Covariant Subtyping Applied to Semantic Predicate Calculi

William Babonnaud

Covariant Subtyping Applied to Semantic Predicate Calculi

William Babonnaud

Loria, Université de Lorraine, CNRS, Inria Nancy Grand-Est, Nancy, France
william.babonnaud@loria.fr

Abstract. Manipulating type hierarchies in formal semantic frameworks is often performed through a subtyping relation which obeys the contravariant rule for the left argument of a function type, due to the traditional representation of predicates as functions. This approach has however serious drawbacks when handling modifiers for first-order predicates. The present paper adopts an opposite view on subtyping by introducing a predicate calculus with a covariant behaviour, endowed with a categorical semantics in which subtyping coercions behave as generalisations of injective functions, and predicates are assimilated to powerobjects. This calculus is type safe in the sense that it prevents unwanted term applications, and is shown to provide a solution for the difficulties faced by a contravariant subtyping.

Keywords: Covariant subtyping · Formal semantics · Category theory

1 Subtyping in Natural Language Semantics

Integrating lexical semantics into formal frameworks, in the form of complex type systems using large hierarchies of base types, has been a challenging step in natural language semantics, opening new questions on the interaction between these hierarchies and compositionality. As in programming languages, the two (possibly overlapping) main strategies introduced in semantic calculi to deal with this multitude of types are *type polymorphism* (e.g. in MGL [27]), and *subtyping* (e.g. in UTT [19] and TCL [1]). The comparison with programming languages is relevant here to the extent that theories of polymorphism and subtyping have emerged firstly in this field, through the works of Milner [22] for the former, and Reynolds [28] and Cardelli [7,8] for the latter. In the present paper, we shall focus on subtyping.

Roughly speaking, subtyping is comparable to the subset relation for sets: if an entity is of type a and a is a subtype of b, then this entity can also be seen as being of type b. Actually, Cardelli [7] proposed a set-theoretic semantics of his calculus in which subtyping corresponds exactly to set inclusion on the semantic side. This approach relies on the idea of a large domain of values V whose type interpretations are subsets, and function types are interpreted as subsets of $V \to V$ in such a way that if α is a subtype of β, then the interpretation of

$\beta \to \gamma$ is a subset of the interpretation of $\alpha \to \gamma$, thus licensing the *contravariant rule* for function types, that is, the fact that the subtyping order for functions is reversed w.r.t. the domain type. However, large domains of values of this kind are better known as denotational semantics for interpreting systems such as the untyped lambda-calculus [31]. When it comes to typed lambda-calculus, a better set-theoretic interpretation of terms of type $\alpha \to \beta$ is given by the set B^A of (continuous) functions from A to B [4,5]; but in such a semantics, the inclusion relation from B^C to B^A when $A \subset C$ does not hold anymore, and the correspondence between contravariant subtyping and set inclusion is lost.[1] Actually, we have almost the converse semantic relation: the domain-restriction function $f \mapsto f|_A$ from B^C to B^A is generally surjective.[2]

Instead of inclusion, a more general idea conveyed by subtyping is related to type safety: whenever α is a subtype of β, a term of type α can be used at any place where a term of type β is expected without threatening the behaviour of the system. An even stronger caracterisation of the subtyping relation has been proposed by Liskov and Wing [16]: they require that whenever α is a subtype of β and ϕ is a property provable for all objects $x : \beta$, then ϕ must be provable for all objects $y : \alpha$, where the intended properties are related to safety. These accounts of safety naturally result in the contravariance rule; it is thus not surprising that applications of subtyping in formal semantic frameworks follow this approach. These applications are most commonly *coercive* [17,19,26], that is, the conversion of a term from a type to its supertype is done explicitly with a specific functional term called *coercion*. Such a choice is grounded in type-theoretical considerations (see e.g. [19, §4]), and in terms of set interpretation, amounts to replace inclusions by injective functions between disjoint domains. If $c : \alpha \to \beta$ is such a coercion, the coercion corresponding to the contravariant rule for functions is easily built as $\lambda f.\lambda x.f(cx) : (\beta \to \gamma) \to \alpha \to \gamma$ for any γ. This ease of usage, along with the amount of theoretical studies that supports it, explain why contravariant subtyping has been naturally imported to computational semantics.

However, as in programming languages [9], contravariance is the source of many difficulties in the interactions between subtyping and classical predicate interpretations inherited from Montague [23]. The following example is adapted from Luo [18]: suppose we are provided with a type v of vehicles and a type p of physical objects, with the subtyping relation $v \leq p$. Let **car** : $v \to t$ and **heavy** : $(p \to t) \to p \to t$ be predicates corresponding the noun "car" and the adjective "heavy". Then, contravariance implies $p \to t \leq v \to t$, which means that **car** is not an acceptable argument for **heavy**. Besides, contravariance also

[1] A different behaviour shows up whenever considering the set of *partial* functions: covariance of domain types is obtained instead of contravariance. However, using partial functions in natural language semantics may produce new kinds of difficulties to cope with.

[2] We may also build an injective function from B^A to B^C which extends the domain of its argument and sets a defaut value on this extension, but if no specific value in B is accessible, the axiom of choice is required to select this defaut value. We will see in Sect. 5 that our proposal uses a similar idea for predicates, since their codomain has precisely an accessible defaut value which must be "false".

implies that general type schemes for adjective representation $(\alpha \to t) \to \alpha \to t$ and $(\beta \to t) \to \beta \to t$ are not comparable even if α and β are, since they appear both positively and negatively in the schemes. These remarks have notably led to criticisms against Montagovian-based predicate models, for instance in [10]. In this paper, we redirect criticms against contravariance subtyping, which seems inadapted to the requirements of semantic-modelling calculi.

The idea of *covariant subtyping* have been pointed out as generally unsafe in programming languages.[3] However, formal frameworks for natural language semantics do not share the object-oriented perspective on types inherited from programming, where objects are thought as collections of methods in records [7,11]. In linguistic applications, an important conceptual shift occurs by replacing objects by *entities*, which do not carry methods of their own: predicates—which generally arise from words through their part of speech, following the initial proposal of Montague [23]—exist independently of entities, and simply may or may not be applicable to them depending on their types. Even if we are right when saying that physical entities have a mass, **mass** is not a method of physical entities but a predicate which is meaningfully applicable only to physical entities.

The flexibility of language is such that predicates may be applied to entities that does not match their type requirements, as for instance in the sentence "the table talks". If predicates are interpreted as functions from their meaningful argument type to propositions or truth values, this means that they may be extended to other arguments without losing their sense nor their distinction from other predicates—an idea which is similar to covariance.[4] The key to convert this remark in a more general framework is to note that, in a semantic calculus, the only functions whose codomain is not the proposition type t are base type coercions themselves. We may thus take a sharper distinction between predicates and other functions in order to provide a well-founded covariant subtyping for predicate calculus. By doing so, we may also retrieve the correspondence between subtyping relations and injective functions, provided that we are able to extend the domain of a predicate in a satisfying way. This property would ensure that whenever coerced to their supertypes, entities and predicates remain distinct, which corresponds to what happens in natural language.[5]

We shall thus introduce the basis of a predicate calculus specifically designed for applications in natural language semantics, abstracting predicate types by new type constructors rather than using the traditional function types of codomain t. Besides, this calculus will be completed with a covariant approach to subtyping, and will be given a general semantics in category theory. The types and terms

[3] In a specific setting, Castagna [9] showed that contravariance and covariance describe different mechanisms which may be adapted to coexist in a type-safe way, but the actual mechanism he defines is not exactly relevant to our purposes.

[4] It also underlies the solution proposed by Asher to overcome the difficulties presented above with the covariant type $\exists x \sqsubseteq \alpha . x \to t$ [1, §4.2].

[5] This has to be opposed to some accounts of subtyping in programming languages where distinct objects may be identified through subtyping coercion, as in the example of stacks as subtypes of bags discussed in [16, §2]: in these accounts, subtyping is surjective rather than injective.

of this calculus are introduced in Sect. 2, and its categorical interpretation is presented in Sect. 3. Through a semantic-driven construction, Sect. 4 develops the rules of subtyping and shows that subtyping coercions thus defined are interpreted as generalisation of injective functions. Sect. 5 supplies additional results on the interaction between covariant subtyping and logical operators, and gives more intuition on the behaviour of coerced predicates. Sect. 6 discusses possible extensions and future perspectives.

2 A Predicate Calculus with Abstracted Functions

Rather than introducing a new version of a full lambda-calculus with some slight changes compared to other proposals of Montagovian inspiration, we propose a system which may appear weaker at first glance, but which actually abstracts upon the simply-typed lambda-calculus. Motivation for such a proposal comes from the fact that the expressive power of a semantic framework, viewed as a system which ultimately aims at building logical formulae to represent the intended semantics of sentences, relies mainly on three ingredients: predicates, logical connectives, and construction rules. The first two of these ingredients are rendered as constants in almost every calculus, and the last one is provided by general rules for constructing lambda-terms, *application* being the most important thereof.

