Fundamenta Informaticae XXI (2001) 1001–1054
DOI 10.3233/FI-2016-0000
IOS Press

The Duality of Classical Intersection and Union Types

Paul Downen*
University of Oregon
pdownen@cs.uoregon.edu

Zena M. Ariola*
University of Oregon
ariola@cs.uoregon.edu

Silvia Ghilezan†
University of Novi Sad
gsilvia@uns.ac.rs

Abstract. For a long time, intersection types have been admired for their surprising ability to *complete* the simply typed lambda calculus. Intersection types are an example of an *implicit* typing feature which can describe program behavior without manifesting itself within the syntax of a program. Dual to intersections, union types are another implicit typing feature which extends the completeness property of intersection types in the lambda calculus to full-fledged programming languages. However, the formalization of union types can easily break other desirable meta-theoretical properties of the type system. But why should unions be troublesome when their dual, intersections, are not?

We look at the issues surrounding the design of type systems for both intersection and union types through the lens of duality by formalizing them within the symmetric language of the classical sequent calculus. In order to formulate type systems which have all of our properties of interest—soundness, completeness, and type safety—we also look at the impact of evaluation strategy on typing. As a result, we present two dual type systems—one for call-by-value and one for call-by-name evaluation—which have all three properties. We also consider the possibility of classical

*This work is supported by the National Science Foundation under grants CCF-1423617 and CCF-1719158.
†Courtesy Research Associate in the Global Studies Institute, University of Oregon. This work has been partly supported by MESTD under the grants ON174026 and III44006.

non-deterministic evaluation, for which there is a choice between two different systems depending on which properties are desired: a *full* type system which is complete, and a *simplified* type system which is sound and type safe.

Keywords: Intersection types, union types, duality, sequent calculus, discipline, type safety, strong normalization, soundness and completeness, reducibility candidates, symmetric candidates

1. Introduction

Intersection types enrich the language of types to say that a single value may satisfy multiple different static properties. Dual to intersection types, union types allow for the ability to weaken the statically-predicted properties of a result, especially in cases where the eventual result of a program is not yet known. When designing type systems for foundational calculi, there are several desirable properties that we might want to hold:

- *Soundness* with respect to normalization: every well-typed expression is strongly normalizing.
- *Completeness* with respect to normalization: every strongly normalizing expression is typable.
- *Subject reduction*: every typed expression only reduces to other expressions of the same type.

Soundness and subject reduction are common properties for typed λ-calculi to have. In contrast, completeness is quite an unusual property for typed λ-calculi (like the simply typed or polymorphic λ-calculus), but it is the hallmark of intersection typing. Together, soundness and completeness means that the type system *exactly* characterizes strongly normalizing expressions.

Intersection types are enough to guarantee completeness for just the lambda calculus, but union types are also needed to scale completeness up to a more full-fledged programming language. However, it is easy for systems with both intersection and union types (or even just intersection types and effects) to lose these nice properties. In particular, the status of subject reduction is fraught in usual formalizations of union types in the lambda calculus. But why should it be that intersection types are more "well-behaved" than union types, since they are supposed to be duals?

In order to investigate this problem—the mismatch between intersection and union types—we consider how they appear in a dual language based on the sequent calculus. Duality presents intersection and union types in a perfectly symmetric way, avoiding the usual problem with formalizing union types in natural deduction, and also extends completeness to a calculus with first-class control. Duality alone is not enough, though, since a naïve presentation of intersection and union types still does not enjoy subject reduction. The issue surrounding subject reduction is related to the issue of type safety of polymorphism in ML, which is solved by the well-known *value restriction*. Building on this, we arrive at a more disciplined type system which enforces a symmetric (co)value restriction: intersection types can only be introduced on *values* (which produce information), and dually union types can only be introduced on *covalues* (which consume information). This (co)value restriction in the disciplined type system corresponds to the notion of *discipline* for restricting substitution in rewriting theories [1], which is useful for ensuring properties such as confluence and strong normalization. That is to say, the

same notion of substitution discipline that tames the non-determinism present in classical cut elimination of the sequent calculus *also* tames the type safety troubles caused by unconstrained intersection and union types.

To begin, we review some of the issues that arise with intersection and union types in the familiar setting of the lambda calculus (Section 2). We then move over to the symmetric setting of the classical sequent calculus and start with an introduction to a programming language based on the sequent calculus (Section 3) before continuing with discussing the issues of extending this language with intersection and union types (Section 4). We begin with a *full* type system $\overline{\lambda}\mu\tilde{\mu}\cap\cup$ for intersection and union types (Section 4.1), which presents plausible typing rules but does not impose any additional, computationally-minded constraints. Unfortunately, some of the same problems arise again in this basic setup, which motivates the search for type systems which have all of the desired properties—soundness, completeness, and subject reduction. Our contributions are:

- (Section 4) An analysis of typing restrictions for establishing type safety in the sequent calculus that leads us to the following well-behaved systems:
  - (Section 4.2) A pair of dual call-by-value $\overline{\lambda}\mu\tilde{\mu}\cap\cup_v$ and call-by-name $\overline{\lambda}\mu\tilde{\mu}\cap\cup_n$ systems for intersection and union types in the sequent calculus based on a value and covalue restriction, respectively, which come with their own notion of focusing.
  - (Section 4.3) A disciplined type system $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ that unifies and subsumes the specialized call-by-value ($\overline{\lambda}\mu\tilde{\mu}\cap\cup_v$), call-by-name ($\overline{\lambda}\mu\tilde{\mu}\cap\cup_n$), and full ($\overline{\lambda}\mu\tilde{\mu}\cap\cup$) systems.
  - (Section 4.4) A further simplified type system $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d^-$ that imposes additional restrictions to safely handle non-deterministic computation.
- (Section 5) A proof that the disciplined type system $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ is complete by typing all strongly normalizing, focused expressions.
- (Section 6) A proof that the $\overline{\lambda}\mu\tilde{\mu}\cap\cup_v$, $\overline{\lambda}\mu\tilde{\mu}\cap\cup_n$, and $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d^-$ type systems enjoy subject reduction, and are therefore type safe, *i.e.,* well-typed programs do not get stuck.
- (Section 7) A pair of models of strong normalization for intersection and union types that is uniform over a chosen discipline:
  - (Section 7.4) The first model is based on reducibility candidates, and proves that the more general $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ type system is sound only for *deterministic* disciplines d.
  - (Section 7.7) The second model is based on symmetric candidates, and proves that the more restrictive $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d^-$ type system is sound even for *non-deterministic* disciplines d.

In summary, our framework for type systems of intersection and union types in the sequent calculus explores three main disciplines of interest–call-by-value (v), call-by-name (n), and non-deterministic (u)—with the following results:

	Deterministic	Sound	Complete	Type Safe
$\overline{\lambda}\mu\tilde{\mu}\cap\cup_u$	No	No	Yes	No
$\overline{\lambda}\mu\tilde{\mu}\cap\cup_v$	Yes	Yes	Yes	Yes
$\overline{\lambda}\mu\tilde{\mu}\cap\cup_n$	Yes	Yes	Yes	Yes
$\overline{\lambda}\mu\tilde{\mu}\cap\cup_d^-$	Either	Yes	No	Yes

$$\texttt{Term} \ni M ::= x \mid \lambda x.M \mid MM$$

$$\frac{}{\Gamma, x : A \vdash x : A}\, Ax$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \to I \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B} \to E$$

Figure 1: λ — The simply typed lambda calculus.

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash M : B}{\Gamma \vdash M : A \cap B} \cap I \qquad \frac{\Gamma \vdash M : A \cap B}{\Gamma \vdash M : A} \cap E \qquad \frac{\Gamma \vdash M : A \cap B}{\Gamma \vdash M : B} \cap E$$

Figure 2: $\lambda\cap$ — Simply typed lambda calculus with intersection types.

In the last row, $\overline{\lambda}\mu\tilde{\mu}\cap\cup_{\bar{d}}^{-}$ is a family of systems that include both deterministic (when d is chosen to be v or n) and non-deterministic (when d is u) evaluation.

2. Intersection and Union Types in the Lambda Calculus

To avoid inconsistencies in the lambda calculus which lead to a formula being both true and false, Alonzo Church introduced a simple theory of Types [2], which is described in Figure 1. All simply typed lambda terms are strongly normalizing (*a.k.a soundness*). However, the converse (*a.k.a completeness*) is not true: some normal forms are not typable by simple types, for example, $\lambda x.xx$. This same, one-directional, strong normalization property extends to all systems of the Barendregt's lambda cube [3], which expresses the programming language features of polymorphism, type operators, and dependent types. A well-typed term in any type system of the lambda cube is strongly normalizing, but, for every corner of the cube, there are untypable strongly normalizing terms. This limitation, where static type checking is only an approximation of strong normalization, was overcome by extending the simply typed lambda calculus with intersection types [4, 5, 6]. We refer to this system as $\lambda\cap$, and recall the typing rules for intersection in Figure 2.

Theorem 2.1. A lambda term is strongly normalizing if and only if it is typable in $\lambda\cap$.

The reason for the surprising strength of $\lambda\cap$ for *exactly* classifying strongly normalizing lambda terms within its type system is that intersection types are not an ordinary connective like functions or products. Instead, intersection types effectively add *new* typing rules to the other proper connectives in the calculus, so that there are more possible ways of typing a normal function. The extra typing possibilities on top of the existing simply typed λ-calculus is what gives $\lambda\cap$ the ability to give a type to more terms, coming up with at least one type to describe any term which is strongly normalizing.

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M : A \cup B} \cup I \quad \frac{\Gamma \vdash M : B}{\Gamma \vdash M : A \cup B} \cup I \quad \frac{\Gamma, x : A \vdash M : C \quad \Gamma, x : B \vdash M : C \quad \Gamma \vdash N : A \cup B}{\Gamma \vdash M[N/x] : C} \cup E$$

Figure 3: $\lambda\cup$ — Simply typed lambda calculus with union types.

Union types were first proposed in [7] and further studied in [8, 9], as a dual counterpart to intersection types. We recall the rules for union types in Figure 3, and refer to $\lambda\cap\cup$ as the extension of the simply typed lambda calculus with *both* intersection and union types. The apparently more permissive system $\lambda\cap\cup$ also types exactly the strongly-normalizing lambda terms, the same terms as $\lambda\cap$, so the two systems are equivalent from the point of view of typability for just the lambda calculus.

Theorem 2.2. A lambda term is strongly normalizing if and only if it is typable in $\lambda\cap\cup$.

However, even though $\lambda\cap$ and $\lambda\cap\cup$ both admit the same lambda terms as well-typed, they are still very different type systems. From a programmer's perspective, intersection and union types offer two different interfaces on how a value of a type can be used. A union type is effectively an untagged union, as found in systems programming languages like C. Whereas an intersection type is akin to a finite version of polymorphism. For example, a function of type $(\texttt{int} \to \texttt{int}) \cap (\texttt{float} \to \texttt{float})$ can be given either integer or floating-point arguments, and returns a result of the same type as it was given. From a language designer's perspective, intersection and union types have different meta-theoretic properties, and naïvely extending a language with intersection or union types can easily lead to some undesirable outcomes.

2.1. Failure of completeness in the λ-calculus with conditionals and no union types

The heart of the iconic completeness property for intersection types is the following fact: all normal forms of the lambda calculus are typable. Without this fact, there is no hope for completeness. For example, the term xy can be given type B by assuming $x : A \to B$ and $y : A$. The bigger challenge is when a variable appears more than once, which can lead to self-application like xx. Here, intersection types come to the rescue by allowing for $xx : B$ under the assumption that $x : (A \to B) \cap A$.

The typability of normal forms works in the lambda calculus with intersection types, in part, because it only has one real type with programming constructs: the function type. All other types are the special intersection type or some abstract type constant which lacks any specific inhabitants. But completeness does not easily extend to a more practical prototypical programming language with conditionals for a boolean or other sum type. For example, consider the conditional term `if` x `then` 1 `else` "`two`"; is it an `int` or a `string`? The answer depends on whether or not x gets instantiated with `true` or `false`, so there is no way of knowing statically during type checking. This perfectly sensible normal form is not typable with only intersection types. Union types are also required to give the type assignment

$$\texttt{if}\ x\ \texttt{then}\ 1\ \texttt{else}\ \text{“}\texttt{two}\text{”} : \texttt{int} \cup \texttt{string}$$

under the assumption that $x : \texttt{bool}$. In other words, once conditionals and base types are added to the lambda calculus, *both* union and intersection types are needed to establish completeness of typability for strongly normalizing terms.

There is one additional complication for completeness when base types are introduced: some normal forms *should not* be typable because they are manifestly ill-typed. For example, it is sensible for the normal form xy to be typable because there are contexts which bind x and y to values (like $x = \lambda x.x$ and $y = 1$) for which evaluating xy results in an answer. However, a normal form with an ill-typed application like $1(\text{“}\texttt{augh}\text{”})$ is *fatally* stuck because 1 is not a function, and can never give an answer in any context. Therefore, in a more general programming language, the completeness property should be further weakened to avoid such possibilities: all strongly normalizing terms with *non-fatal* normal forms are typable. Note that the example conditional normal form above is not fatal in this sense, since binding x to either `true` or `false` will yield an answer. So it should be typable, which requires a union between the types of its two possible branches.

2.2. Failure of subject reduction in the λ-calculus with intersection and union types

The λ-calculus with intersection and union types $\lambda\cap\cup$ does not enjoy subject reduction [8, 9]. This is due to the union elimination rule ($\cup E$); the root cause of the problem is the use of substitution, $M[N/x]$, in the conclusion of the rule.

Example 2.3. The following counter-example is given in [9]. Consider the term

$$M \equiv \lambda xyz.x((\lambda t.t)yz)((\lambda t.t)yz)$$

and its possible β-reducts

$$\begin{array}{rcl} \lambda xyz.x((\lambda t.t)yz)((\lambda t.t)yz) & \to_\beta & \lambda xyz.x(yz)((\lambda t.t)yz) \text{ or } \lambda xyz.x((\lambda t.t)yz)(yz) \\ & \to_\beta & \lambda xyz.x(yz)(yz) \end{array}$$

The terms $\lambda xyz.x((\lambda t.t)yz)((\lambda t.t)yz)$ and $\lambda xyz.x(yz)(yz)$ are typable with the type

$$(A \to A \to C) \cap (B \to B \to C) \to (D \to A \cup B) \to D \to C$$

Let $\Gamma = \{x : (A \to A \to C) \cap (B \to B \to C),\ \ y : D \to A \cup B,\ \ z : D\}$, then

$$\frac{\Gamma,\ \ w : A \vdash xww : C \qquad \Gamma,\ \ w : B \vdash xww : C \qquad \Gamma \vdash (\lambda t.t)yz : A \cup B}{\Gamma \vdash x((\lambda t.t)yz)((\lambda t.t)yz) : C} \cup E$$

Similarly,

$$\frac{\Gamma,\ \ w : A \vdash xww : C \qquad \Gamma,\ \ w : B \vdash xww : C \qquad \Gamma \vdash yz : A \cup B}{\Gamma \vdash x(yz)(yz) : C} \cup E$$

However, the terms $\lambda xyz.x(yz)((\lambda t.t)yz)$ and $\lambda xyz.x((\lambda t.t)yz)(yz)$ which are obtained from M by one step β-reduction are not typable in Γ.

