\Omega \vdash \kappa \qquad \Omega \mid \Gamma, x : \kappa \vdash \tau : \rho}{\Omega \mid \Gamma \vdash \lambda x : \kappa.\tau : \kappa \to \rho} \qquad \frac{\Omega \mid \Gamma \vdash \tau : \forall \mathbf{a} : \mathbf{q}.*}{\Omega \mid \Gamma \vdash \bigvee \tau : *}$$

$$\frac{\Omega \mid \Gamma \vdash \tau : \forall \mathbf{a} : \mathbf{q}.\rho \qquad \Omega \vdash \mathbf{j} : \mathbf{q}}{\Omega \mid \Gamma \vdash \tau\ [\ \mathbf{j}\ ] : \rho[\mathbf{j}/\mathbf{a}]}$$

$$\frac{\Omega \mid \Gamma \vdash \tau : \kappa \to \rho \qquad \Omega \mid \Gamma \vdash \sigma : \kappa}{\Omega \mid \Gamma \vdash \tau\ \sigma : \rho} \qquad \frac{}{\Omega \vdash *}$$

$$\frac{\Omega \vdash \mathbf{q} \qquad \Omega, \mathbf{a} : \mathbf{q} \vdash \kappa}{\Omega \vdash \forall \mathbf{a} : \mathbf{q}.\kappa} \qquad \frac{\Omega \vdash \kappa \qquad \Omega \vdash \rho}{\Omega \vdash \kappa \to \rho} \qquad \frac{}{\Omega \vdash \diamond}$$

$$\frac{\Omega \vdash \Gamma \qquad \Omega \vdash \kappa}{\Omega \vdash \Gamma, x : \kappa}$$

Figure 8. Kinding and kind formation

6.1.1 Categories with Families.

A Category-with-Families (CwF) consists of

- A category $\mathbb{C}$ with a terminal object, called the *category of contexts*. Objects of $\mathbb{C}$ are called semantic contexts and written with the symbols Ω, Ψ, Υ.
- For each semantic context Ω a collection $St(\Omega)$ called the *semantic sorts* of Ω. Semantic sorts are written with the symbols X, Y, and Z.
- For each semantic context Ω and $X \in St(\Omega)$ a collection $In(\Omega, X)$ called the *semantic indices* of sort X.
- For each arrow $f : \Omega \to \Psi$ a mapping $-\{f\} : St(\Psi) \to St(\Omega)$. This preserves composition, in the sense that we have $-\{id_\Omega\} = id_{St(\Omega)}$ and for $g : \Psi \to \Upsilon$ we have $-\{g \circ f\} = -\{g\}\{f\}$.
- For each arrow $f : \Omega \to \Psi$ and sort $X \in St(\Psi)$ a mapping $-\{f\} : In(\Psi, X) \to In(\Omega, X\{f\})$. This preserves composition in the same sense as in the above point.
- For each context Ω and sort $X \in St(\Omega)$, we have an object $\Omega.X$ of $\mathbb{C}$, an arrow $\mathfrak{p}(X) : \Omega.X \to \Omega$, and an index $\mathfrak{v}_X \in In(\Omega.X, X\{\mathfrak{p}(X)\})$ such that for all $f : \Psi \to \Omega$ and $M \in In(\Psi, X\{f\})$ there exists a unique morphism $\langle f, M\rangle_X : \Psi \to \Omega.X$ such that $\mathfrak{p}(X) \circ \langle f, M\rangle_X = f$ and $\mathfrak{v}_X\{\langle f, M\rangle_X\} = M$.

For $M \in In(\Omega, X)$ we write $\overline{M}$ for the arrow $\langle id_\Omega, M\rangle_X : \Omega \to \Omega.X$.

For $f : \Psi \to \Omega$ and $X \in St(\Omega)$ we define $\mathfrak{q}(f, X) : \Psi.X\{f\} \to \Omega.X$, called the *weakening* f by X as

$$\mathfrak{q}(f, X) = \langle f \circ \mathfrak{p}(X\{f\}), \mathfrak{v}_{X\{f\}}\rangle_X$$

A *weakening map* is a morphism of the form $\mathfrak{p}(X) : \Omega.X \to \Omega$ or of the form $\mathfrak{q}(w, Y)$ where w is a weakening map. We write f^+ and X^+ for $f\{w\}$ and $X\{w\}$ when w is a weakening map that is clear from context.

A CwF is said to *support* Π*-sorts* if for any two sorts $X \in St(\Omega)$ and $Y \in St(\Omega.X)$ there is a sort $\Pi(X, Y) \in St(\Omega)$ and a morphism

$$App_{X,Y} : \Omega.X.\Pi(X, Y)^+ \to \Omega.X.Y$$

such that

$$\mathfrak{p}(X) \circ App_{X,Y} = \mathfrak{p}(\Pi(X, Y)^+) \qquad \textit{App-T}$$

and for every $M \in In(\Omega.X, Y)$

$$App_{X,Y} \circ \overline{M\{p(X)\}} = \overline{M} \qquad \Pi\text{-}C'$$

and for every morphism $f : B \to \Gamma$

$$\begin{aligned} &App_{X,Y} \circ \mathfrak{q}(\mathfrak{q}(f, X), \Pi(X, Y)\{\mathfrak{q}(f, X)\}) \\ &= \mathfrak{q}(\mathfrak{q}(f, X), Y) \circ App_{X\{f\}, Y\{\mathfrak{q}(f,\sigma)\}} \end{aligned}$$

6.1.2 Set-theoretic CwF.

Our index language will be interpreted using a standard set-theoretic CwF. This CwF's category of contexts is **Sets**, the category of sets and functions. For a set Ω, $St(\Omega)$ is the collection of all Ω-indexed families of sets. For all Ω and $X \in St(\Omega)$, $In(\Omega, X)$ is the collection of families $(x_\omega \in X_\omega)_{\omega \in \Omega}$ selecting an element $x_\omega \in X_\omega$ for each $\omega \in \Omega$. For all $h : \Omega \to \Psi$, $X \in St(\Psi)$, and $M \in In(\Psi, X)$ we define $X\{h\}_\omega \stackrel{def}{=} X_{f(\omega)}$ and $M\{h\}_\omega \stackrel{def}{=} M_{f(\omega)}$. For each Ω and $X \in St(\Omega)$ we define

$$\Omega.X \stackrel{def}{=} \{(\omega, x) \mid \omega \in \Omega \text{ and } x \in X_\omega\}$$

$\mathfrak{p}(X)(\omega, x) \stackrel{def}{=} \omega$, and $(\mathfrak{v}_X)_{(\omega,x)} \stackrel{def}{=} x$. Finally, letting $f : \Psi \to \Omega$ and $M \in In(\Psi, X\{f\})$, we have $\langle f, M\rangle_X(\psi) = (f(\psi), M_\psi)$.

6.1.3 Interpretation.

Our index langauge is a fragment of the calculus of constructions, with the unnotable addition of strings. Its interpretation appears, for example, in Hoffman's tutorial[2], and is displayed in Figure 9 for convenience.

We also state the fragment of the standard soundness theorem which will be relevant in the sequel:

Theorem 6.1. *Our interpretation satisfies the following properties.*

- *If* $\Omega \vdash$ *then* $[\![\Omega]\!]$ *is an object of the context category* **Sets**.
- *If* $\Omega \vdash \mathbf{q}$ *then* $[\![\Omega; \mathbf{q}]\!]$ *is an element of* $St([\![\Omega]\!])$
- *If* $\Omega \vdash \mathbf{j} : \mathbf{q}$ *then* $[\![\Omega; \mathbf{j}]\!]$ *is an element of* $In([\![\Omega]\!], [\![\Omega; \mathbf{q}]\!])$

$$
\begin{aligned}
&[\![\diamond]\!] \overset{def}{=} \top \\
&[\![\Omega, \mathrm{a} : \mathrm{q}]\!] \overset{def}{=} [\![\Omega]\!].[\![\Omega; \mathrm{q}]\!] \text{ if } x \text{ not in } \Omega\text{, undefined otherwise.} \\
&[\![\Omega; (\mathrm{j} : \mathrm{q}) \Rightarrow \mathrm{r}]\!] \overset{def}{=} \Pi([\![\Omega; \mathrm{q}]\!], [\![\Omega, \mathrm{a} : \mathrm{q}; \mathrm{r}]\!]) \\
&[\![\Omega; \mathbf{prf}\ \mathrm{j}]\!] \overset{def}{=} (\ \{*\} \text{ if } [\![\Omega; \mathrm{j}]\!]_\omega = \mathit{true},\ \emptyset \text{ otherwise }\ )_{\omega \in [\![\Omega]\!]} \\
&[\![\Omega; \mathbf{prop}]\!] \overset{def}{=} (\{\mathit{true}, \mathit{false}\})_{\omega \in [\![\Omega]\!]} \\
&[\![\Omega; \mathbf{str}]\!] \overset{def}{=} (\text{the set of strings})_{\omega \in [\![\Omega]\!]} \\
&[\![\Omega; \mathbf{App}_{[\mathrm{a}:\mathrm{q}],\mathrm{r}}(\mathrm{i}, \mathrm{j})]\!] \overset{def}{=} \\
&\quad \mathit{App}_{[\![\Omega;\mathrm{q}]\!],[\![\Omega,\mathrm{a}:\mathrm{q};\mathrm{r}]\!]} \circ \langle \overline{[\![\Omega; \mathrm{i}]\!]}, [\![\Omega; \mathrm{j}]\!]^+ \rangle_{[\![\Omega; (\mathrm{a}:\mathrm{q}) \to \mathrm{r}]\!]^+} \\
&[\![\Omega, \mathrm{a} : \mathrm{q}\ ;\ \mathrm{a}]\!] \overset{def}{=} \mathfrak{v}_{[\![\Omega, \mathrm{a}:\mathrm{q}]\!]} \\
&[\![\Omega; \mathrm{s}]\!] \overset{def}{=} (s)_{\omega \in [\![\Omega]\!]}
\end{aligned}
$$

Figure 9. Interpretation of index language

$$
\begin{aligned}
&[\![\Omega \vdash *]\!]_\omega \overset{def}{=} \mathcal{PP}(\mathit{Inst}) \\
&[\![\Omega \vdash \forall \mathrm{a} : \mathrm{q}.\kappa]\!] \overset{def}{=} \Pi_{\Omega,\mathrm{q}}[\![\Omega, \mathrm{a} : \mathrm{q} \vdash \kappa]\!] \\
&[\![\Omega \vdash \kappa \to \rho]\!] \overset{def}{=} [\![\Omega \vdash \kappa]\!] \Rightarrow [\![\Omega \vdash \rho]\!]
\end{aligned}
$$

Figure 10. Interpretation of kinding judgments

6.2 Type language

Our kinding and kind formation judgments are interpreted in terms of the fibration $\mathit{Fam}(\mathbf{Sets})$. We assume basic knowledge of fibrations, which can be learned from Jacobs [3]. A kind-in-sorting-context $\Omega \vdash \kappa$ is interpreted as an object of $\mathit{Fam}(\mathbf{Sets})_{[\![\Omega]\!]}$, concretely a $[\![\Omega]\!]$-indexed family of sets. For pre-kind contexts $\Gamma = x_1 : \kappa_1, \ldots, x_n : \kappa_n$, the kinding context formation judgment $\Omega \vdash \Gamma$ is interpreted as the product $[\![\Omega \vdash \kappa_1]\!] \times \cdots \times [\![\Omega \vdash \kappa_n]\!]$ in $\mathit{Fam}(\mathbf{Sets})_{[\![\Omega]\!]}$. A kinding judgment $\Omega \mid \Gamma \vdash \tau : \kappa$ is interpreted as a arrow of $\mathit{Fam}(\mathbf{Sets})_{[\![\Omega]\!]}$ from $[\![\Omega \vdash \Gamma]\!]$ to $[\![\Omega \vdash \kappa]\!]$.

6.2.1 Dependent Simple Products. We define a *CwF-fibration* as a pair (C, p) where C is a CwF and $p : \mathbb{E} \to \mathbb{C}$ is a fibration whose base category $\mathbb{C}$ is C's context category.

We say that a CwF-fibration has *dependent simple products* if

- For all contexts Ω and all $X \in St(\Omega)$ the functor $\mathfrak{p}(X)^*$ has a right adjoint Π_X. (We write $(-)^\sharp$ for the transposition from $\mathfrak{p}(X)^*(A) \to B$ to $A \to \Pi_X(B)$, and $(-)^\flat$ for transposition in the opposite direction.)
- The above adjunction satisfies a *transpose distribution* property: for contexts Ω, Ψ, context arrows $u : \Psi \to \Omega$, sorts $X \in St(\Omega)$, total objects Γ over Ω, total objects Δ over $\Omega.X$, and total arrows $f : \mathfrak{p}(X)^*(\Gamma) \to \Delta$. We have $u^* f^\sharp \cong (\mathfrak{q}(u, X)^* f)^\sharp$.
- For every $u : \Psi \to \Omega$ and $X \in St(\Omega)$ the canonical natural transformation $u^*\Pi_X \Longrightarrow \Pi_{X\{u\}}\mathfrak{q}(u, X)^*$ is an isomorphism. This is typically called a *Beck-Chevalley condition*.

