Indexed Kind Checking for Hierarchical Database Schemas

Anonymous Author(s)

Abstract

Types are an appropriate tool for describing the structure of hierarchical databases, including MUMPS databases, which are commonly used by hospitals. But two common database features, *foreign keys* and *secondary indexes*, cannot be expressed using standard type systems. To express these features, we need a type language that formally distinguishes between the logical layer of the database, which consists of real-world entities and relations between them, and the physical layer of the database, which describes how the logical layer is recorded. We define such a language and call it λ_{schema}; its key feature is an indexed type language, where the index language describes the logical layer and the type language describes the physical layer. We give λ_{schema} kinding judgments with intrinsic denotational semantics, interpreting each closed, well-kinded type as a set of possible database instances. We then define a translation from a closed, normalized, well-kinded λ_{schema} type into a program that detects discrepancies between a MUMPS database instance and the set denoted by the type.

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, 22 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Central to the study of databases is the distinction between the logical and physical layers. The logical layer describes real-world entities and relations between them. The physical layer describes how elements of the logical layer are represented and stored in the database. Database systems such as DynamoDB, CosmosDB, and MUMPS provide the programmer a hierarchical view of the physical layer. As such, we call these systems *hierarchical database systems*.

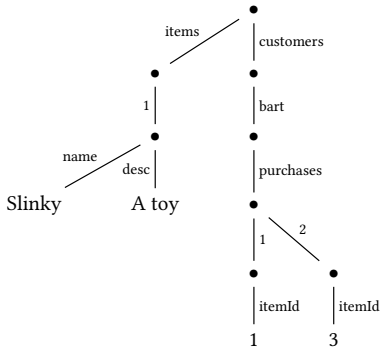

Figure 1. An example database

Because the physical layer is defined in terms of the logical layer, a schema language for a hierarchical database system should describe both logical and physical layers. But to our knowledge, no such language has been defined. We proceed to justify this claim, focusing on the MUMPS database system.

Developed in the 1970s, the MUMPS (Massachusetts General Hospital Utility Multi-Programming System) database system underlies the majority of U.S. News and World Report's top-ranked hospitals today [3] [2]. A MUMPS database instance can be thought of as a partial function from lists of strings to strings; it starts out as everywhere undefined and gradually becomes more defined as *set commands* are executed. For example, after executing the following commands[1]

```
set ["items", "1", "name"]="Slinky"
set ["items", "1", "description"]="A toy"
set ["customers", "kevin", "purchases", "1", "itemId"]="1"
set ["customers", "kevin", "purchases", "2", "itemId"]="3"
```

we obtain the database instance of figure 1.

In the above diagram above, a path $[s_1, \ldots, s_n]$ from the root to a node labeled with string s means that our database instance I is defined as s at $[s_1, \ldots, s_n]$, e.g. we have $I([\text{“items”}, \text{“1”}, \text{“name”}]) = \text{“Slinky”}$. Note all values in a MUMPS database, including integers, are represented as strings.

[1]We use a modified MUMPS syntax to avoid discussing issues orthogonal to the present paper.

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

Figure 1 is inherently physical; because *name* and *desc* are sibling edges that lead to leaves, a programmer understands that they are physically "close together", and hence that he may efficiently access them in sequence. As the physical layer of a database, the above diagram is inadequate, though, because a physical layer should be *about* a logical layer, and the logical layer is only hinted at informally by names such as "items", "customers", "name", and "desc".

Here are some examples of questions that cannot be answered from the above diagram.

1. Is the string "1" supposed to be an identifier for an entity belonging to a set of entities called *items*?
2. Is the string "purchases" supposed to be an identifier for an entity belonging to a set of entities called *barts*?
3. If there is a set of entities called *items*, are all of their identifiers supposed to occur below the edge labelled "items"?
4. Is some potentially strict subset of item identifiers supposed to occur below the edge labelled "items"?
5. Are the leaves labelled "1" and "3" supposed to be item identifiers?

If the answer to points 3 and 5 is affirmative, then we should be able use the values "1" and "3" to find item information rooted at the locations ["*items*", "1"] and ["*items*", "3"]. However, the database depicted in figure 1 is incompatible with these points, because it is not defined at ["*items*", "3"]. If points 3 and 5 were expressed using a formal schema language, we could generate a testing program to detect this inconsistency between our schema and the database. A value encountered at one location in a database that can be used to "look up" additional information at another location in the database is what is known in databases terminology as a *foreign key*. To our knowledge, schema languages for hierarchical database systems, when they exist, lack support for foreign keys.

As an example, consider the Amazon DynamoDB NoSQL Workbench[1]. Its *data modeler* tool allows the developer to describe the database as a homogenous collection of records, selecting one of ten types, such as *string*, *number*, *map*, and *list*, for each of the record fields. It does not provide any means to express foreign keys, but foreign keys occur in the examples provided with Amazon DynamoDB NoSQL Workbench.

In the *Game Player Profiles* example[4], a field called *FriendList* is defined with type *List*. The sample data contains one entry whose *FriendList* field contains a list with two *FriendId*s "player002" and "player003". Presumably, we should be able to use "player002" and "player003" to look up additional data associated with these friends, but while "player001" occurs as a key in the sample data, "player002" and "player003" do not. This mild ambiguity isn't a fundamental problem for instructional data, but it could be a problem for real-world data. Suppose a developer is tasked with writing a routine to remove a player from the database. How are they supposed to know that their routine must also remove any remaining references to the player contained in *FriendList* fields?

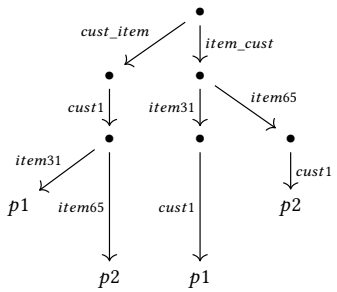

Figure 2. A secondary index, or "reconfiguration" redundantly stores each purchase-customer-item association twice. Some queries can be performed more efficiently on the left subtree, others on the right.

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 compute this list 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 MUMPS, secondary indexes must be created and maintained by the application software rather than the database system. They can become out-of-sync due to programmer errors. A formal schema including specifications of indexes could be compiled to a testing routine to detect these errors. We will not produce such routines in this paper, but it would be a useful goal of future work.