The failure of subject reduction under unrestricted contextual reduction can be recovered in several ways [10]: (i) by using parallel reduction or subtyping [9], or (ii) by using a different notion of contextual closure in a fully type-decorated setting [11].

2.3. Failure of type safety in the λ-calculus with intersection types and effects

Implicit polymorphism, as it appears in ML, has the possibility of being type *unsafe* in the presence of effects [12]. For example, consider the following expression, where `ref` creates a new mutable reference, the $:=$ operator updates a reference, and the ! operator returns the current value of a reference:

$$\begin{aligned} &\texttt{let}\, f \;= \texttt{ref}\ (\lambda x.x) \\ &\texttt{in}\;\; f := (\lambda x.1 + x); \\ &\quad\ (!f)\ \text{“}\texttt{bang}\text{”} \end{aligned}$$

On the surface, the expression $\texttt{ref}\ (\lambda x.x)$ could be given the polymorphic type $\texttt{ref}\ (a \to a)$, for any choice of a, since $\lambda x.x$ is a generic identity function. Assigning this type to f makes the expression type check, since both the update $f := (\lambda x.1 + x)$ and the dereference $(!f)$ “bang” are specializations of the generic type of f: $\texttt{ref}\ (\texttt{int} \to \texttt{int})$ and $\texttt{ref}\ (\texttt{string} \to \texttt{string})$.

However, this expression will result in a type error, since its evaluation will eventually reach the state $1 +$ “bang” which is clearly ill-typed. This problem is not just an issue with mutable state; similar type unsafe examples can be written for other effects like first-class control [13]. ML imposes the *value restriction* on its implicit polymorphism because of these issues, which keeps the language type safe by ruling out counterexamples like the above.

The full generality of polymorphism isn't required for this counterexample, though. We could instead assign the much more specific type $(\texttt{ref}\ (\texttt{int} \to \texttt{int})) \cap (\texttt{ref}\ (\texttt{string} \to \texttt{string}))$ to f, since the generic identity function has both types; $(\lambda x.x) : \texttt{int} \to \texttt{int}$ and $(\lambda x.x) : \texttt{string} \to \texttt{string}$. Doing so again makes the whole expression type check, which is quite undesirable since it leads to a type error. Therefore, once computational effects enter the picture, unrestricted intersection types are no longer safe in a call-by-value language like ML. Dually, unrestricted use of union types is not type safe in a call-by-name language.

3. Computation in a Classical Sequent Calculus

In order to address both issues with intersection and union types—the loss of subject reduction and type safety—we will move away from the λ-calculus and onto Curien and Herbelin's $\overline{\lambda}\mu\tilde{\mu}$-calculus [14]: a core programming language based on the sequent calculus rather than on natural deduction. The syntax, operational semantics, and simple type system of the $\overline{\lambda}\mu\tilde{\mu}$ sequent calculus, which is the starting point for every other type system to follow, are given in Figure 4 and Figure 5. Due to the syntactic structure of the $\overline{\lambda}\mu\tilde{\mu}$-calculus, every step of the operational semantics (denoted by $\mapsto$) is always at the top of a command, and a final command that can no longer take a step has the form c_{fin}. In contrast, the reduction theory, which allows for reduction rules to be applied in *any* context (denoted by $\to$), and not just at the top of the command, is the compatible closure of the operational steps ($\mapsto$).

Notation

As notation, for any set of rules R, we write the operational R-step relation as $\mapsto_R$ and use $\to_R$ to denote the compatible closure of $\mapsto_R$ (*i.e.,* an R-reduction $\to_R$ denotes a $\mapsto_R$ step applied in any

$$\texttt{Command} \ni c ::= \langle v \parallel e \rangle \qquad \texttt{Term} \ni v ::= x \mid \mu\alpha.c \mid \lambda x.v \qquad \texttt{CoTerm} \ni e ::= \alpha \mid \tilde{\mu}x.c \mid v \cdot e$$

$$\langle \mu\alpha.c \parallel e \rangle \mapsto_{\mu} c\,[e/\alpha] \qquad \langle v \parallel \tilde{\mu}x.c \rangle \mapsto_{\tilde{\mu}} c\,[v/x] \qquad \left\langle \lambda x.v \,\middle\|\, v' \cdot e \right\rangle \mapsto_{\beta} \left\langle v\left[v'/x\right] \,\middle\|\, e \right\rangle$$

$$\texttt{FinalCommand} \ni c_{\mathit{fin}} ::= \langle x \parallel \alpha \rangle \mid \langle \lambda x.v \parallel \alpha \rangle \mid \langle x \parallel v \cdot e \rangle$$

Figure 4: $\overline{\lambda}\mu\tilde{\mu}$ — Syntax and operational semantics.

$$\begin{aligned} \texttt{InputEnv} &\ni \quad \Gamma ::= x_1 : A_n, \ldots, x_n : A_n \\ \texttt{OutputEnv} &\ni \quad \Delta ::= \alpha_1 : A_n, \ldots, \alpha_n : A_n \\ \texttt{Judgement} &\ni H, J ::= (\Gamma \vdash v : A \mid \Delta) \mid (\Gamma \mid e : A \vdash \Delta) \mid c : (\Gamma \vdash \Delta) \end{aligned}$$

$$\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma \mid e : A \vdash \Delta}{\langle v \parallel e \rangle : (\Gamma \vdash \Delta)}\ \mathit{Cut}$$

$$\frac{}{\Gamma, x : A \vdash x : A \mid \Delta}\ \mathit{VarR} \qquad \frac{}{\Gamma \mid \alpha : A \vdash \alpha : A, \Delta}\ \mathit{VarL}$$

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

$$\frac{\Gamma, x : A \vdash v : B \mid \Delta}{\Gamma \vdash \lambda x.v : A \to B \mid \Delta} \to R \qquad \frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma \mid e : B \vdash \Delta}{\Gamma \mid v \cdot e : A \to B \vdash \Delta} \to L$$

Figure 5: $\overline{\lambda}\mu\tilde{\mu}$ — A simply typed sequent calculus.

context). The reflexive-transitive closure of $\mapsto_R$ and $\to_R$ is written as $\mapsto\!\!\!\!\twoheadrightarrow_R$ and $\to\!\!\!\!\twoheadrightarrow_R$, respectively. We also denote the reflexive closure of $\mapsto_R$ as $\mapsto_R^?$.

Environments $\Gamma = \{x_1 : A_1, \ldots, x_n : A_n\}$ and $\Delta = \{\alpha_1 : A_1, \ldots, \alpha_n : A_n\}$ are sets (*i.e.,* unordered lists) of basic type assignments, such that every variable x_i in Γ is distinct from all the others, and likewise for every covariable α_i in Δ. We usually will omit the braces when writing environments. The set $\mathit{Dom}(\Gamma) = \{x_1, x_2, \ldots, x_n\}$ is the set of variables assigned in Γ and $\mathit{Dom}(\Delta)$ is the set of covariables assigned in Δ.

The definitions of *free* and *bound* variables are conventional for the $\overline{\lambda}\mu\tilde{\mu}$-calculus. $\lambda x.v$ and $\tilde{\mu}x.c$ binds x in v and c, respectively, and $\mu\alpha.c$ binds α in c. A variable or covariable that is not bound is called free. Capture-avoiding substitution is written as $c[v/x]$ and $c[e/\alpha]$ (and similarly with a term or coterm in place of c) which replaces v for each free occurence of x and e for each free occurence of α so long as these occurences do not appear in a context which binds any free variable or covariable in

$$\texttt{Value}_v \ni V_v ::= x \mid \lambda x.v \qquad \texttt{FinalCommand}_{finv} \ni c_{finv} ::= \langle x \parallel \alpha \rangle \mid \langle \lambda x.v \parallel \alpha \rangle \mid \langle x \parallel V_v \cdot e \rangle$$

$$\langle \mu\alpha.c \parallel e \rangle \mapsto_{\mu_v} c\,[e/\alpha] \qquad \langle \lambda x.v \parallel V_v \cdot e \rangle \mapsto_{\beta_v} \langle v\,[V_v/x] \parallel e \rangle$$

$$\langle V_v \parallel \tilde{\mu}x.c \rangle \mapsto_{\tilde{\mu}_v} c\,[V_v/x] \qquad v \cdot e \mapsto_{\varsigma_v} \tilde{\mu}y.\langle v \parallel \tilde{\mu}x.\langle y \parallel x \cdot e \rangle \rangle \quad (v \notin \texttt{Value}_v)$$

$$\frac{e \mapsto_{\varsigma_v} e'}{\langle V_v \parallel e \rangle \mapsto_{\varsigma_v} \langle V_v \parallel e' \rangle}$$

Figure 6: Call-by-value operational semantics for $\overline{\lambda}\mu\tilde{\mu}$.

v or e. More details about binding can be found in Appendix A.

3.1. Call-by-value and call-by-name computation

The fundamental dilemma of computation in the sequent calculus, going back all the way to Gentzen's original cut elimination procedure, is that there is a non-deterministic choice of which step to take. The non-deterministic choice is neatly summarized by the critical pair between opposed μ- and $\tilde{\mu}$-redexes:

$$c_1\,[\tilde{\mu}x.c_2/\alpha] \leftarrow\!\!\!\shortmid_{\mu} \langle \mu\alpha.c_1 \parallel \tilde{\mu}x.c_2 \rangle \mapsto_{\tilde{\mu}} c_2\,[\mu\alpha.c_1/x]$$

In the worst case, where x is not free in c_2 and α is not free in c_1, we have two completely diverging and unconnected reduction paths:

$$c_1 \leftarrow\!\!\!\shortmid_{\mu} \langle \mu\alpha.c_1 \parallel \tilde{\mu}x.c_2 \rangle \mapsto_{\tilde{\mu}} c_2$$

This non-deterministic choice can be either a weakness or a strength, depending on one's point of view. In Barbanera and Berardi's symmetric λ-calculus [15], non-determinism was embraced as part of the classical reduction system. Whereas in Curien and Herbelin's $\overline{\lambda}\mu\tilde{\mu}$-calculus [14], determinism was restored by imposing a discipline onto reduction which prioritized one side over the other:

> Call-by-value consists in giving priority to the (μ)-redexes (which serve to encode the terms, say, of the form $M\ N$), while call-by-name gives priority to the $(\tilde{\mu})$-redexes.

Call-by-value

The call-by-value semantics can be formalized by making use of a notion of *value* (written V_v) that is either a variable or a λ-abstraction, as given in Figure 6. Call-by-value reduction gives priority to the μ_v-redexes since the $\tilde{\mu}_v$ rule only substitutes values and a μ-abstraction is never a value in call-by-value. And since a μ_v step can substitute any coterm, the fundamental dilemma is resolved in favor of μ. To go along with this value restriction, the β_v step for reducing function calls is also restricted to only substitute a value argument, analogous to the call-by-value λ-calculus. Lastly, a command of the form $\langle x \parallel v \cdot e \rangle$ is only final if v is a value; in other words, the argument of a function call is always eagerly evaluated.

$$\texttt{CoValue}_n \ni E_n ::= \alpha \mid v \cdot E_n \qquad \texttt{FinalCommand}_n \ni c_{finn} ::= \langle x \parallel E_n \rangle \mid \langle \lambda x.v \parallel \alpha \rangle$$

$$\langle \mu\alpha.c \parallel E_n \rangle \mapsto_{\mu_n} c\,[E_n/\alpha] \qquad \left\langle \lambda x.v \parallel v' \cdot E_n \right\rangle \mapsto_{\beta_n} \left\langle v \left[v'/x\right] \parallel E_n \right\rangle$$

$$\langle v \parallel \tilde{\mu}x.c \rangle \mapsto_{\tilde{\mu}_n} c\,[v/x] \qquad v \cdot e \mapsto_{\varsigma_n} \tilde{\mu}y.\,\langle \mu\alpha.\,\langle y \parallel v \cdot \alpha \rangle \parallel e \rangle \quad (e \notin \texttt{CoValue}_n)$$

$$\frac{e \mapsto_{\varsigma_n} e'}{\langle v \parallel e \rangle \mapsto_{\varsigma_n} \langle v \parallel e' \rangle}$$

Figure 7: Call-by-name operational semantics for $\overline{\lambda}\mu\tilde{\mu}$.

In addition to these three reduction rules from $\overline{\lambda}\mu\tilde{\mu}$, we also have a ς rule—first appearing in Wadler's dual calculus [16]—whose purpose is to *lift* out a serious non-value argument v buried in a call-stack $v \cdot e$, thereby refocusing the attention of a command onto v. The ς rule is necessary in the sequent calculus for type safety of call-by-value evaluation by making sure a command does not get stuck prematurely before it reaches a finished state [17], to establish an exact correspondence with the call-by-value reduction in the λ-calculus [18], and to build a uniform model of strong normalization [1]. Similar forms of additional reductions are also known to appear in the call-by-value λ-calculus, which allow to reach more normal forms [19, 20], and to achieve completeness with respect to the continuation-passing style transformation [21]. The fact that a ς_v is applied to the top coterm of a command once the term-side is a value corresponds to a left-to-right evaluation order in the call-by-value λ-calculus, where the argument of an application is evaluated after the function is.

