Bi-orthogonality

Paul Downen

August, 2014

1 Context

I originally wrote these notes in the summer of 2014 when I was trying to work out the strong normalization of a symmetric sequent calculus with user-defined data and codata types for an arbitrary class of evaluation strategies. In particular, I had finished the paper "The duality of construction" [2] and was working on how to extend it to inductive and coinductive types and programs, which would later become the paper "Structures for structural recursion" [6].

Here, I was working with just the system with basic (non-recursive) type constructors, and wanted to prove that was strongly normalizing. Although these notes do not have a formal introduction to that system, you can find the description in [2] which defines the syntax, rewriting theory, and equational theory.

In order to approach the strong normalization problem for [2], I was heavily inspired by Guillaume Munch-Maccagnoni's paper "Focalisation and classical realisability" [9], which studies a similar system based on a symmetric linear sequent calculus. You can see some of the inspiration inside of these notes which rely on some important ideas introduced by Guillaume, notably the idea of "generation" by the (co)values of a type.

In order to preserve the integrity of these notes (and to show my state of mind when I was learning this (bi)orthogonality technique for the first time), I have not modified any of the text that follows from its original form in 2014. In order to ensure the validity of these notes, I have tried keep only the parts that were ultimately sensible. For various complete demonstrations of adequacy of the model in a way that works for a large class of evaluation strategies and languages, see the papers (and appendices) done in collaboration with some of my coauthors Ariola, Johnson-Freyd, and Ghilezan [6, 7, 5, 3, 4, 1, 8].

2 Orthogonality

We're going to look at a syntactic model of understanding computation in the sequent calculus in terms of *orthogonality*. The model hinges on some judgment that isolates commands of the sequent calculus that we deem to be valid execution states for

our purposes. In other words, we isolate a set of commands that can *run*. For example, if we are trying to execute well-typed programs, we might say that the command $\langle \mathsf{true} | \tilde{\mu}[\mathsf{true} \Rightarrow c_1 \mid \mathsf{false} \Rightarrow c_2] \rangle$ runs when c_1 and c_2 do, but a command like $\langle \mathsf{true} | 1 \cdot \mathsf{tp} \rangle$ does not run since it is nonsensical to apply the boolean value true to a number. The idea that only some commands in the untyped sequent calculus go well when run can be specified by a particular subset of all possible commands, $\perp\!\!\!\perp$:

$$\perp\!\!\!\perp \subseteq \mathit{Command}$$

Another way to think of the set $\perp\!\!\!\perp$ is that it specifies pairs of terms and coterms *run together* and can fruitfully cooperate with one another. We may write the fact that a term v goes with a coterm e (and vice versa) as the shorthand $v \perp\!\!\!\perp e$:

$$v \perp\!\!\!\perp e \triangleq \langle v | e \rangle \in \perp\!\!\!\perp$$

Having made some particular choice for $\perp\!\!\!\perp$, we can define two important closure operations that find *everything* that runs with a set of terms or coterms. Starting from an initial set of plausibly "well-behaved" (co)terms of the untyped syntax, $Term_\emptyset$ and $CoTerm_\emptyset$, we define the orthogonal of a chosen set of (co)terms:[1]

- For any particular subset of terms $Term_A$, its orthogonal is the set of coterms that runs with everything in $Term_A$:

$$Term_A^\perp = \{e \in CoTerm_\emptyset \mid \forall v \in Term_A, v \perp\!\!\!\perp e\}$$

- For any particular subset of coterms $CoTerm_A$, its orthogonal is the set of terms that runs with everything in $CoTerm_A$:

$$CoTerm_A^\top = \{v \in Term_\emptyset \mid \forall e \in CoTerm_A, v \perp\!\!\!\perp e\}$$

As with the running set $\perp\!\!\!\perp$, we also have some choice in how to define the initial sets $Term_\emptyset$ and $CoTerm_\emptyset$. This flexibility allows us to capture some additional criteria in the orthogonality operations, and avoid letting in any unwanted miscreants. For instance, Munch-Maccagnoni [9] takes all closed (co)terms to be the initial sets to model classical realizability. If instead we are interested in strong normalization, then we can start with the sets of all strongly normalizing (co)terms as the initial sets. For other purposes, we may not require any additional criteria in the orthogonality operations, so the entire untyped syntax of (co)terms serves sufficiently well as a default choice.

Example 1. For example, suppose we are trying to reason about execution of well-typed programs. In other words, we want to model type-safe, top-level reduction in the sequent calculus. We would need to design our running set of commands that contains all valid states of type-safe execution, and excludes stuck states that are caused by type errors. For example, $\perp\!\!\!\perp$ would not include commands like $\langle \mathsf{true} | 1 \cdot \mathsf{tp} \rangle$, $\langle \mu(x \cdot \alpha.c) | \tilde{\mu}[(x, y).c] \rangle$, $\langle \iota_1(1) | \tilde{\mu}[(x, y).c] \rangle$, and $\langle \mu(x \cdot \alpha.c) | \pi_1[\mathsf{tp}] \rangle$, since they are all stuck

[1] Traditionally, these operations are referred to as either both $\_^\perp$ or $\_^\top$, but here we use the two different symbols to help disambiguate between terms and coterms.

on irrecoverable miscommunications like missing case analysis or data/codata mismatches. Instead, ⫫ would include valid states where we may not have enough information to take the next step, but execution could potentially continue if we learn more. These would be states where we are stuck on a free variable, like $\langle f | 1 \cdot \mathsf{tp} \rangle$ or $\langle z | \tilde{\mu}[(x, y).c] \rangle$, or on a free covariable, like $\langle \mathsf{true} | \alpha \rangle$ or $\langle \mu(x \cdot \alpha.c) | \beta \rangle$. To complete the running set ⫫, we should also ensure that a command that eventually reaches a valid state in some number of steps is also valid. That is, if c' is in ⫫ and $c \twoheadrightarrow c'$ then c is also in ⫫. This is commonly referred to as "closure under expansion" or "closure under backward reduction" and is found in similar models of program evaluation.

Now, we can consider what the orthogonality operations mean for the above description of our choice of the running set. We can begin with every (co)term in the initial sets $Term_\emptyset$ and $CoTerm_\emptyset$. For instance, $\{()\}^\perp$ selects every coterm that runs with the term (). This would include coterms like $\tilde{\mu}[().c]$ and $\tilde{\mu}\_.c$ for commands c that are in ⫫, because they both reduce to the running command c in one step. However, $\{()\}^\perp$ would not include coterms like $1 \cdot \mathsf{tp}$ or $\tilde{\mu}[\iota_1(x).c \mid \iota_2(y).c']$ since commands like $\langle () | 1 \cdot \mathsf{tp} \rangle$ are stuck on an irrecoverable type error and are excluded from ⫫. As another example, $\{\}^\perp$ would instead select *every* coterm, since the condition $\forall v \in \{\}, v \between e$ is vacuously true for any e. Note that this fact about $\{\}^\perp$ (or $\{\}^\top$) holds regardless of the definition of ⫫, so that the orthogonal of the empty set always gives back every term in $Term_\emptyset$ (or coterm in $CoTerm_\emptyset$). *End Example* 1.

While we often have a particular purpose in mind (like the above example of type-safe execution), we can temporarily ignore the particular details and just leave ⫫ abstract for the time being. As we will see, orthogonality already gives us some interesting structure independent of our choice in ⫫ and without knowing anything about the particularities of terms and coterms.

3 Orthogonality and intuitionistic negation

As an operation on sets of (co)terms, orthogonality has some inherently negating behavior: it generates a collection of consumers (terms) from a collection of consumers (coterms), and vice versa. We will see that this simple intuition reveals a fundamental connection between the orthogonality of (co)terms and negation in an intuitionistic setting. As it turns out, basic properties of intuitionistic negation, both from a logical and computational perspective, are shared with the orthogonality operation. Furthermore, classical but non-intuitionistic properties of negation are invalid for the orthogonality model.

To review, a logical presentation of negation in intuitionistic natural deduction can be given by the typical encoding in terms of implication and falsehood: $\neg A = A \to \bot$. The inference rules for connectives like $\to$, $\wedge$, and $\vee$ are defined as usual (see Frank Pfenning's OPLSS lectures, for example). Additionally, the encoding of negation is summarized by the following two inference rules for the $\neg$ connective that are derived

from the rules for $\to$:

$$\dfrac{\begin{array}{c}\overline{A}^{\,x}\\ \vdots\\ \bot\end{array}}{\neg A}\ \neg I^x \qquad\qquad \dfrac{\neg A \quad A}{\bot}\ \neg E$$

Remark 1. For the record, $\bot$ would typically be given the following inference rules:

$$\text{no } \bot I \text{ rules} \qquad\qquad \dfrac{\bot}{C}\ \bot E$$

For our purposes here, we will not actually need to use the $\bot$ elimination rule $\bot E$. Rather, we could have defined negation without any reference to falsehood. However, the use of $\bot$ allows us to use the encoding $\neg A = A \to \bot$, and helps to clarify the connections between the logic, programs, and orthogonality. *End Remark* 1.