We now consider what a schema system for hierarchical database system should look like. Types are a good starting point, because compound type constructors for records and dictionaries can reflect hierarhcical physical structure. A schema for the database in figure 1 might have the form $\{items : \tau_1, customers : \tau_2\}$, a record type whose fields "items" and "customers" are the immediate outgoing edges of the

database roots, and where τ_1 and τ_2 denote the left and right subtrees respectively. The form of τ_2 might be $\{[str] : \tau_3\}$, a dictionary type mapping some set of strings (the customer names) to elements of τ_3, where τ_3 is the type of customers.

We reach a problem at the leaves of the tree: how do we express that the *itemId* field of each purchase is a foreign key? Naively, we could express the foreign key as the location of subtree where we can "look up" using the key, i.e. $\{itemId : ForeignKey(\text{“}items\text{”})\}$. However, in assuming that the set of all item IDs occur as keys at exactly one location, we are confusing the logical and physical layers of the database. Item IDs could be used as keys in two separate locations, one for customers (containing fields such as *description* and *reviews*) and another intended for analysts (containing fields such as *purchaseCount* and *stock*).

Therefore we should define the set of item IDs as a logical entity, and refer to the logical entity both from the *itemId* field of purchase records and from the dictionaries under the "customer_items" and "analyst_items" locations, as in the following schema:

```
1  ⋁ λ (ItemId : str -> prop).
2    { customerItems: { [x : str] ItemId x > τ1 },
3      analystItems : { [x : str] ItemId x > τ2 },
4      customers: {
5        ...
6        {
7          itemId: { x : str | ItemId x }
8        }
9        ...
10     }
11   }
```

In the above schema, types are *layered over* index terms, which denote logical entities and are written using **blue font**. A dictionary type has the form $\{[\mathbf{a} : \mathbf{str}]\ \mathbf{i} > \tau\}$ and represents the set of dictionaries whose values belong to the type τ and whose keys are exactly those strings **a** such that the propostion **i** is true, where the variable **a** can occur in **i**. We've also introduced a new type constructor: a *refinement* $\{\mathbf{a} : \mathbf{str} \mid \mathbf{i}\}$ where **a** occurs in proposition **i** represents the set of all strings **s** such that $\mathbf{i}[\mathbf{s}/\mathbf{x}]$ is true. Blah blah dependent ML. The type and index languages are explained further in section 2.

We make the following contributions.

- We design an expressive schema language for hierarchical databases, which expresses the logical database layer using a pared down version of the calculus of constructions, and expresses the physical database layer using types indexed by the logical layer. This language can express foreign keys and secondary indexes.
- We define an set of indexed kinding rules for types (schemas), and an intrinsic denotational semantics for kinding judgments, where a kinding judgment for a closed proper type is interpreted as a set of possible database instances.
- We provide a category-theoretic characterization of the dependently-sorted analog of simple products, needed to model our index-to-type abstractions. While this has been done for dependently-sorted logics such as Palmgren's[8], the total category we work with is not thin, which requires us to identify a *transpose distribution* property.
- We define the calculus λ_{MUMPS}, designed to simulate reading and traversing a MUMPS database. We define a translation which maps a λ_{schema} kinding judgment to a testing routine in λ_{MUMPS}. The testing routine, which detects foreign key violations, is proven to respect the intrinsic denotational semantics.

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 Preliminaries

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. A *database instance* (also called an *instance*) is a partial mapping 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 $f|_\sigma(\sigma') \stackrel{def}{=} f(\sigma +\!\!+ \sigma')$, 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 $s_1, \ldots, s_n$.

2.2 Foreign Keys

Figure 3 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*. The index language is a 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, i.e. the "cardType" field is a foreign key.

Lines 12-15 define an index-to-type operator called *Purchase*, which maps a string **cust** and a proof that **cust** satisfies

```
1  ⋁λ(ItemId : str -> prop).
2  ⋁λ(CustId : str -> prop).
3  ⋁λ(PurchId : (x:str) -> prf (CustId x) -> 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] : PurchId cust 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 3. Schema for e-commerce database with foreign keys

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.

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 `PurchId 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