It is a standard fact of fibrations that for arrows $u : \Omega \longrightarrow \Psi$ and $v : \Psi \longrightarrow \Upsilon$ of the base category, we have $u^*v^* \cong (v \circ u)^*$. From this we derive a natural isomorphism $\mathfrak{q}(u, X)^*\mathfrak{p}(X)^* \cong \mathfrak{p}(X\{u\})^*u^*$, which will clarify the second and third points:

$$
\begin{aligned}
&\ \mathfrak{q}(u, X)^*\mathfrak{p}(X)^* \\
\cong&\ \langle u \circ \mathfrak{p}^*(X\{u\}), \mathfrak{v}_{X\{u\}} \rangle_X^* \mathfrak{p}(X)^* \\
=&\ (\mathfrak{p}(X) \circ \langle u \circ \mathfrak{p}^*(X\{u\}), \mathfrak{v}_{X\{u\}} \rangle_X)^* \\
=&\ (u \circ \mathfrak{p}(X\{u\})^*)^* \\
\cong&\ \mathfrak{p}(X\{u\})^* u^*
\end{aligned}
$$

In the second point, it is not immediately clear that the right-hand side of the isomorphism is "well-typed". However, applying our isomorphism to the domain of $\mathfrak{q}(z, X)^* f$ gives $\mathfrak{q}(u, X)^*\mathfrak{p}(X)^*(\Gamma) \cong \mathfrak{p}(X\{u\})^*u^*(\Gamma)$. Agreement of codomains follows from the Beck-Chevalley condition.

In the third point, the *canonical transformation* is obtained as the transpose of

$$
\mathfrak{p}(X\{u\})^* u^* \Pi_X \xrightarrow{\cong} \mathfrak{q}(u, X)^* \mathfrak{p}(X)^* \Pi_X \xrightarrow{\mathfrak{q}(u,X)\epsilon} \mathfrak{q}(u, X)^*
$$

6.2.2 Set-theoretic Dependent Simple Products. In our set-theoretic model, for semantic contexts Ω and semantic sorts $X \in St(\Omega)$, we define $\Pi_X : \mathit{Fam}(\mathbf{Sets})_{\Omega.X} \to \mathit{Fam}(\mathbf{Sets})_\Omega$ as:

$$
\begin{aligned}
&\Pi_X(\left(B_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X}) \overset{def}{=} \left(\Pi_{x \in X_\omega} B_{(\omega,x)}\right)_{\omega\in\Omega} \\
&\Pi_X\left(f_{(\omega,x)} : B_{(\omega,x)} \to C_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X} \overset{def}{=} \left(\Pi_{x\in X_\omega}\ f_{(\omega,x)}\right)_{\omega\in\Omega}
\end{aligned}
$$

We show that Π_X is the right adjoint of $\mathfrak{p}(X)^*$ in Appendix A.2, with an underlying correspondence

$$
\frac{(A_\omega)_{\omega\in\Omega} \longrightarrow \left(\Pi_{x\in X_\omega} B_{(\omega,x)}\right)_{\omega\in\Omega} = \Pi_X\left(B_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X}}{\mathfrak{p}(X)^*\ (A_\omega)_{\omega\in\Omega} = (A_\omega)_{(\omega,x)\in\Omega.X} \longrightarrow \left(B_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X}}
$$

From top to bottom, $(-)^\flat$ takes an arrow

$$
\left(\langle f_{(\omega,x)} : A_\omega \to B_{(\omega,x)} \rangle_{x\in X_\omega}\right)_{\omega\in\Omega}
$$

to

$$
\left(f_{(\omega,x)} : A_\omega \to B_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X}
$$

From bottom to top, $(-)^\sharp$ takes an arrow

$$
(f_{(\omega,x)} : A_\omega \to B_{(\omega,x)})_{(\omega,x)\in\Omega.X}
$$

to

$$
\left(\Pi_{x\in X_\omega} f_{(\omega,x)}\right)_{\omega\in\Omega}
$$

In this set-theoretic model, we have that $\Pi_{X\{u\}}\mathfrak{q}(u, X)^* = u^*\Pi_X$, and furthermore that the canonical transformation is the identity at this functor. Clearly, then, this model satisfies the Beck-Chevalley condition for dependent simple products.

6.3 Interpretation

Blah blah blah.

Our interpretation satisfies the following soundness theorems.

$$[\![\{[\mathbf{a}:\mathbf{str}]:\tau\}]\!]_\omega\gamma \stackrel{def}{=} \{f \mid f([]) = \text{“}*\text{”} \wedge \forall s. f|_{[s]} \in [\![\tau]\!]_{(\omega,s)}\gamma^+ \text{ if } [\![\tau]\!]_{(\omega,s)}\gamma^+ \text{ is non-empty and } f|_{[s]} = \emptyset \text{ otherwise}\}$$

$$[\![\{\mathbf{s_i}:\tau_i{}^{i\in 1..n}\}]\!]_\omega\gamma \stackrel{def}{=} \{f \mid f([]) = \text{“}*\text{”} \wedge \forall s_i \in \{s_1,\dots,s_n\}.\ f|_{[s_i]} \in [\![\tau_i]\!]_\omega\gamma \wedge \forall s \notin \{s_1,\dots,s_n\}.f|_{[s]} = \emptyset\}$$

$$[\![\{\mathbf{a}:\mathbf{str} \mid \mathbf{i}\}]\!]_\omega\gamma \stackrel{def}{=} \{f \mid f([])\downarrow\ \wedge ([\![\mathbf{i}]\!]_{(\omega,f([]))} = true) \wedge (\forall s. f|_{[s]} = \emptyset)\}$$

$$[\![\bigvee\tau]\!]_\omega\gamma \stackrel{def}{=} \bigcup_{M\in[\![\mathbf{q}]\!]_\omega}([\![\tau]\!]_\omega\gamma)_M$$

Figure 11. Semantics for "non-operational" kinding rules

$$\frac{\begin{array}{c} f \stackrel{def}{=} [\![\mathbf{\Omega} \mid \Gamma \vdash \tau : \forall \mathbf{a}:\mathbf{q}.\rho]\!] : [\![\mathbf{\Omega} \vdash \mathbf{\Gamma}]\!] \longrightarrow \Pi_{\mathbf{\Omega},\mathbf{q}}[\![\mathbf{\Omega},\mathbf{a}:\mathbf{q} \vdash \rho]\!] \\ M \stackrel{def}{=} [\![\mathbf{\Omega} \vdash \mathbf{j}:\mathbf{q}]\!] \in In([\![\mathbf{\Omega}]\!],[\![\mathbf{\Omega} \vdash \mathbf{q}]\!]) \end{array}}{[\![\mathbf{\Omega} \mid \Gamma \vdash \tau\ [\ \mathbf{j}\ ] : \rho]\!] \stackrel{def}{=} \overline{M}^{\,*}(f^\flat)}$$

$$\frac{\begin{array}{c} X \stackrel{def}{=} [\![\mathbf{\Omega} \vdash \mathbf{q}]\!] \in St(\Omega) \\ f \stackrel{def}{=} [\![\mathbf{\Omega},\mathbf{a}:\mathbf{q} \mid \Gamma \vdash \tau : \kappa]\!] : \mathfrak{p}(X)^*[\![\mathbf{\Omega} \vdash \Gamma]\!] \longrightarrow [\![\mathbf{\Omega},\mathbf{a}:\mathbf{q} \vdash \kappa]\!] \end{array}}{[\![\mathbf{\Omega} \mid \Gamma \vdash \lambda\mathbf{a}:\mathbf{q}.\tau : \forall\mathbf{a}:\mathbf{q}.\kappa]\!] \stackrel{def}{=} f^\sharp}$$

Figure 12. Semantics for abstraction and application

$$\begin{tikzcd}
\mathbb{E}_\Omega \arrow[r, "u^*"] \arrow[d, bend right, "\mathfrak{p}(X)^*"'] & \mathbb{E}_\Psi \arrow[d, bend right, "\mathfrak{p}(X\{u\})^*"'] \\
\mathbb{E}_{\Omega.X} \arrow[r, "\mathfrak{q}(u{,}X)^*"'] \arrow[u, bend right, "\Pi_X"'] & \mathbb{E}_{\Psi.X\{u\}} \arrow[u, bend right, "\Pi_{X\{u\}}"']
\end{tikzcd}$$

Figure 13. Some components of Beck-Chevalley for dependent simple products

7 Semi-Validator Generation

This formal language for describing the structure of MUMPS databases not only provides clarity of thought compared to informal schema techniques, but also the possibility to develop testing and verification tools. We proceed to formalize one such testing tool, called a *semi-validator*: its purpose is to identify discrepancies between a database instance and a schema.

7.1 Restrictions

We impose the following restriction on the form of our schema τ to ease the process of constructing a semi-validator for τ.

- Any abstraction of the form $\lambda(\mathbf{a}:\mathbf{p}).\sigma$ where $\mathbf{p}$ is a function sort must be at the outer level of the schema,

$$\begin{array}{rcl} s & \in & \text{the set of all strings} \\ v, \xi & \in & \mathbf{\Omega} \stackrel{fin}{\rightharpoonup} e \end{array}$$

$$\begin{array}{rcl} e, d & ::= & \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \mid \mathbf{first10}\ e_1\ \mathbf{with}\ x\ \mathbf{in}\ e_2 \\ & & \mid \mathbf{isdefined}\ e \mid \mathbf{read}\ e \mid \lambda x.e \mid e_1\ e_2 \mid true \mid false \\ & & \mid s \mid e_1 = e_2 \mid [e_1,\dots,e_n] \mid v \mid \mathbf{\Omega} \\ v & ::= & true \mid false \mid \lambda x.e \mid s \mid [v_1,\dots,v_n] \mid v \mid \mathbf{\Omega} \end{array}$$

Figure 14. Syntax for MiniMUMPS

  i.e. we must have
  $$\tau = \bigvee\lambda(\mathbf{a_1}:\mathbf{p_1} \longrightarrow \mathbf{q_1}).\dots\bigvee\lambda(\mathbf{a_n}:\mathbf{p_n} \longrightarrow \mathbf{q_n}).\sigma$$
  for some $n \in \mathbb{N}$, where no abstraction of the form
  $$\lambda(\mathbf{a}:\mathbf{p} \longrightarrow \mathbf{q}).\sigma'$$
  is nested anywhere in σ.
- A union over the $\mathbf{str}$ sort i.e. a type of the form
  $$\bigvee\lambda(\mathbf{a}:\mathbf{str}).\sigma$$
  may not occur nested anywhere in τ.
- No function sort is of the form $\mathbf{p} \longrightarrow \mathbf{str}$.

From the first constraint above we infer that any function sort $\mathbf{p} \longrightarrow \mathbf{q}$ inserted into our sort context $\mathbf{\Omega}$, while kind-checking τ will not depend on any index variables with sorts of the form $\mathbf{str}$ or $\mathbf{prf\ j}$. Furthermore, since the sort $\mathbf{str}$ clearly does not depend on any index variables, we can write any sort context $\mathbf{\Omega}$ encountered while type-checking τ as $\mathbf{\Omega}_{str}, \mathbf{\Omega}_{fn}, \mathbf{\Omega}_{prf}$, where $\mathbf{\Omega}_{str}$ contains only bindings of the form $\mathbf{a}:\mathbf{str}$, $\mathbf{\Omega}_{fn}$ includes only bindings of the form $\mathbf{a}:\mathbf{p} \longrightarrow \mathbf{q}$, and $\mathbf{\Omega}_{prf}$ contains only bindings of the form $\mathbf{a}:\mathbf{prf\ j}$.

7.2 MiniMUMPS

In order to compare a database instance to a schema, we need a calculus whose expressions traverse and read a database instance. We call this calculus *MiniMUMPS*. Its syntax is given in figure 14. Its operational semantics is given in figure 15.

A *MiniMUMPS* reduction judgment has the form

$$I \mid e \longrightarrow e'$$

$$E ::= \textbf{if } E \textbf{ then } e \textbf{ else } e \mid \textbf{first10 } E \textbf{ with } x \textbf{ in } e_2 \mid \textbf{isdefined } E \mid \textbf{read } E \mid E :: e \mid v :: E \mid E\ e \mid v\ E$$

$$\frac{I \mid e \hookrightarrow e'}{I \mid E[e] \rightarrow E[e']} \qquad \frac{}{I \mid (\lambda x.e)e' \hookrightarrow e[x/e']}$$

$$\frac{}{I \mid \textbf{if true then } e_1 \textbf{ else } e_2 \hookrightarrow e_1}$$

$$\frac{}{I \mid \textbf{if false then } e_1 \textbf{ else } e_2 \hookrightarrow e_2}$$

$$\frac{I|_{[s_1,\ldots,s_n,t]} \text{ is active only for } t \in \{t_1,\ldots,t_n\} \text{ where } t_1 < \ldots < t_n}{I \mid \textbf{first10 } [s_1,\ldots,s_n] \textbf{ with } x \textbf{ in } e \hookrightarrow e[t_1/x] \wedge \ldots \wedge e[t_{min(n,10)}/x]}$$

$$\frac{I|_{[s_1,\ldots,s_n]} \text{ is active}}{I \mid \textbf{isdefined } [s_1,\ldots,s_n] \hookrightarrow \mathit{true}}$$

$$\frac{I|_{[s_1,\ldots,s_n]} \text{ is not active}}{I \mid \textbf{isdefined } [s_1,\ldots,s_n] \hookrightarrow \mathit{false}}$$

$$\frac{I([s_1,\ldots,s_n]) = s}{I \mid \textbf{read } [s_1,\ldots,s_n] \hookrightarrow s}$$

Figure 15. Reduction relation for MiniMUMPS

It means that the *MiniMUMPS* expression e reduces in one step to the expression e' using read access to the database instance I. We write $I \mid e \Downarrow v$ to mean that e normalizes to v with read access to I, i.e. (e, v) is in the reflexive transitive closure of the relation $I \mid - \rightarrow -$.