Call-by-name

The dual of call-by-value is call-by-name, whose operational semantics is given in Figure 7. Instead of values, call-by-name reduction uses to the notion of *covalue* (written E_n) that is either a covariable or a call-stack $v \cdot E_n$ of an argument and another covalue. Call-by-name reduction gives priority to the $\tilde{\mu}_n$-redexes since a μ_n only substitutes covalues and a $\tilde{\mu}$-abstraction is never a covalue in call-by-name. And since a $\tilde{\mu}_n$ rule can substitute any term, the fundamental dilemma is resolved in favor of $\tilde{\mu}$. The duality between call-by-name and call-by-value is also seen in the rules for functions. The β_n rule only applies when the result of the function call is needed (as expressed by the restriction that the call stack has the form $v \cdot E_n$). Call-by-name also has a dual form of ς reduction which lifts out a non-strict, non-covalue coterm e buried in a call-stack $v \cdot e$. The call-by-name ς rule results in demand-driven computation, where the attention of a command is refocused first onto the calling context of every function call, so that functions are delayed until demanded by their caller. As a result, a command of the form $\langle x \parallel e \rangle$ is only finished in call-by-name if e is a covalue; in other words, computation continues until x is finally demanded by e.

$$c_{fn} ::= \langle v_{fn} \parallel e_{fn} \rangle \quad v_{fn} ::= x \mid \mu\alpha.c_{fn} \mid \lambda x.v_{fn} \quad e_{fn} ::= E_{fn} \mid \tilde{\mu}x.c_{fn} \quad E_{fn} ::= \alpha \mid v_{fn} \cdot E_{fn}$$

$$\texttt{Judgement} \ni H, J ::= (\Gamma \vdash v_{fn} : A \mid \Delta) \mid (\Gamma \mid e_{fn} : A \vdash \Delta) \mid (\Gamma \,;\, E_{fn} : A \vdash \Delta) \mid c_{fn} : (\Gamma \vdash \Delta)$$

$$\frac{}{\Gamma \,;\, \alpha : A \vdash \alpha : A, \Delta}\ VarL \qquad \frac{\Gamma \vdash v_{fn} : A \mid \Delta \quad \Gamma \,;\, E_{fn} : B \vdash \Delta}{\Gamma \,;\, v_{fn} \cdot E_{fn} : A \to B \vdash \Delta} \to L \qquad \frac{\Gamma \,;\, E_{fn} : A \vdash \Delta}{\Gamma \mid E_{fn} : A \vdash \Delta}\ FL$$

Plus the Cut, $VarR$, $ActR$, $ActL$, and $\to I$ rules from Figure 5 (replacing c, v, and e with c_{fn}, v_{fn}, and e_{fn}, respectively).

Figure 8: Focused syntax and types of call-by-name $\overline{\lambda}\mu\tilde{\mu}$.

Focusing in Call-by-name and Call-by-value

The ς reduction rules have the ability to refocus the attention of a command, spurring computation forward in certain execution states. As it turns out, these ς reductions can be done entirely in advance, as a "compile-time" pre-process, instead of waiting until the last possible moment, as a "run-time" step. Because of this equivalence between ς at compile-time and run-time [17], the impact of ς reduction corresponds to *focusing* in logic [22, 23]. So for our purposes, focusing is equivalent to ς-normalization, and since ς reduction is different for call-by-value and call-by-name, it follows that there are different notions of focusing in the sequent calculus.

The result of focusing is a language that appears to be closer to a conventional abstract machine, and can be understood in the grammar of a specialized sub-syntax of $\overline{\lambda}\mu\tilde{\mu}$. The call-by-name focused sub-syntax of $\overline{\lambda}\mu\tilde{\mu}$ is given in Figure 8. In other words, call-by-name focused coterms are either a $\tilde{\mu}$-abstraction (corresponding to a let-binding in the λ-calculus), or a call-stack of the form $v_1 \cdot v_2 \cdot \ldots v_n \cdot \alpha$ (corresponding to the application $\square\ v_1\ v_2\ \ldots\ v_n$). In particular, a coterm of the form $v \cdot \tilde{\mu}x.c$ is not allowed in the call-by-name focused sub-syntax. The impact of focusing can be seen as part of type system as well, through an additional judgement for typing call-by-name covalues, $\Gamma \,;\, E_{fn} : A \vdash \Delta$. The new symbol (;) called the *stoup* [24] signifies that we are typing a covalue *in focus* rather than a general coterm. With this new form of judgement, the type system can express restrictions on the typing rules, along with the new rule FL for placing *focus* on a covalue, as described in Figure 8. Note that the focus on call-by-name covalues is hereditary, because the premise of the $\to L$ typing rule keeps the covalue in the stoup.

Dually, the grammar of the call-by-value focused sub-syntax of $\overline{\lambda}\mu\tilde{\mu}$ is given in Figure 9. In other words, in call-by-value, focused coterms can only push a value onto a call-stack. From the perspective of typing, call-by-value gives a different form of focused judgement, $\Gamma \vdash V_{fv} : A \,;\, \Delta$, wherein a call-by-value value appears in the stoup. With this new form of judgement, the call-by-value type system can express the restricted typing rules, along with the new rule FR for placing *focus* on a value, as given in Figure 9. Notice that the premise of $\to R$ is not in focus (that is, it uses the other form of judgement for general terms) because the body of a λ-abstraction need not be a value for the λ-abstraction itself to be a value. Additionally, focus is *gained* in the premise of the $\to L$ rule which

$$c_{fv} ::= \langle v_{fv} \parallel e_{fv} \rangle \quad V_{fv} ::= x \mid \lambda x.v_{fv} \quad v_{fv} ::= V_{fv} \mid \mu\alpha.c_{fv} \quad e_{fv} ::= \alpha \mid \tilde{\mu}x.c_{fv} \mid V_{fv} \cdot e_{fv}$$

$$\texttt{Judgement} \ni H, J ::= (\Gamma \vdash v_{fv} : A \mid \Delta) \mid (\Gamma \vdash V_{fv} : A \,;\, \Delta) \mid (\Gamma \mid e_{fv} : A \vdash \Delta) \mid c_{fv} : (\Gamma \vdash \Delta)$$

$$\frac{}{\Gamma, x : A \vdash x : A \,;\, \Delta}\ VarR \qquad \frac{\Gamma, x : A \vdash v_{fv} : B \mid \Delta}{\Gamma \vdash \lambda x.v_{fv} : A \to B \,;\, \Delta} \to R \qquad \frac{\Gamma \vdash V_{fv} : A \,;\, \Delta \quad \Gamma \mid e_{fv} : B \vdash \Delta}{\Gamma \mid V_{fv} \cdot e_{fv} : A \to B \vdash \Delta} \to L$$

$$\frac{\Gamma \vdash V_{fv} : A \,;\, \Delta}{\Gamma \vdash V_{fv} : A \mid \Delta}\ FR$$

Plus the Cut, $VarR$, $ActR$, and $ActL$ rules from Figure 5 (replacing c, v, and e with c_{fv}, v_{fv}, and e_{fv}, respectively).

Figure 9: Focused syntax and type system for call-by-value $\overline{\lambda}\mu\tilde{\mu}$.

$$\begin{array}{ccc} \texttt{Value}_v \ni V_v ::= x \mid \lambda x.v & \texttt{Value}_n \ni V_n ::= v & \texttt{Value}_u \ni V_u ::= v \\ \texttt{CoValue}_v \ni E_v ::= e & \texttt{CoValue}_n \ni E_n ::= \alpha \mid v \cdot E_n & \texttt{CoValue}_u \ni E_u ::= e \end{array}$$

Figure 10: The call-by-value (v), call-by-name (n), and non-deterministic (u) disciplines.

type checks the argument, since only values can be pushed onto a focused call-by-value call stack.

3.2. Disciplined computation

So far we have presented three operational semantics for $\overline{\lambda}\mu\tilde{\mu}$: non-deterministic, call-by-value, and call-by-name. Because of focusing and ς reduction, there is no "biggest" system encompassing both call-by-value and call-by-name reduction, so they each must be considered by their own merit. However, we do not need to study each one independently: they can all be united by a common notion that distills their differences. We now show how these different systems can be uniformly characterized by the dual notions of *value* and *covalue*, which already arose in call-by-value and call-by-name above. We refer to the specification of what is substitutable as a *discipline* [25, 1].

Definition 3.1. (Discipline)
A *discipline* d is a subset of terms called *d-values* (denoted by the metavariable V_d) along with a subset of coterms called *d-covalues* (denoted by the metavariable E_d). As shorthand, we denote a *d-non-value* (*i.e.,* a term which is not a d-value) by the metavariable w_d and a *d-non-covalue* (*i.e.,* a coterm which is not a d-covalue) by the metavariable f_d. When the discipline can be inferred from context, we may refer to d-values and d-covalues as just values and covalues, respectively.

$$\texttt{FinalCommand}_d \ni c_{\mathit{find}} ::= \langle x \parallel \alpha \rangle \mid \langle \lambda x.v \parallel \alpha \rangle \mid \langle x \parallel V_d \cdot E_d \rangle$$

$$\langle \mu\alpha.c \parallel E_d \rangle \mapsto_{\mu_d} c\,[E_d/\alpha] \qquad w_d \cdot e \mapsto_{\varsigma_d} \tilde{\mu}y.\, \langle w_d \parallel \tilde{\mu}x.\, \langle y \parallel x \cdot e \rangle \rangle$$

$$\langle V_d \parallel \tilde{\mu}x.c \rangle \mapsto_{\tilde{\mu}_d} c\,[V_d/x] \qquad V_d \cdot f_d \mapsto_{\varsigma_d} \tilde{\mu}y.\, \langle \mu\alpha.\, \langle y \parallel V_d \cdot \alpha \rangle \parallel f_d \rangle$$

$$\langle \lambda x.v \parallel V_d \cdot E_d \rangle \mapsto_{\beta_d} \langle v\,[V_d/x] \parallel E_d \rangle$$

$$\frac{e \mapsto_{\varsigma_d} e'}{\langle V_d \parallel e \rangle \mapsto_{\varsigma_d} \langle V_d \parallel e' \rangle}$$

Figure 11: Operational semantics for the disciplined $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$-calculus, for the discipline d.

The definition of the call-by-value (v), call-by-name (n), and non-deterministic (u) disciplines is given in Figure 10. Once a choice of discipline is made, there is a disciplined operational semantics to go along with it, given in Figure 11. This captures the common ground between each of the three evaluation strategies, where the unifying idea in disciplined reduction rules is that *only* values are substituted for variables and *only* covalues are substituted for covariables. In particular, note that the specific instances of the μ_d, $\tilde{\mu}_d$, β_d, and ς_d rules when d is u, v, and n are *exactly* the rules given in Figures 4, 6, and 7, respectively. In particular, the restrictions on values and covalues—as well as on non-values and non-covalues in the case of ς_d—match what is done by the non-deterministic, call-by-value, and call-by-name operational semantics.

Due to the ς rules, the operational semantics is enough to compute the result of a program by getting a *finished* command of the form $\langle x \parallel \alpha \rangle$, $\langle x \parallel V_d \cdot E_d \rangle$, or $\langle \lambda x.v \parallel \alpha \rangle$ when d is any of the three disciplines v, n, and u discussed here. However, note that the ς rules *never* apply for non-deterministic reduction (since every term is a u-value and every coterm is a u-covalue). It is also for this reason that, once focusing by ς reduction is taken into account, non-deterministic reduction is not the "all-encompassing" system. Instead, none of the call-by-value (v), call-by-name (n), or non-deterministic (u) reduction systems are subsets of the others. This means that we cannot just, say, reduce the study of call-by-name and call-by-value reduction to be special cases of non-deterministic reduction without losing crucial steps.

Disciplined focusing

The notion of discipline also unifies the different focusing regimes in terms of ς reduction.

Definition 3.2. (Focused)
The *d-focused* sub-syntax for a discipline d is exactly the ς_d-normal forms. In other words, call-stack coterms are restricted to the form $V_d \cdot E_d$.

Notice that the special instances for n-focused and v-focused sub-syntaxes given by Definition 3.2 are *exactly* the same as the definitions for call-by-name and call-by-value focusing given previously in Section 3.1. In stark contrast, the u-focused sub-syntax for the non-deterministic discipline u is

exactly the full syntax (Figure 4) and the full typing system (Figure 12). In other words, choosing the non-deterministic u reduction discipline corresponds to the completely unfocused typing discipline.

Admissible disciplines

Our uniform treatment of the (co)value restriction works by not committing to a specific discipline, like call-by-value or -name, as is usually done. Instead, we characterize the necessary properties of the discipline which control substitution in reduction rules, which are the only facts that need to be known about values and covalues. The first such property is the admissibility of the discipline, which says that *enough* terms and coterms are admitted as values and covalues (focusing) and that the status of values and covalues are not changed by certain actions (stability).

Definition 3.3. (Admissibility)
A discipline d is

- *focusing* if (i) x is a d-value and α is a d-covalue, (ii) $\lambda x.v$ is an d-value, and (iii) for any d-value V_d and d-covalue E_d, $V_d \cdot E_d$ is a d-covalue,
- *stable* if the set of d-values and d-covalues are invariant under (i) reduction (*i.e.,* for all $v \to v'$, v is a d-value if and only if v' is, and dually for coterms), and (ii) substitution of d-values for variables and d-covalues for covariables (*i.e.,* $v\,[V_d/x]$ is a d-value if and only if v is, *etc.*), and
- *admissible* if it is both focusing and stable.

Proposition 3.4. The v, n, and u disciplines are all admissible.

The second property which is useful to highlight is determinism (or lack thereof) of a discipline.

Definition 3.5. (Deterministic Discipline)
A discipline is *deterministic* if the $\mapsto$ relation is deterministic.

Proposition 3.6. The v and n disciplines are both deterministic, but the u discipline is not.

4. Intersection and Union Types in the Sequent Calculus

In order to get an intuition of what intersection and union types look like in the symmetric setting of the sequent calculus, we will first look at the unrestricted system. Afterward, we consider how to apply some additional discipline to this naïve system to restore desirable properties like type safety, subject reduction, and strong normalization.

$$\texttt{Type} \ni A, B, C, D ::= p \mid A \to B \mid A \cap B \mid A \cup B$$

$$\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma \vdash v : B \mid \Delta}{\Gamma \vdash v : A \cap B \mid \Delta} \cap R \qquad \frac{\Gamma \mid e : A_i \vdash \Delta}{\Gamma \mid e : A_1 \cap A_2 \vdash \Delta} \cap L \qquad \frac{\Gamma \vdash v : A_1 \cap A_2 \mid \Delta}{\Gamma \vdash v : A_i \mid \Delta} \cap E$$

$$\frac{\Gamma \vdash v : A_i \mid \Delta}{\Gamma \vdash v : A_1 \cup A_2 \mid \Delta} \cup R \qquad \frac{\Gamma \mid e : A \vdash \Delta \quad \Gamma \mid e : B \vdash \Delta}{\Gamma \mid e : A \cup B \vdash \Delta} \cup L \qquad \frac{\Gamma \mid e : A_1 \cup A_2 \vdash \Delta}{\Gamma \mid e : A_i \vdash \Delta} \cup E$$

In the rules $\cap L$, $\cap E$, $\cup R$, and $\cup E$, the index i ranges over 1 or 2.

Figure 12: $\overline{\lambda}\mu\tilde{\mu}\cap\cup$ — Full intersection and union types in the $\overline{\lambda}\mu\tilde{\mu}$ sequent calculus.

4.1. The starting point — Full intersection and union types

Our first attempt at a type system for intersection and union types in the $\overline{\lambda}\mu\tilde{\mu}$ sequent calculus is given in Figure 12, which extends Figure 5. Following the pattern of the simple $\overline{\lambda}\mu\tilde{\mu}$ type system, $\overline{\lambda}\mu\tilde{\mu}\cap\cup$ has both right ($\cap R$ and $\cup R$) and left ($\cap L$ and $\cup L$) inference rules for introducing intersection and union types on terms and coterms. $\overline{\lambda}\mu\tilde{\mu}\cap\cup$ also includes the rule $\cap E$ from $\lambda\cap$ for eliminating intersection types of terms, as well as the dual $\cup E$ for eliminating union types of coterms. These symmetric eliminations are useful for the study of completeness (Section 5).