Using the rules for negation given above, we can give some schematic proofs involving negation and implication that hold in intuitionistic logic. For example, we have double negation introduction, $A \to (\neg\neg A)$,

$$\dfrac{\dfrac{\overline{\neg A}^{\,k} \quad \overline{A}^{\,x}}{\bot}\ \neg E}{\neg\neg A}\ \neg I^k \qquad \dfrac{\neg\neg A}{A \to (\neg\neg A)}\ \to I^x$$

the contrapositive of a function, $(A \to B) \to (\neg B \to \neg A)$,

$$\dfrac{\dfrac{\overline{\neg B}^{\,k} \quad \dfrac{\overline{A \to B}^{\,f} \quad \overline{A}^{\,x}}{B}\ \to E}{\bot}\ \neg E}{\neg A}\ \neg I^x \qquad \dfrac{\neg A}{(\neg B) \to (\neg A)}\ \to I^k \qquad \dfrac{(\neg B) \to (\neg A)}{(A \to B) \to ((\neg B) \to (\neg A))}\ \to I^f$$

and triple negation elimination, $(\neg\neg\neg A) \to (\neg A)$,

$$\dfrac{\dfrac{\overline{\neg\neg\neg A}^{\,w} \quad \dfrac{\dfrac{\overline{\neg A}^{\,k} \quad \overline{A}^{\,x}}{\bot}\ \neg E}{\neg\neg A}\ \neg I^k}{\bot}\ \neg E}{\neg A}\ \neg I^x \qquad \dfrac{\neg A}{(\neg\neg\neg A) \to (\neg A)}\ \to I^w$$

Each of these proofs can also be written as a term in the simply-typed λ-calculus by taking the encoding $\neg A = A \to \bot$ and letting $\bot$ be some unknown type.

$$DNI : A \to \neg\neg A = \lambda x : A.\lambda k : \neg A.k\ x$$
$$ContraPos : (A \to B) \to (\neg B \to \neg A) = \lambda f : A \to B.\lambda k : \neg B.\lambda x : A.k\ (f\ x)$$
$$TNE : \neg\neg\neg A \to \neg A = \lambda w : \neg\neg\neg A.\lambda x : A.w\ (\lambda k : \neg A.k\ x)$$

Remark 2. The three terms DNI, $ContraPos$, and TNE have an important status for pure functional programming in languages like Haskell. In particular, they give us a definition of the *continuation monad* over the return type $\bot$, $Cont\ A = \neg\neg A$. Double negation introduction, DNI, is the *return* (*a.k.a* unit) function. Triple negation elimination, TNE, is the *join* function from $Cont\ (Cont\ A) \to Cont\ A$ with a more general type. And $ContraPos$ is the contravariant mapping function for the underlying $\neg$ functor. We can get the $Functor$ mapping function $fmap$ by taking $ContraPos$ twice on a function, $fmap\ f = ContraPos\ (ContraPos\ f)$. *End Remark* 2.

As it turns out, these three properties of double negation introduction, contrapositive mapping, and triple negation elimination correspond to similar properties of orthogonality. In particular, the two orthogonality operation on sets of (co)terms takes on the role of negation, and the subset relation takes on the role of implication. With this correspondence in mind, we get the following three intuitionistic orthogonality properties:

Property 1. *1. If* $Term_A \subseteq Term_\emptyset$ *then* $Term_A \subseteq Term_A^{\bot\top}$ *and if* $CoTerm_A \subseteq CoTerm_\emptyset$ *then* $CoTerm_A \subseteq CoTerm_A^{\top\bot}$.

2. If $Term_A \subseteq Term_B$ *then* $Term_B^\bot \subseteq Term_A^\bot$ *and if* $CoTerm_A \subseteq CoTerm_B$ *then* $CoTerm_B^\top \subseteq CoTerm_A^\top$.

3. $Term_A^{\bot\top\bot} = Term_A^\bot$ *and* $CoTerm_A^{\top\bot\top} = CoTerm_A^\top$.

Proof. For simplicity, we focus only on the properties for sets of terms. The corresponding properties on sets of coterms follow analogously.

1. By the definition of orthogonality, we know that every coterm in $Term_A^\bot$ runs with every term in $Term_A$. Conversely, that also means that every term in $Term_A$ runs with every coterm in $Term_A^\bot$. Therefore, every term in $Term_A$ must also be a member of $Term_A^{\bot\top}$, since it is one of the terms in $Term_\emptyset$ that runs with everything in $Term_A^\bot$.

2. Suppose that e_B is a coterm in $Term_B^\bot$. By the definition of orthogonality, we know that e_B is in $CoTerm_\emptyset$ and runs with every term in $Term_B$. But since everything in $Term_A$ is included in $Term_B$, then e_B also runs with everything in $Term_A$. Therefore, e_B must also be a member of $Term_A^\bot$.

3. First, we get the fact that $Term_A^\bot \subseteq Term_A^{\bot\top\bot}$ as a consequence of Property 1.1 on coterms. Second, we get the fact that $Term_A^{\bot\top\bot} \subseteq Term_A^\bot$ from Properties 1.1 and 1.2. More specifically, we may introduce the double orthogonality with 1.1, $Term_A \subseteq Term_A^{\bot\top}$, and arrive at the contrapositive with 1.2, $Term_A^{\bot\top\bot} \subseteq Term_A^\bot$. Therefore, the set $Term_A^{\bot\top\bot}$ must be equal to $Term_A^\bot$. □

It is important to point out that when demonstrating the above three properties, we never needed to know anything about what makes up our initial sets of (co)terms and how we defined the running set *runs*. This means that Properties 1.1, 1.2, and 1.3 *must* hold for *any* choice of initial (co)terms and *any* choice of $\perp\!\!\!\perp$. No matter what choices me make, we get to use these intuitionistic reasoning principles when working with orthogonality.

Example 2. One difference between intuitionistic negation versus negation in classical logic is that *double negation elimination*, *i.e.,* $(\neg\neg A) \to A$, is not assumed to hold generically for any A in the intuitionistic setting. To see why "double orthogonal elimination", *i.e.,* $Term_A^{\perp\top} \subseteq Term_A$, cannot hold, let's return to our example of type-safe execution from Example 1. For the moment, let's assume a call-by-value evaluation order for the sequent calculus, so that every coterm is a covalue and so $\langle \mu\alpha.c | e \rangle \to c\{e/\alpha\}$ for any e. Recall that the orthogonal of the empty set of terms, $\{\}^{\perp}$, is the set of all coterms. Now, suppose that the command c is in the running set $\perp\!\!\!\perp$. Notice that $\langle \mu\_.c | e \rangle \to c$, so that for an arbitrary coterm e, the command $\langle \mu\_.c | e \rangle$ reduces in one step to a command in the running set. This means that the term $\mu\_.c$ must be in the double-orthogonal of the empty set, $\{\}^{\perp\top}$. But this also means that we've run into a situation where the double orthogonal of a set of terms (namely the empty set) includes terms that weren't originally there. Therefore, we can't say in general say that the double orthogonal of a set of (co)terms the same as the original set.

Since taking the double orthogonal of a set of (co)terms can introduce new (co)terms, we can view it as a closure operation. Because taking the orthogonal thrice gives the same thing as just once (Property 1.3), flipping back and forth more than twice in this way is redundant: $Term_A^{\perp\top\perp\top} = Term_A^{\perp\top}$ and $CoTerm_A^{\top\perp\top\perp} = Coterm_A^{\top\perp}$. In this regard, $Term_A^{\perp\top}$ can be seen as the completion of $Term_A$ with respect to the running set $\perp\!\!\!\perp$, and $CoTerm_A^{\top\perp}$ is the completion of $CoTerm_A$. *End Example* 2.

By adding more connectives into the mix, like conjunction ($\wedge$) and disjunction ($\vee$), we get additional properties of intuitionistic negation. In particular, we have the *de Morgan law* that allows us to distribute negation over conjunction in both directions: $(\neg(A \vee B)) \leftrightarrow ((\neg A) \wedge (\neg B))$. This law is provable with the rules of our intuitionistic natural deduction as two implications:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg(A \vee B)}\,k \quad \dfrac{\dfrac{}{A}\,x}{A \vee B}\,\vee I_1}{\bot}\,\neg E}{\neg A}\,\neg I^x \quad \dfrac{\dfrac{\dfrac{}{\neg(A \vee B)}\,k \quad \dfrac{\dfrac{}{B}\,y}{A \vee B}\,\vee I_2}{\bot}\,\neg E}{\neg B}\,\neg I^y}{(\neg A) \wedge (\neg B)}\,\wedge I}{(\neg(A \vee B)) \to ((\neg A) \wedge (\neg B))}\,\to I^k$$

$$\dfrac{\dfrac{\dfrac{}{A \vee B}\,x \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{(\neg A) \wedge (\neg B)}\,k}{\neg A}\,\wedge E_1 \quad \dfrac{}{A}\,y}{\bot}\,\neg E \qquad \dfrac{\dfrac{\dfrac{}{(\neg A) \wedge (\neg B)}\,k}{\neg B}\,\wedge E_2 \quad \dfrac{}{B}\,z}{\bot}\,\neg E}{\bot}\,\vee E^{y,z}}{\bot}\,\neg I^x}{((\neg A) \wedge (\neg B)) \to (\neg(A \vee B))}\,\to I^k$$

We can also write down terms in the λ-calculus extended with pairs and sums that have

similar types, expressing the above de Morgan law as two functions:

$$PairNeg : (\neg A) * (\neg B) \to \neg(A + B)$$
$$PairNeg = \lambda k.\lambda x.\, \mathbf{case}\, x\, \mathbf{of}\, \iota_1\, y \Rightarrow \pi_1\, k\, y \mid \iota_2\, z \Rightarrow \pi_2\, k\, z$$

$$NegSum : \neg(A + B) \to (\neg A) * (\neg B)$$
$$NegSum = \lambda k.((\lambda x.k\, (\iota_1\, x)), (\lambda y.k\, (\iota_2\, y)))$$

There is another de Morgan law for distributing a negation over a conjunction in both directions: $(\neg(A \wedge B)) \leftrightarrow ((\neg A) \vee (\neg B))$. However, in an intuitionistic setting, this law does *not* hold both ways. In particular, we can only assume that the right-to-left direction of this law holds in general: $(\neg(A \wedge B)) \leftarrow ((\neg A) \vee (\neg B))$. This implication is provable in intuitionistic natural deduction:

$$\dfrac{\dfrac{\overline{(\neg A) \vee (\neg B)}\ k \qquad \dfrac{\dfrac{\overline{\neg A}\ q \qquad \dfrac{\overline{A \wedge B}\ x}{A}\wedge E_1}{\bot}\neg E \qquad \dfrac{\dfrac{\overline{\neg B}\ r \qquad \dfrac{\overline{A \wedge B}\ x}{B}\wedge E_2}{\bot}\neg E}{\bot}}{\bot}\vee E^{q,r}}{\neg(A \wedge B)} \to I^x}{((\neg A) \vee (\neg B)) \to (\neg(A \wedge B))} \to I^k$$

And we also have an (extended) λ-calculus function that corresponds to the one direction of the law.

$$SumNeg : (\neg A) + (\neg B) \to \neg(A * B)$$
$$SumNeg = \lambda k.\lambda x.\, \mathbf{case}\, k\, \mathbf{of}\, \iota_1\, q \Rightarrow q\, (\pi_1\, x) \mid \iota_2\, r \Rightarrow r\, (\pi_2\, x)$$

We are unable to write the inverse function, $NegPair : \neg(A * B) \to (\neg A) + (\neg B)$, since we don't know up front which of $\neg A$ or $\neg B$ to return in general.