7.3 Semi-deciders

We say that a MiniMUMPS program e is a "*semi-decider* for $\Omega \vdash$ with respect to $\diamond \mid \diamond \vdash \sigma : *$" when for all $I \in [\![\diamond \mid \diamond \vdash \sigma : *]\!]$ and $[s_1,\ldots,s_n] \in [\![\Omega_{str} \vdash]\!]$, if there exists a $[z_1,\ldots,z_m] \in [\![\Omega \vdash]\!]$ such that $z_1 = s_1,\ldots,z_n = s_n$ then $I \mid e\ s_1 \cdots s_n \Downarrow \mathit{true}$.

7.4 Semi-validators

Let $\Omega = \Omega_{str}, \Omega_{fn}, \Omega_{prf}$ and let Δ be a location context such that $\Omega \vdash \Delta$. We say that a closed MiniMUMPS expression e is a "*semi-validator* for $\Omega \mid \diamond \vdash \sigma : *$ at Δ with respect to $\diamond \mid \diamond \vdash \tau : *$" whenever for all $[s_1,\ldots,s_n] \in [\![\Omega_{str} \vdash]\!]$, $l \in [\![\Omega_{str} \vdash \Delta]\!]_{[s_1,\ldots,s_n]}$, and $I \in [\![\diamond \mid \diamond \vdash \tau : *]\!]_*(*)$ such that $I|_l$ is active, if there exists a $[z_1,\ldots z_{|\Omega|}] \in [\![\Omega \vdash]\!]$ with $z_1 = s_1,\ldots,z_n = s_n$ and

$$I|_l \in [\![\Omega \mid \diamond \vdash \sigma : *]\!]_{[z_1,\ldots,z_{|\Omega|}]}(*)$$

$$\begin{array}{rcll} \Delta \text{ (pre-location-context)} & ::= & \Delta, \mathbf{i} & \text{(extension)} \\ & | & \diamond & \text{(empty)} \\ \Xi \text{ (pre-sort-context set)} & \stackrel{def}{=} & \mathcal{P}_{fin}(\Omega) & \\ \varphi \text{ (emptiness grades)} & ::= & + \text{ (occupied)} & \\ & | & ? \text{ (unknown)} & \end{array}$$

Figure 16. Additional Syntax For Semi-Validator Generation

then

$$I \mid e\ s_1\ s_2\ \cdots s_n \Downarrow \mathit{true}$$

Consider a well-kinded schema $\diamond \mid \diamond \vdash \tau :: *$. This schema represents the set of database instances $[\![\diamond \mid \diamond \vdash \tau :: *]\!]_*(*)$. Suppose we have some database instance I which was constructed with the intention that $I \in [\![\diamond \mid \diamond \vdash \tau :: *]\!]_*(*)$. Ideally, we would like a procedure that tests whether $I \in [\![\diamond \mid \diamond \vdash \tau :: *]\!]_*(*)$, but instead of designing such a procedure, we pursue a less ambitions goal: generate a semi-validator e for $\diamond \mid \diamond \vdash \tau : *$ at $[]$ with respect to $\diamond \mid \diamond \vdash \tau : *$, i.e. a MiniMUMPS program e such that if $I \in [\![\diamond \mid \diamond \vdash \tau : *]\!]_*(*)$ then $I \mid e \Downarrow \mathit{true}$. For convenience, we shall refer to such an expression e as a "semi-validator for τ".

The result $I \mid e \Downarrow \mathit{false}$ is then useful to us because it implies that our database instance I does not conform to our target schema τ and hence was constructed incorrectly. The result $I \mid e \Downarrow \mathit{true}$, however, does not guarantee that I conforms to a schema. Thus, a semi-validator is a testing tool rather than a verification tool: it can confirm the presence of errors, but not their absence.

7.5 Semi-validator Generation

7.5.1 Location contexts. As discussed in section ??, a list of strings can be used to "locate" a smaller database instance inside of a larger one. A *pre-location-context* Δ is a list of zero or more subjects. A *location-context under* $\Omega \vdash$ is a list of zero or more subjects $\mathbf{i}$ such that $\Omega \vdash \mathbf{i} : \mathbf{str}$. We write $\Omega \vdash \Delta$ to mean that Δ is a location context under $\Omega \vdash$. We define the interpretation $[\![\Omega \vdash \Delta]\!]$ as the $[\![\Omega \vdash]\!]$-indexed family of lists such that $[\![\Omega \vdash \Delta]\!]_\omega = [[\![\Omega \vdash \mathbf{i_1} : \mathbf{str}]\!]_\omega,\ldots,[\![\Omega \vdash \mathbf{i_n} : \mathbf{str}]\!]_\omega]$.

Operationally, our semi-validator needs a way to translate a valuation $\omega \in [\![\Omega \vdash]\!]$ into an instantiation of Δ list of strings $[s_1,\ldots,s_n]$ where $s_j = \Omega \vdash \mathbf{i_j} : \mathit{str}_\omega$ for $j \in 1..n$.

7.5.2 Preparation. Given a pre-type τ, we first kind check to verify that

$$\diamond \mid \diamond \vdash \tau : *$$

If it fails then τ does not represent a set of database instances, so our tool halts with an error message. If it succeeds then we proceed to generate a semi-validator for τ. τ normalizes to some value θ. By soundness, $[\![\diamond \mid \diamond \vdash \tau : *]\!] = [\![\diamond \mid \diamond \vdash \theta : *]\!]$, and so we need only generate a semi-validator for θ, which is much easier due to θ's simpler form.

$$\frac{\Omega_{str} = \mathbf{a_1} : str, \ldots, \mathbf{a_n} : str}{\Omega_{str} \vdash \mathbf{a_i} : str \Rightarrow \lambda x_1 \cdots x_n . x_i}$$

$$\frac{\Omega_{str} = \mathbf{a_1} : str, \ldots, \mathbf{a_n} : str}{\Omega_{str} \vdash \mathbf{s} : str \Rightarrow \lambda x_1 \cdots x_n . s}$$
where s is the string underlying literal $\mathbf{s}$

$$\frac{\Omega_{str} \vdash \mathbf{i_j} \Rightarrow e_j^{j \in 1..m} \qquad n \overset{def}{=} |\Omega_{str}|}{\Omega_{str} \vdash \mathbf{i_1}, \ldots, \mathbf{i_m} \Rightarrow \lambda x_1 \cdots x_n . [(e_1\ x_1 \cdots x_n), \cdots, (e_m\ x_1 \cdots x_n)]}$$

Figure 17. Location context instantiation

7.5.3 Rules.

Semi-validator generation is performed following the syntactic rules of figure 18. These rules reuse much of the syntax of kind checking, but introduce new syntax defined in figure 16.

A semi-validator generation judgment has the form

$$\Omega @ \Xi_0 \vdash_\tau \theta : \Delta \;\&\; \varphi \;\&\; \Xi_1 \Rightarrow (e, \xi)$$

where $\diamond \mid \diamond \vdash \sigma : *$. In the above judgment, Ω, τ, θ, and Δ are input positions, while Ξ_0, φ, Ξ_1, e, and ξ are output positions.

Theorem 7.1. *If $\Omega @ \Xi_0 \vdash_\tau \theta : \Delta \;\&\; \varphi \;\&\; \Xi_1 \Rightarrow (e, \xi)$ then the following facts hold:*

1. $\Omega \mid \diamond \vdash \theta : *$
2. *If v is a mapping that takes each sort context $\Omega_i \in \Xi_0 = \{\Omega_1, \ldots, \Omega_n\}$ to a semi-decider for $\Omega \vdash_i$, then*
$$e[v(\Omega_1)/x_{\Omega_1}] \cdots [v(\Omega_n)/x_{\Omega_n}]$$
is a semi-validator for $\Omega \mid \diamond \vdash \theta : *$ at Δ with respect to $\diamond \mid \diamond \vdash \tau : *$.
3. *If $\varphi = +$ then for all $\omega \in [\![\Omega \vdash]\!]$ and*
$$I \in [\![\Omega \mid \diamond \vdash \theta : *]\!]_\omega(*)$$
then I is active, i.e. $I(l)$ is defined for some list of strings l.
4. *ξ is a map such that for each $\Omega_i \in \Xi_1 = \{\Omega_1, \ldots, \Omega_m\}$, we have that $\vdash \Omega_i$ and ξ maps Ω_i to a semi-decider for $\Omega \vdash_i$ with respect to $\diamond \mid \diamond \vdash \tau : *$.*

Now is the time to prove thm 7.1.

References

[1] Peter Dybjer. 1995. Internal type theory. In *International Workshop on Types for Proofs and Programs*. Springer, 120–134.

[2] Martin Hofmann. 1997. Syntax and Semantics of Dependent Types. In *Semantics and Logics of Computation*. Cambridge University Press, 79–130.

[3] Bart Jacobs. 1999. *Categorical logic and type theory*. Elsevier.

[4] Erik Palmgren. 2019. Categories with families and first-order logic with dependent sorts. *Annals of Pure and Applied Logic* 170, 12 (2019), 102715.

[5] Hongwei Xi. 2007. Dependent ML: An approach to practical programming with dependent types. *Journal of Functional Programming* 17, 2 (2007), 215–286.

$$\textsc{GenDict}\quad \frac{\Omega, \mathbf{a} : \mathbf{str}@\Xi_0 \vdash_\tau \theta : \Delta, \mathbf{a} \mathbin{\&} ? \mathbin{\&} \Xi_1 \Rightarrow (e, \xi) \qquad \Omega \vdash \Delta \Rightarrow d}{\Omega@\Xi_0 \vdash_\tau \{[\mathbf{a} : \mathbf{str}] : \theta\} : \Delta \mathbin{\&} + \mathbin{\&} \Xi_1 \Rightarrow (e', \xi)}$$

$$\text{where } e' \stackrel{def}{=} \lambda x_1 \ldots x_{|\Omega_{str}|}.\mathbf{first10}\ d\ x_1 \ \cdots\ x_{|\Omega_{str}|}\ \mathbf{with}\ x\ \mathbf{in}\ e\ x_1 \cdots x_{|\Omega_{str}|}\ x$$

$$\textsc{GenRecord}\quad \frac{\Omega@\Xi_{0,i} \vdash_\tau \theta_i : \Delta, \mathbf{s_i} \mathbin{\&} \phi_i \mathbin{\&} \Xi_{1,i} \Rightarrow (e_i, \xi_i)^{\ i\in 1..n} \qquad \Omega \vdash \Delta}{\Omega@\bigcup_{i\in 1..n} \Xi_{0,i} \vdash_\tau \{\mathbf{s}_i : \theta_i^{\ i\in 1..n}\} : \Delta \mathbin{\&} \bigsqcup_{i\in 1..n} \phi_i \mathbin{\&} \bigcup_{i\in 1..n} \Xi_{1,i} \Rightarrow (e', \bigoplus_{i\in 1..n} \xi_i)}$$

$$\text{where } e' \stackrel{def}{=} \lambda x_1 ... x_{|\Omega_{str}|}.(e_1\ x_1 \cdots x_{|\Omega_{str}|}) \wedge \cdots \wedge (e_n\ x_1 \cdots x_{|\Omega_{str}|})$$

$$\textsc{GenRefinement}\quad \frac{\Omega, \mathbf{a} : \mathbf{str} \vdash \mathbf{P}\ \mathbf{b_1} \cdots \mathbf{b_n} : \mathbf{prop} \qquad \Omega \vdash \Delta \Rightarrow d}{\Omega@\{\Omega'\} \vdash_\tau \{\mathbf{a} : \mathbf{str} \mid \mathbf{P}\ \mathbf{b_1} \cdots \mathbf{b_n}\} : \Delta \mathbin{\&} ? \mathbin{\&} \emptyset \Rightarrow (e, \emptyset) \qquad \text{where } \Omega' \stackrel{def}{=} \Omega, \mathbf{a} : \mathbf{str}, \mathbf{b} : \mathbf{prf}\ (\mathbf{P}\ \mathbf{b_1} \cdots \mathbf{b_n})}$$

$$\text{where } e \stackrel{def}{=} \lambda x_1 \cdots x_{|\Omega_{str}|}.\mathbf{let}\ x_{loc} = d\ x_1 \cdots x_n\ \mathbf{in}\ ((\mathbf{isdefined}\ x_{loc}) \wedge (x_{\Omega'}\ x_1 \cdots x_n\ (\mathbf{read}\ x_{loc})))$$

$$\textsc{GenUnionPrf}\quad \frac{\Omega \vdash \mathbf{prf}\ \mathbf{j} \qquad \Omega, \mathbf{a} : \mathbf{prf}\ \mathbf{j}@\Xi_0 \vdash_\tau \theta : \Delta \mathbin{\&} + \mathbin{\&} \Xi_1 \Rightarrow (e, \xi) \qquad \Omega \vdash \Delta \Rightarrow d}{\Omega@\Xi_0 \vdash \bigvee_\tau \mathbf{a} : \mathbf{prf}\ \mathbf{j}.\theta : \Delta \mathbin{\&} ? \mathbin{\&} (\Xi_1 \cup \{\Omega, \mathbf{a} : \mathbf{prf}\ \mathbf{j}\}) \Rightarrow (e, \xi')}$$

$$\text{where } \xi' \stackrel{def}{=} \upsilon \oplus \{\Omega, \mathbf{a} : \mathbf{prf}\ \mathbf{j} \mapsto \lambda x_1 \cdots x_{|\Omega_{str}|}.\mathbf{isdefined}\ (d\ x_1 \cdots x_{|\Omega_{str}|})\}$$

$$\textsc{GenUnionPred}\quad \frac{\Omega \vdash \mathbf{p} \to \mathbf{q} \qquad \Omega, \mathbf{P} : \mathbf{p} \to \mathbf{q}@\Xi_0 \vdash_\tau \theta : \Delta \mathbin{\&} \phi \mathbin{\&} \Xi_1 \Rightarrow (e, \xi) \qquad \Omega \vdash \Delta \Rightarrow d \qquad \Xi_0|_\mathbf{P} \subseteq \Xi_1|_\mathbf{P}}{\Omega@(\Xi_0 - \Xi_0|_\mathbf{P}) \vdash \bigvee \mathbf{P} : \mathbf{p} \to \mathbf{q}.\theta : \Delta \mathbin{\&} \phi \mathbin{\&} (\Xi_1 - \Xi_1|_\mathbf{P}) \Rightarrow (e', \xi)}$$

$$\text{where for } \Xi_0|_\mathbf{P} = \{\Omega_1, \ldots, \Omega_\mathbf{n}\} \text{ we have } e' \stackrel{def}{=} e[\upsilon(\Omega_1)/x_{\Omega_1}] \cdots [\upsilon(\Omega_\mathbf{n})/x_{\Omega_\mathbf{n}}].$$

Figure 18. Semi-Validator Generation

A Set-theoretic Dependent Simple Products

A.1 Transpose Distribution

Here we show that dependent simple products in the standard set-theoretic model satisfy the transpose distribution property.