Since $\overline{\lambda}\mu\tilde{\mu}$ is a calculus that offers first-class control effects, *both* union and intersection types are required for typing arbitrary normal forms, unlike the purely functional lambda calculus.

Example 4.1. Intersection types make it possible to use the same variable in many different contexts. The normal form representing self-application is typable due to the $\cap L$ and $\cap E$ rules. The derivation of $\lambda x.\mu\alpha.\, \langle x \parallel x \cdot \alpha \rangle : A \cap (A \to B) \to B$ proceeds as follows, where $\Gamma = x : A \cap (A \to B)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : A \cap (A \to B) \mid \alpha : B}\, VarR}{\Gamma \vdash x : A \mid \alpha : B}\, \cap E \qquad \dfrac{}{\Gamma \mid \alpha : B \vdash \alpha : B}\, VarL}{\Gamma \mid x \cdot \alpha : A \to B \vdash \alpha : B}\, \to L}{\Gamma \mid x \cdot \alpha : A \cap (A \to B) \vdash \alpha : B}\, \cap L}{\langle x \parallel x \cdot \alpha \rangle : (\Gamma \vdash \alpha : B)}\, Cut}{\Gamma \vdash \mu\alpha.\, \langle x \parallel x \cdot \alpha \rangle : B \mid}\, ActR}{\vdash \lambda x.\mu\alpha.\, \langle x \parallel x \cdot \alpha \rangle : A \cap (A \to B) \to B \mid}\, \to R$$

Example 4.2. Union types make it possible to use the same covariable in many different contexts. The following normal form is typable due to the $\cup R$ and $\cup E$ rules. The derivation of $\mu\alpha.\, \langle \lambda x.\mu\beta.\, \langle x \parallel \alpha \rangle \parallel \alpha \rangle$:

$A \cup (A \to B)$ proceeds as follows, where $\Delta = \alpha : A \cup (A \to B)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : A \vdash x : A \mid \beta : B, \Delta}\,VarR \quad \dfrac{\dfrac{}{x : A \mid \alpha : A \cup (A \to B) \vdash \beta : B, \Delta}\,VarL}{x : A \mid \alpha : A \vdash \beta : B, \Delta}\,\cup E}{\langle x \parallel \alpha \rangle : (x : A \vdash \beta : B, \Delta)}\,Cut}{x : A \vdash \mu\beta.\langle x \parallel \alpha \rangle : B \mid \Delta}\,ActR}{\vdash \lambda x.\mu\beta.\langle x \parallel \alpha \rangle : A \to B \mid \Delta}\,{\to}R}{\vdash \lambda x.\mu\beta.\langle x \parallel \alpha \rangle : A \cup (A \to B) \mid \Delta}\,{\cup}R \qquad \dfrac{}{\mid \alpha : A \cup (A \to B) \vdash \Delta}\,VarL}{\dfrac{\langle \lambda x.\mu\beta.\langle x \parallel \alpha \rangle \parallel \alpha \rangle : (\vdash \Delta)}{\vdash \mu\alpha.\langle \lambda x.\mu\beta.\langle x \parallel \alpha \rangle \parallel \alpha \rangle : A \cup (A \to B) \mid}\,ActR}\,Cut$$

This normal form is not typable without union types. Therefore, in $\overline{\lambda}\mu\tilde{\mu}$ both intersection and union types are needed to type all normal forms. Remember that in the intuitionistic case, *i.e.,* in lambda calculus, intersection types were sufficient to type all normal forms [9]. Adding union to lambda calculus with intersection types did not enlarge the set of typable terms. In contrast, in the symmetric classical case, union types—as the dual to intersection types—in fact allow us to type more expressions.

Lack of subject reduction

Even though intersection and union types allow for the same variable or covariable to appear in many contexts (which is essential for the completeness property that every strongly normalizing expression is well-typed), the typing system from Figure 12 falls short of other desirable properties. For example, it does not enjoy subject reduction, similar to $\lambda\cap\cup$ which has a troublesome elimination rule for union types. However, note that for the symmetric $\overline{\lambda}\mu\tilde{\mu}$ which more fully expresses the duality of types and programs, *both* intersection and union types, and specifically the $\cap R$ and $\cup L$ typing rules, can be responsible for breaking subject reduction.

Example 4.3. Consider the following typing derivation

$$\dfrac{\dfrac{}{\Gamma \vdash x : A_1 \cup A_2 \mid \Delta}\,VarR \quad \dfrac{\begin{array}{cc}\vdots\,\mathcal{D}_1 & \vdots\,\mathcal{D}_2 \\ \Gamma \mid \tilde{\mu}y.\langle z \parallel y \cdot y \cdot \alpha \rangle : A_1 \vdash \Delta & \Gamma \mid \tilde{\mu}y.\langle z \parallel y \cdot y \cdot \alpha \rangle : A_2 \vdash \Delta\end{array}}{\Gamma \mid \tilde{\mu}y.\langle z \parallel y \cdot y \cdot \alpha \rangle : A_1 \cup A_2 \vdash \Delta}\,{\cup}L}{\langle x \parallel \tilde{\mu}y.\langle z \parallel y \cdot y \cdot \alpha \rangle \rangle : (\Gamma \vdash \Delta)}\,Cut$$

given the environments

$$\Gamma = z : (A_1 \to A_1 \to B) \cap (A_2 \to A_2 \to B), x : A_1 \cup A_2 \qquad \Delta = \alpha : B$$

and the similar sub-derivations $\mathcal{D}_1$ and $\mathcal{D}_2$ are instances of the following (with $i = 1, 2$, respectively):

$$\dfrac{\dfrac{\dfrac{}{\Gamma, y : A_i \vdash z : (A_1 \to A_1 \to B) \cap (A_2 \to A_2 \to B) \mid \Delta}\,VarR}{\Gamma, y : A_i \vdash z : A_i \to A_i \to B \mid \Delta}\,{\cap}E \quad \begin{array}{c}\vdots \\ \Gamma, y : A_i \mid y \cdot y \cdot \alpha : A_i \to A_i \to B \vdash \Delta\end{array}}{\dfrac{\langle z \parallel y \cdot y \cdot \alpha \rangle : (\Gamma, y : A_i \vdash \Delta)}{\Gamma \mid \tilde{\mu}y.\langle z \parallel y \cdot y \cdot \alpha \rangle : A_i \vdash \Delta}\,ActL}\,Cut$$

After a single reduction step, we have

$$\langle x \parallel \tilde{\mu} y.\, \langle z \parallel y \cdot y \cdot \alpha \rangle \rangle \to_{\tilde{\mu}} \langle z \parallel x \cdot x \cdot \alpha \rangle$$

but there is no derivation of $\langle z \parallel x \cdot x \cdot \alpha \rangle : (\Gamma \vdash \Delta)$ in $\overline{\lambda}\mu\tilde{\mu}\cap\cup$. In other words, unfortunate applications of $\cup L$ can break subject reduction.

Example 4.4. Consider the following typing derivation

$$\dfrac{\dfrac{\dfrac{\vdots\ \mathcal{D}_1}{\langle x \parallel \lambda y.\mu\delta.\, \langle y \parallel \beta \rangle \cdot \beta \rangle : (\Gamma \vdash \Delta_1)}}{\Gamma \vdash \mu\beta.\, \langle x \parallel \lambda y.\mu\delta.\, \langle y \parallel \beta \rangle \cdot \beta \rangle : A_1 \mid \Delta}\, ActR \quad \dfrac{\dfrac{\vdots\ \mathcal{D}_2}{\langle x \parallel \lambda y.\mu\delta.\, \langle y \parallel \beta \rangle \cdot \beta \rangle : (\Gamma \vdash \Delta_2)}}{\Gamma \vdash \mu\beta.\, \langle x \parallel \lambda y.\mu\delta.\, \langle y \parallel \beta \rangle \cdot \beta \rangle : A_2 \mid \Delta}\, ActR}{\Gamma \vdash \mu\beta.\, \langle x \parallel \lambda y.\mu\delta.\, \langle y \parallel \beta \rangle \cdot \beta \rangle : A_1 \cap A_2 \mid \Delta}\, \cap R \quad \dfrac{}{\Gamma \mid \alpha : A_1 \cap A_2 \vdash \Delta}\, VarL}{\langle \mu\beta.\, \langle x \parallel \lambda y.\mu\delta.\, \langle y \parallel \beta \rangle \cdot \beta \rangle \parallel \alpha \rangle : (\Gamma \vdash \Delta)}\, Cut$$

given the environments

$$\Gamma = x : ((A_1 \to B) \to A_1) \cap ((A_2 \to B) \to A_2) \qquad \Delta = \alpha : A_1 \cap A_2 \qquad \Delta_i = \beta : A_i, \Delta$$

and the similar sub-derivations $\mathcal{D}_1$ and $\mathcal{D}_2$ are instances of the following (with $i = 1, 2$, respectively):

$$\dfrac{\dfrac{}{\Gamma \vdash x : (A_i \to B) \to A_i \mid \Delta_i}\, \cap E,\, VarR \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\langle y \parallel \beta \rangle : (\Gamma, y : A_i \vdash \delta : B, \Delta_i)}\, Cut,\, VarR,\, VarL}{\Gamma, y : A_i \vdash \mu\delta.\, \langle y \parallel \beta \rangle : B \mid \Delta_i}\, ActR}{\Gamma \vdash \lambda y.\mu\delta.\, \langle y \parallel \beta \rangle : A_i \to B \mid \Delta_i}\, \to R \quad \dfrac{}{\Gamma \mid \beta : A_i \vdash \Delta_i}\, VarL}{\Gamma \mid \lambda y.\mu\delta.\, \langle y \parallel \beta \rangle \cdot \beta : (A_i \to B) \to A_i \vdash \Delta_i}\, \to L}{\langle x \parallel \lambda y.\mu\delta.\, \langle y \parallel \beta \rangle \cdot \beta \rangle : (\Gamma \vdash \Delta_i)}\, Cut$$

After a single reduction step, we have

$$\langle \mu\beta.\, \langle x \parallel \lambda y.\mu\delta.\, \langle y \parallel \beta \rangle \cdot \beta \rangle \parallel \alpha \rangle \to_{\mu} \langle x \parallel \lambda y.\mu\delta.\, \langle y \parallel \alpha \rangle \cdot \alpha \rangle$$

but there is no derivation of $\langle x \parallel \lambda y.\mu\delta.\, \langle y \parallel \alpha \rangle \cdot \alpha \rangle : (\Gamma \vdash \Delta)$ in $\overline{\lambda}\mu\tilde{\mu}\cap\cup$. In other words, unfortunate applications of $\cap R$ can break subject reduction. Note that in this example, intersection types alone are enough to break subject reduction in terms which can duplicate their continuation; foregoing union types would not help correct this counterexample.

Since we are not willing to just give up on type safety and subject reduction for the sake of completeness, we must do something to repair the system without losing completeness. As it turns out, the solution to ensuring both subject reduction and soundness (the fact that every well-typed expression is strongly normalizing), is closely connected with the solution to the *fundamental dilemma of computation* in the classical sequent calculus, as discussed in the previous section.

4.2. Call-by-value and call-by-name intersection and union types

Recall from Section 2 that a naïve definition of intersection and union types can easily break desirable properties of the type system, including subject reduction and type safety. At least part of the issue

$$\frac{\Gamma \vdash V_v : A \mid \Delta \quad \Gamma \vdash V_v : B \mid \Delta}{\Gamma \vdash V_v : A \cap B \mid \Delta} \cap R \qquad \frac{\Gamma \mid e : A_i \vdash \Delta}{\Gamma \mid e : A_1 \cap A_2 \vdash \Delta} \cap L \qquad \frac{\Gamma \vdash v : A_1 \cap A_2 \mid \Delta}{\Gamma \vdash v : A_i \mid \Delta} \cap E$$

$$\frac{\Gamma \vdash v : A_i \mid \Delta}{\Gamma \vdash v : A_1 \cup A_2 \mid \Delta} \cup R \qquad \frac{\Gamma \mid e : A \vdash \Delta \quad \Gamma \mid e : B \vdash \Delta}{\Gamma \mid e : A \cup B \vdash \Delta} \cup L \qquad \frac{\Gamma \mid e : A_1 \cup A_2 \vdash \Delta}{\Gamma \mid e : A_i \vdash \Delta} \cup E$$

$$\frac{c : (\Gamma, x : A \vdash \Delta) \quad c : (\Gamma, x : B \vdash \Delta)}{c : (\Gamma, x : A \cup B \vdash \Delta)} \cup x$$

In the rules $\cap L$, $\cap E$, $\cup R$, and $\cup E$, the index i ranges over 1 or 2.

Figure 13: $\overline{\lambda}\mu\tilde{\mu}\cap\cup_v$ — Call-by-value intersection and union types, restricting $\cap R$ to values.

happens when inappropriately generalizing (via either polymorphism or intersections) the type of an effectful term. Since the classical sequent calculus has a built-in notion of control effect due to μ-abstractions, the full type system from Figure 12 is not type safe. A simple way to address the type safety problem in practice is with an ML-like value restriction. As we have seen in the previous section, call-by-value and call-by-name have different notions of values; they require a different, and dual, restriction in their typing rules.

Call-by-value

The call-by-value typing system with intersection and union types, $\overline{\lambda}\mu\tilde{\mu}\cap\cup_v$, is given in Figure 13 as an extension of Figure 5. It uses a *value restriction* for intersection types. Similar to ML's value restriction, which limits polymorphism to syntactic values, $\overline{\lambda}\mu\tilde{\mu}\cap\cup_v$ limits the introduction of intersection types to values. The impact of this restriction is seen in the $\cap R$ rule (which only applies to syntactic values of form V_v). As we will see later (in Section 6 and Section 7), this typing restriction is crucial for ensuring type safety and soundness, but is still permissive enough to allow for the completeness of typing all strongly-normalizing call-by-value expressions (Section 5).