Just like before, these three de Morgan laws correspond to similar properties of orthogonality. The union and intersection operations on sets take on the roles of conjunction and disjunction. This gives us the following intuitionistic de Morgan orthogonality properties.

Property 2. *1.* $(Term_A \cup Term_B)^\perp = (Term_A^\perp \cap Term_B^\perp)$ *and* $(CoTerm_A \cup CoTerm_B)^\top = (CoTerm_A^\top \cap CoTerm_B^\perp)$.

2. $(Term_A \cap Term_B)^\perp \supseteq (Term_A^\perp \cup Term_B^\perp)$ *and* $(CoTerm_A \cap CoTerm_B)^\top \supseteq (CoTerm_A^\top \cup CoTerm_B^\top)$

Proof. Again, for simplicity we will focus attention only on the properties for sets of terms. Coterms follow analogously.

1. First, we show that $(Term_A \cup Term_B)^\perp \subseteq (Term_A^\perp \cap Term_B^\perp)$. Suppose that e is some coterm in $(Term_A \cup Term_B)^\perp$, so that e is in $CoTerm_\emptyset$ and runs with both everything in $Term_A$ and everything in $Term_B$. This means that it is independently a member of both $Term_A^\perp$ and of $Term_B^\perp$, forcing e in $Term_A^\perp \cap Term_B^\perp$.

Second, we show that $(Term_A^\perp \cap Term_B^\perp) \subseteq (Term_A \cup Term_B)^\perp$. Suppose that e is some coterm in $Term_A^\perp \cap Term_B^\perp$. Then we know that e must be in both $Term_A^\perp$ and $Term_B^\perp$, so it is in $CoTerm_\emptyset$ and runs with everything in $Term_A$ and $Term_B$. But that means that e must also be in $(Term_A \cup Term_B)^\perp$.

2. Suppose that e is some coterm in $Term_A^\perp \cup Term_B^\perp$, so that we know that e is in $CoTerm_\emptyset$ and runs with either everything in $Term_A$ or with everything in $Term_B$ (or both). Since every term v in $Term_A \cap Term_B$ must come from both $Term_A$ or $Term_B$, then we know that e must run with v. Therefore, e is also in $(Term_A \cap Term_B)^\perp$. □

Again, take notice that the de Morgan properties of orthogonality don't depend on what (co)terms we're working with, nor what the running set $\perp\!\!\!\perp$ happens to be. They are general laws that come out from the definition of the orthogonality operations and basic operations and relations on sets.

Example 3. To see why the de Morgan Property 2.2 does not go both ways like Property 2.1, let's return again to type-safe execution from Example 1. Now, suppose we begin with two sets of terms, $Term_A = \{True, ()\}$ and $Term_B = \{False, ()\}$, so that their intersection is $Term_A \cap Term_B = \{()\}$. The orthogonal of this intersection is $(Term_A \cap Term_B)^\perp = \{e \mid () \perp\!\!\!\perp e\}$. By our description of the running set from Example 1, given a command c in $\perp\!\!\!\perp$ we have that the coterm $\tilde{\mu}[().c]$ runs with () because $\langle ()|\tilde{\mu}[().c]\rangle \to c$, so the $\tilde{\mu}[().c]$ is in $(Term_A \cap Term_B)^\perp$. However, both $\langle \mathsf{true}|\tilde{\mu}[().c]\rangle$ and $\langle \mathsf{false}|\tilde{\mu}[().c]\rangle$ are not in the running set since they are stuck on a type error. This means that the coterm $\tilde{\mu}[().c]$ is in neither $Term_A^\perp$ nor $Term_B^\perp$, since $\tilde{\mu}[().c]$ fails to run with some of the terms in $Term_A$ and $Term_B$. Therefore, we've stumbled onto a situation where a coterm is in the orthogonal set of an intersection, but does not come from the union of the separate orthogonal sets. In other words, taking the orthogonal of an intersection between two sets of terms permits *more* possible coterms than just forming the orthogonal sets in isolation and putting them together. *End Example* 3.

Aside 1. All of the above properties of orthogonality (except for Proposition 2.2 which I found from the analogy to de Morgan laws for intuitionistic negation) appear in [9] under the proposition for the "Basic properties of the orthogonal." There is only one more property listed under that header that talks about *behaviors*. A set of terms $Term_A$ is a *behavior* when it has the form $Term_A = CoTerm_B^\top$ for some set $CoTerm_B$, and analogously a set of coterms $CoTerm_A$ is a behavior when it has the form $CoTerm_A = Term_B^\perp$. The property then states that

Property 3. *$Term_A$ is a behavior if and only if $Term_A = Term_A^{\perp\top}$, and $CoTerm_A$ is a behavior if and only if $CoTerm_A = CoTerm_A^{\top\perp}$.*

I wonder if this also has some interesting correspondence to known facts about negation in other settings, like the analogy with intuitionistic negation and de Morgan laws above. *End Aside* 1.

4 Modeling constructions

Having looked at orthogonality and some of it's basic properties in detail, let's now see what it can tell us about computation. In particular, we can use orthogonality as the heart of a relational term model (in the general style of logical relations or parametricity model) for classical programs in the sequent calculus.

Every type in the sequent calculus carves out a subset of both terms and coterms that belong to that type. We can then give the meaning of a type in the model, $[\![A]\!]$, by both a set of terms and a set of coterms. We will refer to the meaning of a type A as the *semantic type* $[\![A]\!]$ which defines a set of terms and coterms:

$$[\![A]\!] : \mathcal{P}(Term_\emptyset) \times \mathcal{P}(CoTerm_\emptyset)$$
$$Term[\![A]\!] \subseteq Term_\emptyset \qquad CoTerm[\![A]\!] \subseteq CoTerm_\emptyset$$

The way that terms and coterms of a type interact is through the *cut* rule, which lets us form a command from any term and coterm from the same type. When looking at types from the perspective of the model, the cut rule corresponds to the intuitive property that we should be able to run any term of a type with any coterm of the same type. In other words, we should expect that the following holds for a type A:

$$Term[\![A]\!] = CoTerm[\![A]\!]^\top \qquad CoTerm[\![A]\!] = Term[\![A]\!]^\perp$$

That is, the terms and coterms of a type should be *mutually orthogonal* to one another.

Furthermore, when we begin to consider the types created by user-defined (co)data types, we will see type variables appear from the schematic nature of declarations:

data $\mathsf{F}(\vec{X})$ **where**	**codata** $\mathsf{G}(\vec{X})$ **where**
$\mathsf{K}_1 : \quad \vec{A_1} \vdash \mathsf{F}(\vec{X}) \mid \vec{B_1}$	$\mathsf{H}_1 : \quad \vec{A_1} \mid \mathsf{G}(\vec{X}) \vdash \vec{B_1}$
$\ldots$	$\ldots$
$\mathsf{K}_n : \quad \vec{A_n} \vdash \mathsf{F}(\vec{X}) \mid \vec{B_n}$	$\mathsf{H}_n : \quad \vec{A_n} \mid \mathsf{G}(\vec{X}) \vdash \vec{B_n}$

For the purposes of analyzing the models for these schematic declarations, it is helpful to define the model for *open* types which may contain free type variables, as found in the types $\vec{A}$ and $\vec{B}$ above. To accommodate this generalization, we extend the translation of types to also take a mapping, σ, from type variables to sets of terms and coterms. Then we can define the meaning of a free type variable with respect to a mapping σ as:

$$[\![X]\!]\sigma \triangleq \sigma(X)$$
$$Term[\![X]\!]\sigma \triangleq Term(\sigma(X)) \qquad CoTerm[\![X]\!]\sigma \triangleq CoTerm(\sigma(X))$$

The intention of quantifying over the mapping is to simulate substitution of types that occurs due to instantiations of type constructors like $\mathsf{F}(\vec{C})$ and $\mathsf{G}(\vec{C})$ in the model itself.

Proposition 1. $[\![A]\!]\{[\![B]\!]\sigma/X\}\sigma = [\![A\{B/X\}]\!]\sigma$.

For simplicity, when we are working with closed types, we use the shorthand $[\![A]\!]$ to signify the model of A with respect to the empty mapping, $[\![A]\!]\{\}$.

4.1 Modeling data types

Let's now look at how we can translate data types into the model. Data types in the sequent calculus are defined by the ways that we can construct a term of the type. In other words, to understand a data type, we have to first understand the structure of term constructions for that type. The coterms of a data type are instead given by inversion over the different constructions, so that they consider the possible structures for input that they might interact with. In terms of the model, we can define a data type A by its constructions as a type-specific subset of all possible terms, $Cons[\![A]\!]$. Then, the coterms of A are given by everything that is orthogonal to the constructions, and general terms of type A are found by the double-orthogonal closure of the constructions.

$$Cons[\![A]\!]\sigma \subseteq Term \quad CoTerm[\![A]\!]\sigma \triangleq Cons[\![A]\!]\sigma^{\perp} \quad Term[\![A]\!]\sigma \triangleq Cons[\![A]\!]\sigma^{\perp\top}$$

Note that our requirement that the (co)terms of a type are orthogonal to one another follows from the above definition. In particular, $Term[\![A]\!]\sigma = CoTerm[\![A]\!]\sigma^{\top}$ by definition, and $CoTerm[\![A]\!]\sigma = Term[\![A]\!]\sigma^{\perp}$ by Property 1.3.