It is particularly notable that the lambda-terms appearing in such frameworks are closed ones, and that variables are mainly used for composition purposes as an intermediate step before applying a rule of λ-abstraction. For instance, combining two first-order predicates $u, v : e \to t$ with the logical connective $\wedge : t \to t \to t$ use several applications and an abstraction to result in $\lambda x.u(x) \wedge v(x)$.[6] Provided that we have access to closed extensions of $\wedge$ for use with other types, for instance with the polymorphic AND definable in MGL [27], the same result could be obtained up to η-expansion as $\wedge^{e \to t}\, u\, v$ in only two applications. We shall generalise this reasoning by designing a "weaker" calculus which has no direct access to term variables and no abstraction rule.

Following this idea and the discussions on subtyping from Sect. 1, we shall therefore introduce the covariant subtyping calculus $C\Sigma$. It can be seen as a higher-level lambda-calculus, in the same way as programming languages such as Python are high-level languages w.r.t. machine code or assembly language: its types will be abstractions of common types, its terms will be constants and combinations thereof, and its rules will be restricted to application and other combining rules—but all of these components will stay definable within a more general typed lambda-calculus. As such, $C\Sigma$ is intended to provide the minimal setting for semantic purposes, and to be cleared of any other unnecessary feature.

We will successively define the three main parts of the type system of $C\Sigma$, namely the *type ontology*, the *type constructors*, and the *coercion types*. Altogether, these pieces build the set $\mathcal{T}_{C\Sigma}$ of types of our calculus.

[6] Throughout this paper usual logical connectives will be assumed to carry their usual infix notation.

Definition 1. *A* ***type ontology*** *is a pair* $(\mathbb{B}, \leq)$ *where* $\mathbb{B}$ *is a set of* base types *and* $\leq \subset \mathbb{B} \times \mathbb{B}$ *is a partial order endowed with a greatest element* $e \in \mathbb{B}$*.*

While any type hierarchy may satisfy this definition, the term *ontology* is reminiscent of Fred Sommers' theory of ontological types [32,33], and hints how we would expect the base types to be constructed. In particular, we think the hierarchical structure underlying $\leq$ to be almost a tree.[7]

The next step is to introduce the type constructors. Leaving aside coercions, we will use a unit type as well as products and predicates.

Definition 2. *Let* $(\mathbb{B}, \leq)$ *be a type ontology. The set* $\Xi_{C\Sigma}$ *of* ***complex types upon*** $\mathbb{B}$ *is defined recursively by the grammar*

$$\Xi_{C\Sigma} ::= 1 \mid \mathbb{B} \mid \Xi_{C\Sigma} \times \Xi_{C\Sigma} \mid \mathcal{P}\Xi_{C\Sigma}$$

where 1 *is the unit type, and* $\mathcal{P}$ *is the predicate constructor.*

We may assume the product constructor $\times$ to be associative, so that we can avoid parentheses in successive applications. Compared to common type theories in semantic frameworks, one may notice that we did not introduce a type t for truth values or propositions. It is actually hidden in predicate types: for any $\alpha \in \Xi_{C\Sigma}$, the type $\mathcal{P}\alpha$ of predicates upon α corresponds conceptually to the simple type $\alpha \to t$. Furthermore, a predicate upon a product type corresponds to the uncurried version of a n-ary function of codomain t, and as 1 stands for the empty product, the type t is actually $\mathcal{P}1$. The choice of $\mathcal{P}$ rather than $\to$ is intended to highlight the conceptual differences between predicates and other functions, as will be clarified when introducing subtyping rules in Sect. 4. The arrow constructor is however not ruled out, as it is used for defining coercions.

Definition 3. *Let* $\Xi_{C\Sigma}$ *be a set of complex types upon some type ontology. A* $\Xi_{C\Sigma}$***-coercion*** *is a type of the form* $\alpha \to \beta$*, where* $\alpha, \beta \in \Xi_{C\Sigma}$*.*

As to be seen in Sect. 4, we will generally use exactly one coercion term for each $\Xi_{C\Sigma}$-coercion, that is, we will not allow to use two different coercions for the same arrow type. This assumption enforces the coherence of the system of coercions, and henceforth enables us to identify bijectively each $\Xi_{C\Sigma}$-coercion, which is a type, to a corresponding term which is also called *coercion*. The simplest version of the $C\Sigma$ calculus can use the full set of coercions $\Xi_{C\Sigma} \to \Xi_{C\Sigma}$, but the resulting theory shall be degenerate. Therefore, we will assume to have a specific subset $\mathcal{K} \subset \Xi_{C\Sigma} \to \Xi_{C\Sigma}$ of coercions, whose construction will be constrained by the guidelines in Sect. 4.

Definition 4. *Let* $(\mathbb{B}, \leq)$ *be a type ontology and* $\mathcal{K}$ *a set of* $\Xi_{C\Sigma}$*-coercions. The set* $\mathcal{T}_{C\Sigma}$ *of* $C\Sigma$***-types based upon*** $\mathbb{B}$ ***and*** $\mathcal{K}$ *is defined as:*

$$\mathcal{T}_{C\Sigma} = \Xi_{C\Sigma} \cup \mathcal{K}$$

[7] The reader may also consult [2,29] for recent accounts of Sommers' theory.

We hinted earlier that the $C\Sigma$ calculus should be definable within a lower-level typed lambda-calculus. Being given the set $\mathcal{T}$ of types of such a calculus, including the base types of $\mathbb{B}$, a proposition type t, a unit type 1, products and arrows, we can define an encoding $\theta : \mathcal{T}_{C\Sigma} \to \mathcal{T}$ in the following way:

$$\begin{aligned}
\theta(1) &= 1 \\
\theta(b) &= b && \text{for } b \in \mathbb{B} \\
\theta(\alpha \times \beta) &= \theta(\alpha) \times \theta(\beta) && \text{for } \alpha, \beta \in \Xi_{C\Sigma} \\
\theta(\mathcal{P}(\alpha_1 \times \cdots \times \alpha_n)) &= \theta(\alpha_1) \to \cdots \to \theta(\alpha_n) \to t \\
& && \text{for } \alpha_1, \ldots, \alpha_n \in \Xi_{C\Sigma} \\
\theta(\alpha \to \beta) &= \theta(\alpha) \to \theta(\beta) && \text{for } \alpha \to \beta \in \mathcal{K}
\end{aligned} \tag{1}$$

We now turn to terms. As previously suggested, $C\Sigma$ is mainly constant-based and will not use variables nor lambda-abstraction, at least directly; yet some terms introduced here as constants may be definable as operators in a lower-level calculus precisely by using those hidden constructions. In the rest of this paper we fix a type ontology and a coercion set, so that the set of $C\Sigma$-types is also fixed. We call *constant signature* a pair $(\mathcal{Q}, \tau)$ where $\mathcal{Q}$ is a set of constants and τ is a function $\mathcal{Q} \to \Xi_{C\Sigma}$, and *coercion signature* a pair (K, σ) where K is a set of coercions and σ a bijective function $K \to \mathcal{K}$.

Definition 5. *Let $(\mathcal{Q}, \tau)$ and (K, σ) be constant and coercion signatures. The set $\Lambda_{C\Sigma}$ of **untyped terms of** $C\Sigma$ is recursively defined by the grammar:*

$$\Lambda_{C\Sigma} ::= * \mid \mathcal{Q} \mid K\Lambda_{C\Sigma} \mid \langle \Lambda_{C\Sigma}, \Lambda_{C\Sigma} \rangle \mid \Lambda_{C\Sigma}\Lambda_{C\Sigma} \mid \pi_1\Lambda_{C\Sigma} \mid \pi_2\Lambda_{C\Sigma}$$

Through misuse of language, we will consider $\mathcal{K}$ itself to be a coercion signature, and denote by $c : \alpha \to \beta \in \mathcal{K}$ that c is the symbol representing the coercion from α to β.

The restriction to constants within $C\Sigma$ also makes typing judgements simpler, since variable environments are not needed anymore. As usual, a *typing judgement* will be a sequent $\vdash u : \alpha$ where u is a term and α is a type in $\Xi_{C\Sigma}$. We shall introduce the typing rules in two batches, starting with the more direct rules and postponing the rules using coercions to Sect. 4. The first batch contains the basic rules for constants and pairs, as well as direct application:

$$\begin{gathered}
\frac{}{\vdash * : 1}\,(\textsc{unit}) \qquad \frac{u \in (\mathcal{Q}, \tau)}{\vdash u : \tau(u)}\,(\textsc{const}) \qquad \frac{\vdash u : \alpha \quad \vdash v : \beta}{\vdash \langle u, v \rangle : \alpha \times \beta}\,(\textsc{pair}) \\
\frac{\vdash u : \mathcal{P}(\alpha \times \beta) \quad \vdash v : \alpha}{\vdash uv : \mathcal{P}\beta}\,(\textsc{app}) \qquad \frac{\vdash u : \mathcal{P}\alpha \quad \vdash v : \alpha}{\vdash uv : \mathcal{P}1}\,(\textsc{app}') \\
\frac{\vdash u : \alpha \times \beta}{\vdash \pi_1 u : \alpha}\,(\textsc{proj}_1) \qquad \frac{\vdash u : \alpha \times \beta}{\vdash \pi_2 u : \beta}\,(\textsc{proj}_2)
\end{gathered} \tag{2}$$

Similarly to product types, we may assume that pairs extend to tuples of any size. As for direct application, the rules (APP) and (APP′) distinguish the cases of “partial” and “total” application of predicates. We may have considered an equivalent statement by using only the rule (APP) and an isomorphic operator $\mathcal{P}\alpha \cong \mathcal{P}(\alpha \times 1)$, thus enabling us to derive (APP$'$) as an admissible rule; yet we will keep the latter rule as such for clarity. One may observe that no rule is provided to build predicate terms: all predicates have to be introduced as constants. This implies of course that the constant signature is non-empty and contains at least one predicate, which is consistent with the semantic aim of the calculus: predicates in natural language semantics are generally introduced as constants, and we argue that $C\Sigma$ provides enough material to make explicit construction of predicates as lambda-abstractions unnecessary. In particular, the next section will introduce the logical operators that enables one to build complex predicates from predicate constants.