In addition, there is an alternative version of the $\cup L$ rule, called $\cup x$, that introduces a union type to a free variable of a command. One of our goals is to study of subject reduction and type safety (Section 6), which is the motivation for considering this extra rule. $\cup x$ helps to expose the $\tilde{\mu}$-binder at the conclusion of a typing derivation. For example, we can reduce the following application of the $\cup L$ rule to $\cup x$ by pushing the $\tilde{\mu}$-binder introduced with $ActL$ down into the conclusion of the derivation:

$$\frac{\dfrac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta}\, ActL \quad \dfrac{c : (\Gamma, x : B \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : B \vdash \Delta}\, ActL}{\Gamma \mid \tilde{\mu}x.c : A \cup B \vdash \Delta} \cup L \quad \Rightarrow \quad \frac{\dfrac{c : (\Gamma, x : A \vdash \Delta) \quad c : (\Gamma, x : B \vdash \Delta)}{c : (\Gamma, x : A \cup B \vdash \Delta)} \cup x}{\Gamma \mid \tilde{\mu}x.c : A \cup B \vdash \Delta}\, ActL$$

As a result of this fact, we can circumvent the counter-example to subject reduction in Example 4.3

$$\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma \vdash v : B \mid \Delta}{\Gamma \vdash v : A \cap B \mid \Delta} \cap R \qquad \frac{\Gamma \mid e : A_i \vdash \Delta}{\Gamma \mid e : A_1 \cap A_2 \vdash \Delta} \cap L \qquad \frac{\Gamma \vdash v : A_1 \cap A_2 \mid \Delta}{\Gamma \vdash v : A_i \mid \Delta} \cap E$$

$$\frac{c : (\Gamma \vdash \alpha : A, \Delta) \quad c : (\Gamma \vdash \alpha : B, \Delta)}{c : (\Gamma \vdash \alpha : A \cap B, \Delta)} \cap \alpha$$

$$\frac{\Gamma \vdash v : A_i \mid \Delta}{\Gamma \vdash v : A_1 \cup A_2 \mid \Delta} \cup R \qquad \frac{\Gamma \mid E_n : A \vdash \Delta \quad \Gamma \mid E_n : B \vdash \Delta}{\Gamma \mid E_n : A \cup B \vdash \Delta} \cup L \qquad \frac{\Gamma \mid e : A_1 \cup A_2 \vdash \Delta}{\Gamma \mid e : A_i \vdash \Delta} \cup E$$

In the rules $\cap L$, $\cap E$, $\cup R$, and $\cup E$, the index i ranges over 1 or 2.

Figure 14: $\overline{\lambda}\mu\tilde{\mu}\cap\cup_n$ — Call-by-name intersection and union types, restricting $\cup L$ to covalues.

by typing the result of reduction, $\langle z \parallel x \cdot x \cdot \alpha \rangle$, as follows:

$$\frac{\begin{array}{c}\vdots\ \mathcal{D}_1 \\ \langle z \parallel x \cdot x \cdot \alpha \rangle : (\Gamma', x : A_1 \mid\vdash \Delta)\end{array} \quad \begin{array}{c}\vdots\ \mathcal{D}_2 \\ \langle z \parallel x \cdot x \cdot \alpha \rangle : (\Gamma', x : A_2 \mid\vdash \Delta)\end{array}}{\langle z \parallel x \cdot x \cdot \alpha \rangle : (\Gamma \vdash \Delta)} \cup x$$

given the same environments

$$\Gamma = \Gamma', x : A_1 \cup A_2 \qquad \Gamma' = z : (A_1 \to A_1 \to B) \cap (A_2 \to A_2 \to B) \qquad \Delta = \alpha : B$$

and the similar sub-derivations $\mathcal{D}_1$ and $\mathcal{D}_2$ are instances of the following (with $i = 1, 2$, respectively):

$$\dfrac{\dfrac{\dfrac{}{\Gamma', x : A_i \vdash z : (A_1 \to A_1 \to B) \cap (A_2 \to A_2 \to B) \mid \Delta}\, VarR}{\Gamma', x : A_i \vdash z : A_i \to A_i \to B \mid \Delta} \cap E \qquad \begin{array}{c}\vdots \\ \Gamma', x : A_i \mid x \cdot x \cdot \alpha : A_i \to A_i \to B \vdash \Delta\end{array}}{\langle z \parallel x \cdot x \cdot \alpha \rangle : (\Gamma', x : A_i \vdash \Delta)}\, Cut$$

Call-by-name

The call-by-name typing system, named $\overline{\lambda}\mu\tilde{\mu}\cap\cup_n$, is given in Figure 14 as an extension of Figure 5. It is dual to $\overline{\lambda}\mu\tilde{\mu}\cap\cup_v$, and imposes a *covalue restriction* on union types. This limits the introduction of union types to covalues instead of general coterms, as seen in the $\cup L$ rule (which only applies to syntactic covalues of the form E_n). Additionally, there is an $\cap\alpha$ rule (dual to the $\cup x$ rule of $\overline{\lambda}\mu\tilde{\mu}\cap\cup_v$) which is an alternative to $\cap R$ for introducing an intersection type to a free covariable of a command, which circumvents the counter-example to subject reduction in Example 4.4. Similar to the $\overline{\lambda}\mu\tilde{\mu}\cap\cup_v$, the call-by-name $\overline{\lambda}\mu\tilde{\mu}\cap\cup_n$ strikes another compromise that achieves each of soundness, completeness, and type safety.

$$\frac{\Gamma \vdash V_d : A \mid \Delta \quad \Gamma \vdash V_d : B \mid \Delta}{\Gamma \vdash V_d : A \cap B \mid \Delta} \cap R \qquad \frac{\Gamma \mid e : A_i \vdash \Delta}{\Gamma \mid e : A_1 \cap A_2 \vdash \Delta} \cap L \qquad \frac{\Gamma \vdash v : A_1 \cap A_2 \mid \Delta}{\Gamma \vdash v : A_i \mid \Delta} \cap E$$

$$\frac{c : (\Gamma \vdash \alpha : A, \Delta) \quad c : (\Gamma \vdash \alpha : B, \Delta) \quad (\mu\alpha.c \in \mathtt{Value}_d)}{c : (\Gamma \vdash \alpha : A \cap B, \Delta)} \cap \alpha$$

$$\frac{\Gamma \vdash v : A_i \mid \Delta}{\Gamma \vdash v : A_1 \cup A_2 \mid \Delta} \cup R \qquad \frac{\Gamma \mid E_d : A \vdash \Delta \quad \Gamma \mid E_d : B \vdash \Delta}{\Gamma \mid E_d : A \cup B \vdash \Delta} \cup L \qquad \frac{\Gamma \mid e : A_1 \cup A_2 \vdash \Delta}{\Gamma \mid e : A_i \vdash \Delta} \cup E$$

$$\frac{c : (\Gamma, x : A \vdash \Delta) \quad c : (\Gamma, x : B \vdash \Delta) \quad (\tilde{\mu}x.c \in \mathtt{CoValue}_d)}{c : (\Gamma, x : A \cup B \vdash \Delta)} \cup x$$

In the rules $\cap L$, $\cap E$, $\cup R$, and $\cup E$, the index i ranges over 1 or 2.

Figure 15: $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ — The disciplined intersection and union type system, with the (co)value restriction given by the discipline d.

Focused call-by-name and call-by-value type systems

Focusing is relevant for intersection and union types, since both completeness (Section 5) and soundness (Section 7) rely on focusing. We extend the call-by-name typing system of Figure 8 with the typing rule:

$$\frac{\Gamma \; ; \; E_{fn} : A \vdash \Delta \quad \Gamma \; ; \; E_{fn} : B \vdash \Delta}{\Gamma \; ; \; E_{fn} : A \cup B \vdash \Delta} \cup L$$

Notice that the premises of the $\cup L$ typing rule keep the covalue in the stoup. Dually, the call-by-value typing system of Figure 9 is extended with the typing rule:

$$\frac{\Gamma \vdash V_{fv} : A \; ; \; \Delta \quad \Gamma \vdash V_{fv} : B \; ; \; \Delta}{\Gamma \vdash V_{fv} : A \cup B \; ; \; \Delta} \cap R$$

Notice that the premises of the $\cap R$ typing rule keeps the value in the stoup.

4.3. Disciplined intersection and union types

Just like the notion of discipline unifies call-by-value and call-by-name computation, it also unifies the call-by-value and call-by-name typing restrictions for intersection and union types. The common type system that subsumes $\overline{\lambda}\mu\tilde{\mu}\cap\cup$, $\overline{\lambda}\mu\tilde{\mu}\cap\cup_v$, and $\overline{\lambda}\mu\tilde{\mu}\cap\cup_n$ is shown in Figure 15, which extends Figure 5. This parametric presentation shows that the sequent calculus analog of the value restriction is dual: a *(co)value restriction* limits certain typing rules to only apply to a value or covalue as appropriate. The (co)value restriction only appears in two places: the introduction of an intersection type on the right ($\cap R$ and $\cap\alpha$) and the introduction of a union type on the left ($\cup L$ and $\cup x$). Thankfully, this generic characterization of the value restriction in terms of values and covalues gives a single description

$$\texttt{SimpleValue} ::= x \mid \lambda x.v \qquad\qquad \texttt{SimpleCoValue} ::= \alpha \mid V_d \cdot E_d$$

$$\frac{\Gamma \vdash x : A \mid \Delta \quad \Gamma \vdash x : B \mid \Delta}{\Gamma \vdash x : A \cap B \mid \Delta} \cap Rx \qquad \frac{\Gamma \vdash \lambda x.v : A \mid \Delta \quad \Gamma \vdash \lambda x.v : B \mid \Delta}{\Gamma \vdash \lambda x.v : A \cap B \mid \Delta} \cap R\lambda$$

$$\frac{\Gamma \vdash v : A_1 \cap A_2 \mid \Delta}{\Gamma \vdash v : A_i \mid \Delta} \cap E \qquad \frac{\Gamma \mid e : A_i \vdash \Delta}{\Gamma \mid e : A_1 \cap A_2 \vdash \Delta} \cap L$$

$$\frac{\Gamma \vdash v : A_i \mid \Delta}{\Gamma \vdash v : A_1 \cup A_2 \mid \Delta} \cup R \qquad \frac{\Gamma \mid e : A_1 \cup A_2 \vdash \Delta}{\Gamma \mid e : A_i \vdash \Delta} \cup E$$

$$\frac{\Gamma \mid \alpha : A \vdash \Delta \quad \Gamma \mid \alpha : B \vdash \Delta}{\Gamma \mid \alpha : A \cup B \vdash \Delta} \cup L\alpha \qquad \frac{\Gamma \mid V_d \cdot E_d : A \vdash \Delta \quad \Gamma \mid V_d \cdot E_d : B \vdash \Delta}{\Gamma \mid V_d \cdot E_d : A \cup B \vdash \Delta} \cup L.$$

In the $\cap L$, $\cap E$, $\cup R$, and $\cup E$ rules, the index i ranges over 1 or 2.

Figure 16: $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d^-$ — The simplified intersection and union type system, further restricting $\cap R$ and $\cup L$ rules.

of the *safe* type system for certain (deterministic) disciplines d, in the sense of both type safety (in Section 6) and strong normalization (in Section 7). The restrictions on intersections and unions depend on the chosen discipline: in call-by-value $\cap R$ is restricted to v-values and $\cap\alpha$ is missing (exactly as in Figure 13), in call-by-name $\cup L$ is restricted to n-covalues and $\cup x$ is missing (exactly as in Figure 14), and there are no restrictions to $\cap R$ and $\cup L$ with non-deterministic u reduction (as in Figure 12). Furthermore, the $\overline{\lambda}\mu\tilde{\mu}\cap\cup_u$ type system extends $\overline{\lambda}\mu\tilde{\mu}\cap\cup$ from Figure 12 with the $\cap\alpha$ and $\cup x$ rules.

As we will see later in Section 7, deterministic disciplines d guarantee that all well-typed $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ are strongly normalizing, whereas non-deterministic disciplines like u require even more constraints than the (co)value restriction, which we will now consider.

4.4. Simplified intersection and union types

From the perspective of both disciplined typing (Section 4.3) and focusing (Section 3.2), the non-deterministic discipline u, which represents unrestricted *classical* reduction à la Gentzen's original cut elimination procedure, is indistinguishable from the full system of intersection and union types (Section 4.1). However, as we saw, the full system does not enjoy all the properties that we would want, like subject reduction. If we want a system of non-deterministic reduction which has these properties, we need to go further than just the (co)value restriction.

The type system can be further restricted beyond just the (co)value restriction by placing even more limitations on the tricky $\cap R$ and $\cup L$ rules, as shown in Figure 16. This simplified type system is called $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d^-$, and is a sub-system of $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ for any admissible discipline d. In $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d^-$, the $\cap R$ rule for introducing an intersection on the right only applies to simple terms of the form x or $\lambda x.v$, and the $\cup L$ for introducing a union on the left only applies to simple coterms of the form α or

$V_d \cdot E_d$. This rules out the possibility of applying $\cap R$ to a complex μ-abstraction or $\cup L$ to a complex $\tilde{\mu}$-abstraction, even if they happen to be considered (co)values. The difference is that complex (co)terms ($\mu\tilde{\mu}$-abstractions or non-covalue call stacks) have the ability to take over control of execution on their own, without regard to their partner in a command, so they might be responsible for introducing non-determinism when given the opportunity. In contract, simple (co)terms ((co)variables, λ-abstractions, and covalue call stacks) never participate in a critical pair, and so they always form deterministic commands regardless of who they are paired with.

As a result, well-typed expressions in $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d^-$ enjoy both strong normalization and subject reduction, as seen later in Sections 6 and 7, even for classical non-deterministic reduction. But to be sure, this does *not* subsume the strong normalization and subject reduction results $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$, since neither is more general than the other: $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d^-$ can admit more disciplines d, but $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ types more expressions. Therefore, there is a trade-off between flexibility of typing (for which $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ is more general) and flexibility of computation (for which $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d^-$ is more general).

4.5. Intermezzo — Subtyping

As it happens, we do not use subtyping for the purpose of establishing the syntactic properties of subject reduction (Section 6) and completeness (Section 5). However, subtyping is closely connected with intersection and union types, and the two features quite naturally arise in concert with one another. Furthermore, a semantic version of subtyping implicitly appears anyway in our study of soundness (Section 7). So while we do not formally use subtyping in the following sections, it is worth mentioning how subtyping could be integrated with intersection and union types in the sequent calculus.

The only new typing rules needed to add subtyping to the system are for subsumption. Since $\overline{\lambda}\mu\tilde{\mu}$ has two dual typed entities—terms on the right and coterms on the left—there are two matching dual subsumption rules:

$$\frac{\Gamma \vdash v : A \mid \Delta \quad A <: B}{\Gamma \vdash v : B \mid \Delta}\ SubR \qquad \frac{A <: B \quad \Gamma \mid e : B \vdash \Delta}{\Gamma \mid e : A \vdash \Delta}\ SubL$$

The premise $A <: B$ denotes that A is a subtype of B, and we will write the reverse $B :> A$ as an alternative notation for $A <: B$ and $A :=: B$ to mean that both $A <: B$ and $A :> B$ are derivable. The intuitive idea of a subtyping relation $A <: B$ is that every term of type A can be used in a context (*i.e.,* coterm) expecting a type B, or vice versa, every coterm of type B can be accept every term of type A. These two subsumption rules $SubR$ and $SubL$ apply this intuition by composing an established subtyping fact $A <: B$ in the appropriate direction with the type of a term or coterm.