To make things more concrete, consider how we can use the general model of data types for basic data types like pairs and sums. Additionally, for now we will only look at closed types, so that we do not have to keep track of the mapping of type variables. The pair type $A \otimes B$ is defined by the rule for introducing the pair constructor:

$$\textbf{data}\ A \otimes B\ \textbf{where}$$
$$(\_,\_) : A, B \vdash A \otimes B \mid$$

$$\frac{\Gamma \vdash v_A : A \mid \Delta \quad \Gamma \vdash v_B : B \mid \Delta}{\Gamma \vdash (v_A, v_B) : A \otimes B \mid \Delta}\ \otimes R$$

In other words, we can build a pair whenever we have an arbitrary term of type A *and* an arbitrary term of type B. Therefore, the meaning of the type $A \otimes B$ in the model identifies all pairs constructed out of terms from $Term[\![A]\!]$ and $Term[\![B]\!]$.

$$Cons[\![A \otimes B]\!] \triangleq \{(v_A, v_B) \mid v_A \in Term[\![A]\!], v_B \in Term[\![B]\!]\}$$

And by unfolding the definitions of orthogonality in $CoTerm[\![A \otimes B]\!]$ and $Term[\![A \otimes B]\!]$, we get:

$$CoTerm[\![A \otimes B]\!] = \{e \in CoTerm_{\emptyset} \mid \forall v_A \in Term[\![A]\!], \forall v_B \in Term[\![B]\!], (v_A, v_B) \perp\!\!\!\perp e\}$$
$$Term[\![A \otimes B]\!] = \{v \in Term_{\emptyset} \mid \forall e, \forall v_A \in Term[\![A]\!], \forall v_B \in Term[\![B]\!], (v_A, v_B) \perp\!\!\!\perp e \implies v \perp\!\!\!\perp e\}$$

For sums, we define the type $A \oplus B$ by the rules for introducing one of the two sum constructors:

$$\textbf{data}\ A \oplus B\ \textbf{where}$$
$$\iota_1(\_) : A \vdash A \oplus B \mid$$
$$\iota_2(\_) : B \vdash A \oplus B \mid$$

$$\frac{\Gamma \vdash v_A : A \mid \Delta}{\Gamma \vdash \iota_1(v_A) : A \oplus B \mid \Delta}\ \oplus R_1 \qquad \frac{\Gamma \vdash v_B : B \mid \Delta}{\Gamma \vdash \iota_2(v_B) : A \oplus B \mid \Delta}\ \oplus R_2$$

In other words, we can build a sum whenever we have an arbitrary term of type A *or* an arbitrary term of type B. Therefore, the meaning of the type $A \oplus B$ in the model identifies all sums constructed from $Term[\![A]\!]$ as well as all sums constructed from $Term[\![B]\!]$.

$$Cons[\![A \oplus B]\!] \triangleq \{\iota_1(v_A) \mid v_A \in Term[\![A]\!]\} \cup \{\iota_2(v_B) \mid v_B \in Term[\![B]\!]\}$$

Unfolding the definitions, we get the following (co)terms for $A \oplus B$:

$$\begin{aligned} CoTerm[\![A \oplus B]\!] &= \{e \in CoTerm_\emptyset \mid \forall v_A \in Term[\![A]\!], \iota_1(v_A) \perp\!\!\!\perp e\} \\ &\quad \cap \{e \in CoTerm_\emptyset \mid \forall v_B \in Term[\![B]\!], \iota_2(v_B) \perp\!\!\!\perp e\} \\ &= \{e \in CoTerm_\emptyset \mid \forall v_A \in Term[\![A]\!], \forall v_B \in Term[\![B]\!], \\ &\qquad \iota_1(v_A) \perp\!\!\!\perp e \wedge \iota_2(v_b) \perp\!\!\!\perp e\} \\ Term[\![A \oplus B]\!] &= \{v \in Term_\emptyset \mid \forall e, \forall v_A \in Term[\![A]\!], \forall v_B \in Term[\![B]\!], \\ &\qquad (\iota_1(v_A) \perp\!\!\!\perp e \wedge \iota_2(v_b) \perp\!\!\!\perp e) \implies v \perp\!\!\!\perp e\} \end{aligned}$$

It is also interesting to consider the nullary versions of pairs and sums, which is the unit type 1 and empty type 0, respectively. The 1 type is defined by the singleton unit constructor, and the 0 type is defined by the fact that it has *no* possible constructors.

data 1 **where**	**data** 0 **where**
$() : \vdash 1 \mid$	
$\dfrac{}{\Gamma \vdash () : 1 \mid \Delta}\ 1R$	no $0R$ rules

In the model, the constructions of these data types correspond to the singleton set and the empty set:

$$Cons[\![1]\!] \triangleq \{()\} \qquad\qquad Cons[\![0]\!] \triangleq \{\}$$

By unfolding the definitions, we get the following general (co)terms of the 1 and 0 types:

$$\begin{aligned} CoTerm[\![1]\!] &= \{e \in CoTerm_\emptyset \mid () \perp\!\!\!\perp e\} \\ Term[\![1]\!] &= \{v \in Term_\emptyset \mid \forall e \in CoTerm_\emptyset, () \perp\!\!\!\perp e \implies v \perp\!\!\!\perp e\} \end{aligned}$$

$$\begin{aligned} CoTerm[\![0]\!] &= \{e \in CoTerm_\emptyset\} \\ Term[\![0]\!] &= \{v \in Term_\emptyset \mid \forall e \in CoTerm_\emptyset, v \perp\!\!\!\perp e\} \end{aligned}$$

The representation of 0 is especially interesting here. The set $CoTerm[\![0]\!]$ includes *every* coterm of $CoTerm_\emptyset$, and the set $Term[\![0]\!]$ includes only those terms that are capable of running with *any* coterm in $CoTerm_\emptyset$.

We can model user-defined data types in general based on their declarations. Given a declaration for the data type constructor F:

$$\begin{array}{l}\textbf{data}\ \mathsf{F}(\vec{X})\ \textbf{where}\\ \quad \mathsf{K}_1 :\quad \overrightarrow{A_1} \vdash \mathsf{F}(\vec{X}) \mid \overrightarrow{B_1}\\ \qquad\qquad \ldots\\ \quad \mathsf{K}_n :\quad \overrightarrow{A_n} \vdash \mathsf{F}(\vec{X}) \mid \overrightarrow{B_n}\end{array}$$

we can model its constructions as:

$$\begin{array}{rl} Cons[\![\mathsf{F}(\vec{C})]\!]\sigma \triangleq & \{\mathsf{K}_1(\overrightarrow{e_i}^i, \overrightarrow{v_i}^i) \mid \overrightarrow{e_i \in CoTerm[\![B_{1i}]\!]\sigma'}^i, \overrightarrow{v_i \in Term[\![A_{1i}]\!]\sigma'}^i\}\\ & \cup \ldots\\ & \cup \{\mathsf{K}_n(\overrightarrow{e_i}^i, \overrightarrow{v_i}^i) \mid \overrightarrow{e_i \in CoTerm[\![B_{ni}]\!]\sigma'}^i, \overrightarrow{v_i \in Term[\![A_{ni}]\!]\sigma'}^i\}\\ & \textbf{where}\ \sigma' = \{\overrightarrow{[\![C]\!]\sigma}/\vec{X}\}\sigma \end{array}$$

Notice that now the mapping from type variables to semantic types becomes useful to handle the placeholders used by a data type declaration, in order to plug in the meaning of the types $\vec{C}$ for the occurrences of the type variables $\vec{X}$ that show up in the types of each constructor.

It is worthwhile to discuss why this definition of $[\![\mathsf{F}(\vec{C})]\!]\sigma$ is well-defined. In all of the above models for basic data types, $[\![A]\!]$ is defined by induction on the structure of the type A. In $Cons[\![\mathsf{F}(\vec{C})]\!]\sigma$ above, the models for each of $\vec{C}$ follows from structural induction on $\mathsf{F}(\vec{C})$. However, the types $\vec{A}$ and $\vec{B}$ that are introduced by the declaration of F appear to come out of nowhere. Fortunately, this does not pose a problem when the declaration for F is *non-recursive*. In other words, the types $\vec{A}$ and $\vec{B}$ never mention the data type constructor F. Moreover, combinations of many type constructor declarations $\mathsf{F}_1, \ldots, \mathsf{F}_n$ should also be *non-recursive*, meaning that there is a well-founded ordering of dependencies between each F_i. Therefore, the model for a type, $[\![A]\!]\sigma$, follows by lexicographical induction on (1) the set of type constructors used to build A, and (2) the structure of the type A. In each of $\overrightarrow{[\![C]\!]\sigma}$, C may be built from the same set of connectives but is a sub-component of the original type. In each of $\overrightarrow{[\![A]\!]\sigma'}$ and $\overrightarrow{[\![B]\!]\sigma'}$, $\vec{A}$ and $\vec{B}$ are not sub-components of the original type, but they are built from a strictly smaller set of connectives (namely, the set with F removed). Therefore, $[\![A]\!]$ is well-defined for user-declared data types.

This general definition encompasses all of the above definitions for basic data types. For example, the declaration for $\otimes$ gives us the following set of constructions for a closed pair type:

$$\begin{array}{rl} Cons[\![A \otimes B]\!] &= \{(v_A, v_B) \mid v_A \in Term[\![X]\!]\{[\![A]\!], [\![B]\!]/X, Y\}, v_B \in Term[\![Y]\!]\{[\![A]\!], [\![B]\!]/X, Y\}\}\\ &= \{(v_A, v_B) \mid v_A \in Term[\![A]\!] v_B \in Term[\![B]\!]\}\end{array}$$

And similarly for the data type constructors $\oplus$, 1, and 0.