We conclude the present section by stating the mandatory term equalities. Besides the traditional rules of congruence (REFL), (SYM) and (TRANS), we assert in (ONE) that $*$ is the only term of type 1, in (EQPR) and (EQPAR) that equalities propagate to products and applications, in (PRJ$_1$) and (PRJ$_2$) that projections work on pairs as expected, and in (AP*) that successive applications of a predicate to several arguments amount to applying it to the tuple of these arguments. Furthermore, the rule (ASSOC) states the tuple equality which licenses the use of $\times$ as an associative type constructor. Both (EQAP) and (AP*) correspond to the previous typing rule (APP); the definition of their counterparts for the rule (APP$'$) is left as an exercise to the reader. Here again, the rules involving coercions are postponed to Sect. 4.

$$\frac{\vdash u : \alpha}{\vdash u = u : \alpha}\,(\text{REFL}) \qquad \frac{\vdash u = v : \alpha}{\vdash v = u : \alpha}\,(\text{SYM}) \qquad \frac{\vdash u : 1}{\vdash u = * : 1}\,(\text{ONE})$$

$$\frac{\vdash u = v : \alpha \quad \vdash v = w : \alpha}{\vdash u = w : \alpha}\,(\text{TRANS}) \qquad \frac{\vdash u = u' : \alpha \quad \vdash v = v' : \beta}{\vdash \langle u, v\rangle = \langle u', v'\rangle : \alpha \times \beta}\,(\text{EQPR})$$

$$\frac{\vdash u : \alpha \quad \vdash v : \beta}{\vdash \pi_1\langle u, v\rangle = u : \alpha}\,(\text{PRJ}_1) \qquad \frac{\vdash u = u' : \mathcal{P}(\alpha \times \beta) \quad \vdash v = v' : \alpha}{\vdash uv = u'v' : \mathcal{P}\beta}\,(\text{EQAP}) \tag{3}$$

$$\frac{\vdash u : \alpha \quad \vdash v : \beta}{\vdash \pi_2\langle u, v\rangle = v : \beta}\,(\text{PRJ}_2) \qquad \frac{\vdash u : \mathcal{P}(\alpha \times \beta \times \gamma) \quad \vdash v : \alpha \quad \vdash w : \beta}{\vdash (uv)w = u\langle v, w\rangle : \mathcal{P}\gamma}\,(\text{AP}^*)$$

$$\frac{\vdash u : \alpha \quad \vdash v : \beta \quad \vdash w : \gamma}{\vdash \langle u, \langle v, w\rangle\rangle = \langle\langle u, v\rangle, w\rangle : \alpha \times \beta \times \gamma}\,(\text{ASSOC})$$

3 Categorical Model of $C\Sigma$

In order to study more efficiently its underlying type theory, we define an interpretation of $C\Sigma$ in category theory. Many equivalences between type systems and classes of categories have been identified, notable examples including the correspondences between simply-typed λ-calculi and cartesian closed categories [14], as well as between Martin-Löf type theories and locally cartesian closed categories [30]. As $C\Sigma$ is "high-level", we may expect its categorical model to have weaker assumptions; however, this will not actually be the case: we need all the power of the underlying "low-level" model to interpret efficiently our calculus.

The rest of this paper assumes preliminary knowledge of category theory, including the notions of categories, initial and terminal objects, products, exponentials, pullbacks, functors, natural transformations, and adjunctions.[8] The reader may consult [20,25] for an introduction; useful elements (with increasing difficulty) can also be found in [12,13,21]. Only three definitions will be recalled here: *monomorphisms*, *subobject classifiers*, and *toposes*.

Definition 6. *A morphism $f : A \to B$ is a **monomorphism** (or "is mono", noted $f : A \rightarrowtail B$) if for any pair of morphisms $g, h : C \to A$, $f \circ g = f \circ h$ implies $g = h$. A is a **subobject** of B if there is a monomorphism $A \rightarrowtail B$.*

Definition 7. *In any category with a terminal object, a **subobject classifier** is an object Ω along with a morphism $\top : 1 \to \Omega$ such that for any monomorphism $m : A \rightarrowtail B$, there is a unique morphism $\chi_m : B \to \Omega$ such that the following diagram is a pullback:*

$$\begin{tikzcd}
B \arrow[r, "\chi_m"] & \Omega \\
A \arrow[u, tail, "m"] \arrow[r, "!_A"] & 1 \arrow[u, "\top"']
\end{tikzcd}$$

Definition 8. *A **topos** is a cartesian closed category with a subobject classifier.*

The type ontology $(\mathbb{B}, \leq)$ can be seen as a category $\mathcal{B}$ by taking elements of $\mathbb{B}$ as objects, and by introducing a morphism $a \to b$ whenever $a \leq b$. We consider from now on a topos $\mathcal{C}$ containing $\mathcal{B}$ as a subcategory, and we note $\zeta : \mathcal{B} \to \mathcal{C}$ the associated faithful functor. In particular, we define the object E of entities as ζe, where e is the greatest element of $\mathcal{B}$. Besides, we require the image ζf of each morphism f in $\mathcal{B}$ to be a monomorphism. Notice then that $\mathcal{B}$ as a poset is contained in $\mathrm{Sub}(E)$, the set of subobjects of E.

For any object A of $\mathcal{C}$, we note $\mathcal{P}A$ the exponential Ω^A, which is called the *powerobject of* A. The associated *evaluation map* is $\mathrm{ev}_A : \mathcal{P}A \times A \to \Omega$. Notice that, by a well-known property of exponentials, we have the isomorphism $\mathcal{P}(A \times B) \cong \mathcal{P}A \times \mathcal{P}B$ for any objects A and B. If $f : A \times B \to \Omega$ is a morphism, we call *name* of f the morphism $\mathrm{name}(f) : A \to \mathcal{P}B$ obtained from f by the universal property of exponentials. As $1 \times B \cong B$, this definition extends up to isomorphism to any morphism $B \to \Omega$. As a consequence, a predicate in a topos has three equivalent modes of presentation: as an arrow $f : B \to \Omega$, as its name $1 \to \mathcal{P}B$, or as the subobject $A \rightarrowtail B$ obtained by pullback of $\top$ along f.

We have now enough theoretical support to properly define an interpretation of $C\Sigma$ in the topos $\mathcal{C}$. This interpretation will be denoted as a map $[\![\cdot]\!]$ which

[8] The composition of $f : A \to B$ and $g : B \to C$ will be noted $g \circ f$ or gf whenever no ambiguity may occur. The identity on A is id_A. The product map of $f' : A \to B$ and $g' : A \to C$ will be noted $\langle f', g' \rangle : A \to B \times C$. The initial and terminal objects are noted 0 and 1, and the associated morphisms are $0_A : 0 \to A$ and $!_A : A \to 1$.

assigns to each type in $\Xi_{C\Sigma}$ an object of $\mathcal{C}$, and to each well-typed term a *global element* in $\mathcal{C}$, that is, a morphism of the form $1 \to A$. For types the definition of $[\![\cdot]\!]$ is straightforward since every constructor has an obvious categorical counterpart:

$$\begin{aligned} [\![1]\!] &= 1 && \\ [\![b]\!] &= \zeta b && \text{for } b \in \mathbb{B} \\ [\![\alpha \times \beta]\!] &= [\![\alpha]\!] \times [\![\beta]\!] && \text{for } \alpha, \beta \in \Xi_{C\Sigma} \\ [\![\mathcal{P}\alpha]\!] &= \mathcal{P}[\![\alpha]\!] && \text{for } \alpha \in \Xi_{C\Sigma} \end{aligned} \tag{4}$$