Subsumption is only as useful as the subtyping relation provides, so the remainder of extending the system with subtyping lies in deciding which subtyping rules to allow. Of course, standard reflexivity and transitivity of subtyping are a given:

$$\frac{}{A <: A}\ Refl \qquad \frac{A <: B \quad B <: C}{A <: C}\ Trans$$

The more interesting subtyping rules involve the connectives. For intersection and union types, the

following subtyping rules can be derived from the standard lattice properties:

$$\frac{A_i <: C \quad (i = 1, 2)}{A_1 \cap A_2 <: C} \cap Low \qquad \frac{C <: A \quad C <: B}{C <: A \cap B} \cap Up$$

$$\frac{C <: A_i \quad (i = 1, 2)}{C <: A_1 \cup A_2} \cup Up \qquad \frac{A <: C \quad B <: C}{A \cup B <: C} \cup Low$$

These rules imply that intersections are the *greatest* ($\cap Up$) *lower bound* ($\cap Low$) and that unions are the *least* ($\cup Low$) *upper bound* ($\cup Up$) of two types.[1] Note that the generality of the $\cap Up$ and $\cup Low$ rules (captured by the generic lower and upper bound C) implies that intersection and union types are idempotent (*i.e.,* that $A \cap A :=: A :=: A \cup A$); to achieve a system of non-idempotent intersection and union types these two rules would have to be weakened or omitted. Also notice that with these subtyping rules, the $\cap E$, $\cap L$, $\cup E$, and $\cup R$ rules become special cases of subsumption. For example, the derivation of the $\cup R$ and $\cup E$ rules is

$$\frac{\Gamma \vdash v : A_i \mid \Delta \quad \dfrac{\dfrac{}{A_i <: A_i}\, Refl}{A_i <: A_1 \cup A_2}\, \cup Up}{\Gamma \vdash v : A_1 \cup A_2 \mid \Delta}\, SubR \qquad \frac{\dfrac{\dfrac{}{A_i <: A_i}\, Refl}{A_i <: A_1 \cup A_2}\, \cup Up \quad \Gamma \vdash e : A_1 \cup A_2 \vdash \Delta}{\Gamma \mid e : A_i \vdash \Delta}\, SubL$$

and the other ones are dual.

The only other connective that we have considered is the function arrow, $A \to B$. Functions can be integrated into the subtyping system via the standard rule

$$\frac{A :> A' \quad B <: B'}{A \to B <: A' \to B'} \to Sub$$

which states that subtyping distributes over a function arrow: covariantly in the result and contravariantly in the input. Combining the subtyping rule for functions along with the ones for intersections and unions allows for the derivation of some more complex, interesting subtyping relations. For example, the subtyping relation $(A_1 \cup A_2) \to (B_1 \cap B_2) <: (A_1 \to B_1) \cap (A_2 \to B_2)$—stating that an intersection of two different function types is a supertype of functions from the union of possible input to an intersection of possible output—can be derived from the above rules as

$$\frac{\begin{array}{cc}\vdots\ \mathcal{D}_1 & \vdots\ \mathcal{D}_2 \\ (A_1 \cup A_2) \to (B_1 \cap B_2) <: A_1 \to B_1 & (A_1 \cup A_2) \to (B_1 \cap B_2) <: A_2 \to B_2\end{array}}{(A_1 \cup A_2) \to (B_1 \cap B_2) <: (A_1 \to B_1) \cap (A_2 \to B_2)} \cap Up$$

where the similar sub-derivations $\mathcal{D}_1$ and $\mathcal{D}_2$ are (for $i = 1, 2$, respectively):

$$\frac{\dfrac{\dfrac{}{A_i :> A_i}\, Refl}{A_1 \cup A_2 :> A_i}\, \cup Up \quad \dfrac{\dfrac{}{B_i <: B_i}\, Refl}{B_1 \cap B_2 <: B_i}\, \cap Low}{(A_1 \cup A_2) \to (B_1 \cap B_2) <: A_i \to B_i}\, \to Sub$$

[1] Note that the $\cap Low$ and $\cup Up$ subtyping rules mimic the ordinary left-hand conjunction introduction and right-hand disjunction introduction rules of the sequent calculus. As a consequence, these subtyping rules enjoy *transitivity elimination* that is analogous to cut elimination in the sequent calculus: every derivable subtyping relation can be inferred without use of the *Trans* rule. Just taking $A_1 \cap A_2 <: A_i$ and $A_i <: A_1 \cup A_2$ as axioms would be sound, but break transitivity elimination.

Dually, the relation $(A_1 \to B_1) \cup (A_2 \to B_2) <: (A_1 \cap A_2) \to (B_1 \cup B_2)$—stating that a union of two different function types is a subtype of functions from the intersection of possible input to the union of possible output—is also derivable from the above inference rules.

The above two subtyping relations don't make sense in general when reversed, but some special cases do. Namely when the input types (A_1 and A_2) or output types (B_1 and B_2) are the same. In these special cases, the reverse of the above derivations may be sensibly taken as additional axioms to the subtyping system:

- $(A_1 \to B) \cap (A_2 \to B) <: (A_1 \cup A_2) \to B$
- $(A \to B_1) \cap (A \to B_2) <: A \to (B_1 \cap B_2)$
- $(A_1 \cap A_2) \to B <: (A_1 \to B) \cup (A_2 \to B)$
- $A \to (B_1 \cup B_2) <: (A \to B_1) \cup (A \to B_2)$

Note that since the reverse directions of each of these are derivable from the above inference rules, assuming any of these subtyping relations as an additional subtyping axiom is the same as assuming it as a symmetric type *equality* (up to subtyping). For example, the particular subtyping relation $(A_1 \to B) \cap (A_2 \to B) :> (A_1 \cap A_2) \to B$ is already derivable from the above standard rules for subtyping of function, intersection, and union types, so in the context of the other inference rules, the first axiom above is equivalent to the assumption that $(A_1 \to B) \cap (A_2 \to B) :=: (A_1 \cap A_2) \to B$.

5. Strongly Normalizing Expressions are Typable — Completeness

Like in the simply typed λ-calculus, not all normal forms in the simply typed $\overline{\lambda}\mu\tilde{\mu}$ are typable, *e.g.,* the term $\lambda x.\mu\alpha.\langle x \parallel x \cdot \alpha\rangle$. Even though polymorphism can type many more terms, as in system F, some strongly normalizing terms are still not typable [26]. For example, the "monster" term, given in [27], is not typable in F_ω but is typable with intersection types. This shortcoming is overcome in the disciplined $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ type system wherein all strongly normalizing terms are typable.

The reason that completeness is even possible is due to the following two unusual properties about intersection and union types that do not normally hold in a simply-typed calculus:

- Generalized weakening (Proposition 5.2): every typing derivation can be weakened by a type assignment for a (co)variable *whether or not* it already appears in the environment. If the (co)variable does not already appear, then the assignment can be added as an unused free (co)variable as normal. But if the (co)variable is already being used, then the assignment can be added as an unused intersection type on a variable (*i.e.,* $x : A$ to $x : A \cap B$) or an unused union type on a covariable (*i.e.,* $\alpha : A$ to $\alpha : A \cup B$).
- Anti-substitution (Proposition 5.6): if the result of substituting a (co)value into an expression is typable, then there is some type for that (co)value such that the original expression is typable.

These two properties allow us to prove the following two main lemmas:

- Every final normal form is typable (Proposition 5.4).

- Non-erasable subject expansion (Proposition 5.7): if an expression reduces to a typed expression by a step which does not erase any sub-expression, then the starting expression has the same type as the result of the reduction.

From here, the proof that every strongly normalizing and focused expression is typable follows by expanding out from a chosen reduction path to a normal form. Note that, since the reduction rules will only ever attempt to substitute a value for a variable or a covalue for a covariable (as defined by the chosen discipline), there is only ever a need to perform anti-substitution on values and covalues. As a result, the (co)value restriction in $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ that only allows us to introduce intersection types on values and introduce union types on covalues is never an obstacle for establishing the completeness of typability for strongly normalizing expressions. In contrast, the smaller type system $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d^-$ only allows intersection and union types to be introduced on a subset of values and covalues, which makes it incomplete.

Definition 5.1. The generalized extension of environments, which combines together two environments as usual (where they are different) or with intersection and union types (where they overlap), is defined as follows:

$$\begin{aligned}\Gamma \cap \Gamma' = \{x : A \cap B \mid x{:}A \in \Gamma \text{ and } x{:}B \in \Gamma'\} \\ \cup \{x : A \mid x{:}A \in \Gamma \text{ and } x \notin \mathit{Dom}(\Gamma')\} \\ \cup \{x : B \mid x \notin \mathit{Dom}(\Gamma) \text{ and } x{:}B \in \Gamma'\}\end{aligned} \qquad \begin{aligned}\Delta \cup \Delta' = \{\alpha : A \cup B \mid \alpha{:}A \in \Delta \text{ and } \alpha{:}B \in \Delta'\} \\ \cup \{\alpha : A \mid \alpha{:}A \in \Delta \text{ and } \alpha \notin \mathit{Dom}(\Delta')\} \\ \cup \{\alpha : B \mid \alpha \notin \mathit{Dom}(\Delta) \text{ and } \alpha{:}B \in \Delta'\}\end{aligned}$$

Proposition 5.2. (Generalized Weakening)
For every discipline d, in both the $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ and $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d^-$ type systems:

(i) If $c : (\Gamma \vdash \Delta)$ is derivable then so are $c' : (\Gamma \cap \Gamma' \vdash \Delta \cup \Delta')$ and $c' : (\Gamma' \cap \Gamma \vdash \Delta' \cup \Delta)$ for some $c' =_\alpha c$.

(ii) If $\Gamma \vdash v : A \mid \Delta$ is derivable then so are $\Gamma \cap \Gamma' \vdash v' : A \mid \Delta \cup \Delta'$ and $\Gamma' \cap \Gamma \vdash v' : A \mid \Delta' \cup \Delta$ for some $v' =_\alpha v$.

(iii) If $\Gamma \mid e : A \vdash \Delta$ is derivable then so are $\Gamma \cap \Gamma' \mid e' : A \vdash \Delta \cup \Delta'$ and $\Gamma' \cap \Gamma \mid e' : A \vdash \Delta' \cup \Delta$ for some $e' =_\alpha e$.

Proof:
By mutual induction on the given typing derivation.

- In the base cases where the derivation is just

$$\frac{}{\Gamma, x : A \vdash x : A \mid \Delta}\ \mathit{VarR} \qquad\qquad \frac{}{\Gamma \mid \alpha : A \vdash \alpha : A, \Delta}\ \mathit{VarL}$$

  then there are two possibilities (where, weakening the VarL rule follows dually):

  - If $x \notin \mathit{Dom}(\Gamma')$ then $(\Gamma, x : A)\cap\Gamma' = (\Gamma\cap\Gamma'), x : A$ and $\Gamma'\cap(\Gamma, x : A) = (\Gamma'\cap\Gamma), x : A$, so the same inference rule applies in the weakened environment.

– Otherwise, we have the two weakenings $(\Gamma, x : A) \cap (\Gamma', x : B) = (\Gamma \cap \Gamma'), x : A \cap B$ and $(\Gamma', x : B) \cap (\Gamma, x : A) = (\Gamma' \cap \Gamma), x : B \cap A$ so

$$\dfrac{\dfrac{}{(\Gamma \cap \Gamma'), x : A \cap B \vdash x : A \cap B \mid \Delta \cup \Delta'}\ VarR}{(\Gamma \cap \Gamma'), x : A \cap B \vdash x : A \mid \Delta \cup \Delta'}\ \cap E$$

$$\dfrac{\dfrac{}{(\Gamma' \cap \Gamma), x : B \cap A \vdash x : B \cap A \mid \Delta' \cup \Delta}\ VarR}{(\Gamma' \cap \Gamma), x : B \cap A \vdash x : A \mid \Delta' \cup \Delta}\ \cap E$$

- The cases for binding rules $ActR$, $ActL$, and $\to R$ require renaming. For example, suppose we have a derivation ending in

$$\dfrac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta}\ ActL$$

If it happens that $x \in Dom(\Gamma')$, then rename the bound x to some other $y \notin Dom(\Gamma')$. From this point, both weakenings follow directly from the inductive hypothesis. The case for $ActR$ is dual to $ActL$, and $\to R$ is analogous to $ActL$.

- The cases for the remaining rules Cut, $\to L$, *etc.,* follow directly from the inductive hypothesis.

□

Definition 5.3. A *normal form* for the discipline d is any expression (command, term, or coterm) such that no μ_d, $\tilde{\mu}_d$, β_d, or ς_d reduction applies *anywhere* in the expression, and moreover a *final normal form* is one in which every sub-command is final. Equivalently, the final normal forms of any discipline d are generated by the following syntax:

$$\begin{array}{lll} c_{nfd} ::= \langle v_{pnfd} \parallel \alpha \rangle \mid \langle x \parallel e_{pnfd} \rangle & & \\ v_{nfd} ::= v_{pnfd} \mid \mu\alpha.c_{nfd} & & v_{pnfd} ::= x \mid \lambda x.v_{nfd} \\ e_{nfd} ::= e_{pnfd} \mid \tilde{\mu}x.c_{nfd} & & e_{pnfd} ::= \alpha \mid V_{nfd} \cdot E_{nfd} \end{array}$$

Where V_{nfd} denotes the intersection of the set of values of d and the normal terms above, and E_{nfd} denotes the intersection of the set of covalues of d and the normal coterms above. Note that normal terms v_{nfd} are further refined as *passive normal terms* v_{pnfd} which cannot be a μ-abstraction, which prevents $\langle v_{pnfd} \parallel \alpha \rangle$ from being a μ_d-redex. Dually, normal coterms e_{nfd} are further refined as *passive normal coterms* which cannot be a $\tilde{\mu}$-abstraction, preventing $\langle x \parallel e_{pnfd} \rangle$ from being a $\tilde{\mu}_d$-redex.

Proposition 5.4. For every discipline d, every final d-normal form is typable in both $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ and $\overline{\lambda}\mu\tilde{\mu}\cap\cup_{\bar{d}}$.

Proof:
We show the following typing derivations that are valid in both $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ and $\overline{\lambda}\mu\tilde{\mu}\cap\cup_{\bar{d}}$ by mutual induction on the syntax of final normal forms in Definition 5.3:

(i) for every v_{nfd}, there are environments Γ and Δ and a type A such that $\Gamma \vdash v_{nfd} : A \mid \Delta$,

(ii) for every e_{nfd}, there are environments Γ and Δ and a type A, such that $\Gamma \mid e_{nfd} : A \vdash \Delta$, and

(iii) for every c_{nfd}, there are environments Γ and Δ, such that $c_{nfd} : (\Gamma \vdash \Delta)$.