4.2 Modeling codata types

Codata types in the sequent calculus follow exactly the dual pattern as data types, and are defined by the ways that we can construct a coterm of the type. In other words, to understand a codata type, we first have to understand the structure of the observations (*i.e.,* coterm constructions) for that type. The terms of a codata type are instead given by inversion over the different possible observations, so that they consider the possible structures of the output they might interact with. In terms of the model, we can define a codata type A by its observations as a type-specific subset of all possible coterms, $Obs[\![A]\!]$. Then, the terms of A are given by everything that is orthogonal to the observations, and general coterms of type A are found by the double-orthogonal closure of the specific observations.

$$Obs[\![A]\!]\sigma \subseteq CoTerm_{\emptyset} \quad Term[\![A]\!]\sigma \triangleq Obs[\![A]\!]\sigma^{\top} \quad CoTerm[\![A]\!]\sigma \quad \triangleq Obs[\![A]\!]\sigma^{\top\perp}$$

Since this is just the oppositely oriented version of our treatment of data types, it should be no surprise that our requirement that the (co)terms of a type are orthogonal to one another. Specifically, $CoTerm[\![A]\!]\sigma = Term[\![A]\!]\sigma^{\perp}$ by definition, and $Term[\![A]\!]\sigma = CoTerm[\![A]\!]\sigma^{\top}$ by Property 1.3.

To help explain the general model of codata types, lets consider some pervasive basic co types like products and functions. The product type $A \,\&\, B$ is defined by the two rules for projection:

$$\begin{array}{c}\textbf{codata}\ A \,\&\, B\ \textbf{where} \\ \pi_1[\_] :| A \,\&\, B \vdash A \\ \pi_2[\_] :| A \,\&\, B \vdash B\end{array}$$

$$\frac{\Gamma \mid e_A : A \vdash \Delta}{\Gamma \mid \pi_1[e_A] : A \,\&\, B \vdash \Delta}\ \&L_1 \qquad \frac{\Gamma \mid e_B : A \vdash \Delta}{\Gamma \mid \pi_2[e_B] : A \,\&\, B \vdash \Delta}\ \&L_2$$

In other words, we are allowed to observe a product $A \,\&\, B$ whenever we have an arbitrary coterm accepting type A or an arbitrary coterm accepting type B. Therefore, the meaning of the type $A \,\&\, B$ in the model identifies all first projections built from $CoTerm[\![A]\!]$ as well as all second projections built from $CoTerm[\![B]\!]$.

$$Obs[\![A \,\&\, B]\!] \triangleq \{\pi_1[e_A] \mid e_A \in CoTerm[\![A]\!]\} \cup \{\pi_2[e_B] \mid e_B \in CoTerm[\![B]\!]\}$$

$$\begin{aligned} Term[\![A \,\&\, B]\!] &= \{v \in Term_{\emptyset} \mid \forall e_A \in CoTerm[\![A]\!], v \perp\!\!\!\perp \pi_1[e_A]\} \\ &\quad \cap \{v \in Term_{\emptyset} \mid \forall e_B \in CoTerm[\![B]\!], v \perp\!\!\!\perp \pi_2[e_B]\} \\ &= \{v \in Term_{\emptyset} \mid \forall e_A \in CoTerm[\![A]\!], \forall e_B \in CoTerm[\![B]\!], \\ &\qquad v \perp\!\!\!\perp \pi_1[e_A] \wedge v \perp\!\!\!\perp \pi_2[e_B]\} \\ CoTerm[\![A \,\&\, B]\!] &= \{e \in CoTerm_{\emptyset} \mid \forall v Term_{\emptyset}, \forall e_A \in CoTerm[\![A]\!], \forall e_B CoTerm[\![B]\!], \\ &\qquad (v \perp\!\!\!\perp \pi_1[e_A] \wedge v \perp\!\!\!\perp \pi_2[e_B]) \implies v \perp\!\!\!\perp e\} \end{aligned}$$

For functions, we define the type $A \to B$ by the rule for introducing a function call:

$$\begin{array}{c}\textbf{codata}\, A \to B\, \textbf{where} \\ \_ \cdot \_ : A \mid A \to B \vdash B\end{array}$$

$$\frac{\Gamma \vdash v_A : A \mid \Delta \quad \Gamma \mid e_B : B \vdash \Delta}{\Gamma \mid v_A \cdot e_B : A \to B \vdash \Delta}\ {\to}L$$

In other words, we are allowed to observe a function $A \to B$ whenever we have an arbitrary term of type A and an arbitrary coterm accepting type B. Therefore, the meaning of the type $A \to B$ in the model identifies all function calls built out of terms from $Term[\![A]\!]$ and coterms from $CoTerm[\![B]\!]$.

$$Obs[\![A \to B]\!] \triangleq \{v_A \cdot e_B \mid v_A \in Term[\![A]\!], e_B \in CoTerm[\![B]\!]\}$$

And for the terms and general coterms of $A \to B$ we have:

$$\begin{aligned} Term[\![A \to B]\!] &= \{v \in Term_\emptyset \mid \forall v_A \in Term[\![A]\!], \forall e_B \in CoTerm[\![B]\!], v \perp\!\!\!\perp v_A \cdot e_B\} \\ CoTerm[\![A \to B]\!] &= \{e \in Term_\emptyset \mid \forall v \in Term_\emptyset, \forall v_A \in Term[\![A]\!], \forall e_B \in CoTerm[\![B]\!], \\ &\qquad (v \perp\!\!\!\perp v_A \cdot e_B) \implies (v \perp\!\!\!\perp e)\} \end{aligned}$$

$$\frac{\Gamma \mid e_A : A \vdash \Delta \quad \Gamma \mid e_B : B \vdash \Delta}{\Gamma \mid [e_A, e_B] : A \parr B \vdash \Delta}\ \parr L$$

We can also give a model of classical connectives, like $A \parr B$, which make use of multiple (or no) conclusions. The codata type $A \parr B$ is defined as the dual of the pair $A \otimes B$:

$$\begin{array}{c}\textbf{codata}\, A \parr B\, \textbf{where} \\ [\_, \_] :| A \parr B \vdash A, B\end{array}$$

And likewise, the observations of $A \parr B$ are a mirror image of the constructions of $A \otimes B$:

$$Obs[\![A \parr B]\!] \triangleq \{[e_A, e_B] \mid e_A \in CoTerm[\![A]\!], e_B \in CoTerm[\![B]\!]\}$$

This definition is also similar to functions, except that the first component A of the structure is also an output like B.

As before, it is interesting to take a look at the nullary versions of products and copairs (*i.e.,* $\parr$).

$$\begin{array}{c}\textbf{codata}\, \bot\, \textbf{where} \\ \mathsf{tp} : \mid \bot \vdash\end{array} \qquad\qquad \textbf{codata}\, \top\, \textbf{where}$$

$$\frac{}{\Gamma \mid \mathsf{tp} : \bot \vdash \Delta}\ \bot L \qquad\qquad \text{no } \top L \text{ rules}$$

In the model, the observations of these codata types correspond to the singleton set and the empty set:

$$Obs[\![\bot]\!] \triangleq \{\mathsf{tp}\} \qquad\qquad Obs[\![\top]\!] \triangleq \{\}$$

which gives us the following general (co)terms of the $\bot$ and $\top$ types:

$$\begin{aligned} Term[\![\bot]\!] &= \{v \in Term_\emptyset \mid v \perp\!\!\!\perp \mathsf{tp}\} \\ CoTerm[\![\bot]\!] &= \{e \in CoTerm_\emptyset \mid \forall v \in Term_\emptyset, v \perp\!\!\!\perp \mathsf{tp} \implies v \perp\!\!\!\perp e\} \end{aligned}$$

$$\begin{aligned} Term[\![\top]\!] &= \{v \in Term_\emptyset\} \\ CoTerm[\![\top]\!] &= \{e \in CoTerm_\emptyset \mid \forall v \in Term_\emptyset, v \perp\!\!\!\perp e\} \end{aligned}$$

As the dual to 0, the $\top$ type is especially interesting. The set $Term[\![\top]\!]$ includes *every* term of the language, and the set $CoTerm[\![\top]\!]$ includes only the coterms that can run with *any* term.

As with data types, we can model user-defined codata types in general based on their declarations. Given a declaration for the codata type constructor G:

$$\begin{aligned} &\textbf{codata}\ \mathsf{G}(\vec{X})\ \textbf{where} \\ &\quad \mathsf{H}_1 : \quad \overrightarrow{A_1} \mid \mathsf{G}(\vec{X}) \vdash \overrightarrow{B_1} \\ &\qquad\qquad \cdots \\ &\quad \mathsf{H}_n : \quad \overrightarrow{A_n} \mid \mathsf{G}(\vec{X}) \vdash \overrightarrow{B_n} \end{aligned}$$

we can model its observations as:

$$\begin{aligned} Obs[\![\mathsf{G}(\vec{C})]\!]\sigma \triangleq\ & \{\mathsf{H}_1[\vec{v_i}^i, \vec{e_i}^i] \mid \overrightarrow{v_i \in Term[\![A_{1i}]\!]\sigma'}^i, \overrightarrow{e_i \in CoTerm[\![B_{1i}]\!]\sigma'}^i\} \\ & \cup \ldots \\ & \cup \{\mathsf{H}_n[\vec{v_i}^i, \vec{e_i}^i] \mid \overrightarrow{v_i \in Term[\![A_{ni}]\!]\sigma'}^i, \overrightarrow{e_i \in CoTerm[\![B_{ni}]\!]\sigma'}^i\} \end{aligned}$$

The (non-recursive) user-declared codata type $[\![\mathsf{G}(\vec{C})]\!]$ is well-defined for similar reasons as user-defined data types. Just as before, this general schematic definition encompasses all of the models for basic codata types discussed above. For instance, the declaration for the function type constructor gives us the following observations for closed types:

$$\begin{aligned} Obs[\![A \to B]\!] &= \{v_A \cdot e_B \mid v_A \in Term[\![X]\!]\{[\![A]\!], [\![B]\!]/X, Y\}, e_B \in CoTerm[\![Y]\!]\{[\![A]\!], [\![B]\!]/X, Y\}\} \\ &= \{v_A \cdot e_B \mid v_A \in Term[\![A]\!], e_B \in CoTerm[\![B]\!]\} \end{aligned}$$

Remark 3. As observed above, the set of terms for the empty codata type $\top$ and the set of coterms for the empty data type 0 include everything in the initial sets of terms and coterms: $Term[\![\top]\!] = Term_\emptyset$ and $CoTerm[\![0]\!] = CoTerm_\emptyset$. Another way of looking at this fact is that the choice of $Term_\emptyset$ and $CoTerm_\emptyset$ defines the terms of the type $\top$ and the coterms of the type 0. Furthermore, the $\top$ type corresponds to the terminal object of a category (as it's the unit of the product former &), and the 0 type corresponds to initial object of a category (as it's the unit of the sum former $\oplus$). So another viewpoint is that $CoTerm_\emptyset$ is the *initial* set defining the *initial* type, and dually $Term_\emptyset$ is the *term*inal set defining the *term*inal type. *End Remark* 3.