For contexts Ω, Ψ, arrows $u : \Psi \to \Omega$, semantic sorts $X \in St(\Omega)$, total objects A over Ω, and total objects B over $\Omega.X$, an arrow $f : \mathfrak{p}(X)^*(A) \to B$ has the form

$$\left(f_{(\omega,x)} : A_\omega \to B_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X}$$

We then have:

$$\begin{aligned}
&u^* f^\sharp \\
&u^* \left(f_{(\omega,x)}\right)^\sharp_{(\omega,x)\in\Omega.X} \\
&= u^* \left(\langle f_{(\omega,x)}\rangle_{x\in X_\omega}\right)_{\omega\in\Omega} \\
&= \left(\langle f_{(u(\psi),x)}\rangle_{x\in X_{u(\psi)}}\right)_{\psi\in\Psi} \\
&= \left(f_{(u(\psi),x)}\right)^\sharp_{(\psi,x)\in\Psi.X\{u\}} \\
&= (\langle u\circ\mathfrak{p}(X\{u\}), \mathfrak{v}_{X\{u\}}\rangle^*_X \left(f_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X})^\sharp \\
&= (\langle u\circ\mathfrak{p}(X\{u\}), \mathfrak{v}_{X\{u\}}\rangle^*_X f)^\sharp \\
&= (\mathfrak{q}(u,X)^* f)^\sharp
\end{aligned}$$

A.2 The Beck-Chevalley Condition

In our set-theoretic model, the canonical natural transformation of dependent simple products is an identity and therefore satisfies the Beck-Chevalley condition. We proceed to demonstrate this.

For semantic contexts Ω and sorts $X \in St(\Omega)$, we define $\Pi_X : Fam(\mathbf{Sets})_{\Omega.X} \to Fam(\mathbf{Sets})_\Omega$ as

$$\Pi_X(\left(B_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X}) = \left(\Pi_{x\in X_\omega} B_{(\omega,x)}\right)_{\omega\in\Omega}$$

$$\Pi_X\left(f_{(\omega,x)} : B_{(\omega,x)} \to C_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X} = \left(\Pi_{x\in X_\omega}\ \ f_{(\omega,x)}\right)_{\omega\in\Omega}$$

We have a bijection of the following form.

$$\frac{(A_\omega)_{\omega\in\Omega} \longrightarrow \left(\Pi_{x\in X_\omega} B_{(\omega,x)}\right)_{\omega\in\Omega} = \Pi_X \left(B_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X}}{\mathfrak{p}(X)^*\left(A_\omega\right)_{\omega\in\Omega} = \left(A_\omega\right)_{(\omega,x)\in\Omega.X} \longrightarrow \left(B_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X}}$$

From top to bottom, this bijection, which we'll call $(-)^\flat$, takes an arrow

$$\left(\langle f_{(\omega,x)} : A_\omega \to B_{(\omega,x)}\rangle_{x\in X_\omega}\right)_{\omega\in\Omega}$$

to

$$\left(f_{(\omega,x)} : A_\omega \to B_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X}$$

From bottom to top, $(-)^\flat$'s inverse $(-)^\sharp$ takes an arrow

$$\left(f_{(\omega,x)} : A_\omega \to B_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X}$$

to

$$\left(\Pi_{x\in X_\omega} f_{(\omega,x)}\right)_{\omega\in\Omega}$$

$(-)^\flat$ underlies an adjunction $\mathfrak{p}(X)^* \dashv \Pi_{\Omega,X}$. To prove this, first consider an arrow g, where

$$g = (g_\omega)_{(\omega)\in\Omega} : (C_\omega)_{\omega\in\Omega} \to (A_\omega)_{\omega\in\Omega}$$

and a arrow f where

$$f = \left(f_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X} : (A_\omega)_{(\omega,x)\in\Omega.X} \to \left(B_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X}$$

We then have

$$(f\circ\mathfrak{p}^*(g))^\sharp$$

$$
\begin{aligned}
&= \left(\left(f_{(\omega,x)} \circ g_\omega\right)_{(\omega,x)\in\Omega.X}\right)^\sharp \\
&= \left(\Pi_{x\in X_\omega} f_{(\omega,x)} \circ g_\omega\right)_{\omega\in\Omega} \\
&= \left((\Pi_{x\in X_\omega} f_{(\omega,x)}) \circ g_\omega\right)_{\omega\in\Omega} \\
&= f^\sharp \circ g
\end{aligned}
$$

Next, consider a arrow f, where

$$f = \left(\langle f_{(\omega,x)}\rangle_{x\in X_\omega}\right)_{\omega\in\Omega} : (A_\omega)_{\omega\in\Omega} \to \left(\Pi_{x\in X_\omega} B_{(\omega,x)}\right)_{\omega\in\Omega}$$

and a arrow g, where

$$g = \left(g_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X} : \left(B_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X} \to \left(D_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X}$$

We then have

$$
\begin{aligned}
&(\Pi_X(g) \circ f)^\flat \\
&= \left(\left(\langle g_{(\omega,x)} \circ f_{(\omega,x)}\rangle_{x\in X_\omega}\right)_{\omega\in\Omega}\right)^\flat \\
&= \left(g_{(\omega,x)} \circ f_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X} \\
&= g \circ f^\flat
\end{aligned}
$$

To obtain the counit of this bijection at component $\left(B_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X}$, writing π_x for the projection $\Pi_{x\in X_\omega} B_{(x,\omega)} \to B_{(x,\omega)}$, we map the identity arrow

$$\left(\langle \pi_x\rangle_{x\in X_\omega}\right)_{\omega\in\Omega} : \Pi_X \left(B_{(\omega,x)}\right)_{(\omega,x)\in\Omega} \to \Pi_X \left(B_{(\omega,x)}\right)_{(\omega,x)\in\Omega}$$

through $(-)^\flat$, obtaining the counit ϵ as

$$\left(\pi_{(\omega,x')} : \Pi_{x\in X_\omega} B_{(\omega,x)} \to B_{(\omega,x')}\right)_{(\omega,x')\in\Omega.X}$$

It is a arrow of type

$$\mathfrak{p}(X)^*\Pi_X \left(B_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X} \longrightarrow \left(B_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X}$$

To prove the Beck-Chevalley condition, we must first concretely describe the canonical natural transformation $u^*\Pi_X \Longrightarrow \Pi_{X\{u\}}\mathfrak{q}(u,X)^*$. As a first step, we concretely describe the natural transformation

$$\mathfrak{p}(X\{u\})^* u^* \Pi_X \xrightarrow{\cong} \mathfrak{q}(u,X)^*\mathfrak{p}(X)^*\Pi_X \xrightarrow{\mathfrak{q}(u,X)\epsilon} \mathfrak{q}(u,X)^*$$

at component $\left(B_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X}$

$$
\begin{aligned}
&\mathfrak{p}(X\{u\})^* u^* \Pi_X \left(B_{\omega,x}\right)_{(\omega,x)\in\Omega.X} \\
&= \mathfrak{p}(X\{u\})^* u^* \left(\Pi_{x\in X_\omega} B_{(\omega,x)}\right)_{\omega\in\Omega} \\
&= \mathfrak{p}(X\{u\})^* \left(\Pi_{x\in X_{u(\psi)}} B_{(u(\psi),x)}\right)_{\psi\in\Psi} \\
&= \left(\Pi_{x\in X_{u(\psi)}} B_{(u(\psi),x)}\right)_{(\psi,x')\in\Psi.X\{u\}} \\
&= \left(\Pi_{x\in X_{\pi\langle u\circ\mathfrak{p}(X\{u\}),\mathfrak{v}_{X\{u\}}\rangle(\psi,x')}} B_{(\pi\langle u\circ\mathfrak{p}(X\{u\}),\mathfrak{v}_{X\{u\}}\rangle(\psi,x'),x)}\right)_{(\psi,x')\in\Psi.X\{u\}} \\
&= \langle u\circ\mathfrak{p}(X\{u\}), \mathfrak{v}_{X\{u\}}\rangle^* \left(\Pi_{x\in X_{\pi(\omega,x')}} B_{(\pi(\omega,x'),x)}\right)_{(\omega,x')\in\Omega.X} \\
&= \langle u\circ\mathfrak{p}(X\{u\}), \mathfrak{v}_{X\{u\}}\rangle^* \left(\Pi_{x\in X_\omega} B_{(\omega,x)}\right)_{(\omega,x')\in\Omega.X}
\end{aligned}
$$

$$\Big\downarrow{\scriptstyle \langle u\circ\mathfrak{p}(X\{u\}),\mathfrak{v}_{X\{u\}}\rangle^*(\pi_{(\omega,x')})_{(\omega,x')\in\Omega.X}}$$

$$\langle u\circ\mathfrak{p}(X\{u\}), \mathfrak{v}_{X\{u\}}\rangle^* \left(B_{(\omega,x')}\right)_{(\omega,x')\in\Omega.X}$$

The above arrow is equal to

$$\left(\pi_{(u(\psi),x')} : \Pi_{x\in X_{u(\psi)}} B_{(u(\psi),x)} \to B_{(u(\psi),x')}\right)_{(\psi,x')\in\Psi.X\{u\}}$$

Transposing gives

$$\left(\Pi_{x'\in X_{u(\psi)}}\left(\pi_{(u(\psi),x')} : (\Pi_{x\in X_{u(\psi)}} B_{(u(\psi),x)}) \longrightarrow B_{(u(\psi),x')}\right)\right)_{\psi\in\Psi}$$

This is the identity arrow on the object

$$\left(\Pi_{x'\in X_{u(\psi)}} B_{(u(\psi),x')}\right)_{\psi\in\Psi}$$

which has the following "type signature"

$$\Pi_{X\{u\}}\mathfrak{q}(u,X)^{*}\left(B_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X} \longrightarrow u^{*}\Pi_{X}\left(B_{(\omega,x)}\right)_{(\omega,x)\in\Omega.X}$$

Because it is an identity, it is clearly invertible.

B Substitution lemmas

We first reproduce some standard definitions and theorems from the semantics of dependent types. These definitions and lemmas will subsequently be used to establish our own soundness proofs.

Lemma B.1. *If $\Omega, \mathrm{a} : \mathrm{p}, \Psi \vdash \mathrm{q}$ and $\Omega \vdash \mathrm{j} : \mathrm{p}$ then $\Omega, \Psi[\mathrm{j}/\mathrm{a}] \vdash \mathrm{q}[\mathrm{j}/\mathrm{a}]$.*

Proof. TODO ◻

For pre-contexts Ω, Ψ and pre-sorts q, r we define the expression $\mathbf{P}(\Omega; \mathrm{p}; \Psi)$ inductively by

$$\mathbf{P}(\Omega; \mathrm{q}; \diamond) \overset{def}{=} \mathfrak{p}([\![\Omega; \mathrm{p}]\!])$$
$$\mathbf{P}(\Omega; \mathrm{q}; \Psi, \mathrm{a}:\mathrm{r}) \overset{def}{=} \mathfrak{q}(\mathbf{P}(\Omega; \mathrm{q}; \Psi), [\![\Omega, \Psi; \mathrm{r}]\!])$$

The idea is that $\mathbf{P}(\Omega; \mathrm{q}; \Psi)$ is a morphism from $[\![\Omega, \mathrm{a} : \mathrm{q}, \Psi]\!]$ to $[\![\Omega, \Psi]\!]$ projecting the q part.

Now let $\Omega, \Psi, \mathrm{p}, \mathrm{q}$ be as before and i a pre-index. We define

$$\mathbf{T}(\Omega; \mathrm{q}; \diamond; \mathrm{i}) \overset{def}{=} \overline{[\![\Omega; \mathrm{i}]\!]}$$
$$\mathbf{T}(\Omega; \mathrm{q}; \Psi, \mathrm{a} : \mathrm{r}; \mathrm{i}) \overset{def}{=} \mathfrak{q}(\mathbf{T}(\Omega; \mathrm{q}; \Psi; \mathrm{i}), [\![\Omega, \mathrm{b} : \mathrm{q}, \Psi; \mathrm{r}]\!]) \qquad \mathrm{b} \text{ fresh}$$

The idea here is that $\mathbf{T}(\Omega; \mathrm{q}; \Psi; \mathrm{i})$ is a morphism from $[\![\Omega, \Psi[\mathrm{i}/\mathrm{b}]]\!]$ to $[\![\Omega, \mathrm{b} : \mathrm{q}, \Psi]\!]$ yielding $[\![\Omega; \mathrm{i}]\!]$ at the $\mathrm{b}:\mathrm{q}$ position and variables otherwise.

The above ideas must be proven simultaneously in the form of weakening and substitution lemmas.