```
1  ⋁λ(ItemId : str -> prop).
2  ⋁λ(CustId : str -> prop).
3  ⋁λ(Purchased :
4        (x : str) -> prf (CustId x) ->
5        (y : str) -> prf (ItemId y) -> prop).
6
7  {
8    custToItem : {
9      [c,i : str] : CustId c,ItemId i,Purchased c i > "*"
10   }
11   itemToCust : {
12     [i,c : str] : CustId c,ItemId i,Purchased c i > "*"
13   }
14 }
```

Figure 4. Reconfigurations in an e-commerce database

- Map ["customers"] to a dictionary that maps every `x` satisfying the `CustId` predicate to a subschema of type `Customer x`.

Reconfigurations

Figure 4 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 (or *pre-indices*) $\mathbf{i}, \mathbf{j}, \mathbf{k}$ consist of string literals, variables, applications, and the proposition constant `true`. We take an intrinsic approach in which only well-formed pre-indices, called *indices* are endowed with meaning. An application is annotated with the domain and codomain of the applied 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 $\mathbf{j}\ \mathbf{k}$ as shorthand for $\mathbf{App}_{[a:q],p}(\mathbf{j}, \mathbf{k})$.

Following [9], index level classifiers are called sorts. But unlike [9], we distinguish between arbitrary *pre-sorts* and well formed pre-sorts, which we call *sorts*. Pre-sorts are written using the metavariables $\mathbf{p}, \mathbf{q}, \mathbf{r}$. We include the pre-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.

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

$$
\begin{array}{lcll}
\mathbf{i}, \mathbf{j}, \mathbf{k} \text{ (pre-index)} & ::= & \mathbf{s} & \text{(string literal)} \\
& | & \mathbf{App}_{[\mathbf{a}:\mathbf{q}],\mathbf{p}}(\mathbf{j}, \mathbf{k}) & \text{(index application)} \\
& | & \mathbf{a} & \text{(index variable)} \\
& | & \mathbf{true} & \text{(true proposition)} \\
\\
\mathbf{p}, \mathbf{q}, \mathbf{r} \text{ (pre-sort)} & ::= & \mathbf{str} & \text{(string sort)} \\
& | & \mathbf{prop} & \text{(proposition sort)} \\
& | & \mathbf{prf}\ \mathbf{j} & \text{(proof sort)} \\
& | & (\mathbf{a} : \mathbf{q}) \to \mathbf{r} & \text{(function sort)} \\
\\
\tau, \sigma \text{ (pre-type)} & ::= & \{[\mathbf{a} : \mathbf{str}] : \mathbf{i}_1, \ldots, \mathbf{i}_n > \tau\} & \text{(dictionary)} \\
& | & \{\mathbf{s}_i : \tau_i{}^{i \in 1..n}\} & \text{(record)} \\
& | & \{\mathbf{a} : \mathbf{str} \mid \mathbf{i}\} & \text{(string refinement)} \\
& | & \lambda \mathbf{a} : \mathbf{q}.\tau & \text{(index-to-type abstr.)} \\
& | & \lambda x : \kappa.\tau & \text{(type-to-type abstr.)} \\
& | & \bigvee \tau & \text{(union)} \\
& | & \tau\ \sigma & \text{(type app.)} \\
& | & \tau\ [\ \mathbf{j}\ ] & \text{(index-to-type app.)} \\
\\
\kappa, \rho \text{ (pre-kind)} & ::= & * & \text{(proper type)} \\
& | & \kappa \to \rho & \text{(type-to-type op)} \\
& | & \forall \mathbf{a} : \mathbf{q}.\ \kappa & \text{(index-to-type op)} \\
\\
\Omega, \Psi \text{ (pre-sort-context)} & ::= & \Omega, \mathbf{a} : \mathbf{q} & \text{(extension)} \\
& | & \diamond & \text{(empty)} \\
\\
\Gamma \text{ (pre-kind-context)} & ::= & \Gamma, x : \kappa & \text{(extension)} \\
& | & \diamond & \text{(empty)}
\end{array}
$$

Figure 5. Syntax

Our pre-types τ, σ are also called *types* or *schemas* when well formed. A well formed pre-type $\{[\mathbf{a} : \mathbf{str}] : \mathbf{i}_1, \ldots, \mathbf{i}_n > \tau\}$ is a dictionary type. The type τ and prop-sorted indices $\mathbf{i}_1, \ldots, \mathbf{i}_n$ depend on the index variable $\mathbf{a}$. Furthermore, each $\mathbf{i}_k$ may depend implicitly on proofs of previous indices

$$\mathbf{prf}\ \mathbf{i}_1, \ldots, \mathbf{prf}\ \mathbf{i}_{k-1}$$

and τ may depend implicitly on all proofs $\mathbf{prf}\ \mathbf{i}_1, \ldots, \mathbf{prf}\ \mathbf{i}_n$. The type $\{[\mathbf{a} : \mathbf{str}] : \mathbf{i}_1, \ldots, \mathbf{i}_n > \tau\}$ denotes the set of database instances which map each string $\mathbf{s}$ such that the interpretations of $\mathbf{i}_1[\mathbf{s}/\mathbf{a}], \ldots, \mathbf{i}_n[\mathbf{s}/\mathbf{a}]$ are true to a database instance in the set denoted by $\tau[\mathbf{s}/\mathbf{a}]$. Record types, type-type applications, and type-type abstractions in our pre-type syntax are standard. A string refinement type of the form

$$
\begin{array}{lcl}
\theta & ::= & \{[\mathbf{a}] : \mathbf{i}_1, \ldots, \mathbf{i}_n > \theta\} \\
& | & \{s_i : \theta_i{}^{i \in 1..n}\} \\
& | & \lambda \mathbf{a} : \mathbf{q}.\theta \\
& | & \lambda x : \kappa.\theta \\
& | & \bigvee \theta
\end{array}
$$

Figure 6. Type values

$$\textsc{Red-Ind-App}\quad \frac{}{(\lambda(\mathbf{a} : \mathbf{q}).\tau)\ [\ \mathbf{j}\ ] \hookrightarrow \tau[\mathbf{j}/\mathbf{a}]} \qquad \textsc{Red-Ty-App}\quad \frac{}{(\lambda(x : \kappa).\tau)\ \sigma \hookrightarrow \tau[\sigma/x]}$$

Figure 7. Top-level Reduction Rules

$$
\begin{array}{lcl}
E & ::= & [\,] \\
& | & \{[\mathbf{a}] : \mathbf{i}_1, \ldots, \mathbf{i}_n > E\} \\
& | & \{s_i : \theta_i{}^{i \in 1..(j-1)},\ s_j : E,\ s_i : \tau_i{}^{i \in (j+1)..n}\} \\
& | & \lambda \mathbf{a} : \mathbf{q}.E \\
& | & \lambda x : \kappa.E \\
& | & \bigvee E \\
& | & E\ \tau \\
& | & \theta\ E \\
& | & E\ [\ \mathbf{i}\ ]
\end{array}
$$

Figure 8. Evaluation Contexts

$\{\mathbf{a} : \mathbf{str} \mid \mathbf{i}\}$ represents the set of all strings $\mathbf{s}$ such that the interperetation of $\mathbf{i}[\mathbf{s}/\mathbf{a}]$ is true.

Finally, our pre-type syntax includes index-type abstractions $\lambda(\mathbf{a} : \mathbf{q}).\tau$, index-type applications $\tau\ [\ \mathbf{j}\ ]$, and unions $\bigvee \tau$. For a type $\bigvee \tau$ to be well-kinded, the type τ must be 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.

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 6 gives the syntax of type values. The binary relation $\hookrightarrow$ on types is defined in figure 7; it represents the substitution of a type or index into an abstraction. An evaluation context E, defined in Figure 8, is a type containing a single hole $[\,]$. We write $E[\tau]$ for the substitution of τ for the hole of E. We write $\tau \to \tau'$ 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 $\to^*$ for the reflexive transitive closure of $\to$. We write $\tau \not\to$