5 Static and semantic typing

Before in Section 4, we mentioned briefly why the recursive meaning of a type, $[\![A]\!]$, is well-defined. As a quick warm-up, we can more formally describe the well-defined nature of the semantic model for open types with free type variables. This proof gives a simple example of our general induction principle on types built from declared, but non-recursive, (co)data type constructors.

Lemma 1. *For any open type A built from the non-recursive (co)data type constructors in $\mathcal{F}$, $[\![A]\!]\sigma$ is defined when σ is defined for every free type variable A.*

Proof. By lexicographic induction on (1) inclusion of the set of type constructors $\mathcal{F}$, and (2) the structure of A.

- $A = X$: $[\![X]\!]\sigma = \sigma(X)$ and $\sigma(X)$ is defined by assumption.
- $A = \mathsf{F}(\vec{C})$, where F is a non-recursive data type constructor declared as:

$$\begin{aligned}&\textbf{data}\ \mathsf{F}(\vec{X})\ \textbf{where}\\ &\quad \mathsf{K}_1 :\ \ \overrightarrow{A_{1j}}^{j} \vdash \mathsf{F}(\vec{X}) \mid \overrightarrow{B_{1j}}^{j}\\ &\qquad\qquad \cdots\\ &\quad \mathsf{K}_n :\ \ \overrightarrow{A_{nj}}^{j} \vdash \mathsf{F}(\vec{X}) \mid \overrightarrow{B_{nj}}^{j}\end{aligned}$$

  Recall that $[\![\mathsf{F}(\vec{C})]\!]\sigma$ is defined as

$$Term[\![\mathsf{F}(\vec{C})]\!]\sigma \triangleq Cons[\![\mathsf{F}(\vec{C})]\!]\sigma^{\bot\top} \quad CoTerm[\![\mathsf{F}(\vec{C})]\!]\sigma \triangleq Cons[\![\mathsf{F}(\vec{C})]\!]\sigma^{\bot}$$

  where the constructions $Cons[\![\mathsf{F}(\vec{C})]\!]\sigma$ are defined as

$$Cons[\![\mathsf{F}(\vec{C})]\!]\sigma \triangleq \bigcup_{i=1}^{n}\{\mathsf{K}_1(\vec{e_j}^{j}, \vec{v_j}^{j}) \mid \overrightarrow{e_j \in CoTerm[\![B_{ij}]\!]\sigma'}^{j}, \overrightarrow{v_j \in Term[\![A_{ij}]\!]\sigma'}^{j}\}$$
$$\textbf{where}\ \sigma' = \{\overrightarrow{[\![C]\!]\sigma}/\vec{X}\}\sigma$$

  Observe that each of $\overrightarrow{[\![C]\!]\sigma}$ are defined by the inductive hypothesis (with the same set $\mathcal{F}$ and decreasing on the structure of the type). Also, because the data declaration is non-recursive, we know that each of the types A_{ij} and B_{ij} do not contain the type constructor F, but they *may* contain free occurrences of the additional type variables $\vec{X}$. Therefore, each $[\![A_{ij}]\!]\sigma'$ and $[\![B_{ij}]\!]\sigma'$ is defined by the inductive hypothesis (with the decreasing set of type constructors $\mathcal{F} \setminus \{\mathsf{F}\}$). It follows that $Cons[\![\mathsf{F}(\vec{C})]\!]\sigma$, and therefore $[\![\mathsf{F}(\vec{C})]\!]\sigma$ in general, is defined.

- $A = \mathsf{G}(\vec{C})$, where G is a non-recursive codata type constructor declared as:

$$\begin{aligned}&\textbf{codata}\ \mathsf{G}(\vec{X})\ \textbf{where}\\ &\quad \mathsf{H}_1 :\ \ \overrightarrow{A_{1j}}^{j} \mid \mathsf{G}(\vec{X}) \vdash \overrightarrow{B_{1j}}^{j}\\ &\qquad\qquad \cdots\\ &\quad \mathsf{H}_n :\ \ \overrightarrow{A_{nj}}^{j} \mid \mathsf{G}(\vec{X}) \vdash \overrightarrow{B_{nj}}^{j}\end{aligned}$$

  Analogous to the case for a data type constructor by duality. □

In addition to the fact that the recursively semantic model for types is well-defined, it also follows the expected substitution principle for open types.

Lemma 2. *For any open types A built from the non-recursive (co)data type constructors in $\mathcal{F}$ which are closed under σ, $[\![A]\!]\{[\![D]\!]\sigma/Y\}\sigma = [\![A\{D/Y\}]\!]\sigma$ when σ is defined for every free type variable in A (besides Y) and D.*

Proof. By lexicographic induction on (1) inclusion of the set of type constructors F, and (2) the structure of A.

- $A = X$: then $[\![X]\!]\{[\![C]\!]\sigma/X\}\sigma = [\![C]\!]\sigma = [\![X\{C/X\}]\!]\sigma$.
- $A = \mathsf{F}(\vec{C})$, where F is a non-recursive data type constructor declared as:

$$\begin{aligned}&\textbf{data}\ \mathsf{F}(\vec{X})\ \textbf{where}\\ &\quad \mathsf{K}_1 : \ \overrightarrow{A_{1j}}^{j} \vdash \mathsf{F}(\vec{X}) \mid \overrightarrow{B_{1j}}^{j}\\ &\qquad\qquad \cdots\\ &\quad \mathsf{K}_n : \ \overrightarrow{A_{nj}}^{j} \vdash \mathsf{F}(\vec{X}) \mid \overrightarrow{B_{nj}}^{j}\end{aligned}$$

  Observe that each of $\overrightarrow{[\![C]\!]\{[\![D]\!]\sigma/Y\}\sigma} = \overrightarrow{[\![C\{D/Y\}]\!]\sigma}$ by the inductive hypothesis (with the same set $\mathcal{F}$ and decreasing on the structure of the type). Also, because the data declaration is non-recursive, we know that each of the types A_{ij} and B_{ij} do not contain the type constructor F. Furthermore, because the data declaration is closed under σ, we know that each of the types A_{ij} and B_{ij} do not contain the free type variable Y (unless it is in $\vec{X}$). Let $\sigma' = \{\overrightarrow{[\![C]\!]\{[\![D]\!]\sigma/Y\}\sigma}/\vec{X}\}\sigma$ and note that $\sigma' = \{\overrightarrow{[\![C\{D/Y\}]\!]\sigma}/\vec{X}\}\sigma$ by the inductive hypothesis on $\overrightarrow{[\![C]\!]\{[\![D]\!]\sigma/Y\}\sigma}$. Then each $[\![A_{ij}]\!]\sigma'\{[\![D]\!]\sigma/Y\} = [\![A_{ij}]\!]\sigma'$ and $[\![B_{ij}]\!]\sigma'\{[\![D]\!]\sigma/Y\} = [\![B_{ij}]\!]\sigma'$ by one of two cases:

  - Y is not in $\vec{X}$: then each $A_{ij}\{D/Y\} = A_{ij}$ and $B_{ij}\{D/Y\} = B_{ij}$, and so each $[\![A_{ij}]\!]\sigma'\{[\![D]\!]\sigma/Y\} = [\![A_{ij}]\!]\sigma'$ and $[\![B_{ij}]\!]\sigma'\{[\![D]\!]\sigma/Y\} = [\![B_{ij}]\!]\sigma'$ by the inductive hypothesis.
  - Y is in $\vec{X}$: then $\sigma'\{[\![D]\!]\sigma/Y\} = \sigma'$ by shadowing.

  Therefore, $[\![\mathsf{F}(\vec{C})]\!]\sigma' = [\![\mathsf{F}(\overrightarrow{C\{D/Y\}})]\!]\sigma$.

- $A = \mathsf{G}(\vec{C})$, where G is a non-recursive codata type constructor declared as:

$$\begin{aligned}&\textbf{codata}\ \mathsf{G}(\vec{X})\ \textbf{where}\\ &\quad \mathsf{H}_1 : \ \overrightarrow{A_{1j}}^{j} \mid \mathsf{G}(\vec{X}) \vdash \overrightarrow{B_{1j}}^{j}\\ &\qquad\qquad \cdots\\ &\quad \mathsf{H}_n : \ \overrightarrow{A_{nj}}^{j} \mid \mathsf{G}(\vec{X}) \vdash \overrightarrow{B_{nj}}^{j}\end{aligned}$$

  Analogous to the case for a data type constructor by duality. □

Now that we have a well-defined semantic model for types that we can use to talk about the run-time behavior of commands, we would like to relate the static type system to the model. Suppose we have a well-typed, open command with free (co)variables, as described by a derivation ending in the sequent $c : (\Gamma \vdash \Delta)$. Intuitively, the environments Γ and Δ give the range of significance for the free (co)variables of c, so that if $x : A$ is in Γ and we plug in any value of type A for x in c, then the resulting command should also be well-typed (and similarly for covariables). Likewise for (co)terms, every well-typed substitution for a free variable of the (co)term should yield a well-typed (co)term of the same type.