Terms however need a few more assumptions to be correctly interpreted. First, even if by Definition 5 coercions are not considered as terms, they are given an interpretation by $[\![\cdot]\!]$. We will thus assume that each coercion $c : \alpha \to \beta \in \mathcal{K}$ is assigned a morphism $\kappa(c) : [\![\alpha]\!] \to [\![\beta]\!]$. Moreover, we require that each constant $u \in (\mathcal{Q}, \tau)$ has a corresponding morphism $\rho(u) : 1 \to [\![\tau(u)]\!]$, and that the induced map ρ from Q to the morphisms of $\mathcal{C}$ is injective. Then, any well-typed term $u : \alpha$ is interpreted as a morphism $1 \to [\![\alpha]\!]$ according to the following definition:

$$\begin{aligned} [\![*]\!] &= \mathrm{id}_1 : 1 \to 1 && \\ [\![u]\!] &= \rho(u) : 1 \to [\![\tau(u)]\!] && \text{for } u \in (Q, \tau) \\ [\![c]\!] &= \kappa(c) : [\![\alpha]\!] \to [\![\beta]\!] && \text{for } c : \alpha \to \beta \in \mathcal{K} \\ [\![cu]\!] &= [\![c]\!] \circ [\![u]\!] : 1 \to [\![\beta]\!] && \text{for } c : \alpha \to \beta \in \mathcal{K} \text{ and } u : \alpha \\ [\![\langle u, v\rangle]\!] &= \langle [\![u]\!], [\![v]\!] \rangle : 1 \to [\![\alpha]\!] \times [\![\beta]\!] && \text{for } u : \alpha \text{ and } v : \beta \\ [\![uv]\!] &= \mathrm{comp}_{[\![\alpha]\!],[\![\beta]\!]} \circ \langle [\![u]\!], [\![v]\!] \rangle : 1 \to \mathcal{P}[\![\beta]\!] && \text{for } u : \mathcal{P}(\alpha \times \beta) \text{ and } v : \alpha \\ [\![\pi_1 u]\!] &= \pi_1^{[\![\alpha]\!],[\![\beta]\!]} \circ [\![u]\!] : 1 \to [\![\alpha]\!] && \text{for } u : \alpha \times \beta \\ [\![\pi_2 u]\!] &= \pi_2^{[\![\alpha]\!],[\![\beta]\!]} \circ [\![u]\!] : 1 \to [\![\beta]\!] && \text{for } u : \alpha \times \beta \end{aligned} \tag{5}$$

where, for any objects A, B, $\pi_1^{A,B}$ and $\pi_2^{A,B}$ are the usual projections $A \times B \to A$ and $A \times B \to B$, and $\mathrm{comp}_{A,B}$ is defined to be the name of the composite:[9]

$$(\mathcal{P}(A \times B) \times A) \times B \xrightarrow{\sim} \mathcal{P}(A \times B) \times (A \times B) \xrightarrow{\mathrm{ev}_{A \times B}} \Omega$$

Then, we need to ensure that this interpretation respects the term equalities introduced in (3). This is straightforward for the congruence rules and for the propagation rules (EQPR) and (EQAP), and the well-foundedness of the rule (ONE) is guaranteed by the fact that the unit type is interpreted as the terminal object 1, for which id_1 is the unique morphism $1 \to 1$. Moreover, the projection rules (PRJ$_1$) and (PRJ$_2$) are easily retrieved by the definition of categorical products. The major subtlety lies in the translation of the last two rules, stating the associativity of the product and its consequence on direct application. For any terms u, v and w, the interpretations of $\langle u, \langle v, w\rangle\rangle$ and $\langle\langle u, v\rangle, w\rangle$ are only equal up to some isomorphism which belongs to the class of isomorphisms of the form

[9] Notice in particular that, up to isomorphism, $\mathrm{comp}_{A,1} = \mathrm{ev}_A$. This is consistent because the isomorphism $\mathcal{P}1 \cong \Omega$ holds in any topos.

$\iota_{A,B,C} : (A \times B) \times C \to A \times (B \times C)$. These isomorphisms also propagates to n-ary predicates through isomorphisms $\exists\iota_{A,B,C} : \mathcal{P}((A \times B) \times C) \to \mathcal{P}(A \times (B \times C))$, whose notation will become clearer at the end of this section. As a result, the term equalities may be correctly retrieved by considering the equivalence classes of the categorical interpretations for the relation on morphisms of being equal up to composition with some isomorphism of the form ι, $\exists\iota$, or combination thereof.[10]

So far, in the construction of $C\Sigma$ and its interpretation above we only discussed two of the three main ingredients of a semantic calculus as presented at the beginning of Sect. 2: predicates and rules. What about logical connectives? Obviously we may introduce them as constants as well, but since we do not use directly a proposition type nor lambda-abstractions, the usual definition of logical constants does not fit in $C\Sigma$: we would be unable to connect predicates before all their arguments are provided, whereas construction of lambda-abstracted compounds is common in natural language semantics. Furthermore, the introduction of these connectives should be carefully studied from a categorical perspective because toposes have their own internal logic: we then expect the logical constants of $C\Sigma$ to reflect this logic. Toposes are powerful enough to model higher-order logic [15], and depending on the properties of the given topos, this logic can be classical or intuitionistic [12].

The formal basis of topos logic lies on morphisms $\neg : \Omega \to \Omega$ and $\wedge, \vee, \Rightarrow: \Omega \times \Omega \to \Omega$. For any A, we can extend these arrows to "generalised" operators $\neg_A : \mathcal{P}A \to \mathcal{P}A$ and $\wedge_A, \vee_A, \Rightarrow_A: \mathcal{P}A \times \mathcal{P}A \to \mathcal{P}A$ as names of composites of the above morphisms with evaluation maps. For instance, $\wedge_A$ names the composite:

$$\begin{tikzcd}
(\mathcal{P}A \times \mathcal{P}A) \times A \arrow[d, "\mathrm{id} \times \Delta_A"] & \Omega \times \Omega \arrow[r, "\wedge"] & \Omega \\
(\mathcal{P}A \times \mathcal{P}A) \times (A \times A) \arrow[r, "\sim"] & (\mathcal{P}A \times A) \times (\mathcal{P}A \times A) \arrow[u, "\mathrm{ev}_A \times \mathrm{ev}_A"] &
\end{tikzcd}$$

where $\Delta_A : A \to A \times A$ is the diagonal map $\langle \mathrm{id}_A, \mathrm{id}_A \rangle$. Other logical connectives are obtained in the same way. In $C\Sigma$, we may think of $\neg$, $\wedge$, $\vee$ and $\Rightarrow$ as polymorphic constants of respective types $\forall\alpha.\mathcal{P}(\mathcal{P}\alpha \times \alpha)$ for the former and $\forall\alpha.\mathcal{P}(\mathcal{P}\alpha \times \mathcal{P}\alpha \times \alpha)$ for the others. They are however distinct from other constants and more generally from other terms in the sense that their interpretations are not global elements: to keep things consistent, we have for instance to posit for each type α in $\Xi_{C\Sigma}$ the interpretation $[\![\wedge^\alpha]\!] = \wedge_{[\![\alpha]\!]}$. Thus, logical connectives play a very specific role in $C\Sigma$, and may be added to the calculus as a separate set of constants.

The treatment of quantifiers is slightly more complex, as it requires additional constructions of topos theory. As pulling back along a morphism $f : A \to B$ preserves monomorphisms, it induces a pullback function $f^{-1} : \mathrm{Sub}(B) \to \mathrm{Sub}(A)$

[10] As suggested at the end of Sect. 2, we may extend this reasoning to the class of isomorphisms of the form $A \times 1 \to A$ to avoid the typing rule (APP$'$) and its counterparts in term equalities.

which has both a left adjoint $\exists_f$ and a right adjoint $\forall_f$. As $\mathcal{P}A$ is the internal version of the set $\mathrm{Sub}(A)$ for any object A, those adjoints themselves induce internal morphisms $\exists f : \mathcal{P}A \to \mathcal{P}B$ and $\forall f : \mathcal{P}A \to \mathcal{P}B$ [21, §IV.9]. To produce models of the usual quantifiers, we use versions of these morphisms obtained from product projections: if $\pi : A \times B \to B$ is such a projection, $\exists\pi$ and $\forall\pi$ are morphisms $\mathcal{P}(A \times B) \to \mathcal{P}B$ which can serve as interpretations for polymorphic constants $\exists^{\alpha,\beta}$ and $\forall^{\alpha,\beta}$ of type $\mathcal{P}(\mathcal{P}(\alpha\times\beta)\times\beta)$ in $C\Sigma$. Other kinds of generalised quantifiers may also be introduced as morphisms of this family of types; their own properties would depend on their respective definitions, and they would differs from the universal and existential quantifiers only by the absence of obvious relation to the pullback function. This completes the introduction of logical connectives in the calculus.

4 Monomorphic Subtyping Discipline

Monomorphism is the categorical generalisation of the notion of injective function; in particular, both notions coincide in the category Set (see e.g. [12, §3.1]). As discussed in Sect. 1, a set-based interpretation of the subtyping relation in natural language semantics should be injective as well, for entities and for predicates. From a categorical perspective, we come naturally to think of the subtyping relation in terms of monomorphisms. The present section details how we can construct such a "monic" subtyping relation in $C\Sigma$, and how such a relation is necessarily a covariant one.

In Sect. 2, we hinted that defining the subtyping relation amounts to give constraints for building the set of coercions $\mathcal{K}$. If α and β are types in $\Xi_{C\Sigma}$, we note $\alpha \sqsubseteq \beta$ to express that α is a subtype of β, and whenever it is the case, we enforce the constraint $\alpha \to \beta \in \mathcal{K}$. We shall generate subtyping relations and categorical interpretations of the corresponding coercions simultaneously. As a starting point, recall that the construction of $C\Sigma$ is based on a type ontology $(\mathbb{B}, \leq)$ which includes the foundations of the subtyping relation. For every pair $a, b \in \mathbb{B}$, we naturally posit $a \sqsubseteq b$ whenever $a \leq b$ holds.[11] By assumptions made on the topos $\mathcal{C}$ in Sect. 3, such a pair provides also a monomorphism $\zeta a \rightarrowtail \zeta b$ which is exactly the interpretation of the related coercion $a \to b \in \mathcal{K}$. This sets up the basis of the subtyping relation, and we have now to investigate how it propagates to type constructors.