Lemma B.2. *(Weakening) Let Ω, Ψ be pre-contexts, p, q pre-sorts, i a pre-index, and b a fresh variable. Let $\mathrm{A} \in \{\mathrm{p}, \mathrm{i}\}$. The expression $\mathbf{P}(\Omega; \mathrm{p}; \Psi)$ is defined iff $[\![\Omega, \mathrm{p}:\mathrm{q}, \Psi]\!]$ and $[\![\Omega, \Psi]\!]$ are defined and in this case is a morphism from the former to the latter. If $[\![\Omega, \Psi; \mathrm{A}]\!]$ is defined then*

$$[\![\Omega, \mathrm{b}:\mathrm{p}, \Psi; \mathrm{A}]\!] \simeq [\![\Omega, \Psi; \mathrm{A}]\!]\{\mathbf{P}(\Omega; \mathrm{p}; \Psi)\}$$

Lemma B.3. *(Substitution) Let Ω, Ψ be pre-contexts, p, q pre-sorts, i, j pre-indices, and b a fresh variable. Let $\mathrm{A} \in \{\mathrm{p}, \mathrm{i}\}$ and suppose that $[\![\Omega; \mathrm{i}]\!]$ is defined.*

The expression $\mathbf{T}(\Omega; \mathrm{p}; \Psi; \mathrm{i})$ is defined iff $[\![\Omega, \Psi[\mathrm{i}/\mathrm{b}]]\!]$ and $[\![\Omega, \mathrm{b}:\mathrm{p}, \Psi]\!]$ are both defined and in this case is a morphism from the former to the latter. If $[\![\Omega, \mathrm{b}:\mathrm{p}, \Psi; \mathrm{A}]\!]$ is defined then

$$[\![\Omega, \Psi[\mathrm{i}/\mathrm{b}]; \mathrm{A}[\mathrm{i}/\mathrm{b}]]\!] \simeq [\![\Omega, \mathrm{b}:\mathrm{p}, \Psi; \mathrm{A}]\!]\{\mathbf{T}(\Omega; \mathrm{p}; \Psi; \mathrm{i})\}$$

Proof. TODO ◻

Now that the above standard lemmas have been established, we state and prove our substitution lemmas.

Lemma B.4. *If $\Omega, \mathrm{a} : \mathrm{p}, \Psi \vdash \kappa$ and $\Omega \vdash \mathrm{j} : \mathrm{p}$ then $\Omega, \Psi[\mathrm{j}/\mathrm{a}] \vdash \kappa[\mathrm{j}/\mathrm{a}]$ and*

$$\mathbf{T}(\Omega; \mathrm{a} : \mathrm{p}; \Psi; j)^{*}[\![\Omega, \mathrm{a} : \mathrm{p}, \Psi \vdash \kappa]\!] \cong [\![\Omega, \Psi[\mathrm{j}/\mathrm{a}] \vdash \kappa[\mathrm{j}/\mathrm{a}]]\!]$$

Proof. By induction on the proof of $\Omega, \mathrm{a} : \mathrm{p}, \Psi \vdash \kappa$.

Case WFK-IndAbs:

We have $\kappa = \forall \mathbf{a} : \mathbf{q}.\kappa'$. Our premises are $\Omega, \mathbf{a} : \mathbf{p}, \Psi \vdash \mathbf{q}$ and $\Omega, \mathbf{a} : \mathbf{p}, \Psi, \mathbf{b} : \mathbf{q} \mid \Gamma \vdash \kappa'$. Applying lemma B.1 we have $\Omega, \Psi[\mathbf{j}/\mathbf{a}] \vdash \mathbf{q}[\mathbf{j}/\mathbf{a}]$ and hence $[\![\Omega, \Psi[\mathbf{j}/\mathbf{a}] \vdash \mathbf{q}[\mathbf{j}/\mathbf{a}]]\!] \downarrow$. By lemma B.3 we have

$$[\![\Omega, \mathbf{a} : \mathbf{p}, \Psi \vdash \mathbf{q}]\!]\{\mathrm{T}(\Omega; \mathbf{p}; \Psi; \mathbf{j})\} = [\![\Omega, \Psi[\mathbf{j}/\mathbf{a}] \vdash \mathbf{q}[\mathbf{j}/\mathbf{a}]]\!]$$

Applying the IH to the second premise we have

$$\Omega, \Psi[\mathbf{j}/\mathbf{a}], \mathbf{b} : \mathbf{q}[\mathbf{j}/\mathbf{a}] \mid \Gamma[\mathbf{j}/\mathbf{a}] \vdash \kappa'[\mathbf{j}/\mathbf{a}]$$

and

$$\mathbf{T}(\Omega; \mathbf{p}; \Psi, \mathbf{b} : \mathbf{q}; \mathbf{j})^*[\![\Omega, \mathbf{a} : \mathbf{p}, \Psi, \mathbf{b} : \mathbf{q} \mid \Gamma \vdash \kappa']\!] \cong [\![\Omega, \Psi[\mathbf{j}/\mathbf{a}], \mathbf{b} : \mathbf{q}[\mathbf{j}/\mathbf{a}] \mid \Gamma[\mathbf{j}/\mathbf{a}] \vdash \kappa'[\mathbf{j}/\mathbf{a}]]\!]$$

Applying WFK-IndAbs we get

$$\frac{\Omega, \Psi[\mathbf{j}/\mathbf{a}] \vdash \mathbf{q}[\mathbf{j}/\mathbf{a}] \qquad \Omega, \Psi[\mathbf{j}/\mathbf{a}], \mathbf{b} : \mathbf{q}[\mathbf{j}/\mathbf{a}] \mid \Gamma[\mathbf{j}/\mathbf{a}] \vdash \kappa'[\mathbf{j}/\mathbf{a}]}{\Omega, \Psi[\mathbf{j}/\mathbf{a}] \mid \Gamma[\mathbf{j}/\mathbf{a}] \vdash \forall \mathbf{q}[\mathbf{j}/\mathbf{a}].\kappa'[\mathbf{j}/\mathbf{a}]}$$

Additionally, we have

$$\begin{array}{ll}
& \mathbf{T}(\Omega; \mathbf{a} : \mathbf{p}; \Psi; \mathbf{j})^*[\![\Omega, \mathbf{a} : \mathbf{p}, \Psi \vdash \forall \mathbf{a} : \mathbf{q}.\kappa']\!] \\
= & \quad \{\text{Interpretation of KF-Forall}\} \\
& \mathbf{T}(\Omega; \mathbf{a} : \mathbf{p}; \Psi; \mathbf{j})^* \Pi_{[\![\Omega, \mathbf{a}:\mathbf{p}, \Psi \vdash \mathbf{q}]\!]}[\![\Omega, \mathbf{a} : \mathbf{p}, \Psi, \mathbf{b} : \mathbf{q} \vdash \kappa']\!] \\
\cong & \quad \{\text{Beck-Chevalley For Dependent Simple Products, taking } \mathbf{T}(\Omega; \mathbf{a} : \mathbf{p}; \Psi; \mathbf{j}) \text{ as } u\} \\
& \Pi_{[\![\Omega, \mathbf{a}:\mathbf{p}, \Psi \vdash \mathbf{q}]\!]\{\mathrm{T}(\Omega; \mathbf{a}:\mathbf{p}; \Psi; \mathbf{j})\}} \mathrm{q}(\mathbf{T}(\Omega; \mathbf{a} : \mathbf{p}; \Psi; \mathbf{j}), [\![\Omega, \mathbf{a} : \mathbf{p}, \Psi \vdash \mathbf{q}]\!])^*[\![\Omega, \mathbf{a} : \mathbf{p}, \Psi, \mathbf{b} : \mathbf{q} \vdash \kappa']\!] \\
= & \quad \{\text{Lemma B.3}\} \\
& \Pi_{[\![\Omega, \Psi[\mathbf{j}/\mathbf{a}] \vdash \mathbf{q}[\mathbf{j}/\mathbf{a}]]\!]} \mathrm{q}(\mathbf{T}(\Omega; \mathbf{a} : \mathbf{p}; \Psi; \mathbf{j}), [\![\Omega, \mathbf{a} : \mathbf{p}, \Psi \vdash \mathbf{q}]\!])^*[\![\Omega, \mathbf{a} : \mathbf{p}, \Psi, \mathbf{b} : \mathbf{q} \vdash \kappa']\!] \\
= & \quad \{\text{Definition of } \mathbf{T}\} \\
& \Pi_{[\![\Omega, \Psi[\mathbf{j}/\mathbf{a}] \vdash \mathbf{q}[\mathbf{j}/\mathbf{a}]]\!]} \mathbf{T}(\Omega; \mathbf{p}; (\Psi, \mathbf{a} : \mathbf{q}); \mathbf{j})^*[\![\Omega, \mathbf{a} : \mathbf{p}, \Psi, \mathbf{b} : \mathbf{q} \vdash \kappa']\!] \\
= & \quad \{\text{IH}\} \\
& \Pi_{[\![\Omega, \Psi[\mathbf{j}/\mathbf{a}] \vdash \mathbf{q}[\mathbf{j}/\mathbf{a}]]\!]}[\![\Omega, \Psi[\mathbf{j}/\mathbf{a}], \mathbf{b} : \mathbf{q}[\mathbf{j}/\mathbf{a}] \vdash \kappa'[\mathbf{j}/\mathbf{a}]]\!] \\
= & \quad \{\text{Interpretation of KF-Forall}\} \\
& [\![\Omega, \Psi[\mathbf{j}/\mathbf{a}] \vdash \forall \mathbf{q}[\mathbf{j}/\mathbf{a}].\kappa'[\mathbf{j}/\mathbf{a}]]\!] \\
= & \quad \{\text{Definition of Index-to-Type Substitution}\} \\
& [\![\Omega, \Psi[\mathbf{j}/\mathbf{a}] \vdash (\forall \mathbf{q}.\kappa')[\mathbf{j}/\mathbf{a}]]\!]
\end{array}$$

Other cases:

TODO

□

Lemma B.5. *If* $\Omega, \mathbf{a} : \mathbf{p}, \Psi \mid \Gamma \vdash \tau : \kappa$ *and* $\Omega \vdash \mathbf{j} : \mathbf{p}$ *then* $\Omega, \Psi[\mathbf{j}/\mathbf{a}] \mid \Gamma[\mathbf{j}/\mathbf{p}] \vdash \tau[\mathbf{j}/\mathbf{p}] : \kappa[\mathbf{j}/\mathbf{p}]$ *and*

$$\mathbf{T}(\Omega; \mathbf{a} : \mathbf{p}; \Psi; \mathbf{j})^*[\![\Omega, \mathbf{a} : \mathbf{p}, \Psi \mid \Gamma \vdash \tau : \kappa]\!] \cong [\![\Omega, \Psi[\mathbf{j}/\mathbf{a}] \mid \Gamma[\mathbf{j}/\mathbf{a}] \vdash \tau[\mathbf{j}/\mathbf{a}] : \kappa[\mathbf{j}/\mathbf{a}]]\!]$$

Proof. By induction on the proof of $\Omega, \mathbf{a} : \mathbf{p}, \Psi \mid \Gamma \vdash \tau : \kappa$.

Case K-IndAbs:

We have $\tau = \lambda \mathbf{b} : \mathbf{q}.\tau'$ and $\kappa = \forall \mathbf{q}.\kappa'$. Our premises are $\Omega, \mathbf{a} : \mathbf{p}, \Psi \vdash \mathbf{q}$ and $\Omega, \mathbf{a} : \mathbf{p}, \Psi, \mathbf{b} : \mathbf{q} \mid \Gamma \vdash \tau' : \kappa'$. Applying lemma B.1 we have $\Omega, \Psi[\mathbf{j}/\mathbf{a}] \vdash \mathbf{q}[\mathbf{j}/\mathbf{a}]$ and hence $[\![\Omega, \Psi[\mathbf{j}/\mathbf{a}] \vdash \mathbf{q}[\mathbf{j}/\mathbf{a}]]\!] \downarrow$. By lemma B.3 we have

$$[\![\Omega, \mathbf{a} : \mathbf{p}, \Psi \vdash \mathbf{q}]\!]\{\mathrm{T}(\Omega; \mathbf{p}; \Psi; \mathbf{j})\} = [\![\Omega, \Psi[\mathbf{j}/\mathbf{a}] \vdash \mathbf{q}[\mathbf{j}/\mathbf{a}]]\!]$$

Applying the IH to the second premise we have

$$\Omega, \Psi[\mathbf{j}/\mathbf{a}], \mathbf{b} : \mathbf{q}[\mathbf{j}/\mathbf{a}] \mid \Gamma[\mathbf{j}/\mathbf{a}] \vdash \tau'[\mathbf{j}/\mathbf{a}] : \kappa'[\mathbf{j}/\mathbf{a}]$$

and

$$\mathbf{T}(\Omega; \mathbf{p}; \Psi, \mathbf{b} : \mathbf{q}; \mathbf{j})^*[\![\Omega, \mathbf{a} : \mathbf{p}, \Psi, \mathbf{b} : \mathbf{q} \mid \Gamma \vdash \tau' : \kappa']\!] \cong [\![\Omega, \Psi[\mathbf{j}/\mathbf{a}], \mathbf{b} : \mathbf{q}[\mathbf{j}/\mathbf{a}] \mid \Gamma[\mathbf{j}/\mathbf{a}] \vdash \tau'[\mathbf{j}/\mathbf{a}] : \kappa'[\mathbf{j}/\mathbf{a}]]\!]$$

Now, applying the K-IndAbs rule, we get

$$\frac{\Omega, \Psi[j/a] \vdash q[j/a] \qquad \Omega, \Psi[j/a], b : q[j/a] \mid \Gamma[j/a] \vdash \tau'[j/a] : \kappa'[j/a]}{\Omega, \Psi[j/a] \mid \Gamma[j/a] \vdash \lambda b : q[j/a].\tau'[j/a] : \forall q[j/a].\kappa'[j/a]}$$

We also have

$$\begin{aligned}
& [\![\Omega, \Psi'[j/a] \mid \Gamma[j/a] \vdash (\lambda b : q.\tau')[j/a] : (\forall q.\kappa')[j/a]]\!] \\
= \quad & \{\text{Definition of substitution}\} \\
& [\![\Omega, \Psi'[j/a] \mid \Gamma[j/a] \vdash \lambda b : q[j/a].\tau'[j/a] : \forall q[j/a].\kappa'[j/a]]\!] \\
= \quad & \{\text{Interpretation of K-IndAbs}\} \\
& [\![\Omega, \Psi'[j/a], b : q[j/a] \mid \Gamma[j/a] \vdash \tau'[j/a] : \kappa'[j/a]]\!]^\sharp \\
\cong \quad & \{\text{IH}\} \\
& (\mathbf{T}(\Omega; p; (\Psi, b : q); j)^*[\![\Omega, a : p, \Psi, b : q \mid \Gamma \vdash \tau' : \kappa']\!])^\sharp \\
\cong \quad & \{\text{Transpose distribution}\} \\
& \mathbf{T}(\Omega; p; \Psi; j)^*[\![\Omega, a : p, \Psi, b : q \mid \Gamma \vdash \tau' : \kappa']\!]^\sharp \\
= \quad & \{\text{Interpretation of K-IndAbs}\} \\
& \mathbf{T}(\Omega; p; \Psi; j)^*[\![\Omega, a : p, \Psi \mid \Gamma \vdash \lambda b : q.\tau' : \forall q.\kappa']\!]
\end{aligned}$$

Other cases:
TODO.