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

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

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

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

Figure 9. Sorting, sort formation, and sort context formation

if there exists no τ' with $\tau \to \tau'$. We write $\tau \Downarrow \tau'$ if $\tau \to^* \tau'$ and $\tau' \not\to$.

5 Static semantics

5.1 Index-level static semantics

Index-level judgment rules are shown in Figure 24. 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 25. 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 [5]. 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.

6.1.1 Categories with Families.

A Category-with-Families (CwF) consists of

$$\frac{\Omega, \mathbf{a} : \mathbf{str}, \mathbf{a}_1 : \mathbf{prf}\ \mathbf{i}_1, \ldots, \mathbf{a}_n : \mathbf{prf}\ \mathbf{i}_n \vdash \qquad \Omega, \mathbf{a} : \mathbf{str}, \mathbf{a}_1 : \mathbf{prf}\ \mathbf{i}_1, \ldots, \mathbf{a}_n : \mathbf{prf}\ \mathbf{i}_n \mid \Gamma \vdash \tau : *}{\Omega \mid \Gamma \vdash \{[\mathbf{a}] : \mathbf{i}_1, \ldots \mathbf{i}_n > \tau\} : *}$$

$$\frac{\Omega \mid \Gamma \vdash \tau_i : *^{\,i \in 1..n}}{\Omega \mid \Gamma \vdash \{\mathbf{s_i} : \tau_i^{\,i \in 1..n}\} : *} \qquad \frac{\Omega, \mathbf{a} : \mathbf{str} \vdash \mathbf{i} : \mathbf{prop}}{\Omega \mid \Gamma \vdash \{\mathbf{a} : \mathbf{str} \mid \mathbf{i}\} : *}$$

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 10. Kinding and kind formation

- 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.Semantic indices are written with the symbols M and N.
- 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 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 for each $M \in In(\Omega.X, Y)$ there is a term $\lambda_{X,Y}(M) \in In(\Omega, \Pi(X, Y))$ and for each $M \in In(\Omega, \Pi(X, Y))$ and $N \in In(\Omega, X)$ there is an index $App_{X,Y}(M, N) \in In(\Omega, X\{\overline{N}\})$ such that the following equations hold for each $f \in \Psi \to \Omega$.

$$\begin{array}{lcl} App_{X,Y}(\lambda_{X,Y}(M), N) & = & M\{\overline{N}\} \\ \Pi(X, Y)\{f\} & = & \Pi(X\{f\}, Y\{\mathfrak{q}(f, X)\}) \in St(\Psi) \\ \lambda_{X,Y}(M)\{f\} & = & \lambda_{X\{f\},Y\{\mathfrak{q}(f,X)\}}(M\{\mathfrak{q}(f, \sigma)\}) \\ App_{X,Y}(M, N)\{f\} & = & App_{X\{f\},Y\{\mathfrak{q}(f,X)\}}(M\{f\}, N\{f\}) \end{array}$$

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 $f : \Omega \to \Psi, X \in St(\Psi)$, and $M \in In(\Psi, X)$ we define $X\{f\}_\omega \stackrel{def}{=} X_{f(\omega)}$ and $M\{f\}_\omega \stackrel{def}{=} M_{f(\omega)}$. 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, without abstractions but with with the unnotable addition of strings. Its interpretation appears, for example, in [6], and is displayed in Figure 11 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* $In([\![\Omega]\!], [\![\Omega; \mathbf{q}]\!])$

6.2 Type language

Our kinding and kind formation judgments are interpreted in terms of the fibration $Fam(\mathbf{Sets})$. We assume basic knowledge of fibrations, which can be learned from Jacobs [7]. A kind-in-sorting-context $\Omega \vdash \kappa$ is interpreted as an object of $Fam(\mathbf{Sets})_{[\![\Omega]\!]}$, 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 $Fam(\mathbf{Sets})_{[\![\Omega]\!]}$. A kinding judgment $\Omega \mid \Gamma \vdash \tau : \kappa$ is interpreted as a arrow of $Fam(\mathbf{Sets})_{[\![\Omega]\!]}$ from $[\![\Omega \vdash \Gamma]\!]$ to $[\![\Omega \vdash \kappa]\!]$.

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

Figure 11. Interpretation of index language

$$\begin{array}{l} [\![\Omega \vdash *]\!]_\omega \stackrel{def}{=} \mathcal{PP}(Inst) \\ [\![\Omega \vdash \forall \mathbf{a} : \mathbf{q}.\kappa]\!] \stackrel{def}{=} \Pi_{\Omega,\mathbf{q}}[\![\Omega, \mathbf{a} : \mathbf{q} \vdash \kappa]\!] \\ [\![\Omega \vdash \kappa \to \rho]\!] \stackrel{def}{=} [\![\Omega \vdash \kappa]\!] \Rightarrow [\![\Omega \vdash \rho]\!] \end{array}$$

Figure 12. Interpretation of kinding judgments

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 $u : \Omega \to \Psi$ and $v : \Psi \to \Upsilon$ 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:

$$I \in [\![\{[\mathbf{a}] : \mathbf{i_1}, \ldots, \mathbf{i_n} > \tau\}]\!]_\omega\gamma \overset{def}{\Leftrightarrow}$$
$$I(\epsilon) = \text{“}*\text{”} \wedge$$
$$\forall s \in Strings.\ I|_{[s]} \in [\![\tau]\!]_{\omega +\!\!+ (s,*,\ldots,*)}\gamma^+ \text{ if } \omega +\!\!+ (s,*,\ldots,*) \in [\![\Omega, \mathbf{a} : \mathbf{str}, \mathbf{a_1} : \mathbf{prf}\ \mathbf{i_1}, \ldots \mathbf{i_n} : \mathbf{prf}\ \mathbf{i_n}]\!], \text{and } I|_{[s]} = \emptyset \text{ otherwise}$$

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

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

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

Figure 13. Semantics for “non-operational” kinding rules

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

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

Figure 14. Semantics for abstraction and application

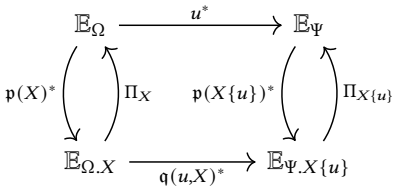

Figure 15. Some components of Beck-Chevalley for dependent simple products

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

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 $\Pi_X : Fam(\mathbf{Sets})_{\Omega.X} \longrightarrow Fam(\mathbf{Sets})_\Omega$ as:

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

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

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

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

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

to

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