To avoid inconsistency, the unit type should only be comparable with itself, that is, $1 \sqsubseteq 1$ is the only valid subtyping relation involving 1. To see how to deal with products and predicates, we turn to their categorical interpretations. The topos $\mathcal{C}$, as a cartesian category, is equipped with a bifunctor $\times : \mathcal{C} \otimes \mathcal{C} \to \mathcal{C}$, where $\otimes$ stands for the product of categories. The following lemma is a common result whose proof is easy to retrieve:

[11] Actually, we may remove superfluous coercions by limiting $\mathcal{K}$ to the strict part of the subtyping relation, that is, $\alpha \to \beta$ in $\mathcal{K}$ only if $\alpha \sqsubset \beta$ holds. Indeed, the coercion corresponding to $\alpha \to \alpha$ is the identity map $\mathrm{id}_{[\![\alpha]\!]}$, which brings no useful additional information.

Lemma 1. *The bifunctor $\times$ preserves monomorphisms, that is, if $f : A \rightarrowtail C$ and $g : B \rightarrowtail D$ are monos, then so is $f \times g : A \times B \rightarrowtail C \times D$.*

A similar result can be established for powerobjects through a less common view than usual. Indeed, the operator $\mathcal{P}$ is generally extended to a contravariant functor $\mathcal{C}^{\mathrm{op}} \to \mathcal{C}$ by taking $\mathcal{P}f$ to be the internal counterpart of the map f^{-1} introduced in Sect. 3, but this definition does not satisfy our requirements. We shall use instead the *covariant powerobject functor* $\mathcal{P}^+ : \mathcal{C} \to \mathcal{C}$ defined on any object A by $\mathcal{P}^+A = \mathcal{P}A$ and on any morphism f by $\mathcal{P}^+f = \exists f$. This definition is correct since $\exists(gf) = \exists g \circ \exists f$ for any morphisms f, g, and $\exists\, \mathrm{id}_A = \mathrm{id}_{\mathcal{P}A}$ for any object A [13, §A2.3]. The following lemma appears in [13, Cor. A2.2.5] and [21, Cor. 3, §IV.3]:

Lemma 2. *If $m : A \rightarrowtail B$ is mono, then $\mathcal{P}m \circ \exists m = \mathrm{id}_{\mathcal{P}A}$, i.e. $\exists m$ is split mono.*

A weaker way to put it is to say that $\mathcal{P}^+$ preserves monomorphisms. Nevertheless we shall see below that the existence of the retraction $\mathcal{P}m$ will be useful for a special extension of $C\Sigma$. Altogether, the lemmata above provide the keys to complete the definition of the subtyping relation for all the type constructors. Hence we assert the following subtyping rules:

$$\frac{a, b \in \mathbb{B} \quad a \leq b}{a \sqsubseteq b}\,(\textsc{base}) \qquad \frac{\alpha \sqsubseteq \gamma \quad \beta \sqsubseteq \delta}{\alpha \times \beta \sqsubseteq \gamma \times \delta}\,(\textsc{prod}) \qquad \frac{\alpha \sqsubseteq \beta}{\mathcal{P}\alpha \sqsubseteq \mathcal{P}\beta}\,(\textsc{pred}) \qquad (6)$$

The coercion set $\mathcal{K}$ must be built consequently. The map κ introduced in Sect. 3, which sends each coercion in $\mathcal{K}$ to a corresponding morphism in $\mathcal{C}$, can also be properly defined using similar rules: if c, c' are coercions, the corresponding product coercion introduced by the rule (PROD), noted $c \times c'$, shall be interpreted by $\kappa(c \times c') = \kappa(c) \times \kappa(c')$; and similarly, if c is a coercion, the predicate coercion introduced by the rule (PRED), noted $\mathcal{P}c$, shall be interpreted by $\kappa(\mathcal{P}c) = \exists\kappa(c)$. Recall then that we put $[\![c]\!] = \kappa(c)$ for a coercion c. Lemmata 1 and 2, as well as the construction of $\mathcal{C}$ above the type ontology, can be used to prove inductively the following result, which puts monomorphisms at the heart of the subtyping interpretation.

Proposition 1. *For any coercion $c \in \mathcal{K}$, if c has been constructed using the rules in* (6)*, then $[\![c]\!]$ is a monomorphism.*

As promised in Sect. 2, we now turn to the second batch of typing rules for $C\Sigma$, which shall constrain the use of coercions. Our main concern when defining these rules is to provide a strong discipline on coercion uses in order to preserve *type safety*, even if we take this notion in a weaker understanding compared to its definition for programming languages. Our objective is to prevent overgeneration in $C\Sigma$ by constraining the use of coercions: for instance, we may forbid the application of coercions both to a predicate and its argument at the same time, because it would allow any predicate to take any argument. Moreover, we may need to prevent n-ary predicates with argument-places related in types to accept arguments coerced from different types if the underlying semantics intends to link them, as it could lead to unwanted semantic interpretations.

Assume we add a base type $\Box$ to $C\Sigma$, and define a *type context* to be a type in $\Xi^{\Box}_{C\Sigma}$, that is, a type with at least one occurrence of $\Box$. If d is such a context, define for any base type $a \neq \Box$ the type $d[a]$ to be d where all occurrences of $\Box$ are replaced by a. We may generally think of a type $d[a]$ for a term to represent a product of types in $\Xi_{C\Sigma}$, some of them involving the type a. We propose the following formulation:

Definition 9. *A $C\Sigma$ calculus is **type safe** if:*

1. *for all pairs a, b of incompatible base types, type context d, term $u : \mathcal{P}d[a]$ and constant $v : d[b]$, and coercions c and c' of respective domains $\mathcal{P}d[a]$ and $d[b]$, no combinations of typing rules make $(cu)(c'v)$ a well-typed term ;*
2. *for all type a, pair of incompatibles types b, c, type contexts d, d', d'' and γ such that $d = d' \times d'' \times \gamma$, term $u : \mathcal{P}d[a]$, constants $v : d'[b]$ and $w : d''[c]$, and coercions $c' : d'[b] \to d'[a]$ and $c'' : d''[c] \to d''[a]$, no combinations of typing rules make $u((c' \times c'')\langle v, w\rangle)$ a well-typed term.*

This formulation is intended to translate into formal conditions the discussion of the previous paragraph: the first condition forbids to coerce predicates and arguments at the same time, and the second condition forbids the simultaneous coercion of two incompatible types to the same type between arguments of a given predicate. Recall that, by the equality rule (AP*) and the definition of the product of coercion, the second condition also applies to terms of the form $u(c'v)(c''w)$. In practice, the first condition states for instance that the following rule of *free coerced application* (FCA) is *not* acceptable for type safety:

$$\text{unsafe!} \qquad \frac{\vdash u : \mathcal{P}\alpha \quad \vdash v : \beta \quad c : \alpha \to \gamma \in \mathcal{K} \quad c' : \beta \to \gamma \in \mathcal{K}}{\vdash ((\mathcal{P}c)u)\,(c'v) : \mathcal{P}1}\,(\text{FCA}) \tag{7}$$

In a similar vein, the second condition states for instance that the rule of *partial coerced application* (PCA) defined below is also *not* type safe:

$$\text{unsafe!} \qquad \frac{\vdash u : \mathcal{P}(\beta \times \gamma) \quad \vdash v : \alpha \quad c : \alpha \to \beta \in \mathcal{K}}{\vdash u\,(cv) : \mathcal{P}\gamma}\,(\text{PCA}) \tag{8}$$

Indeed, consider base types h, v and p standing for humans, vehicles and physical entities respectively, with $h \leq p$, $v \leq p$ and h, v incompatible, and assume constant predicates $\mathbf{j} : h$, $\mathbf{car} : \mathcal{P}v$ and $\mathbf{heavy} : \mathcal{P}(\mathcal{P}p \times p)$. By construction, we have coercions $c : h \to p$ and $c' : \mathcal{P}v \to \mathcal{P}p$. Now, by two successive applications of the rule (PCA), the term $\mathbf{heavy}(c'\,\mathbf{car})(c\,\mathbf{j})$ is well-typed, which explicitly breaks the second condition, hence the unsafeness of (PCA).[12] This term could be the semantics of a category-mistaken sentence such as "?John is a heavy car". Of course, we might want to have semantic representation of such sentences in our calculus, but it should not be obtained by means of the subtyping relation.

To preserve type safety, we need a weaker version of the rule (PCA), which will be called *restricted total coerced application* (RTCA). The idea behind this

[12] The same reason explains why we did not introduce a typing rule to build directly coerced terms of the form cu in Sect. 2.

rule is to force all the arguments of a given predicate to be gathered before application and to restrict coercion uses so that the arguments filling slots with the same expected base type are themselves of the same base type, thus avoiding simultaneous application of a predicate to terms such as **car** and **j**.

$$\frac{\vdash u : \mathcal{P}d[b] \quad \vdash v : d[a] \quad c : d[a] \to d[b] \in \mathcal{K}}{\vdash u\,(cv) : \mathcal{P}1}\,(\textsc{rtca}) \tag{9}$$

It follows directly from Definition 9 that (RTCA) is type safe. It can even be safely extended for simultaneous subtyping of several base types: generalise the notion of type context for finite number of holes $\square_1, \ldots, \square_n$, the corresponding extended rule is defined for $\vdash u : \mathcal{P}d[b_1, \ldots, b_n]$ and $\vdash v : d[a_1, \ldots, a_n]$ with the additional condition that all b_i must be pairwise distinct. This extension is useful when dealing with n-ary predicates, for instance transitive verbs.