□

C Validator Generation

Theorem C.1. *Suppose $\Omega @ \Xi_0 \vdash_\tau \theta : \Delta \,\&\, \phi \,\&\, \Xi_1 \Rightarrow (e, \xi)$. If υ is a mapping that takes each sort context $\Omega_i \in \Xi_0 = \{\Omega_1, \ldots, \Omega_n\}$ to a semi-decider for $\Omega \vdash_i$, then $e[\upsilon(\Omega_1)/x_{\Omega_1}] \cdots [\upsilon(\Omega_n)/x_{\Omega_n}]$ is a semi-validator for $\Omega \mid \diamond \vdash \theta : *$ at Δ with respect to $\diamond \mid \diamond \vdash \tau : *$.*

Proof. By induction on the proof of $\Omega @ \Xi_0 \vdash_\tau \theta : \Delta \,\&\, \phi\ \Xi_1 \Rightarrow (e, \xi)$.

Case GenDict:
By the IH, if υ is a mapping that takes each sort context $\Omega_i \in \Xi_0 = \{\Omega_1, \ldots, \Omega_n\}$ to a semi-decider for $\Omega \vdash_i$ then $e[\upsilon(\Omega_1)/x_{\Omega_1}] \cdots [\upsilon(\Omega_n)/x_{\Omega_n}]$ is a semi-validator for $\Omega, a : str \mid \diamond \vdash \theta : *$ at Δ, a with respect to $\diamond \mid \diamond \vdash \tau : *$.

For the inductive step, we need to prove that if υ is a mapping that takes each sort context $\Omega_i \in \Xi_0 = \{\Omega_1, \ldots, \Omega_n\}$ to a semi-decider for $\Omega \vdash_i$ then

$$e'[\upsilon(\Omega_i)/x_{\Omega_i}]^{i \in 1..n} = \lambda x_1 \ldots x_{|\Omega_{str}|}.\mathbf{first10}\ d\ x_1 \cdots x_{|\Omega_{str}|}\ \mathbf{with}\ x\ \mathbf{in}\ e[\upsilon(\Omega_i)/x_{\Omega_i}]^{i \in 1..n}\ x_1 \cdots x_{\Omega_{str}}\ x$$

is a semi-validator for $\Omega \mid \diamond \vdash \{[a : str] : \theta\} : *$ at Δ with respect to $\diamond \mid \diamond \vdash \tau : *$.

To this end, let $[s_1, \ldots, s_m] \in [\![\Omega \vdash_{str}]\!]$ and $[t_1, \ldots, t_k] \stackrel{def}{=} [\![\Omega \vdash \Delta]\!]_{[s_1,\ldots,s_m]}$. Suppose there exists $[z_1, \ldots, z_{|\Omega|}] \in [\![\Omega \vdash]\!]$ such that $z_1 = s_1, \ldots, z_m = s_m$ and a database instance I such that $I|_{[t_1,\ldots,t_k]} \in [\![\Omega \mid \diamond \vdash \{[a : str] : \theta\} : *]\!]_{[z_1,\ldots,z_{|\Omega|}]}(*)$. Then, for each string t' such that $I|_{[t_1,\ldots,t_k,t']}$ is active, we have $[\![\Omega, a : str \vdash \Delta, a]\!]_{[s_1,\ldots,s_m,t']} = [t_1, \ldots, t_k, t']$. The interpretation of dictionary types in figure 11 then gives us $I|_{[t_1,\ldots,t_k,t']} \in [\![\Omega, a : str \mid \diamond \vdash \theta : *]\!]_{[z_1,\ldots,z_m,t',z_{m+1},\ldots z_{|\Omega|}]}(*)$. We apply our IH to obtain

$$I \mid e[\upsilon(\Omega_i)/x_{\Omega_i}]^{i \in 1..n}\ s_1 \cdots s_n\ t' \Downarrow true \qquad (i)$$

Now, we have $I \mid e'[\upsilon(\Omega_i)/x_{\Omega_i}]^{i \in 1..n}\ s_1 \cdots s_m \longrightarrow^* (e[\upsilon(\Omega_i)/x_{\Omega_i}]^{i \in 1..n}\ s_1 \cdots s_m\ t'_1) \wedge \ldots \wedge (e[\upsilon(\Omega_i)/x_{\Omega_i}]^{i \in 1..n}\ s_1 \cdots s_m\ t'_l)$ where $t'_1, \ldots, t'_l$ are up to 10 of the lexicographically smallest strings t' such that $I|_{[t_1,\ldots,t_k,t']}$ is active. By (i) we have:

$$I \mid (e[\upsilon(\Omega_i)/x_{\Omega_i}]^{i \in 1..n}\ s_1 \cdots s_m\ t'_1) \wedge \ldots \wedge (e[\upsilon(\Omega_i)/x_{\Omega_i}]^{i \in 1..n}\ s_1 \cdots s_m\ t'_l) \longrightarrow^* true \wedge \cdots \wedge true \longrightarrow^* true$$

Hence, $I \mid e'[\upsilon(\Omega_i)/x_{\Omega_i}]^{i \in 1..n}\ s_1 \ldots s_m \Downarrow true$.

Case GenRecord**:**

By the IH, for each $i \in 1..n$, if v_i is a mapping that takes each sort context $\Omega_{i,j} \in \Xi_{0,i} = \{\Omega_{i,1}, \ldots, \Omega_{i,n_i}\}$ to a semi-decider for $\Omega_{i,j} \vdash$ then $e_i[v(\Omega_{i,1})/x_{\Omega_{i,1}}] \cdots [v(\Omega_{i,n_i})/x_{\Omega_{i,n_i}}]$ is a semi-validator for $\Omega \mid \diamond \vdash \theta_i : *$ at Δ, s_i with respect to $\diamond \mid \diamond \vdash \tau : *$.

For the inductive step, we need to prove that if v is a mapping that takes each sort context $\Omega_{i,j} \in \bigcup_{i\in 1..n} \Xi_{0,i} = \{\Omega_{1,1}, \ldots, \Omega_{1,n_1}, \ldots, \Omega_{n,1}, \ldots, \Omega_{n,n_n}\}$ to a semi-decider for $\Omega_{i,j} \vdash$ then

$$\begin{aligned}
&e'[v(\Omega_{i,j})/x_{\Omega_{i,j}}]^{i\in 1..n, j\in 1..n_i}\\
&= \lambda x_1, \ldots, x_{|\Omega_{str}|}.(e_1[v(\Omega_{i,j})/x_{\Omega_{i,j}}]^{i\in 1..n, j\in 1..n_i}\ x_1 \cdots x_{|\Omega_{str}|}) \land \cdots \land (e_n[v(\Omega_{i,j})/x_{\Omega_{i,j}}]^{i\in 1..n, j\in 1..n_i}\ x_1 \cdots x_{|\Omega_{str}|})\\
&= \lambda x_1, \ldots, x_{|\Omega_{str}|}.(e_1[v(\Omega_{1,j})/x_{\Omega_{1,j}}]^{j\in 1..n_1}\ x_1 \cdots x_{|\Omega_{str}|}) \land \cdots \land (e_n[v(\Omega_{n,j})/x_{\Omega_{n,j}}]^{j\in 1..n_n}\ x_1 \cdots x_{|\Omega_{str}|})
\end{aligned}$$

is a semi-validator for $\Omega \mid \diamond \vdash \{s_i : \theta_i{}^{i\in 1..n}\} : *$ at Δ with respect to $\diamond \mid \diamond \vdash \tau : *$.

To this end, let $[r_1, \ldots, r_m] \in [\![\Omega_{str} \vdash]\!]$ and $[t_1, \ldots, t_k] = [\![\Omega \vdash \Delta]\!]_{[r_1,\ldots,r_m]}$. Suppose there exists $[z_1, \ldots, z_{|\Omega|}] \in [\![\Omega \vdash]\!]$ such that $z_1 = r_1, \ldots, z_m = r_m$ and a database instance I such that $I|_{[t_1,\ldots,t_k]} \in [\![\Omega \mid \diamond \vdash \{s_i : \theta_i{}^{i\in 1..n}\}]\!]_{[t_1,\ldots,t_k]}(*)$. Then the interpretation of record types in figure 11 gives us $I|_{[t_1,\ldots,t_k,s_i]} \in [\![\Omega \mid \diamond \vdash \theta_i : *]\!]_{[z_1,\ldots,z_{|\Omega|}]}(*)$ for $i \in 1..n$.

Then by the IH, $I \mid e_i[v(\Omega_{i,j})/x_{\Omega_{i,j}}]^{j\in 1..n_i}\ r_1 \cdots r_m \Downarrow \mathit{true}$ for $i \in 1..n$. Hence, we have

$$\begin{aligned}
&I \mid e'[v(\Omega_{i,j})/x_{\Omega_{i,j}}]^{i\in 1..n, j\in 1..n_i}\ r_1 \cdots r_m\\
&\to^* (e_1[v(\Omega_{1,j})/x_{\Omega_{1,j}}]^{j\in 1..n_1}\ r_1 \cdots r_m) \land \cdots \land (e_n[v(\Omega_{n,j})/x_{\Omega_{n,j}}]^{j\in 1..n_n}\ r_1 \cdots r_m)\\
&\to^* \mathit{true} \land \cdots \land \mathit{true}\\
&\Downarrow \mathit{true}.
\end{aligned}$$

Case GenRefinement**:**

We need to prove that if v is a mapping that takes $\Omega' = \Omega, \mathbf{a} : \mathbf{str}, \mathbf{b} : \mathbf{prf}\ (\mathbf{P}\ \mathbf{b_1} \cdots \mathbf{b_n})$ to a semi-decider for Ω' then

$$e[v(\Omega')/x_{\Omega'}] \ = \ (\lambda x_1 \ldots x_n.\mathbf{let}\ x_{loc} = d\ x_1 \cdots x_n\ \mathbf{in}\ ((\mathbf{isdefined}\ x_{loc}) \land (v(\Omega')\ x_1 \cdots x_n\ (\mathbf{read}\ x_{loc}))))$$

is a semi-validator for $\Omega \mid \diamond \vdash \{\mathbf{a} : \mathbf{str} \mid \mathbf{P}\ \mathbf{b_1}\ \cdots\ \mathbf{b_n}\} : *$ at Δ with respect to $\diamond \mid \diamond \vdash \tau : *$.

Let $[s_1, \ldots, s_m] \in [\![\Omega_{str} \vdash]\!]$ and $[t_1, \ldots, t_k] = [\![\Omega \vdash \Delta]\!]_{[s_1,\ldots,s_m]}$. Suppose there exists $[z_1, \ldots, z_{|\Omega|}] \in [\![\vdash \Omega]\!]$ such that $s_1 = z_1, \ldots, s_m = z_m$ and a database instance I such that $I|_{[t_1,\ldots,t_k]} \in [\![\Omega \mid \diamond \vdash \{\mathbf{a} : \mathbf{str} \mid \mathbf{P}\ \mathbf{b_1} \cdots \mathbf{b_n}\} : *]\!]_{[z_1,\ldots,z_{|\Omega|}]}(*)$. Expanding the interpretation of refinement types in figure 11 gives $I|_{[t_1,\ldots,t_k]}([\,]) \downarrow$ and $[\![\Omega, \mathbf{a} : \mathbf{str} \vdash \mathbf{P}\ \mathbf{b_1} \cdots \mathbf{b_n}]\!]_{[z_1,\ldots,z_{|\Omega|}, I|_{[t_1,\ldots,t_k]}([\,])]} = \mathit{true}$. By the fourth line of figure 9 we have $[\![\Omega, \mathbf{a} : \mathbf{str} \vdash \mathbf{prf}\ (\mathbf{P}\ \mathbf{b_1} \cdots \mathbf{b_n})]\!]_{[z_1,\ldots,z_{|\Omega|}, I|_{[t_1,\ldots,t_k]}([\,])]} = \{*\}$. By the second line of figure 9 we have $[\![\Omega, \mathbf{a} : \mathbf{str}, \mathbf{prf}\ (\mathbf{P}\ \mathbf{b_1} \cdots \mathbf{b_n})]\!] = [\![\Omega, \mathbf{a} : \mathbf{str}]\!].[\![\Omega, \mathbf{a} : \mathbf{str} \vdash \mathbf{prf}\ (\mathbf{P}\ \mathbf{b_1} \cdots \mathbf{b_n})]\!]$. Hence, the set $[\![\Omega']\!] = [\![\Omega, \mathbf{a} : \mathbf{str}]\!].[\![\Omega, \mathbf{a} : \mathbf{str} \vdash \mathbf{prf}\ (\mathbf{P}\ \mathbf{b_1} \cdots \mathbf{b_n})]\!]$ contains the element $[z_1, \ldots, z_{|\Omega|}, I|_{[t_1,\ldots,t_k]}([\,]), *]$.