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.

7 Schema Tester 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

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

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

Figure 16. Syntax for MiniMUMPS

one 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 schema tester 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, i.e. we must have

  $$\tau = \bigvee \lambda(\mathbf{a_1} : \mathbf{p_1} \to \mathbf{q_1}). \ldots \bigvee \lambda(\mathbf{a_n} : \mathbf{p_n} \to \mathbf{q_n}).\sigma$$

  for some $n \in \mathbb{N}$, where no abstraction of the form

  $$\lambda(\mathbf{a} : \mathbf{p} \to \mathbf{q}).\sigma'$$

  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 $\mathbf{p} \to \mathbf{str}$.

From the first constraint above we infer that any function sort $\mathbf{p} \to \mathbf{q}$ inserted into our sort context Ω 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 Ω encountered while type-checking τ as $\Omega_{str}, \Omega_{fn}, \Omega_{prf}$, where Ω_{str} contains only bindings of the form $\mathbf{a} : \mathbf{str}$, Ω_{fn} includes only bindings of the form $\mathbf{a} : \mathbf{p} \to \mathbf{q}$, and Ω_{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 16. Its operational semantics is given in figure 17.

A *MiniMUMPS* reduction judgment has the form

$$I \mid e \to e'$$

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 $I \mid - \to -$.

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

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

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

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

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

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

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

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

Figure 17. Reduction relation for MiniMUMPS

7.3 Context testers

We say that a MiniMUMPS program e is a "*context tester* 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 Schema testers

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 "*schema tester* 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

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

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 *validation* procedure that determines whether $I \in [\![\diamond \mid \diamond \vdash \tau :: *]\!]_*(*)$, but instead of designing such a procedure, we pursue a less ambitious goal: generate a *testing* procedure which examines I and either determines that $I \notin [\![\diamond \mid \diamond \vdash \tau :: *]\!]_*(*)$ or provides no information about whether $I \in [\![\diamond \mid \diamond \vdash \tau :: *]\!]_*(*)$. Formally, it generates a schema tester 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 "schema tester 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 schema tester is a testing tool rather than a verification tool: it can confirm the presence of errors, but not their absence.

7.5 Schema Tester 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. We say that Δ is a *location-context under* $\Omega \vdash$, written $\Omega \vdash \Delta$, if Δ is a list of zero or more indices $\mathbf{i}$ such that $\Omega \vdash \mathbf{i} : \mathbf{str}$. 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 schema tester needs a way to translate a valuation $\omega \in [\![\Omega \vdash]\!]$ into $[\![\Omega \vdash \Delta]\!]_\omega$. A *location instantiator* for $\Omega \vdash \Delta$ with $\Omega = \Omega_{str}, \Omega_{fn}, \Omega_{prop}$ is a MiniMUMPS expression e such that for all $[z_1, \ldots, z_m] \in [\![\Omega]\!]$ and $n \leq m$ with $[s_1, \ldots, s_n] \in [\![\Omega_{str} \vdash]\!]$ and $s_1 = z_1, \ldots, s_n = z_n$ we have $e\ s_1\ \cdots\ s_n \Downarrow [\![\Omega \vdash \Delta]\!]_{[z_1,\ldots,z_m]}$. For each derivable judgment $\Omega \vdash \Delta$, the rules of figure 19 produce a judgment of the form $\Omega \vdash \Delta \Rightarrow e$, where e is a location instantiator for $\Omega \vdash \Delta$.

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

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

If kind checking 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 schema tester 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 schema tester for θ, which is much easier due to θ's simpler form.

$$\begin{array}{rcll} \Delta \text{ (pre-location-context)} & ::= & \Delta, \mathbf{i} & \text{(extension)} \\ & \mid & \diamond & \text{(empty)} \\ \Xi \text{ (pre-sort-context set)} & \stackrel{def}{=} & \mathcal{P}_{fin}(\Omega) & \end{array}$$

Figure 18. Additional Syntax For Schema Tester Generation

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

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

where s is the string underlying literal $\mathbf{s}$

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

Figure 19. Location instantiator generation

7.5.3 Rules. Schema tester generation is performed following the syntactic rules of figure 20. These rules reuse much of the syntax of kind checking, but introduce new syntax defined in figure 18.

A schema tester generation judgment has the form

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

where $\diamond \mid \diamond \vdash \tau : *$. 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* $\Omega @ \Xi_0 \vdash_\tau \theta : \Delta\ \&\ \Xi_1 \Rightarrow (e, \xi)$ *then the following facts hold:*

1. $\Omega \mid \diamond \vdash \theta : *$ *is derivable*
2. *If* υ *is a mapping that takes each sort context* $\Omega_\mathbf{i} \in \Xi_0 = \{\Omega_1, \ldots, \Omega_n\}$ *to a context tester for* $\Omega_\mathbf{i} \vdash$ *with respect to* τ*, then*
$$e[\upsilon(\Omega_1)/x_{\Omega_1}] \cdots [\upsilon(\Omega_n)/x_{\Omega_n}]$$
is a schema tester for $\Omega \mid \diamond \vdash \theta : *$ *at* Δ *with respect to* τ.