We acknowledge however that (RTCA), even in its extended form, may be too restrictive for semantic uses, in particular in a calculus like $C\Sigma$ where no variables and λ-abstractions are available. What if we need to perform the partial application of **heavy** on **car** as the semantic representation of the phrase "a heavy car" requires? A solution to retrieve the flexibility of partial coercion application is to exploit the retraction of the corresponding predicate coercion. For each predicate coercion $c : \mathcal{P}\alpha \to \mathcal{P}\beta$ in $\mathcal{K}$, define its *reverse coercion* $\bar{c} : \mathcal{P}\beta \to \mathcal{P}\alpha$ with $[\![\bar{c}]\!] = \mathcal{P}m$, where m is the mono $[\![\alpha]\!] \to [\![\beta]\!]$ such that $[\![c]\!] = \exists m$. Instead of using coercions to embed the type of the argument term into the type expected by the predicate, reverse coercions enable us to specialise the type of the predicate to the type of its argument. Once again, predicate specialisation must be performed on all occurrences of a given base type to prevent unsafe applications as examplified with (PCA). However, this global specialisation makes subsequent partial applications possible. If we extend our set of coercions with reversed ones, the *specialised partial coerced application* rule (SPCA) is given by:

$$\frac{\vdash u : \mathcal{P}(d[b] \times d'[b]) \quad \vdash v : d[a] \quad c : a \to b \in \mathcal{K}}{\vdash (\bar{c}'\,u)\,v : \mathcal{P}d'[a]}\,(\textsc{spca}) \tag{10}$$

where $\bar{c}'$ is the reverse of the coercion $c' : \mathcal{P}(d[a] \times d'[a]) \to \mathcal{P}(d[b] \times d'[b])$ built upon c. This rule is again type safe since it guarantees that after applying it, all subsequent applications will be done with base types which are subtypes of a.[13]

We insist on the fact that (SPCA), while ressembling a contravariant subtyping rule—justified by the fact that the retraction $\mathcal{P}m$ of a mono comes indeed from a contravariant functor—, is actually a direct consequence of the covariant property of our subtyping relation. As highlighted in Sect. 1, a contravariant subtyping

[13] As an anonymous reviewer pointed out, the rule (SPCA) is similar in spirit to a typing rule for *bounded polymorphism*, using a predicate of type $\forall b.\mathcal{P}(d[b] \times d'[b])$. The connections between this kind of polymorphism and covariant subtyping may be even deeper and may be worth investigating, but are beyond the scope of this paper. Incidentally, the author also took part in the developpment of another framework using bounded polymorphism and record types [3].

is unable to compare the types $(a \to t) \to a \to t$ and $(b \to t) \to b \to t$, even if a and b are comparable base types, while the covariant subtyping is able to perform the corresponding comparison between $\mathcal{P}(\mathcal{P}a \times a)$ and $\mathcal{P}(\mathcal{P}b \times b)$, and provides coercions in both directions depending on the situation. Notice however that in this case the interpretation of the corresponding reverse coercion is not a monomorphism, but has the dual property of *epimorphism*, as another consequence of Lemma 2.[14]

The last remark to make on the previous typing rules is to observe that (RTCA) is actually a special case of (SPCA) with $d'[a] = d'[b] = 1$.[15] To understand why, consider the coercion $c : d[a] \to d[b]$. Using (SPCA), from $u : \mathcal{P}d[b]$ and $v : d[a]$ we can construct the term $(\bar{c}'u)\,v$ with $[\![\bar{c}']\!] = \mathcal{P}[\![c]\!]$. In the categorical model, the following diagram commutes by general property of the contravariant powerobject functor [21, §IV.1]:

$$\begin{tikzcd}
1 \arrow[r, "{\langle [\![u]\!], [\![v]\!] \rangle}"] & \mathcal{P}[\![d[b]]\!] \times [\![d[a]]\!] \arrow[r, "\mathcal{P}[\![c]\!] \times \mathrm{id}"] \arrow[d, "\mathrm{id} \times [\![c]\!]"] & \mathcal{P}[\![d[a]]\!] \times [\![d[a]]\!] \arrow[d, "\mathrm{ev}_{[\![d[a]]\!]}"] \\
& \mathcal{P}[\![d[b]]\!] \times [\![d[b]]\!] \arrow[r, "\mathrm{ev}_{[\![d[b]]\!]}"] & \Omega
\end{tikzcd}$$

In other words, $[\![(\bar{c}'u)v]\!] = [\![u(cv)]\!]$. This semantic result licenses the introduction on the syntactic side of the following coerced equational rule:

$$\frac{\vdash u : \mathcal{P}d[b] \qquad \vdash v : d[a] \qquad c : d[a] \to d[b]}{\vdash (\bar{c}'u)v = u(cv) : \mathcal{P}1}\,(\textsc{ceq}) \tag{11}$$

It is then clear that with (CEQ), the rule (SPCA) entails the rule (RTCA). We can even extend the rule (CEQ) to a more general form for partial application, provided that we take care of all the details in order to preserve type safety. The resulting rule, call it (PCEQ), is given by:

$$\frac{\vdash u : \mathcal{P}(d[b] \times d'[b]) \qquad \vdash v : d[a] \qquad c : a \to b}{\vdash (\bar{c}'u)v = \bar{c}''(u(c'''v)) : \mathcal{P}d'[a]}\,(\textsc{pceq}) \tag{12}$$

where c' is as before, and c'' and c''' are the coercions $\mathcal{P}d'[a] \to \mathcal{P}d'[b]$ and $d[a] \to d[b]$ built upon c. It captures the previous rule when $d'[a] = d'[b] = 1$, by noticing that c'' and its reverse are then the coercion $\mathcal{P}1 \to \mathcal{P}1$, an identity which can be safely removed from the term. Moreover, the right-hand side of the equality does not fall under the forbidden terms of Definition 9. To complete the construction of $C\Sigma$, we shall add the rules (SPCA) and (PCEQ) to the first batch of rules introduced in Sect. 2.

[14] For this reason and despite the name of "coercion", reverse coercions must not be part of $\mathcal{K}$. Their use is only licensed by explicit involvement of the bar notation in the typing rules.

[15] This fact is more precisely verified under the assumption of interpreting terms up to the isomorphism $A \times 1 \cong A$, cf. footnote 10.

5 Understanding Predicate Covariance

The aim of this section is to provide further intuition on covariance for predicates, as the formalisms of the previous sections have been kept rather abstract. A few more results on the interactions between predicates and logical connectives will also be stated. But for now, let us start with this simple question: if $u : \mathcal{P}\alpha$ is a predicate constant and $c : \mathcal{P}\alpha \to \mathcal{P}\beta$ is a coercion, what does the predicate $cu : \mathcal{P}\beta$ actually describe? To answer it, we turn once again to the categorical model of $C\Sigma$. If c' is the coercion for $\alpha \to \beta$, we have $[\![cu]\!] = \exists[\![c']\!] \circ [\![u]\!]$ by (5) and (6). Thus, we need to explain how the covariant powerobject functor $\mathcal{P}^+$ works in $\mathcal{C}$.

Let $m : A \rightarrowtail B$ be a mono in $\mathcal{C}$. As the powerobject $\mathcal{P}A$ is an internal representation of the set $\mathrm{Sub}(A)$, it will be sometimes convenient to study the external function $\exists_m : \mathrm{Sub}(A) \to \mathrm{Sub}(B)$ instead of $\exists m$ in order to derive internal properties as an application of the Yoneda lemma (see [21, §IV.9] for details). To give even more intuition, suppose in this paragraph only that A and B are sets. Then $\mathrm{Sub}(A)$ and $\mathrm{Sub}(B)$ are their respective powersets, and if $U \in \mathrm{Sub}(A)$, then $\exists_m(U) = \{m(x) \mid x \in U\}$. Besides, if we further assume that the injective function m actually stands for the inclusion $A \subseteq B$, then $\exists_m(U)$ is U itself viewed as a subset of B. The category Set has $\{0, 1\}$ as subobject classifier, and the characteristic χ_U^A of U in A is the classical one, with $\chi_U^A(x) = 1$ if $x \in U$, and $\chi_U^A(x) = 0$ otherwise. As there is a one-to-one correspondence between subsets of A and characteristic functions on A, $\exists_m$ induces a map sending each function χ_U^A to χ_U^B, and it is clear that for all $x \in A$, $\chi_U^A(x) = \chi_U^B(x)$.