Then we have
$I \mid e[v(\Omega')/x_{\Omega'}]\ s_1 \cdots s_m \to^* (\mathbf{isdefined}\ [t_1, \ldots, t_k]) \ \land \ (v(\Omega')\ s_1 \cdots s_m\ I([t_1, \ldots, t_k])) \to^* \mathit{true} \land \mathit{true} \to \mathit{true}$

Case GenUnionPrf**:**

By the IH, if v is a mapping that takes each sort context $\Omega_i \in \Xi_0 = \{\Omega_1, \ldots, \Omega_n\}$ to a semi-decider for $\Omega_i \vdash$ then $e[v(\Omega_1)/x_{\Omega_1}] \cdots [v(\Omega_n)/x_{\Omega_n}]$ is a semi-validator for $\Omega, \mathbf{a} : \mathbf{prf}\ \mathbf{j} \mid \diamond \vdash \theta : *$ at Δ with respect to $\diamond \mid \diamond \vdash \tau : *$.

For the inductive step, we need to prove that if v is a mapping that takes each sort context $\Omega_i \in \Xi_0 = \{\Omega_1, \ldots, \Omega_n\}$ to a semi-decider for $\Omega_i \vdash$ then $e[v(\Omega_1)/x_{\Omega_1}] \cdots [v(\Omega_n)/x_{\Omega_n}]$ is a semi-validator for $\Omega \mid \diamond \vdash \bigvee \mathbf{a} : \mathbf{prf}\ \mathbf{j}.\theta : *$ at Δ with respect to $\diamond \mid \diamond \vdash \tau : *$.

To this end, suppose v takes each sort context $\Omega_i \in \Xi_i = \{\Omega_1, \ldots, \Omega_n\}$ to a semi-decider for $\Omega_i \vdash$. Then applying the IH tells us that $e[v(\Omega_1)/x_{\Omega_1}] \cdots [v(\Omega_n)/x_{\Omega_n}]$ is a semi-validator for $\Omega, \mathbf{a} : \mathbf{prf}\ \mathbf{j} \mid \diamond \vdash \theta : *$ at Δ with respect to

$\diamond \mid \diamond \vdash \tau : *$. Expanding the definition of semi-validator gives us that for all $[s_1, \ldots, s_n] \in [\![\Omega_{str} \vdash]\!]$, $l \in [\![\Omega_{str} \vdash \Delta]\!]_{[s_1,\ldots,s_n]}$, and $I \in [\![\diamond \mid \diamond \vdash \tau : *]\!]_*(*)$, if there exists a $[z_1, \ldots, z_{|\Omega,\mathbf{a}:\mathbf{prf\ j}|}] \in [\![\vdash \Omega, \mathbf{a} : \mathbf{prf\ j}]\!]$ with $z_1 = s_1, \ldots, z_n = s_n$ and

$$I|_l \in [\![\Omega, \mathbf{a} : \mathbf{prf\ j} \mid \diamond \vdash \theta : *]\!]_{[z_1,\ldots,z_{|\Omega,\mathbf{a}:\mathbf{prf\ j}|}]}(*)$$

then

$$I \mid e\ s_1 \cdots s_n \Downarrow true$$

Let $[s_1, \ldots, s_n] \in [\![\Omega_{str} \vdash]\!]$, $l \in [\![\Omega_{str} \vdash \Delta]\!]_{[s_1,\ldots,s_n]}$, and $I \in [\![\diamond \mid \diamond \vdash \tau : *]\!]$. Suppose there exists $[z_1, \ldots, z_{|\Omega|}] \in [\![\Omega \vdash]\!]$ with $z_1 = s_1, \ldots, z_n = s_n$ and

$$I|_l \in [\![\Omega \mid \diamond \vdash \bigvee(\mathbf{a} : \mathbf{prf\ j}).\theta : *]\!]_{[z_1,\ldots,z_{|\Omega|}]}(*)$$

Referring to the interpretation of union types in figure 11, we see that there exists a $M \in [\![\Omega \vdash \mathbf{prf\ j}]\!]_{[z_1,\ldots,z_{|\Omega|}]}$ such that

$$I|_l \in ([\![\Omega \mid \diamond \vdash \lambda \mathbf{a} : \mathbf{prf\ j}.\theta : \forall \mathbf{a} : \mathbf{prf\ j}.*]\!]_{[z_1,\ldots,z_{|\Omega|}]}(*))_M$$

that is,

$$I|_l \in ([\![\Omega, \mathbf{a} : \mathbf{prf\ j} \mid \diamond \vdash \theta : *]\!]^{\sharp}_{[z_1,\ldots,z_{|\Omega|}]}(*))_M$$

□

D Garbage After this Point

E A Non-Categorical Attempt

Our judgments are of the form

$$\mathbf{P} : \mathbf{str} \to \mathbf{prop}, \mathbf{Q} : (\mathbf{x} : \mathbf{str}) \to \mathbf{prf}\ (\mathbf{P\ x}) \to \mathbf{prop}, \ldots \mid \mathbf{a} : \mathbf{str}, \mathbf{b} : \mathbf{str} \mid \mathbf{prf}\ (\mathbf{P\ a}), \ldots \vdash \tau :: * \ \&\ r \ \&\ (S \mid T)$$

We have rearraged our subject context into a predicate context, followed by a string (location) context, followed by a proof context. In our original system it is technically possible to have predicates that depend on proofs or strings, but I haven't seen a need for this in actual schemas. On the right side of the turnstile, we have a normalized type (schema), followed by a kind (should be proper) followed by an effect scalar (either + meaning that each instance satisfying the schema is defined in at least one location, or ? meaning no information), followed by $(S \mid T)$ where S is a set of sources and T is a set of sinks.

A source $\Theta \mid \Xi \in S$ is a location context Θ followed by a formula set Ξ. It tells us that we can decide Ξ by checking if Θ is populated. A sink $\Xi \mid \Theta$ is also a location context and a formula set. It conveys the requirement that we can decide the formula set Θ under location context Ξ.

In addition to types and terms, we can associate with each context a collection of deciders. A decider determines if its context has any points, using a database instance as input. Like types and terms, deciders can be reindexed. If we can reindex a source into a sink, we then obtain a decider for the sink.

Let $\Omega_{\mathbf{Pred}} \vdash \Xi$ and $\Omega_{\mathbf{Pred}}, \Omega_{\mathbf{Loc}}, \Omega_{\mathbf{Prf}} \vdash \tau :: *$. For $\Omega'_{\mathbf{Loc}} \mid \Omega'_{\mathbf{Prf}} \in \Xi$ we define the set $[\Omega'_{\mathbf{Loc}} \mid \Omega'_{\mathbf{Prf}}]_\tau$ of τ-*deciders* for $\Omega'_{\mathbf{Loc}} \mid \Omega'_{\mathbf{Prf}}$ as the set of all expressions e such that for all $I \in \mathit{Inst}$ we have that $I \mid e \Downarrow \mathbf{true}$ if and only if there exist $\omega_{\mathbf{Pred}} \in [\![\Omega_{\mathbf{Pred}}]\!]$, $\omega \in [\![\Omega_{\mathbf{Pred}} \mid \Omega_{\mathbf{Loc}} \mid \Omega_{\mathbf{Pref}}]\!]$, $\omega' \in [\![\Omega_{\mathbf{Pred}} \mid \Omega'_{\mathbf{Loc}} \mid \Omega'_{\mathbf{Prf}}]\!]$ such that ω and ω' extend $\omega_{\mathbf{Pred}}$ and $I \in [\![\tau]\!]_\omega$.

$\forall I \in \mathit{Inst}$.

$(I \mid e \Downarrow \mathbf{true}) \Leftrightarrow$

$(\exists \omega_{\mathbf{Pred}} \in [\![\Omega_{\mathbf{Pred}}]\!], \omega \in [\![\Omega_{\mathbf{Pred}} \mid \Omega_{\mathbf{Loc}} \mid \Omega_{\mathbf{Pref}}]\!], \omega' \in [\![\Omega_{\mathbf{Pred}} \mid \Omega'_{\mathbf{Loc}} \mid \Omega'_{\mathbf{Prf}}]\!]$. ω and ω' extend $\omega_{\mathbf{Pred}}$ and $I \in [\![\tau]\!]_\omega)$

a valuation ξ of Ξ is a mapping from $\mathit{dom}(\Xi)$ to terms such that for each $\mathbf{x} : \Omega \in \Xi$ we have $\xi(\Omega)$

We interpret a judgment $\Omega_{\mathbf{Pred}} \mid \Omega_{\mathbf{Loc}} \mid \Omega_{\mathbf{Prf}} @ \Xi \vdash \tau : * \ \&\ \phi \ \&\ \Upsilon$ as a term e such that for each valuation ξ of Ξ we have e_ξ

Theorem E.1. *If* $\Omega_{\mathbf{Pred}} \mid \Omega_{\mathbf{Loc}} \mid \Omega_{\mathbf{Prf}} @ \Xi \vdash \tau : * \ \&\ \phi \ \&\ \Upsilon$ *then* $\Omega_{\mathbf{Pred}}, \Omega_{\mathbf{Loc}}, \Omega_{\mathbf{Prf}} \mid \diamond \vdash \tau : *$.

F An Order-Theoretic Model (and Redesign)

To avoid dangling foreign keys, industrial databases often avoid removing data entries, giving their evolution over time an inflationary character. In such a situation we can interpret the sort **prop** as the ordinal 2, containing the two elements *known* and *unknown*, ordered such that *unknown* ≤ *known*. A predicate $\mathbf{P} : \mathbf{str} \to \mathbf{prop}$ then represents a set of elements such that membership in the set is either known (definitely a member) or undetermined (may or may not be a member).

F.1 Contexts, Sorts, Indices, and Subsitution

We capture the above intuition with a CwF. Its category of contexts is **Posets**. For semantic contexts P, we define $St(\Omega)$ (read the *semantic sorts* of context Ω) as the collection of Ω-indexed families of posets. Such a family $(X_\omega)_{\omega\in\Omega}$ is a poset-indexed family rather than a set-indexed family; i.e., for $\omega_1, \omega_2 \in \Omega$ with $\omega_1 \leq \omega_2$ we have a chosen monotone injection $i_{\omega_1\leq\omega_2} : X_{\omega_1} \to X_{\omega_2}$ such that for $\omega \in \Omega$ we have $i_{\omega\leq\omega} = id_{X_\omega}$ and for $\omega_1 \leq \omega_2 \leq \omega_3$ we have $i_{\omega_1\leq\omega_3} = i_{\omega_2\leq\omega_3} \circ i_{\omega_1\leq\omega_2}$. A poset-indexed family of posets $(X_\omega)_{\omega\in\Omega}$ can itself be considered a poset whose elements are families $(M_\omega \in X_\omega)_{\omega\in\Omega}$ and $(M_\omega)_{\omega\in\Omega} \leq \left(M'_\omega\right)_{\omega\in\Omega} \Leftrightarrow (M_\omega \leq M'_\omega)$ for all $\omega \in \Omega$.

For semantic contexts Ω and all $X \in St(P)$ we define $In(P, X)$ as the collection of all Ω-indexed families $(M_\omega \in X_\omega)_{\omega\in\Omega}$ such that for $\omega_1 \leq_P \omega_2$ we have $i(M_{\omega_1}) \leq_{X_{\omega_2}} M_{\omega_2}$.

For each monotone function $f : \Omega \to \Psi$ we define a sort-level semantic substitution operator $-\{f\} : St(\Psi) \to St(\Omega)$ as

$$X\{f\} \stackrel{def}{=} \left(X_{f(\omega)}\right)_{\omega\in\Omega}$$

For $\omega_1 \leq \omega_2$ we have $f(\omega_1) \leq f(\omega_2)$ and so our chosen monotone injection is

$$i_{\omega_1\leq\omega_2} : X\{f\}_{\omega_1} \to X\{f\}_{\omega_2} \stackrel{def}{=} i_{f(\omega_1)\leq f(\omega_2)}$$

For $X \in St(\Psi)$ a index-level semantic substution operator $-\{f\} : In(\Psi, X) \to In(\Omega, X\{f\})$.