We can reflect this intuition in the semantic world by defining the model of an open command, $c : (\Gamma \models \Delta)$, by the fact that c runs for every sensible substitution for its free (co)variables. Note that we only ever substitute (co)values for (co)variables, so we now need to specify the (co)values of a semantic type, rather than just the general terms. Since the notion of "what is a (co)value" comes from the strategy, the (co)values of a semantic type depend on the chosen strategy $\mathcal{S}$:

$$Value[\![A]\!]\sigma \triangleq Term[\![A]\!]\sigma \cap Value_{\mathcal{S}}$$
$$CoValue[\![A]\!]\sigma \triangleq CoTerm[\![A]\!]\sigma \cap CoValue_{\mathcal{S}}$$

The model of an environment is given by the set of all possible (parallel) substitutions for all specified (co)variables:

$$[\![\overrightarrow{x_i : A_i}^i]\!]\sigma \triangleq \{\overrightarrow{V_i}^i / \overrightarrow{x_i}^i \mid \overrightarrow{V_i \in Value[\![A_i]\!]\sigma}^i\}$$
$$[\![\overrightarrow{\alpha_i : A_i}^i]\!]\sigma \triangleq \{\overrightarrow{E_i}^i / \overrightarrow{\alpha_i}^i \mid \overrightarrow{E_i \in CoValue[\![A_i]\!]\sigma}^i\}$$

Then, the model for open commands is the statement that the command runs for every possible substitution for its environment, and the model for (co)terms is the statement that the (co)term is a member of the particular type for every possible substitution for its environment.

$$c : (\Gamma \models_\sigma \Delta) \triangleq \forall \gamma \in [\![\Gamma]\!]\sigma, \forall \delta \in [\![\Delta]\!]\sigma, c\{\gamma, \delta\} \in \perp\!\!\!\perp$$
$$\Gamma \models_\sigma v : A \mid \Delta \triangleq \forall \gamma \in [\![\Gamma]\!]\sigma, \forall \delta \in [\![\Delta]\!]\sigma, v\{\gamma, \delta\} \in Term[\![A]\!]$$
$$\Gamma \mid e : A \models_\sigma \Delta \triangleq \forall \gamma \in [\![\Gamma]\!]\sigma, \forall \delta \in [\![\Delta]\!]\sigma, e\{\gamma, \delta\} \in CoTerm[\![A]\!]$$

Now that we have given an interpretation for open commands and (co)terms in the model, we can express the soundness of the type system with respect to the model: static typing implies semantic typing.

Property 4 (Soundness)**.** *For a particular running set* $\perp\!\!\!\perp$*,* $\mu\tilde{\mu}^{\mathcal{T}}_{\mathcal{S}}$ *typing is* sound *with respect to* $[\![\mu\tilde{\mu}^{\mathcal{T}}_{\mathcal{S}}]\!]$ *iff:*

1. $c : (\Gamma \vdash \Delta)$ *implies* $c : (\Gamma \models \Delta)$*,*
2. $\Gamma \vdash v : A \mid \Delta$ *implies* $\Gamma \models v : A \mid \Delta$*, and*
3. $\Gamma \mid e : A \vdash \Delta$ *implies* $\Gamma \mid e : A \models \Delta$*.*

It is now our goal to prove that for a general class of choices — (co)data type constructors $\mathcal{F}$, strategies $\mathcal{S}$, initial sets $Term_\emptyset$ and $CoTerm_\emptyset$, and running set $\perp\!\!\!\perp$ — the type system of $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}$ is sound with respect to $[\![\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}]\!]$. In other words, we need to justify each inference rule of the $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}$ sequent calculus in terms of the model, and in doing so may need to require some additional properties about $\mathcal{F}$, $\mathcal{S}$, $Term_\emptyset$, $CoTerm_\emptyset$, and $\perp\!\!\!\perp$. These properties arise pragmatically out of the needs of the ultimate proof of soundness, giving a general characterization of the reasoning that pushes the inductive argument through.

To begin establishing the preliminaries for soundness, let's consider the most fundamental rule of the sequent calculus: the *Cut* rule. Recall that the *Cut* rule declares that for *any* type A, we can meaningfully put together an arbitrary term and coterm of type A:

$$\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma \mid e : A \vdash \Delta}{\langle v|e\rangle : \Gamma \vdash \Delta}\ Cut$$

In terms of the model, this rule requires that the terms and coterms of every semantic type $[\![A]\!]$ be *mutually orthogonal* to one another, so that we can run a command formed by any combination from the sets of (co)terms.

Property 5 (Mutual orthogonality)**.** *A set of terms $Term_A$ and coterms $CoTerm_A$ are* mutually orthogonal *iff $Term_A = CoTerm_A^\top$ and $CoTerm_A = Term_A^\perp$. Furthermore, a semantic type R is mutually orthogonal iff $Term(R)$ and $CoTerm(R)$ are mutually orthogonal.*

It follows that the meaning of every static type is a mutually orthogonal semantic type.

Lemma 3 (Mutual orthogonality of types)**.** *For every type A built from the (co)data type constructors $\mathcal{F}$, $[\![A]\!]\sigma$ is mutually orthogonal whenever every semantic type in the codomain of σ is mutually orthogonal.*

Proof. By cases on the structure of A:

- $A = X$: then $[\![X]\!]\sigma = \sigma(X)$, which is mutually orthogonal by assumption.
- $A = \mathsf{F}(\vec{C})$ where F is a data type constructor: then $Term[\![\mathsf{F}(\vec{C})]\!]\sigma = CoTerm[\![\mathsf{F}(\vec{C})]\!]\sigma^\top$ by definition and

  $$CoTerm[\![\mathsf{F}(\vec{C})]\!]\sigma = Cons[\![\mathsf{F}(\vec{C})]\!]\sigma^\perp = Cons[\![\mathsf{F}(\vec{C})]\!]\sigma^{\perp\top\perp} = Term[\![\mathsf{F}(\vec{C})]\!]\sigma^\perp$$

  by triple orthogonal elimination.
- $A = \mathsf{G}(\vec{C})$ where G is a codata type constructor: then $CoTerm[\![\mathsf{G}(\vec{C})]\!]\sigma = Term[\![\mathsf{G}(\vec{C})]\!]\sigma^\perp$ by definition and

  $$Term[\![\mathsf{G}(\vec{C})]\!]\sigma = Obs[\![\mathsf{G}(\vec{C})]\!]\sigma^\top = Obs[\![\mathsf{G}(\vec{C})]\!]\sigma^{\top\perp\top} = CoTerm[\![\mathsf{G}(\vec{C})]\!]\sigma^\top$$

  by triple orthogonal elimination. □

As a consequence of the mutual orthogonality of static types, we also have that the terms and coterms of every static type belong to the initial sets $Term_\emptyset$ and $CoTerm_\emptyset$, due to the definition of the dual orthogonality operations.

Lemma 4. *For every type A built from the (co)data type constructors $\mathcal{F}$, $Term[\![A]\!]\sigma \subseteq Term_\emptyset$ and $CoTerm[\![A]\!]\sigma \subseteq CoTerm_\emptyset$ whenever every semantic type in the codomain of σ is mutually orthogonal.*

Proof. By Lemma 3 and the definition of $\_^\perp$ and $\_^\top$. □

The rest of the inference rules assert the typing of certain forms of terms and coterms. Therefore, we need to make sure that these certain (co)terms are in the initial sets $Term_\emptyset$ and $CoTerm_\emptyset$, so that they are not lost during the double orthogonal procedure. In other words, we need to make sure that $Term_\emptyset$ and $CoTerm_\emptyset$ and *initialized* properly, so that they contain "enough" (co)terms that we expect to have available. In the remainder of the section, let the data type constructor F and codata type constructor G declared as:

$$\begin{array}{ll}
\textbf{data}\ \mathsf{F}(\vec{X})\ \textbf{where} & \textbf{codata}\ \mathsf{G}(\vec{X})\ \textbf{where} \\
\mathsf{K}_1 :\ \ \overrightarrow{A_{1j}}^j \vdash \mathsf{F}(\vec{X}) \mid \overrightarrow{B_{1j}}^j & \mathsf{H}_1 :\ \ \overrightarrow{A_{1j}}^j \mid \mathsf{G}(\vec{X}) \vdash \overrightarrow{B_{1j}}^j \\
\cdots & \cdots \\
\mathsf{K}_n :\ \ \overrightarrow{A_{nj}}^j \vdash \mathsf{F}(\vec{X}) \mid \overrightarrow{B_{nj}}^j & \mathsf{H}_n :\ \ \overrightarrow{A_{nj}}^j \mid \mathsf{G}(\vec{X}) \vdash \overrightarrow{B_{nj}}^j
\end{array}$$

stand in place for some generic (co)data type constructor.