3. *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* $\Omega_\mathbf{i} \in \Xi_1 = \{\Omega_1, \ldots, \Omega_m\}$, *we have that* $\Omega_\mathbf{i} \vdash$ *and* ξ *maps* $\Omega_\mathbf{i}$ *to a context tester for* $\Omega_\mathbf{i} \vdash$ *with respect to* $\diamond \mid \diamond \vdash \tau : *$.

8 Future Work

8.1 Multiplicities

Example goes here.

GenDict

$$\frac{\Omega, \mathbf{a} : \mathbf{str}, \mathbf{a}_1 : \mathbf{prf}\ \mathbf{i}_1, \ldots, \mathbf{a}_{\mathbf{n}_{prf}} : \mathbf{prf}\ \mathbf{i}_{\mathbf{n}_{prf}} @ \Xi_0 \vdash_\tau \theta : \Delta, \mathbf{a} \ \&\ \Xi_1 \Rightarrow (e, \xi) \qquad \Omega \vdash \Delta \Rightarrow d_0 \qquad \Omega, \mathbf{a} : \mathbf{str} \vdash \Delta, \mathbf{a} \Rightarrow d_1}{\Omega @ \Xi_0 \vdash_\tau \{[\mathbf{a}] : \mathbf{i}_1, \ldots, \mathbf{i}_{\mathbf{n}_{prf}} > \theta\} : \Delta \ \&\ (\Xi_1 \cup \{\Omega, \mathbf{a} : \mathbf{str}, \mathbf{a}_1 : \mathbf{prf}\ \mathbf{i}_1, \ldots, \mathbf{a}_{\mathbf{n}_{prf}} : \mathbf{prf}\ \mathbf{i}_{\mathbf{n}_{prf}}\}) \Rightarrow (e', \xi')}$$

where $e' \stackrel{def}{=} \lambda x_1 \ldots x_{|\Omega_{str}|}.\mathbf{first10}\ d_0\ x_1\ \cdots\ x_{|\Omega_{str}|}\ \mathbf{with}\ x\ \mathbf{in}\ e\ x_1 \cdots x_{|\Omega_{str}|}\ x$

and $\xi' \stackrel{def}{=} \xi \oplus \{\Omega, \mathbf{a} : \mathbf{str}, \mathbf{a}_1 : \mathbf{prf}\ \mathbf{i}_1, \ldots, \mathbf{a}_{\mathbf{n}_{prf}} : \mathbf{prf}\ \mathbf{i}_{\mathbf{n}_{prf}} \mapsto \lambda x_1, \ldots, x_{|\Omega_{str}|+1}.\mathbf{isdefined}\ (d_1\ x_1 \cdots x_{|\Omega_{str}|+1})\}$

GenRecord

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

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

GenRefinement

$$\frac{\Omega, \mathbf{a} : \mathbf{str} \vdash \mathbf{P}\ \mathbf{b}_1 \cdots \mathbf{b}_\mathbf{n} : \mathbf{prop} \qquad \Omega \vdash \Delta \Rightarrow d}{\Omega @ \{\Omega'\} \vdash_\tau \{\mathbf{a} : \mathbf{str} \mid \mathbf{P}\ \mathbf{b}_1 \cdots \mathbf{b}_\mathbf{n}\} : \Delta \ \&\ \emptyset \Rightarrow (e, \emptyset)}$$

where $\Omega' \stackrel{def}{=} \Omega, \mathbf{a} : \mathbf{str}, \mathbf{b} : \mathbf{prf}\ (\mathbf{P}\ \mathbf{b}_1 \cdots \mathbf{b}_\mathbf{n})$

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

GenUnionPred

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

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

Figure 20. Schema Tester Generation

8.2 Full Validation

- All predicates must have finite support.

References

[1] [n.d.]. Amazon DynamoDB NoSQL Workbench. https://aws.amazon.com/dynamodb/nosql-workbench/. Accessed: 2024-06-30.

[2] [n.d.]. Epic Health System Community. http://www.epic.com/community. Accessed: 2024-06-27.

[3] [n.d.]. From Healthcare to Mapping the Milky Way: 5 Things You Didn't Know About Epic's Tech. http://www.epic.com/epic/post/healthcare-mapping-milky-way-5-things-didnt-know-epics-tech/. Accessed: 2024-06-27.

[4] [n.d.]. Gaming Player Profiles. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/data-modeling-schema-gaming-profile.html. Accessed: 2024-06-30.

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

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

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

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

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

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:

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

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

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

and a arrow f where

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

We then have

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

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

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

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

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

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

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

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

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

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

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* $\Omega, \mathbf{a} : \mathbf{p}, \Psi \vdash \mathbf{q}$ *and* $\Omega \vdash \mathbf{j} : \mathbf{p}$ *then* $\Omega, \Psi[\mathbf{j}/\mathbf{a}] \vdash \mathbf{q}[\mathbf{j}/\mathbf{a}]$.

Proof. TODO □

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

$$\mathbf{P}(\Omega; \mathbf{q}; \diamond) \stackrel{def}{=} \mathfrak{p}(\llbracket \Omega; \mathbf{p} \rrbracket)$$
$$\mathbf{P}(\Omega; \mathbf{q}; \Psi, \mathbf{a} : \mathbf{r}) \stackrel{def}{=} \mathfrak{q}(\mathbf{P}(\Omega; \mathbf{q}; \Psi), \llbracket \Omega, \Psi; \mathbf{r} \rrbracket)$$

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

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

$$\mathbf{T}(\Omega; \mathbf{q}; \diamond; \mathbf{i}) \stackrel{def}{=} \overline{\llbracket \Omega; \mathbf{i} \rrbracket}$$
$$\mathbf{T}(\Omega; \mathbf{q}; \Psi, \mathbf{a} : \mathbf{r}; \mathbf{i}) \stackrel{def}{=} \mathfrak{q}(\mathbf{T}(\Omega; \mathbf{q}; \Psi; \mathbf{i}), \llbracket \Omega, \mathbf{b} : \mathbf{q}, \Psi; \mathbf{r} \rrbracket) \qquad \mathbf{b} \text{ fresh}$$

The idea here is that $\mathbf{T}(\Omega; \mathbf{q}; \Psi; \mathbf{i})$ is a morphism from $\llbracket \Omega, \Psi[\mathbf{i}/\mathbf{b}] \rrbracket$ to $\llbracket \Omega, \mathbf{b} : \mathbf{q}, \Psi \rrbracket$ yielding $\llbracket \Omega; \mathbf{i} \rrbracket$ at the $\mathbf{b} : \mathbf{q}$ 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,* $\mathbf{p}, \mathbf{q}$ *pre-sorts,* $\mathbf{i}$ *a pre-index, and* $\mathbf{b}$ *a fresh variable. Let* $\mathbf{A} \in \{\mathbf{p}, \mathbf{i}\}$. *The expression* $\mathbf{P}(\Omega; \mathbf{p}; \Psi)$ *is defined iff* $\llbracket \Omega, \mathbf{p} : \mathbf{q}, \Psi \rrbracket$ *and* $\llbracket \Omega, \Psi \rrbracket$ *are defined and in this case is a morphism from the former to the latter. If* $\llbracket \Omega, \Psi; \mathbf{A} \rrbracket$ *is defined then*