- (Co)variables are typable as $x : A \vdash x : A \mid$ and $\mid \alpha : A \vdash \alpha : A$ by the $VarR$ and $VarL$ axioms (respectively).

- For an abstraction $\mu\alpha.c_{nfd}$, we know by the inductive hypothesis that there is a derivation $\mathcal{D}$ of $c_{nfd} : (\Gamma \vdash \Delta)$ for some Γ and Δ. We may assume (by weakening via Proposition 5.2 if necessary) that $\Delta = \alpha : A, \Delta'$ for some type A. Therefore,

$$\frac{\begin{array}{c}\vdots\ \mathcal{D} \\ c_{nfd} : (\Gamma \vdash \alpha : A, \Delta')\end{array}}{\Gamma \vdash \mu\alpha.c_{nfd} : A \mid \Delta'}\ ActR$$

  The typability of normal $\tilde{\mu}$-abstractions is dual to the above.

- For an abstraction $\lambda x.v_{nfd}$, we know by the inductive hypothesis that there is a derivation $\mathcal{D}$ of $\Gamma \vdash v_{nfd} : B \mid \Delta$ for some Γ, Δ, and B. We may assume (by weakening via Proposition 5.2 if necessary) that $\Gamma = \Gamma', x : A$ for some type A. Therefore,

$$\frac{\begin{array}{c}\vdots\ \mathcal{D} \\ \Gamma', x : A \vdash v_{nfd} : B \mid \Delta\end{array}}{\Gamma' \vdash \lambda x.v_{nfd} : A \to B \mid \Delta}\ {\to}R$$

- For a call stack $V_{nfd} \cdot E_{nfd}$, we know by the inductive hypothesis that there is a derivation $\mathcal{D}$ of $\Gamma \vdash V_{nfd} : A \mid \Delta$ for some Γ, Δ, A and a derivation $\mathcal{E}$ of $\Gamma' \mid E_{nfd} : B \vdash \Delta'$ for some Γ', Δ', B. Therefore, we have by generalized weakening of Proposition 5.2

$$\frac{\begin{array}{c}\vdots\ \mathcal{D}' \\ \Gamma \cap \Gamma' \vdash V_{nfd} : A \mid \Delta \cup \Delta'\end{array} \quad \begin{array}{c}\vdots\ \mathcal{E}' \\ \Gamma \cap \Gamma' \mid E_{nfd} : B \vdash \Delta \cup \Delta'\end{array}}{\Gamma \cap \Gamma' \mid V_{nfd} \cdot E_{nfd} : A \to B \vdash \Delta \cup \Delta'}\ {\to}L$$

- For a passive-term command $\langle v_{pnfd} \parallel \alpha \rangle$, we know by the inductive hypothesis that there is a derivation $\mathcal{D}$ of $\Gamma \vdash v_{pnf} : A \mid \Delta$ for some Γ, Δ, and A. We therefore have one of the two following cases via Proposition 5.2, depending on whether or not α already appears in Δ. If α is not in the domain of Δ then $(\alpha : A) \cup \Delta = \alpha : A, \Delta$, so we can weaken $\mathcal{D}$ to $\mathcal{D}'$ such that

$$\frac{\begin{array}{c}\vdots\ \mathcal{D}' \\ \Gamma \vdash v_{pnfd} : A \mid \alpha : A, \Delta\end{array} \quad \dfrac{}{\Gamma \mid \alpha : A \vdash \alpha : A, \Delta}\ VarL}{\langle v_{pnfd} \parallel \alpha \rangle : (\Gamma \vdash \alpha : A, \Delta)}\ Cut$$

Otherwise, if $\Delta = \alpha : B, \Delta'$ (for some given type B), then $(\alpha : A) \cup \Delta = \alpha : A \cup B, \Delta'$, so we can weaken $\mathcal{D}$ to $\mathcal{D}'$ such that

$$\dfrac{\begin{array}{c}\vdots\ \mathcal{D}' \\ \Gamma \vdash v_{pnfd} : A \mid \alpha : A \cup B, \Delta'\end{array} \qquad \dfrac{\dfrac{}{\Gamma \mid \alpha : A \cup B \vdash \alpha : A \cup B, \Delta'}\, VarL}{\Gamma \mid \alpha : A \vdash \alpha : A \cup B, \Delta'}\, \cup E}{\langle v_{pnfd} \parallel \alpha \rangle : (\Gamma \vdash \alpha : A \cup B, \Delta')}\, Cut$$

The typability of a passive-coterm command $\langle x \parallel e_{pnfd} \rangle$ is dual to the above using intersection types instead of union types.

□

Definition 5.5. (Non-erasing)
In general, a reduction is *non-erasing* when every sub-expression on the left-hand side of the reduction appears as a sub-expression on the right-hand side of reduction. In particular, the following applications of reduction rules are non-erasing:

- (μ_d) $\langle \mu\alpha.c \parallel E_d \rangle \mapsto c\,[E_d/\alpha]$ is non-erasing when α is a free (co)variable of c.
- $(\tilde{\mu}_d)$ $\langle V_d \parallel \tilde{\mu}x.c \rangle \mapsto c\,[V_d/x]$ is non-erasing when x is a free variable of c.
- (β_d) $\langle \lambda x.v \parallel V_d \cdot E_d \rangle \mapsto \langle v\,[V_d/x] \parallel E_d \rangle$ is non-erasing when x is a free variable of v.

Note that non-erasing reduction is a congruence relation (*e.g.,* if $v \to v'$ is non-erasing then so is $C[v] \to C[v']$ for any context C).

Proposition 5.6. (Anti-substitution)
For every admissible discipline d, in the $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ type system:

(i) If $c\,[V_d/x] : (\Gamma \vdash \Delta)$ and $x \in FV(c)$ then there is a type A such that $c : (\Gamma, x : A \vdash \Delta)$ and $\Gamma \vdash V_d : A \mid \Delta$. Dually, if $c\,[E_d/\alpha] : (\Gamma \vdash \Delta)$ and $\alpha \in FV(c)$ then there is a type A such that $c : (\Gamma \vdash \alpha : A, \Delta)$ and $\Gamma \mid E_d : A \vdash \Delta$.

(ii) If $\Gamma \vdash v\,[V_d/x] : B \mid \Delta$ and $x \in FV(v)$ then there is a type A such that $\Gamma, x : A \vdash v : B \mid \Delta$ and $\Gamma \vdash V_d : A \mid \Delta$. Dually, if $\Gamma \vdash v\,[E_d/\alpha] : B \mid \Delta$ and $\alpha \in FV(v)$ then there is a type A such that $\Gamma \vdash v : B \mid \alpha : A, \Delta$ and $\Gamma \mid E_d : A \vdash \Delta$.

(iii) If $\Gamma \mid e\,[V_d/x] : B \vdash \Delta$ and $x \in FV(e)$ then there is a type A such that $\Gamma, x : A \mid e : B \vdash \Delta$ and $\Gamma \vdash V_d : A \mid \Delta$. Dually, if $\Gamma \mid e\,[E_d/\alpha] : B \vdash \Delta$ and $\alpha \in FV(e)$ then there is a type A such that $\Gamma \mid e : B \vdash \alpha : A, \Delta$ and $\Gamma \mid E_d : A \vdash \Delta$.

Proof:
By induction on the given typing derivation. Substituting a value for a variable has the cases:

- The case for a typed variable $\Gamma \vdash x\,[V_d/x] : A \mid \Delta$ is immediate.
- The cases for a typed abstraction of the form $\lambda y.v$, $\mu\beta.c$, and $\tilde{\mu}y.c$ follow directly from the inductive hypothesis, the definition of substitution, and the stability of d (to ensure that if $\cap R$ and $\cup L$ are used, they are still applicable before substitution has been done).

- The cases for a command typed by $\cap\alpha$ and $\cup x$ rule follow from the inductive hypothesis.
- In the case for a command typed by a Cut rule

$$\frac{\Gamma \vdash v\,[V_d/x] : C \mid \Delta \quad \Gamma \mid e\,[V_d/x] : C \vdash \Delta}{\langle v \parallel e \rangle\,[V_d/x] : (\Gamma \vdash \Delta)}\ Cut$$

  there are the three following sub-cases depending on the reason why $x \in FV(\langle v \parallel e \rangle)$:

  - If $x \in FV(v)$ and $x \notin FV(e)$, then we know that $e\,[V_d/x] = e$ and by the inductive hypothesis on v, there is a type A such that $\Gamma, x : A \vdash v : C \mid \Delta$ and $\Gamma \vdash V_d : A \mid \Delta$ are both derivable. Therefore, by weakening (via Proposition 5.2) $\Gamma \mid e : C \vdash \Delta$, we have

$$\frac{\Gamma, x : A \vdash v : C \mid \Delta \quad \Gamma, x : A \mid e : C \vdash \Delta}{\langle v \parallel e \rangle : (\Gamma, x : A \vdash \Delta)}\ Cut$$

  - If $x \notin FV(v)$ and $x \in FV(e)$, then the result follows dually to the above case.
  - If $x \in FV(v)$ and $x \in FV(e)$, then by the inductive hypothesis on both v and e there are types A and B such that
    - $\Gamma, x : A \vdash v : C \mid \Delta$,
    - $\Gamma \vdash V_d : A \mid \Delta$,
    - $\Gamma, x : B \mid e : C \vdash \Delta$, and
    - $\Gamma \vdash V_d : B \mid \Delta$.

    Since $(\Gamma, x : A) \cap (x : B) = \Gamma, x : A \cap B$ and $(x : A) \cap (\Gamma, x : B) = \Gamma, x : A \cap B$, we also have derivations of $\Gamma, x : A \cap B \vdash v : C \mid \Delta$ and $\Gamma, x : A \cap B \mid e : C \vdash \Delta$ by generalized weakening (Proposition 5.2). Therefore, we have the following two derivations:

$$\frac{\Gamma, x : A \cap B \vdash v : C \mid \Delta \quad \Gamma, x : A \cap B \mid e : C \vdash \Delta}{\langle v \parallel e \rangle : (\Gamma, x : A \cap B \vdash \Delta)}\ Cut$$

$$\frac{\Gamma \vdash V_d : A \vdash \Delta \quad \Gamma \vdash V_d : B \vdash \Delta}{\Gamma \vdash V_d : A \cap B \vdash \Delta}\ \cap R$$

- The case for a call stack $\Gamma \mid v \cdot e\,[V_d/x] : A \to B \vdash \Delta$ typed by an $\to L$ rule is similar to the above case for a command typed by a Cut rule.
- The case for a call stack typed by a $\cup L$ rule follows from the inductive hypothesis and the stability of d to ensure that $\cup L$ still applies.

The cases for the substitution of a covalue for a covariable are exactly dual to the above, making use of union types rather than intersection types when the covalue is duplicated during substitution. □

Proposition 5.7. (Non-erasing Subject Expansion)
For every admissible discipline d, in the $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ type system:

(i) If $c \to_{\mu_d\tilde{\mu}_d\beta_d} c'$ by a non-erasing reduction and $c' : (\Gamma \vdash \Delta)$ then $c : (\Gamma \vdash \Delta)$.

(ii) If $v \to_{\mu_d\tilde{\mu}_d\beta_d} v'$ by a non-erasing reduction and $\Gamma \vdash v' : A \mid \Delta$ then $\Gamma \vdash v : A \mid \Delta$.

(iii) If $e \to_{\mu_d\tilde{\mu}_d\beta_d} e'$ by a non-erasing reduction and $\Gamma \mid e' : A \vdash \Delta$ then $\Gamma \mid e : A \vdash \Delta$.

Proof:
We begin by showing non-erasing subject expansion for any operational step $c \mapsto_{\mu_d\tilde{\mu}_d\beta_d} c'$. Consider the non-erasing μ_d step $\langle \mu\alpha.c \parallel E_d \rangle \mapsto_{\mu_d} c\,[E_d/\alpha]$ where $\alpha \in FV(c)$ and suppose that we have a derivation of $c\,[E_d/\alpha] : (\Gamma \vdash \Delta)$. By anti-substitution (Proposition 5.6) we have some type A and derivations $\mathcal{D}$ and $\mathcal{E}$ such that:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\,\mathcal{D}\\ c : (\Gamma \vdash \alpha : A, \Delta)\end{array}}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}\,ActR \qquad \begin{array}{c}\vdots\,\mathcal{E}\\ \Gamma \mid E_d : A \vdash \Delta\end{array}}{\langle \mu\alpha.c \parallel E_d \rangle : (\Gamma \vdash \Delta)}\,Cut$$

The cases for $\tilde{\mu}_d$ and β_d steps are similar to the above case for μ_d.

Finally, the fact that non-erasing subject expansion holds for *any* reduction, follows by induction on the given typing derivation for the reduct. □

Proposition 5.8. (Completeness)
For d ranging over v, n, and u, every strongly d-normalizing and d-focused command, term, and coterm is typable in the $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ type system.

Proof:
By mutual induction on the length of the longest reduction sequence beginning with the given command, term, and coterm along with the following variant of Proposition 5.7:

(i) If $c \to_{\mu_d\tilde{\mu}_d\beta_d} c'$, c is strongly normalizing, and $c' : (\Gamma \vdash \Delta)$, then $c : (\Gamma \vdash \Delta)$.

(ii) If $v \to_{\mu_d\tilde{\mu}_d\beta_d} v'$ v is strongly normalizing, and $\Gamma \vdash v' : A \mid \Delta$, then $\Gamma \vdash v : A \mid \Delta$.

(iii) If $e \to_{\mu_d\tilde{\mu}_d\beta_d} e'$, e is strongly normalizing, and $\Gamma \mid e' : A \vdash \Delta$, then $\Gamma \mid e : A \vdash \Delta$.

Let the length of the longest reduction sequence starting from c, v or e be written as $|c|$, $|v|$, and $|e|$, respectively.

First, note that the d-focused sub-syntax is closed under reduction for every d, so only $\mu_d\tilde{\mu}_d\beta_d$ reductions are possible. Furthermore, when d is one of the admissible v, n, or u, every d-normal form is a final d-normal form. It follows that every strongly normalizing and focused c (or analogously v or e) is typable because either

- c is a final d-normal which is typable by Proposition 5.4, or
- $c \to_{\mu_d\tilde{\mu}_d\beta_d} c'$ for some $|c'| < |c|$, and by the inductive hypothesis we know that c' is typable as some $c' : (\Gamma \vdash \Delta)$ so that $c : (\Gamma \vdash \Delta)$ is also derivable (by the expansion property (i)) above.