F.2 Comprehensions

Let Ω be a semantic context and X a semantic sort in context Ω. The comprehension $\Omega.X$ is the poset of pairs (ω, x) with $\omega \in \Omega$ and $x \in X_\omega$ such that

$$(\omega_1, x_1) \leq_{\Omega.X} (\omega_2, x_2) \stackrel{def}{\Leftrightarrow} (\omega_1 \leq \omega_2) \wedge (i(x_1) \leq x_2)$$

We have a monotone function $\mathfrak{p}(X) : \Omega.X \to \Omega$ defined as

$$\mathfrak{p}(X)(\omega, x) \stackrel{def}{=} \omega$$

and also a semantic index term $\mathfrak{v}_X \in In(\Omega.X, X\{\mathfrak{p}(X)\})$ defined as

$$(\mathfrak{v}_X)_{(\omega,x)} \stackrel{def}{=} x$$

Lemma F.1. *Let $f : \Omega \to \Psi$ be a monotone function, $X \in St(\Psi)$, and $M \in In(\Omega, X\{f\})$. Then there exists a unique morphism $\langle f, M\rangle_X : \Omega \to \Psi.X$ satisfying $\mathfrak{p}(X) \circ \langle f, M\rangle_X = f$ and $\mathfrak{v}_X\{\langle f, M\rangle_X\} = M$.*

Proof. The morphism is

$$\omega \stackrel{\langle f,M\rangle_X}{\mapsto} (f(\omega), M_\omega)$$

Clearly we have $\mathfrak{p}(X) \circ \langle f, M\rangle_X = f$ since

$$\omega \stackrel{\langle f,M\rangle_X}{\mapsto} (f(\omega), M_\omega) \stackrel{\mathfrak{p}(X)}{\mapsto} f(\omega)$$

Also, we have $\mathfrak{v}_X\{\langle f, M\rangle_X\} = M$ since

$$\mathfrak{v}_X\{\langle f, M\rangle_X\}_\omega = (\mathfrak{v}_X)_{\langle f,M\rangle_X(\omega)} = (\mathfrak{v}_X)_{(f(\omega),M_\omega)} = M_\omega$$

Also, $\langle f, M\rangle_X$ is monotone since if $\omega \leq \omega'$ we have

$$\langle f, M\rangle_X(\omega) = (f(\omega), M_\omega) \leq (f(\omega'), M_{\omega'}) = \langle f, M\rangle_X(\omega')$$

(Reminder: Since $M \in X\{f\}$ we have $M_\omega \in X_{f(\omega)}$.)

TODO: prove uniqueness ◻

F.3 Π-sorts

This model does not have arbitrary Π-sorts. However, identifying the *pointed sorts* $PSt(\Omega)$ as those families of posets over Ω whose component posets all have $\perp$ (minimum) elements, our model has those Π-sorts whose codomains are pointed.

Theorem F.2. *For all contexts Ω, $X \in St(\Omega)$, $Y \in PSt(\Omega.X)$, our model supports the Π-sort $\Pi(X, Y)$.*

$$
\begin{array}{lcl}
\mathit{TypeVars} & \stackrel{def}{=} & \text{the set of all type variables} \\
\mathit{FormulaVars} & \stackrel{def}{=} & \text{the set of all index variables} \\
\mathit{SubjectVars} & \stackrel{def}{=} & \text{the set of all index variables} \\
x, y, z & \in & \mathit{TypeVars} \\
\mathbf{a}, \mathbf{b}, \mathbf{P} & \in & \mathit{FormulaVars} \\
\mathbf{s}, \mathbf{t} & \in & \mathit{Strings} \\
\mathbf{u}, \mathbf{v} & \in & \mathit{SubjectVars}
\end{array}
$$

$$
\begin{array}{llll}
\mathbf{i}, \mathbf{j}, \mathbf{k}\ \text{(formula)} & ::= & \mathbf{App}_{[\mathbf{a}:\mathbf{q}],\mathbf{p}}(\mathbf{j}, \mathbf{k}) & \text{(formula application)} \\
 & | & \mathbf{a} & \text{(formula variable)} \\
 & | & \mathbf{true} & \text{(true formula)} \\
 & & & \\
\mathbf{m}, \mathbf{n}\ \text{(multiplicity)} & ::= & \mathbf{one} \mid \mathbf{lone} \mid \mathbf{some} \mid \mathbf{set} & \\
 & & & \\
\mathbf{g}, \mathbf{h}\ \text{(subject)} & ::= & \mathbf{s} & \text{(string)} \\
 & | & \mathbf{u} & \text{(subject var)} \\
 & & & \\
\mathbf{d}, \mathbf{e}, \mathbf{f}\ \text{(sort)} & ::= & \mathbf{str} & \text{(string sort)} \\
 & & & \\
\mathbf{p}, \mathbf{q}, \mathbf{r}\ \text{(pre-signature)} & ::= & \mathbf{prop} & \text{(prop sig)} \\
 & | & \mathbf{prf}\ \mathbf{j} & \text{(proof sig)} \\
 & | & (\mathbf{u} : \mathbf{d}) \xrightarrow{\mathbf{m}} \mathbf{r} & \text{(subject-to-formula function sort)} \\
 & | & (\mathbf{a} : \mathbf{q}) \longrightarrow \mathbf{r} & \text{(formula-to-formula function sort)}
\end{array}
$$

$$
\begin{array}{llll}
\Omega, \Psi\ \text{(pre-formula-context)} & ::= & \Omega, \mathbf{a} : \mathbf{q} & \text{(extension)} \\
 & | & \diamond & \text{(empty)} \\
 & & & \\
\Xi, \Theta\ \text{(pre-sort-context)} & ::= & \Xi, \mathbf{u} :^{\mathbf{m}} \mathbf{d} & \text{(extension)} \\
 & | & \diamond & \text{(empty)}
\end{array}
$$

Figure 19. Syntax

Proof. UNFINISHED. TODO. Given $X \in St(\Omega)$ and $Y \in St(\Omega.X)$ we define the semantic sort $\Pi(X, Y) \in St(\Omega)$ as

$$\omega \in \Omega \mapsto \left(Y_{(\omega,x)}\right)_{x \in X_\omega}$$

where above, the right-hand-side is a poset-indexed family of posets considered as a poset. Indeed, given $x \le x' \in X_\omega$ we have $(\omega, i(x)) = (\omega, x) \le (\omega, x')$ and hence a monotone injection $i : Y_{(\omega,x)} \longrightarrow Y_{(\omega,x')}$. ◻

G An Order-Theoretic Model (and Redesign) with Multiplicities

I could compose two fibrations where strings are in the bottom layer, formulas in the middle, types on top.

I could restrict Pi sorts so that only strings may occur in the domain. We're still giving coeffects to all indices, though.

G.1 Syntax

G.2 Static semantics

G.3 Normalized validation

$$\frac{}{\Omega \vdash \mathbf{s} : \mathbf{str}} \qquad \frac{}{\Omega, \mathbf{a} : \mathbf{q}, \Omega' \vdash \mathbf{a} : \mathbf{q}} \qquad \frac{\Omega \vdash \mathbf{j} : (\mathbf{a} : \mathbf{q}) \longrightarrow \mathbf{r} \qquad \Omega \vdash \mathbf{k} : \mathbf{q}}{\Omega \vdash \mathbf{App}_{[\mathbf{a}:\mathbf{q}],\mathbf{r}}(\mathbf{j}, \mathbf{k}) : \mathbf{r}[\mathbf{k}/\mathbf{a}]} \qquad \frac{}{\Omega \vdash \mathbf{true} : \mathbf{prop}} \qquad \frac{}{\Omega \vdash \mathbf{str}}$$

$$\frac{\Omega \vdash \mathbf{q} \qquad \Omega, \mathbf{a} : \mathbf{q} \vdash \mathbf{r}}{\Omega \vdash (\mathbf{a} : \mathbf{q}) \Rightarrow \mathbf{r}} \qquad \frac{\Omega \vdash \mathbf{j} : \mathbf{prop}}{\Omega \vdash \mathbf{prf}\ \mathbf{j}} \qquad \frac{}{\diamond \vdash} \qquad \frac{\Omega \vdash \qquad \Omega \vdash \mathbf{p}}{\Omega, \mathbf{a} : \mathbf{p} \vdash}$$

Figure 20. Sorting, sort formation, and sort context formation

$$\begin{array}{lcl} \mathbf{a}, \mathbf{b}, \mathbf{P} & \in & \mathit{IndexVars} \\ \mathbf{s}, \mathbf{t} & \in & \mathit{Strings} \end{array}$$

$$\begin{array}{llll}
\mathbf{i}, \mathbf{j}, \mathbf{k} \text{ (pre-index)} & ::= & \mathbf{s} & \text{(string literal)} \\
 & | & \mathbf{App}_{[\mathbf{a}:\mathbf{q}],\mathbf{p}}(\mathbf{j}, \mathbf{k}) & \text{(index application)} \\
 & | & \mathbf{a} & \text{(index variable)} \\
 & | & \mathbf{true} & \text{(true proposition)} \\
 & & & \\
\mathbf{p}, \mathbf{q}, \mathbf{r} \text{ (pre-sort)} & ::= & \mathbf{str} & \text{(string sort)} \\
 & | & \mathbf{prop} & \text{(proposition sort)} \\
 & | & \mathbf{prf}\ \mathbf{j} & \text{(proof sort)} \\
 & | & (\mathbf{a} : \mathbf{q}) \longrightarrow \mathbf{r} & \text{(function sort)} \\
 & & & \\
\tau, \sigma \text{ (pre-type)} & ::= & \{[\mathbf{a} : \mathbf{str}] : \tau\} & \text{(dictionary)} \\
 & | & \{\mathbf{s}_i : \tau_i{}^{i \in 1..n}\} & \text{(record)} \\
 & | & \{\mathbf{a} : \mathbf{str} \mid \mathbf{i}\} & \text{(string refinement)} \\
 & | & \langle \mathbf{str} \rangle & \text{(string)} \\
 & | & \bigvee \mathbf{a} : \mathbf{q}.\tau & \text{(union over index-to-type abstr.)} \\
 & & & \\
\phi, \psi \text{ (effect scalar)} & ::= & + \text{ (non-empty)} \mid ? \text{ (possibly empty)} & \\
 & & & \\
\Xi \text{ (requirements set)} & ::= & \text{Sets of pairs } (\Omega, \{\mathbf{a} : \mathbf{str} \mid \mathbf{P}\ \mathbf{b_1} \cdots \mathbf{b_n}\}) & \\
 & & & \\
\kappa, \rho \text{ (pre-kind)} & ::= & * & \text{(proper type pre-kind)} \\
 & | & \underline{*} & \text{(populated proper type pre-kind)}
\end{array}$$

$$\begin{array}{llll}
\Omega, \Psi \text{ (pre-sort-context)} & ::= & \Omega, \mathbf{a} : \mathbf{q} & \text{(extension)} \\
 & | & \diamond & \text{(empty)}
\end{array}$$

Figure 21. Normalized Syntax

$$\frac{}{\Omega \vdash \mathbf{s} : \mathbf{str}} \qquad \frac{}{\Omega, \mathbf{a} : \mathbf{q}, \Omega' \vdash \mathbf{a} : \mathbf{q}} \qquad \frac{\Omega \vdash \mathbf{j} : (\mathbf{a} : \mathbf{q}) \longrightarrow \mathbf{r} \qquad \Omega \vdash \mathbf{k} : \mathbf{q}}{\Omega \vdash \mathbf{App}_{[\mathbf{a}:\mathbf{q}],\mathbf{r}}(\mathbf{j}, \mathbf{k}) : \mathbf{r}[\mathbf{k}/\mathbf{a}]} \qquad \frac{}{\Omega \vdash \mathbf{true} : \mathbf{prop}} \qquad \frac{}{\Omega \vdash \mathbf{str}}$$

$$\frac{\Omega \vdash \mathbf{q} \qquad \Omega, \mathbf{a} : \mathbf{q} \vdash \mathbf{r}}{\Omega \vdash (\mathbf{a} : \mathbf{q}) \Rightarrow \mathbf{r}} \qquad \frac{\Omega \vdash \mathbf{j} : \mathbf{prop}}{\Omega \vdash \mathbf{prf}\ \mathbf{j}} \qquad \frac{}{\diamond \vdash} \qquad \frac{\Omega \vdash \qquad \Omega \vdash \mathbf{p}}{\Omega, \mathbf{a} : \mathbf{p} \vdash}$$

Figure 22. Sorting, sort formation, and sort context formation

$$\frac{\Omega, \mathbf{a} : \mathbf{str}@\Xi \vdash \tau : * \,\&\, ? \,\&\, \Upsilon}{\Omega@\Xi \vdash \{[\mathbf{a} : \mathbf{str}] : \tau\} : * \,\&\, ? \,\&\, \Upsilon} \qquad \frac{\Omega@\Xi_i \vdash \tau_i : * \,\&\, \phi_i \,\&\, \Upsilon_i{}^{i\in 1..n}}{\Omega@\bigcup_{i\in 1..n} \Xi_i \vdash \{\mathbf{s}_i : \tau_i{}^{i\in 1..n}\} : * \,\&\, \bigvee_{i\in 1..n} \phi_i \,\&\, \bigoplus_{i\in 1..n} \Upsilon_i}$$

$$\frac{\Omega, \mathbf{a} : \mathbf{str} \vdash \mathbf{P}\ \mathbf{b_1} \cdots \mathbf{b_n} : \mathbf{prop}}{\Omega@\{(\Omega, \{\mathbf{a} : \mathbf{str} \mid \mathbf{P}\ \mathbf{b_1} \cdots \mathbf{b_n}\})\} \vdash \{\mathbf{a} : \mathbf{str} \mid \mathbf{P}\ \mathbf{b_1} \cdots \mathbf{b_i}\} : * \,\&\, + \,\&\, \emptyset}$$

K-UnionAbs

$$\frac{\Omega \vdash \mathbf{q} \qquad \Omega, \mathbf{a} : \mathbf{q} \vdash \tau : * \,\&\, + \,\&\, \Upsilon \qquad a \notin FV(\Gamma)}{\Omega \vdash \bigvee \mathbf{a} : \mathbf{q}.\tau : * \,\&\, ? \,\&\, (\Upsilon \cup \{\Omega, \mathbf{a} : \mathbf{q}\})}$$

$$\frac{}{\Omega \vdash *} \qquad \frac{}{\Omega \vdash \diamond} \qquad \frac{\Omega \vdash \Gamma \qquad \Omega \vdash \kappa}{\Omega \vdash \Gamma, x : \kappa}$$

Figure 23. Kinding and kind formation