$$\llbracket \Omega, \mathbf{b} : \mathbf{p}, \Psi; \mathbf{A} \rrbracket \simeq \llbracket \Omega, \Psi; \mathbf{A} \rrbracket \{\mathbf{P}(\Omega; \mathbf{p}; \Psi)\}$$

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

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

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

Proof. TODO □

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

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

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

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

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

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

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{aligned}
& \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}]\!]\{\mathbf{T}(\Omega; \mathbf{a}:\mathbf{p}; \Psi; \mathbf{j})\}} \mathfrak{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}]]\!]} \mathfrak{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{aligned}$$

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

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

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

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

We also have

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

Other cases:
TODO.

□

C Validator Generation

Theorem C.1. *Suppose* $\Omega@\Xi_0 \vdash_\tau \theta : \Delta \;\&\; \Xi_1 \Rightarrow (e, \xi)$. *If* v *is a mapping that takes each sort context* $\Omega_\mathbf{i} \in \Xi_0 = \{\Omega_1, \ldots, \Omega_\mathbf{n}\}$ *to a context tester for* $\Omega_\mathbf{i} \vdash$, *then* $e[v(\Omega_1)/x_{\Omega_1}] \cdots [v(\Omega_\mathbf{n})/x_{\Omega_\mathbf{n}}]$ *is a schema tester for* $\Omega \mid \diamond \vdash \theta : * \text{ at } \Delta$ *with respect to* $\diamond \mid \diamond \vdash \tau : *$.

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

Case GenDict**:**
By the IH, if v is a mapping that takes each sort context $\Omega_\mathbf{i} \in \Xi_0 = \{\Omega_1, \ldots, \Omega_\mathbf{n}\}$ to a context tester for $\Omega \vdash_\mathbf{i}$ then $e[v(\Omega_1)/x_{\Omega_1}] \cdots [v(\Omega_\mathbf{n})/x_{\Omega_\mathbf{n}}]$ is a schema tester for $\Omega, \mathbf{a} : \mathbf{str}, \mathbf{a}_1 : \mathbf{prf}\ \mathbf{i}_1, \ldots, \mathbf{a}_{\mathbf{n}_{prf}} : \mathbf{prf}\ \mathbf{i}_{\mathbf{n}_{prf}} \mid \diamond \vdash \theta : *$ at $\Delta, \mathbf{a}$ 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 $\Omega_\mathbf{i} \in \Xi_0 = \{\Omega_1, \ldots, \Omega_\mathbf{n}\}$ to a context tester for $\Omega \vdash_\mathbf{i}$ then

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

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

To this end, let $[s_1, \ldots, s_m] \in [\![\Omega_{str} \vdash]\!]$ and $[t_1, \ldots, t_k] \stackrel{def}{=} [\![\Omega_{str} \vdash \Delta]\!]_{[s_1,\ldots,s_m]}$. 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 $I|_{[t_1,\ldots,t_k]} \in [\![\Omega \mid \diamond \vdash \{[\mathbf{a} : \mathbf{str}] : \theta\} : *]\!]_{[z_1,\ldots,z_{|\Omega|}]}(*)$. Then, for each string t' such that $I|_{[t_1,\ldots,t_k,t']}$ is active, we have $[\![\Omega_{str}, \mathbf{a} : \mathbf{str} \vdash \Delta, \mathbf{a}]\!]_{[s_1,\ldots,s_m,t']} = [t_1, \ldots, t_k, t']$. The interpretation of dictionary types in figure 13 then gives us

$$I|_{[t_1,\ldots,t_k,t']} \in [\![\Omega, \mathbf{a} : \mathbf{str}, \mathbf{a}_1 : \mathbf{prf}\ \mathbf{i}_1, \ldots, \mathbf{a}_{\mathbf{n}_{prf}} : \mathbf{prf}\ \mathbf{i}_{\mathbf{n}_{prf}} \mid \diamond \vdash \theta : *]\!]_{[z_1,\ldots,z_m,t',z_{m+1},\ldots z_{|\Omega|,*,\ldots,*}]}(*)$$

We apply our IH to obtain

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

Now, we have $I \mid e'[v(\Omega_\mathbf{i})/x_{\Omega_\mathbf{i}}]^{i \in 1..n}\ s_1 \cdots s_m \longrightarrow^* (e[v(\Omega_\mathbf{i})/x_{\Omega_\mathbf{i}}]^{i \in 1..n}\ s_1 \cdots s_m\ t'_1) \wedge \ldots \wedge (e[v(\Omega_\mathbf{i})/x_{\Omega_\mathbf{i}}]^{i \in 1..n}\ s_1 \cdots s_m\ t'_l)$ 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:

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

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

Case GenRecord**:**

By the IH, for each $i \in 1..n$, if v_i is a mapping that takes each sort context $\Omega_{i,j} \in \Xi_{0,i} = \{\Omega_{i,1}, \dots, \Omega_{i,n_i}\}$ to a context tester 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 schema tester for $\Omega \mid \diamond \vdash \theta_i : *$ at $\Delta, \mathbf{s_i}$ 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 $\Omega_{i,j} \in \bigcup_{i\in 1..n} \Xi_{0,i} = \{\Omega_{1,1}, \dots, \Omega_{1,n_1}, \dots, \Omega_{n,1}, \dots, \Omega_{n,n_n}\}$ to a context tester for $\Omega_{i,j} \vdash$ then

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

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

To this end, let $[r_1, \dots, r_m] \in [\![\Omega_{str} \vdash]\!]$ and $[t_1, \dots, t_k] = [\![\Omega \vdash \Delta]\!]_{[r_1,\dots,r_m]}$. Suppose there exists $[z_1, \dots, z_{|\Omega|}] \in [\![\Omega \vdash]\!]$ such that $z_1 = r_1, \dots, z_m = r_m$ and a database instance I such that $I|_{[t_1,\dots,t_k]} \in [\![\Omega \mid \diamond \vdash \{\mathbf{s_i} : \theta_i{}^{i\in 1..n}\}]\!]_{[t_1,\dots,t_k]}(*)$. Then the interpretation of record types in figure 13 gives us $I|_{[t_1,\dots,t_k,s_i]} \in [\![\Omega \mid \diamond \vdash \theta_i : *]\!]_{[z_1,\dots,z_{|\Omega|}]}(*)$ 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

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

Case GenRefinement**:**

We need to prove that if v is a mapping that takes $\Omega' = \Omega, \mathbf{a : str, b : prf\ (P\ b_1 \cdots b_n)}$ to a context tester for Ω' with respect to $\diamond \mid \diamond \vdash \tau : *$ then

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

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