Property 6 (Initialization)**.**

- *$Term_\emptyset$ is* initialized with A neutrality *iff* $\mu\alpha.c \in Term_\emptyset$ *whenever* $c : (\models \alpha : A)$. *$CoTerm_\emptyset$ is* initialized with A neutrality *iff* $\tilde{\mu}x.c \in CoTerm_\emptyset$ *whenever* $c : (x : A \models)$.
- *For a data type constructor* F*, $Term_\emptyset$ is* initialized with F data *iff for all constructors* K_i *of* F*, $Cons[\![\mathsf{F}(\vec{C})]\!]\sigma \subseteq Term_\emptyset$ whenever $\overrightarrow{e \in CoTerm_\emptyset}^i$ and $\vec{v} \in Term_\emptyset$. $CoTerm_\emptyset$ is* initialized with F data *iff* $\tilde{\mu}[\mathsf{K}_i(\overrightarrow{\alpha_j}^j, \overrightarrow{x_j}^j).c_i\,]^i \in CoTerm_\emptyset$ *whenever* $c_i : (\overrightarrow{x_j : A_{ij}}^j \models_{\{\overrightarrow{[\![C]\!]}/\vec{X}\}} \overrightarrow{\alpha_j : B_{ij}}^j)$ *for any types* $\vec{C}$.
- *For a codata type constructor* G*, $CoTerm_\emptyset$ is* initialized with G codata *iff for all coconstructors* H_i *of* G*, $\mathsf{H}_i[\vec{v}, \vec{e}\,] \in CoTerm_\emptyset$ whenever $\vec{v} \in Term_\emptyset$ and $\vec{e} \in CoTerm_\emptyset$. $Term_\emptyset$ is* initialized with G codata *iff* $\mu(\overrightarrow{\mathsf{H}_i[\overrightarrow{x_j}^j, \overrightarrow{\alpha_j}^j].c_i}^i) \in Term_\emptyset$ *whenever* $c_i : (\overrightarrow{x_j : A_{ij}}^j \models_{\{\overrightarrow{[\![C]\!]}/\vec{X}\}} \overrightarrow{\alpha_j : B_{ij}}^j)$ *for any types* $\vec{C}$.

Additionally, we also need to make sure that the running set $\perp\!\!\!\perp$ contains "enough" commands, based on the run-time behavior of programs. That is to say, if a command — made up of plausibly well-behaved (co)terms from $Term_\emptyset$ and $CoTerm_\emptyset$ — reduces to a second command that runs, then the first command also runs. This assertion gives us a generalized version of *saturation* from classical realizability [9] and reducibility candidates.

Property 7 (Saturation)**.** *For a reduction relation L, a running set* $\perp\!\!\!\perp$ *is* saturated by *L iff the following condition holds:*

- *For any* $v \in Term_\emptyset$ *and* $e \in CoTerm_\emptyset$, $\langle v|e\rangle \in \perp\!\!\!\perp$ *whenever* $\langle v|e\rangle \to_L c$ *and* $c \in \perp\!\!\!\perp$.

Furthermore, a set of term $Term_A$ *and a set of coterms* $CoTerm_A$ *are* saturated by *L iff the following conditions hold, respectively:*

- *For any* $v \in Term_\emptyset$, $v \in Term_A$ *whenever* $v \to_L v'$ *and* $v' \in Term_A$.
- *For any* $e \in CoTerm_\emptyset$, $e \in CoTerm_A$ *whenever* $e \to_L e'$ *and* $e' \in Term_A$.

For the remainder of this section, we will generally assume that the initial sets $Term_\emptyset$ and $CoTerm_\emptyset$ are initialized with neutrality at every type, and with every (co)-data type constructor in the particular $\mathcal{F}$ under consideration. Also, we will generally assume that the running set $\perp\!\!\!\perp$ is saturated by $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}$ head reduction for our choice of type constructors $\mathcal{F}$ and strategy $\mathcal{S}$. That is, $\perp\!\!\!\perp$ will be some set of commands saturated by reduction with the μ_E, $\tilde{\mu}_V$, or β rules applied to the top-level command, and with the ς rules applied to the immediate sub-(co)term of the top-level command.

Now, let's consider the *Act* and *CoAct* inference rules for introducing the generic input and output abstractions. These rules assert that for any well-typed command, we can abstract over a variable or covariable of any type to get a well-typed coterm or term of that type:

$$\frac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}\ Act \qquad\qquad \frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta}\ CoAct$$

For it to be possible that $Term[\![A]\!]$ contains the term $\mu\alpha.c$ and $CoTerm[\![A]\!]$ contains the coterm $\tilde{\mu}x.c$, we need to ensure that $Term_\emptyset$ and $CoTerm_\emptyset$ are initialized with A neutrality. Next, the saturation of $\perp\!\!\!\perp$ by μ_E and $\tilde{\mu}_V$ head reduction gets us part of the way there, guaranteeing a weaker form of activation stating that $\mu\alpha.c$ runs with any *covalue* of A, and $\tilde{\mu}x.c$ runs with any *value* of A.

Lemma 5 (Weak activation)**.**

- $\mu\alpha.c \in CoValue[\![A]\!]^\top$ *whenever* $c : (\models \alpha : A)$.
- $\tilde{\mu}x.c \in Value[\![A]\!]^\perp$ *whenever* $c : (x : A \models)$.

Proof.

- For the first point, observe that for any $E_A \in CoValue[\![A]\!]$ we have the head reduction:

  $$\langle \mu\alpha.c|E_A\rangle \to_{\mu_E} c\{E_A/\alpha\}$$

  where $c\{E_A/\alpha\} \in \perp\!\!\!\perp$ by the assumption that $c : (\models \alpha : A)$. Furthermore, $\mu\alpha.c \in Term_\emptyset$ by initialization of $Term_\emptyset$ by A neutrality, and $E_A \in CoTerm_\emptyset$ because $CoValue[\![A]\!] \subseteq CoTerm[\![A]\!]$ by definition and $CoTerm[\![A]\!] \subseteq CoTerm_\emptyset$ by Lemma 4. Therefore, $\mu\alpha.c \perp\!\!\!\perp E_A$ by saturation of $\perp\!\!\!\perp$ by μ_E head reduction, and so $\mu\alpha.c \in CoValue[\![A]\!]^\top$.

- The second point for $\tilde{\mu}x.c \in Value[\![A]\!]^\perp$ follows analogously by duality. □

In order to extend this weak form of activation to a stronger form that demonstrates the soundness of the *Act* rule (and dually the *CoAct* rule), we need to show that because $\mu\alpha.c$ runs with any value of $[\![A]\!]$ it *also* must run with any term of $[\![A]\!]$. Therefore, we need to establish something more about the relationship between the (co)terms and (co)values of the semantic types in question. In particular, we would like to say that the terms of every semantic type $[\![A]\!]$ are *generated* by the double orthogonal closure on the values of $[\![A]\!]$ (and dually for the covalues generating the coterms). That would imply that the values of $[\![A]\!]$ describe all of its interesting intricacies, and that we are not missing out on any different subtleties by only testing $\tilde{\mu}x.c$ on just the values of $[\![A]\!]$.

Property 8 (Generation)**.** *For a chosen strategy $\mathcal{S}$:*

- *A set of terms $Term_A$ is generated by its values iff $Term_A = (Term_A \cap Value_{\mathcal{S}})^{\perp\top}$.*
- *A set of coterms $CoTerm_A$ is generated by its covalues iff $CoTerm_A = (CoTerm_A \cap CoValue_{\mathcal{S}})^{\top\perp}$.*
- *A semantic type R is generated by its (co)values iff $Term(R)$ is generated by its values and $CoTerm(R)$ is generated by its covalues.*

With the assumption that the meaning of a type A is generated by its values, we get the stronger version of activation that we have been seeking.

Lemma 6 (Strong activation)**.**
- *$\mu\alpha.c \in Term[\![A]\!]$ whenever $c : (\models \alpha : A)$ and $CoTerm[\![A]\!]$ is generated by its covalues.*
- *$\tilde{\mu}x.c \in CoTerm[\![A]\!]$ whenever $c : (x : A \models)$ and $Term[\![A]\!]$ is generated by its values.*

Proof.
- For the first point, we know that $\mu\alpha.c \in CoValue[\![A]\!]^{\top}$ by Lemma 5. From the assumption that $CoTerm[\![A]\!] = CoValue[\![A]\!]^{\top\perp}$ we know that $CoTerm[\![A]\!]^{\top} = CoValue[\![A]\!]^{\top\perp\top} = CoValue[\![A]\!]^{\top}$ by triple orthogonal elimination, and that $Term[\![A]\!] = CoTerm[\![A]\!]^{\top} = CoValue[\![A]\!]^{\top}$ by Lemma 3. Therefore, $\mu\alpha.c$ is also in $Term[\![A]\!]$.
- The second point for $\tilde{\mu}x.c \in CoTerm[\![A]\!]$ follows analogously by duality. □

References

[1] P. Downen. *Sequent Calculus: A Logic and a Language for Computation and Duality*. PhD thesis, The University of Oregon, 2017.

[2] P. Downen and Z. M. Ariola. The duality of construction. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 249–269. Springer, 2014.

[3] P. Downen and Z. M. Ariola. Beyond polarity: Towards a multi-discipline intermediate language with sharing. In *27th EACSL Annual Conference on Computer Science Logic, CSL 2018, September 4-7, 2018, Birmingham, UK*, volume 119 of *LIPIcs*, pages 21:1–21:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[4] P. Downen and Z. M. Ariola. A computational understanding of classical (co)recursion. In *PPDP '20: 22nd International Symposium on Principles and Practice of Declarative Programming, Bologna, Italy, 9-10 September, 2020*, pages 5:1–5:13. ACM, 2020.

[5] P. Downen, Z. M. Ariola, and S. Ghilezan. The duality of classical intersection and union types. *Fundamenta Informaticae*, 170(1-3):39–92, 2019.

[6] P. Downen, P. Johnson-Freyd, and Z. M. Ariola. Structures for structural recursion. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP '15, pages 127–139, New York, NY, USA, 2015. ACM.

[7] P. Downen, P. Johnson-Freyd, and Z. M. Ariola. Abstracting models of strong normalization for classical calculi. *Journal of Logical and Algebraic Methods in Programming*, 111:100512, 2020.

[8] P. Johnson-Freyd. *Properties of Sequent-Calculus-Based Languages*. PhD thesis, The University of Oregon, 2017.

[9] G. Munch-Maccagnoni. Focalisation and classical realisability. In *Computer Science Logic*, pages 409–423. Springer, 2009.