Now, abstracting over these set-theoretic considerations, we can generalise some observed properties to any topos $\mathcal{C}$. For any object X, define the morphism $\mathrm{true}_X : X \to \Omega$ as the composite $\top \circ\, !_X$. By definition of the subobject classifier, any subobject $k : U \rightarrowtail A$ is classified by some $f : A \to \Omega$, such that $fk = \mathrm{true}_U$. The following result appears in [21, Prop. 1, §IV.3] with slight differences in notation:

Proposition 2. *Let $g : B \to \Omega$ be the map such that* $\mathrm{name}(g) = \exists m \circ \mathrm{name}(f)$. *Then, g classifies the subobject $mk : U \rightarrowtail B$.*

In other words, mk is the pullback of $\top$ along g, which also means that g shares the same conditions as f to be evaluated to true. In $C\Sigma$, this means that, being given terms $u : \mathcal{P}\alpha$, $v : \beta$ and the coercion $c : \alpha \to \beta$, the term $((\mathcal{P}c)u)v : \mathcal{P}1$, as a logical formula, is true if and on only if there is a term $w : \alpha$ such that $v = cw$ and uw is true.[16] It is legitimate to ask whether the same result holds for conditions of falsity. We shall investigate this question from an external point of view, and study several results related to logical connectives in the way to answer it.

[16] This property, grounded in topos theory, ignores the requirements of type safety as given in Definition 9, according to which the term $((\mathcal{P}c)u)v$ cannot be typed. It shows nonetheless that terms of the form given in the first conditions are not necessary from a truth-theoretical point of view, since it brings "true" application of a coerced form of a predicate u back to a direct application of u.

Recall that for any X, $\langle \mathrm{Sub}(X), \leq, 0, X, \cup, \cap, \Rightarrow\rangle$ is a Heyting algebra (see e.g. [12, §8.3]), where $\cup$, $\cap$ and $\Rightarrow$ are external counterparts to $\wedge_X$, $\vee_X$ and $\Rightarrow_X$ introduced in Sect. 3. If $U \in \mathrm{Sub}(X)$, write $\overline{U}$ for the pseudo-complement $U \Rightarrow 0$. Being given any mono $m : A \rightarrowtail B$, we would like to know whether these logical morphisms, as well as the quantifiers $\exists_{A,X}, \forall_{A,X} : \mathcal{P}(A \times X) \to \mathcal{P}X$, are preserved by the predicate subtyping $\exists m$. The following proposition states a stronger property for some of these connectives: they are actually natural transformations from $\mathcal{P}^+$ to itself.

Proposition 3. *The transformations $\wedge_A$, $\vee_A$ are natural in A, and $\exists_{A,X}$ is natural in A and X.*

Due to the lack of space, we omit proofs of this proposition and of the following results below. The naturality of conjunction, disjunction and existential quantifier amounts to say that $C\Sigma$ can be completed with new equational rules describing the good interaction of these connectives with subtyping coercions. For the first two, the equational rules are the following:

$$\begin{gathered}\dfrac{\vdash u : \mathcal{P}\alpha \quad \vdash v : \mathcal{P}\alpha \quad c : \mathcal{P}\alpha \to \mathcal{P}\beta \in \mathcal{K}}{\vdash \wedge^{\beta}(c \times c)\langle u, v\rangle = c\,(\wedge^{\alpha}\langle u, v\rangle) : \mathcal{P}\beta}\,(\wedge\text{-EQ}) \\[2ex] \dfrac{\vdash u : \mathcal{P}\alpha \quad \vdash v : \mathcal{P}\alpha \quad c : \mathcal{P}\alpha \to \mathcal{P}\beta \in \mathcal{K}}{\vdash \vee^{\beta}(c \times c)\langle u, v\rangle = c\,(\vee^{\alpha}\langle u, v\rangle) : \mathcal{P}\beta}\,(\vee\text{-EQ})\end{gathered} \tag{13}$$

As for the existential quantifier, there are two corresponding rules to account for the double naturality:

$$\begin{gathered}\dfrac{\vdash u : \mathcal{P}(\alpha \times \gamma) \quad c' : \mathcal{P}(\alpha \times \gamma) \to (\beta \times \gamma) \in \mathcal{K}}{\vdash \exists^{\beta,\gamma}(c'\,u) = \exists^{\alpha,\gamma} u : \mathcal{P}\gamma}\,(\exists\text{-EQ}_1) \\[2ex] \dfrac{\vdash u : \mathcal{P}(\alpha \times \gamma) \quad c : \mathcal{P}\gamma \to \mathcal{P}\delta \quad c' : \mathcal{P}(\alpha \times \gamma) \to (\alpha \times \delta)}{\vdash \exists^{\alpha,\delta}(c'\,u) = c\,(\exists^{\alpha,\gamma} u) : \mathcal{P}\delta}\,(\exists\text{-EQ}_2)\end{gathered} \tag{14}$$

In terms of truth-theoretical interpretation, these rules mean that coercions interacts well with the truth conditions of conjunctions, disjunctions and existential quantifiers applied to predicates: in each of the equalities above, the left-hand and right-hand sides have the same conditions of truth and falsity.[17]

However, the naturality of logical connectives w.r.t. $\mathcal{P}^+$ does not propagate to the relative pseudo-complement, nor to the universal quantifier. There is a strict entailment between logical formulae using these connectives through the subtyping relation, under the sufficient condition that the codomain of the subtyping coercion contains the disjoint union of the domain with a *non zero* object, where X non zero means $X \ncong 0$. Within the type ontologies used for formal semantics, this condition generally holds: for instance, the types p, h and v from the example in Sect. 4 are such that $p \geq h \vee v$. The key properties are stated below:

[17] Notice that this property applies regardless of the ambient logic, be it classical or intuitionistic.

Proposition 4. *Let $f : A \to B$ be any morphism. If $\overline{\exists_f(A)}$ is non zero, then for all $U, V \in \mathrm{Sub}(A)$ we have the strict inclusion $\exists_f(U \Rightarrow V) < \exists_f(U) \Rightarrow \exists_f(V)$.*

Corollary 1. *If $\overline{\exists_f(A)}$ is non zero, $\exists_f\big(\overline{U}\big) < \overline{\exists_f(U)}$.*

Proposition 5. *Let $m : A \rightarrowtail B$ be a monomorphism and X any object, and suppose $m' = m \times \mathrm{id}_X$. If $\overline{\exists_m(A)}$ is non zero, then for all $U \in \mathrm{Sub}(A \times X)$ we have the strict inclusion $\forall_{\pi_{B,X}}(\exists_{m'}(U)) < \forall_{\pi_{A,X}}(U)$.*

These propositions show in particular that neither the equational rules in (13) nor ($\exists$-EQ$_1$) have counterparts for implication, negation, and universal quantifier on its first parameter. In term of truth conditions, Prop. 4 shows for instance that there are arguments on which the predicate $\Rightarrow^{\beta}(c \times c)\langle u, v\rangle : \mathcal{P}\beta$ can be proved true, whereas $c\,(\Rightarrow^{\alpha}\langle u, v\rangle) : \mathcal{P}\beta$ will be proved false, the arguments in question being those on which $\Rightarrow^{\alpha}\langle u, v\rangle$ fails to have a truth value due to type mismatch—and similar results hold for negation and universal quantifier. However, we have for universal quantifiers the following proposition which is weaker than naturality on the second parameter, but suffices for interaction with coercions:

Proposition 6. *Let $m : X \rightarrowtail Y$ be a monomorphism and A any object. Then, $\forall_{\pi_{A,Y}}\exists_{(\mathrm{id}_A \times m)} = \exists_m \forall_{\pi_{A,X}}$.*

The consequence of the latter result is the following counterpart to ($\exists$-EQ$_2$):

$$\frac{\vdash u : \mathcal{P}(\alpha \times \gamma) \qquad c : \mathcal{P}\gamma \to \mathcal{P}\delta \qquad c' : \mathcal{P}(\alpha \times \gamma) \to (\alpha \times \delta)}{\vdash \forall^{\alpha,\delta}(c'\,u) = c\,(\forall^{\alpha,\gamma}u) : \mathcal{P}\delta}\,(\forall\text{-EQ}) \tag{15}$$

All the previous properties are expected to provide a better understanding of the covariant subtyping. Prop. 2 showed in particular that if $u : \mathcal{P}a$ is a first-order predicate and $c : \mathcal{P}a \to \mathcal{P}b$ a predicate coercion, then $c\,u$ is a predicate which is true on the same entities as u. By Cor. 1, we can affirm that if u is false on some entity, then $c\,u$ is also false on that very entity. However, $c\,u$ can be false on an entity without it even being in the span of u at all, that is, even if it is not of type α. In terms of a lower-level lambda-calculus, $c\,u$ can be expressed by $\lambda x{:}b.\,\exists y{:}a.\,(c'(y) = x) \wedge u(y)$, where $c' : a \to b$ is the coercion that underlies c, that is, $[\![c]\!] = \exists[\![c']\!]$. Thus, the behaviour of $c\,u$ is simply explained: if x is an entity of type β, then either x is not in the image of c', in which case $c\,u\,x$ is false, or there is an antecedent y of x through c' and $c\,u\,x = u\,y$.