Let $[s_1, \dots, s_m] \in [\![\Omega_{str} \vdash]\!]$ and $[t_1, \dots, t_k] = [\![\Omega \vdash \Delta]\!]_{[s_1,\dots,s_m]}$. Suppose there exists $[z_1, \dots, z_{|\Omega|}] \in [\![\vdash \Omega]\!]$ such that $s_1 = z_1, \dots, s_m = z_m$ and a database instance I such that $I|_{[t_1,\dots,t_k]} \in [\![\Omega \mid \diamond \vdash \{\mathbf{a : str \mid P\ b_1 \cdots b_n}\} : *]\!]_{[z_1,\dots,z_{|\Omega|}]}(*)$. Expanding the interpretation of refinement types in figure 13 gives $I|_{[t_1,\dots,t_k]}([]) \downarrow$ and $[\![\Omega, \mathbf{a : str \vdash P\ b_1 \cdots b_n}]\!]_{[z_1,\dots,z_{|\Omega|},I|_{[t_1,\dots,t_k]}([])]} = \mathit{true}$. By the fourth line of figure 11 we have $[\![\Omega, \mathbf{a : str \vdash prf\ (P\ b_1 \cdots b_n)}]\!]_{[z_1,\dots,z_{|\Omega|},I|_{[t_1,\dots,t_k]}([])]} = \{*\}$. By the second line of figure 11 we have $[\![\Omega, \mathbf{a : str, prf\ (P\ b_1 \cdots b_n)}]\!] = [\![\Omega, \mathbf{a : str}]\!].[\![\Omega, \mathbf{a : str \vdash prf\ (P\ b_1 \cdots b_n)}]\!]$. Hence, the set $[\![\Omega']\!] = [\![\Omega, \mathbf{a : str}]\!].[\![\Omega, \mathbf{a : str \vdash prf\ (P\ b_1 \cdots b_n)}]\!]$ contains the element $[z_1, \dots, z_{|\Omega|}, I|_{[t_1,\dots,t_k]}([]), *]$.

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

□

D Garbage After this Point

E A Non-Categorical Attempt

Our judgments are of the form

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

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 $\mathbf{\Theta} \mid \Xi \in S$ is a location context $\mathbf{\Theta}$ followed by a formula set Ξ. It tells us that we can decide Ξ by checking if $\mathbf{\Theta}$ is populated. A sink $\Xi \mid \mathbf{\Theta}$ is also a location context and a formula set. It conveys the requirement that we can decide the formula set $\mathbf{\Theta}$ 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 $\mathbf{\Omega_{Pred}} \vdash \Xi$ and $\mathbf{\Omega_{Pred}}, \mathbf{\Omega_{Loc}}, \mathbf{\Omega_{Prf}} \vdash \tau :: *$. For $\mathbf{\Omega'_{Loc}} \mid \mathbf{\Omega'_{Prf}} \in \Xi$ we define the set $[\mathbf{\Omega'_{Loc}} \mid \mathbf{\Omega'_{Prf}}]_\tau$ of *τ-deciders* for $\mathbf{\Omega'_{Loc}} \mid \mathbf{\Omega'_{Prf}}$ as the set of all expressions e such that for all $I \in Inst$ we have that $I \mid e \Downarrow \mathbf{true}$ if and only if there exist $\omega_{\mathbf{Pred}} \in [\![\mathbf{\Omega_{Pred}}]\!], \omega \in [\![\mathbf{\Omega_{Pred}} \mid \mathbf{\Omega_{Loc}} \mid \mathbf{\Omega_{Pref}}]\!], \omega' \in [\![\mathbf{\Omega_{Pred}} \mid \mathbf{\Omega'_{Loc}} \mid \mathbf{\Omega'_{Prf}}]\!]$ such that ω and ω' extend $\omega_{\mathbf{Pred}}$ and $I \in [\![\tau]\!]_\omega$.

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

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

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

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

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* $\leq$ *known*. A predicate $\mathbf{P : str \to prop}$ 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

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

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

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

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 $\omega \le \omega'$ we have

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

(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 $\bot$ (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 $\Pi(X, Y)$.*

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 $i : Y_{(\omega,x)} \to Y_{(\omega,x')}$.

□

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

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

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

Figure 21. Syntax

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

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

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

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

$\mathbf{i}, \mathbf{j}, \mathbf{k}$ (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)
	|	$(\mathbf{a} : \mathbf{q}) \longrightarrow \mathbf{r}$	(function sort)
τ, σ (pre-type)	::=	$\{[\mathbf{a} : \mathbf{str}] : \mathbf{i}_1, \ldots, \mathbf{i}_n \tau\}$	(dictionary)
	|	$\{\mathbf{s}_i : \tau_i^{\,i \in 1..n}\}$	(record)
	|	$\{\mathbf{a} : \mathbf{str} \mid \mathbf{i}\}$	(string refinement)
	|	$\langle \mathbf{str} \rangle$	(string)
	|	$\bigvee \mathbf{a} : \mathbf{q}.\tau$	(union over index-to-type abstr.)
ϕ, ψ (effect scalar)	::=	+ (non-empty) | ? (possibly empty)	
Ξ (requirements set)	::=	Sets of pairs $(\Omega, \{\mathbf{a} : \mathbf{str} \mid \mathbf{P}\ \mathbf{b}_1 \cdots \mathbf{b}_n\})$	
κ, ρ (pre-kind)	::=	$*$	(proper type pre-kind)
	|	$\underline{*}$	(populated proper type pre-kind)
Ω, Ψ (pre-sort-context)	::=	$\Omega, \mathbf{a} : \mathbf{q}$	(extension)
	|	$\diamond$	(empty)

Figure 23. Normalized Syntax

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

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

Figure 24. Sorting, sort formation, and sort context formation

$$\frac{\Omega, \mathbf{a} : \mathbf{str}@\Xi \vdash \tau : * \;\&\; ? \;\&\; \Upsilon}{\Omega@\Xi \vdash \{[\mathbf{a} : \mathbf{str}] : \tau\} : * \;\&\; ? \;\&\; \Upsilon}$$

$$\frac{\Omega@\Xi_i \vdash \tau_i : * \;\&\; \phi_i \;\&\; \Upsilon_i^{\,i\in 1..n}}{\Omega@ \bigcup_{i\in 1..n} \Xi_i \vdash \{\mathbf{s}_i : \tau_i^{\,i\in 1..n}\} : * \;\&\; \bigvee_{i\in 1..n} \phi_i \;\&\; \bigoplus_{i\in 1..n} \Upsilon_i}$$

$$\frac{\Omega, \mathbf{a} : \mathbf{str} \vdash \mathbf{P}\ \mathbf{b}_1 \cdots \mathbf{b}_n : \mathbf{prop}}{\Omega@\{(\Omega, \{\mathbf{a} : \mathbf{str} \mid \mathbf{P}\ \mathbf{b}_1 \cdots \mathbf{b}_n\})\} \vdash \{\mathbf{a} : \mathbf{str} \mid \mathbf{P}\ \mathbf{b}_1 \cdots \mathbf{b}_i\} : * \;\&\; + \;\&\; \emptyset}$$

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 25. Kinding and kind formation