It turns out that the complex coercion c can be itself interpreted as an operator $\lambda u.\lambda x.\exists y.\,(c'(y) = x) \wedge u(y)$, which resembles the functor used by Asher in [1, §6.1] to transform a first-order predicate on a dot type to another one whose argument type is an aspect of the initial one. Even if dot type projections are not exactly subtyping relations as intended in this paper—neither it is for Asher, incidentally—, this similitude should not be surprising since both operators carry the same idea of a covariant transformation of predicates. Moreover, the reverse coercion $\bar{c}$ coincides with the usual contravariant subtyping for first-order predicates as the operator $\lambda u \lambda x.u(c'x)$, and as c' is intended to be mono, the composition $\lambda u.\bar{c}(cu)$ amounts to the identity up to isomorphism, which is also a consequence of Lemma 2. Thus, a last rule of term equality can be added:

$$\frac{\vdash u : \mathcal{P}\alpha \qquad c : \mathcal{P}\alpha \to \mathcal{P}\beta}{\vdash u = \bar{c}(c\,u) : \mathcal{P}\alpha}\,(\textsc{retract}) \tag{16}$$

However, for $v : \mathcal{P}\beta$, we have $v \neq c(\bar{c}\,v)$ in general.

6 Conclusion and Future Works

We introduced $C\Sigma$, a general semantic predicate calculus using a constructor $\mathcal{P}$ instead of the traditional functions of codomain t, and completed by a covariant subtyping. The typing rules constraining the use of subtyping coercions ensure a property of type safety which is sufficient for semantic purposes. The general typing rules of $C\Sigma$ are obtained by gathering the rules in (2) and (10), and the term equality rules consist in those given in (3), to which we add the equalities given in (12–16). The typing rule in (9) as well as the term equality rule in (11) are also admissible. Altogether, these rules enable $C\Sigma$ to be as efficient as other proposals for natural language semantics. However, $C\Sigma$ gains in flexibility by its covariant approach of subtyping, which enables us to deal easily with second-order types that pose difficulties to other semantic frameworks.

In the previous section, we generalised reasoning in Set to any topos in such a way that the interpretation of $C\Sigma$ is not restricted to sets—even if toposes have a general “set-like” behaviour. It may be relevant then to ask whether interpreting $C\Sigma$ directly in Set would not have sufficed for our purposes, instead of carrying on with the generalisation we presented. There are actually two reasons for such a move. Firstly, in spite of its specific status amongst toposes (and in mathematics in general), Set has a few restrictive properties which we may want to dismiss, the most important one being the fact that, as a Boolean topos, the internal logic of Set is always classical: we may want to use another topos with intuitionistic logic, for instance. Secondly, some semantic phenomena, such as vagueness [6], suggests that predicate modelling might need to go beyond the set-theoretical basis: if this happens to be the case, we believe that the generalisation we presented offers more flexibility to help accounting for any new proposal.

We ought to add that the genericity of topos theory, as compared to set theory, make the choice of a canonical covariant subtyping coercion more difficult. We may however think of such a canonical coercion as similar to an inclusion in Set, in the sense that a coerced entity is roughly the same entity viewed in another perspective. It may be hard however to systematise this idea in a general topos, unless we introduce an entity constant by a global element for each supertype the entity has, and then choose coercions which preserve the resulting families of global elements. Nevertheless, this issue may be less harmful than it seems, to the extent that covariant coercions ensure at least that two distinct entities remain distinct when coerced, thus preserving the relations of entities w.r.t. each other.

Finally, we may extend the calculus with additional non-subtyping coercions to improve its abilities, in the spirit of the reverse coercions introduced in Sect. 4. The projection maps from product terms π_1 and π_2 have been introduced here as term constructors, but could actually be added as such coercions, provided that we enforce a strong distinction between subtyping coercions and other ones when defining the rules of our calculus. Other possible coercions—whose introduction did unfortunately not fit in those pages although studied by the author—are what we could call *transstructural coercions*, that is, coercions that do not preserve the structure of type constructors. This case includes the introduction of *dot types* [26] as subtypes of products (as proposed in [1,2]), and other transformations such as *type shifts* [24]. Finally, creative uses of language and other transfers of meaning may be handled by more general coercions. Overall, future studies on $C\Sigma$-like frameworks will be devoted to explore these potential extensions of the calculus.

Acknowledgments. The author is grateful to the three anonymous reviewers for their helpful and valuable comments.

References

1. Asher, N.: Lexical Meaning in Context: A Web of Words. Cambridge University Press (2011)
2. Babonnaud, W.: A topos-based approach to building language ontologies. In: Bernardi, R., Kobele, G.M., Pogodalla, S. (eds.) Formal Grammar. 24th International Conference, FG 2019, Riga, Latvia, August 11, 2019, Proceedings. pp. 18–34. Springer, Berlin (2019)
3. Babonnaud, W., de Groote, P.: Lexical selection, coercion, and record types. In: LENLS17: Logic & Engineering of Natural Language Semantics, Online, November 15-17, 2020 (2020)
4. Barendregt, H.P.: The Lambda-Calculus: Its Syntax and Semantics. Elsevier (1984)
5. Berry, G.: Stable models of typed λ-calculi. In: Ausiello, G., Böhm, C. (eds.) Automata, Languages and Programming. pp. 72–89. Springer (1978)
6. Burnett, H., Sutton, P.: Vagueness and natural language semantics. In: Gutzmann, D., Matthewson, L., Meier, C., Rullmann, H., Zimmermann, T.E. (eds.) The Wiley Blackwell Companion to Semnatics. John Wiley & Sons (2020)
7. Cardelli, L.: A semantics of multiple inheritance. In: Kahn, G., MacQueen, D.B., Plotkin, G. (eds.) Semantics of Data Types. pp. 51–67. Springer, Berlin (1984)
8. Cardelli, L.: Structural subtyping and the notion of power type. In: Ferrante, J., Mager, P. (eds.) POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 70–79. Association for Computing Machinery, New York (1988)
9. Castagna, G.: Covariance and contravariance: Conflict without a cause. ACM Transactions on Programming Languages and Systems **17**(3), 431–447 (1995)
10. Chatzikyriakidis, S., Luo, Z.: On the interpretation of common nouns: Types versus predicates. In: Chatzikyriakidis, S., Luo, Z. (eds.) Modern Perspectives in Type-Theoretical Semantics, Studies in Linguistics and Philosophy, vol. 98, pp. 43–70. Springer (2017)

11. Cook, W.R., Hill, W., Canning, P.S.: Inheritance is not subtyping. In: Allen, F.E. (ed.) POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 125–135. Association for Computing Machinery, New York (1990)
12. Goldblatt, R.: Topoi: The Categorial Analysis of Logic, Studies in logic and the foundations of mathematics, vol. 98. North-Holland Publishings (1979)
13. Johnstone, P.T.: Sketches of an Elephant: A Topos Theory Compendium. Oxford University Press (2002)
14. Lambek, J.: From λ-calculus to cartesian closed categories. In: Hindley, J.R., Seldin, J.P. (eds.) To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, pp. 375–402. Academic Press, London (1980)
15. Lambek, J., Scott, P.J.: Introduction to Higher Order Categorical Logic. Cambridge University Press (1986)
16. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems **16**(6), 1811–1841 (1994)
17. Luo, Z.: Coercive subtyping. Journal of Logic and Computation **9**(1), 105–130 (1999)
18. Luo, Z.: Type-theoretical semantics with coercive subtyping. In: Li, N., Lutz, D. (eds.) Proceedings of SALT 20. pp. 38–56 (2010)
19. Luo, Z., Soloviev, S., Xue, T.: Coercive subtyping: Theory and implementation. Information and Computation **223**, 18–42 (2013)
20. MacLane, S.: Categories for the Working Mathematician. Springer (1971)
21. MacLane, S., Moerdijk, I.: Sheaves in Geometry and Logic: A First Introduction to Topos Theory. Springer, New York (1992), corr. 2nd edition 1994
22. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences **17**(3), 348–375 (1978)
23. Montague, R.: The proper treatment of quantification in ordinary english. In: Suppes, P., Moravcsik, J., Hintikka, J. (eds.) Approaches to Natural Language, pp. 221–242. Reidel, Dordrecht (1973)
24. Partee, B.H.: Noun phrase interpretation and type-shifting principles. In: Groenendijk, J., de Jongh, D., Stokhof, M. (eds.) Studies in discourse representation theory and the theory of generalized quantifiers, pp. 115–143. De Gruyter (1986)
25. Pierce, B.C.: Basic Category Theory for Computer Scientists. The MIT Press, Cambridge (1991)
26. Pustejovsky, J.: The Generative Lexicon. MIT Press, Cambridge (1995)
27. Retoré, C.: The montagovian generative lexicon ΛTy_n: a type theoretical framework for natural language semantics. In: Matthes, R., Schubert, A. (eds.) Proceedings of the 19th International Conference on Types for Proofs and Programs. LIPICS, vol. 26, pp. 202–229 (2014)
28. Reynolds, J.C.: Using category theory to design implicit conversions and generic operators. In: Jones, N.D. (ed.) Semantics-Directed Compiler Generation. pp. 211–258. Springer (1980)
29. Saba, W.S.: Language and its commonsense: Where formal semantics went wrong, and where it can (and should) go. Journal of Knowledge Structures and Systems **1**(1), 40–62 (2020)
30. Seely, R.A.G.: Locally cartesian closed categories and type theory. Mathematical Proceedings of the Cambridge Philosophical Society **95**(1), 33–48 (1984)
31. Smyth, M.B., Plotkin, G.: The category-theoretic solution of recursive domain equations. SIAM Journal on Computing **11**(4), 761–783 (1982)
32. Sommers, F.: The ordinary language tree. Mind **68**(2), 160–185 (1959)
33. Sommers, F.: Type and ontology. The Philosophical Review **72**(3), 327–363 (1963)