Second, note that the expansion properties above are subsumed by Proposition 5.7 for any non-erasing reductions, so it only remains to show expansion for erasing reductions. Consider the erasing reduction $\langle \mu\alpha.c \parallel E_d \rangle \mapsto_{\mu_d} c$ where $\alpha \notin FV(c)$ and $\mathcal{D}$ is a derivation of $c : (\Gamma \vdash \Delta)$. We know that $|E_d| < |\langle \mu\alpha.c \parallel E_d \rangle|$, and so by the inductive hypothesis there is a derivation $\mathcal{E}$ of $\Gamma' \mid E_d : A \vdash \Delta'$ for some Γ', Δ', A. By applying generalized weakening on these $\mathcal{D}$ and $\mathcal{E}$, the following is derivable

$$\dfrac{\dfrac{\begin{array}{c}\vdots\ \mathcal{D}'\\ c : (\Gamma \cap (\Gamma', \alpha : A) \vdash \Delta \cup \Delta')\end{array}}{\Gamma \cap \Gamma' \vdash \mu\alpha.c : A \mid \Delta \cup \Delta'}\ ActR \qquad \begin{array}{c}\vdots\ \mathcal{E}'\\ \Gamma \cap \Gamma' \mid E_d : A \vdash \Delta \cup \Delta'\end{array}}{\langle \mu\alpha.c \parallel E_d \rangle : (\Gamma \cap \Gamma' \vdash \Delta \cup \Delta')}\ Cut$$

The cases for the operational $\tilde{\mu}_d$ and β_d steps is similar to μ_d above. Finally, the generalization to general reductions follows by induction on the typing derivation given for the reduct and the fact that the sets of values and covalues is closed under $\mu_d\tilde{\mu}_d\beta_d$ expansion (for the cases of $\cap R$ and $\cup L$ inference rules). □

6. Subject Reduction and Type Safety

Type safety—the theorem that "well-typed programs cannot 'go wrong'" [28]—is often broken down into two simpler syntactic properties about the operational semantics [29]:

- *Progress* says that every well-typed expression is either *finished*, or it can take a step, and
- *Preservation* says that if a well-typed expression takes a step, then its reduct is well-typed (at the same type and environment as before).

Furthermore, the property of *subject reduction* goes even further than preservation as stated above and ensures that types are preserved after *any* reduction, not just the steps of the operational semantics. When considering intersection and union types, the progress half of type safety is relatively straightforward, following the usual procedure as a simply-typed language.

Proposition 6.1. (Progress)
For any d ranging over v, n, and u, if $c : (\Gamma \vdash \Delta)$ is derivable then either c is a final command (*i.e.,* a c_{final} as defined in Figure 11) or there is a c' such that $c \mapsto_{\mu_d\tilde{\mu}_d\beta_d\varsigma_d} c'$.

However, the preservation half of type safety is much more difficult for intersection and union types. The first main problem was seen in Section 2, where the $\cup E$ elimination rule for union types in the λ-calculus performs a substitution in its conclusion. This kind of rule, which effectively unifies on the result of a substitution, is very brittle under reduction. For example, the term $x\ M\ M$ matches the substitution $(x\ y\ y)\,[M/y]$. However, if $M \to M'$ then $x\ M\ M \to x\ M\ M'$, which doesn't match either $(x\ y\ y)\,[M/y]$ or $(x\ y\ y)\,[M'/y]$, thereby breaking the unification. The sequent calculus presentation of union types avoids this problem, since the rules for union types are perfectly dual to the rules for intersection types. Instead of eliminating union types with substitution, the sequent calculus

can eliminate union types with rules about coterms. As a result, the problem above simply does not arise in the symmetric language of the sequent calculus.

The only remaining potential issue is that unions can be introduced on $\tilde{\mu}$-abstractions which take an input (and dually intersections can be introduced on μ-abstraction which give an output) which is syntactically distinct from a command with a free variable or covariable representing some other side input or output. This distinction is a real issue for subject reduction because the $\tilde{\mu}$ and μ reduction rules can eliminate the cut in which a union or intersection was introduced. In the special case of just substituting one (co)variable for another

$$\langle x \parallel \tilde{\mu}y.c \rangle \mapsto_{\tilde{\mu}} c\,[x/y] \qquad\qquad \langle \mu\beta.c \parallel \alpha \rangle \mapsto_{\mu} c\,[\alpha/\beta]$$

then we have to be sure that the result of reduction is still typable even if a union was introduced on the $\tilde{\mu}$-abstraction (which is possible in call-by-value) or dually an intersection was introduced on the μ-abstraction (which is possible in call-by-name).

Recall Example 4.3, which begins with the following typing derivation under the initial environments are $\Gamma = x : A_1 \cup A_2, z : (A_1 \to A_1 \to B) \cap (A_2 \to A_2 \to B_2)$ and $\Delta = \alpha : B$,

$$\dfrac{\dfrac{}{\Gamma \vdash x : A_1 \cup A_2 \mid \Delta}\ \mathit{VarR} \qquad \dfrac{\dfrac{\dfrac{\vdots\ \mathcal{D}_1}{\langle z \parallel y \cdot y \cdot \alpha \rangle : (\Gamma, y : A_1 \vdash \Delta)}}{\Gamma \mid \tilde{\mu}y.\langle z \parallel y \cdot y \cdot \alpha \rangle : A_1 \vdash \Delta}\ \mathit{ActL} \quad \dfrac{\dfrac{\vdots\ \mathcal{D}_2}{\langle z \parallel y \cdot y \cdot \alpha \rangle : (\Gamma, y : A_2 \vdash \Delta)}}{\Gamma \mid \tilde{\mu}y.\langle z \parallel y \cdot y \cdot \alpha \rangle : A_2 \vdash \Delta}\ \mathit{ActL}}{\Gamma \mid \tilde{\mu}y.\langle z \parallel y \cdot y \cdot \alpha \rangle : A_1 \cup A_2 \vdash \Delta}\ \cup L}{\langle x \parallel \tilde{\mu}y.\langle z \parallel y \cdot y \cdot \alpha \rangle \rangle : (\Gamma \vdash \Delta)}\ \mathit{Cut}$$

This derivation is only possible because of the convenient use of $\cup L$ on the introduction of y to split the remaining proof into two depending on whether y is instantiated with a value of type A_1 or a value of type A_2, wherein both applications of z are well-typed (since both of its arguments must be of the *same* type). But after one step

$$\langle x \parallel \tilde{\mu}y.\langle z \parallel y \cdot y \cdot \alpha \rangle \rangle \mapsto_{\tilde{\mu}_v} \langle z \parallel x \cdot x \cdot \alpha \rangle$$

a typing derivation of $\langle z \parallel x \cdot x \cdot \alpha \rangle : (\Gamma \vdash \Delta)$ is not possible using the rules in Figure 15 alone, because there is no covalue being used to consume x, so that there is no place to insert a similar application $\cup L$. This is not just a problem with union types; the same issue can occur when a μ-abstraction duplicates its bound covariable as was seen in Example 4.4. The fact that the μ and $\tilde{\mu}$ reductions can rename one (co)variable for another is the reason that we need the additional rules for introducing intersection and union types on free (co)variables to establish subject reduction.

The $\cap\alpha$ and $\cup x$ rules are a necessary addition to prevent the basic renaming counterexamples to subject reduction like the one above. However, they have the unfortunate consequence of making the standard substitution lemma (that a (co)value can be substituted for a (co)variable of the same type) which is usually straightforward into a surprisingly difficult process. If we are encountering a substitution $c\,[V/x]$ and we know that the type derivation of c has the form

$$\dfrac{\vdots\ \mathcal{D}_1 \quad \vdots\ \mathcal{D}_2}{c : (\Gamma, x : A_1 \vdash \Delta) \quad c : (\Gamma, x : A_2 \vdash \Delta)}\ \cup x$$

then it's not necessarily obvious which of the sub-derivations $\mathcal{D}_1$ or $\mathcal{D}_2$ to choose in order to proceed. Fortunately, call-by-value and call-by-name are particularly well-behaved disciplines, for which it is still possible to show that substitution preserves types. That's because v-values are so simple that, given any $V_v : A_1 \cup A_2$ which might be *either* an A_1 or a A_2, we can check the typing derivation to see which one it *actually* is. This is not possible if we have an abstraction $\mu\alpha.c : A_1 \cup A_2$, since the result might need to be computed before finding out which type it really belongs to. Dually, given any $E_n : A_1 \cap A_2$ which might want *either* an A_1 or a A_2 as input, we can check which one is actually used. Due to these simplifications, it's possible to determine the correct typing derivation from the result of substitution even with rules that split the derivation on a free (co)variable.

Proposition 6.2. (Substitution)
For every admissible discipline d, in the $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d^-$ type system, given any $\Gamma \vdash V_d : A \mid \Delta$,

(i) if $c : (\Gamma, x : A \vdash \Delta)$ is derivable then so is $c\,[V_d/x] : (\Gamma \vdash \Delta)$,

(ii) if $\Gamma, x : A \vdash v : B \mid \Delta$ is derivable then so is $\Gamma \vdash v\,[V_d/x] : B \mid \Delta$, and

(iii) if $\Gamma, x : A \mid e : B \vdash \Delta$ is derivable then so is $\Gamma \mid e\,[V_d/x] : B \vdash \Delta$.

Dually, given any $\Gamma \mid E_d : A \vdash \Delta$,

(i) if $c : (\Gamma \vdash \alpha : A, \Delta)$ is derivable then so is $c\,[E_d/\alpha] : (\Gamma \vdash \Delta)$,

(ii) if $\Gamma \vdash v : B \mid \alpha : A, \Delta$ is derivable then so is $\Gamma \vdash v\,[E_d/\alpha] : B \mid \Delta$, and

(iii) if $\Gamma \mid e : B \vdash \alpha : A, \Delta$ is derivable then so is $\Gamma \mid e\,[E_d/\alpha] : B \vdash \Delta$.

Likewise, the same properties hold in the $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ type system when d is n or v.

Proof:
When just considering the simplified $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d^-$ type system, the substitution properties all follow by the standard mutual induction on the given derivation of the command, term, or coterm being substituted into. The stability of d comes into play for the $\cap R$ and $\cup L$ rules, to ensure they still apply after substitution. The only challenge arises with the $\cap\alpha$ and $\cup x$ typing rules of the more general $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ type system when d is n or v.

Consider the call-by-value case where $d = v$ (the call-by-name case is symmetric to this one). Given a derivation $\mathcal{E}$ of a typed value $\Gamma \vdash V_v : A \mid \Delta$ and a derivation $\mathcal{D}$ of the main judgement, the substitution operation $\mathcal{D}\,[\mathcal{E}/x]$ is defined by lexicographic induction on $\mathcal{E}$ and $\mathcal{D}$ (with $\mathcal{E}$ of higher significance). The interesting case of substitution is when $\mathcal{D}$ ends in a $\cup x$ inference, since the $\cap\alpha$ inference rule is never allowed when $d = v$ (because $\mu\alpha.c$ is never a v-value).

First, given that the derivation $\mathcal{D}$ of the main judgement is

$$\mathcal{D} = \frac{\begin{array}{cc} \vdots\ \mathcal{D}_1 & \vdots\ \mathcal{D}_2 \\ c : (\Gamma, x : A_1 \vdash \Delta) & c : (\Gamma, x : A_2 \vdash \Delta) \end{array}}{c : (\Gamma, x : A_1 \cup A_2 \vdash \Delta)}\ \cup x$$

then the substitution $\mathcal{D}\left[\mathcal{E}/x\right]$ proceeds as follows by inversion on the possible last inference of $\mathcal{E}$:

$$\mathcal{D}\left[\begin{array}{c}\vdots\,\mathcal{E}'\\ \Gamma \vdash V_v : A_i \mid \Delta\end{array} \cup R \Big/ x\right] \triangleq \mathcal{D}_i\left[\mathcal{E}'/x\right]$$

$$\mathcal{D}\left[\overline{\Gamma \vdash y : A_1 \cup A_2 \mid \Delta}\ \mathit{VarR} \Big/ x\right] \triangleq \frac{\begin{array}{cc}\vdots\,\mathcal{D}_1\left[\mathcal{E}_1/x\right] & \vdots\,\mathcal{D}_2\left[\mathcal{E}_1/x\right]\\ c\left[y/x\right] : (\Gamma', y : A_1 \vdash \Delta) & c\left[y/x\right] : (\Gamma', y : A_2 \vdash \Delta)\end{array}}{c\left[y/x\right] : (\Gamma', y : A_1 \cup A_2 \vdash \Delta)} \cup y$$

where $\Gamma = \Gamma', y : A_1 \cup A_2$

and $\mathcal{E}_i = \overline{\Gamma', y : A_i \vdash y : A_i \mid \Delta}\ \mathit{VarR}$

$$\mathcal{D}\left[\begin{array}{c}\vdots\,\mathcal{E}'\\ \Gamma \vdash V_v : C \mid \Delta\end{array} \cap E \Big/ x\right] \triangleq \left(\begin{array}{c}\vdots\,\mathcal{D}'\\ c : (\Gamma, x : C \vdash \Delta)\end{array}\right)\left[\mathcal{E}'/x\right]$$

where $C = (A_1 \cup A_2) \cap B$ or $C = B \cap (A_1 \cup A_2)$

and $\mathcal{D}'$ is the weakening of $\mathcal{D}$ by Proposition 5.2

Note that in the $\cup R$ case both $\mathcal{D}$ and $\mathcal{E}$ decrease, in the VarR case $\mathcal{D}$ decreases and $\mathcal{E}$ stays the same (modulo changing the type of the free variable), and in the $\cap E$ case $\mathcal{D}$ *increases* but $\mathcal{E}$ decreases. Also note that the cases where $\mathcal{E}$ ends with $\to R$ or $\cap R$ is impossible (because types don't match) as is ActR (because a μ-abstraction is not a value in call-by-value). Substitution therefore follows by lexicographic induction on $\mathcal{E}$ and $\mathcal{D}$, where the remaining cases for defining substitution are standard, and follow by propagating the substitution up the derivation of the main judgement by induction and putting back together the same inference rules (except for VarR which might be replaced). Additionally, the fact that the set of values and covalues is closed under substitution is used when the main derivation ends with a $\cap R$ or $\cup L$ inference (so that the (co)value restriction is still met after substitution occurs). □

With the substitution property above for both call-by-value and call-by-name disciplines of substitution, it is now possible to show that the ActR and ActL typing rules for μ- and $\tilde{\mu}$-abstractions are invertible, which gives us the standard preservation and subject reduction properties for call-by-value and -name intersection and union types.

Proposition 6.3. (Typing Inversion)
For every discipline d, in both the $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ and $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d^-$ type systems:

(i) if $\Gamma \vdash \mu\alpha.c : A \mid \Delta$ is derivable then so is $c : (\Gamma \vdash \alpha : A, \Delta)$, and

(ii) if $\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta$ is derivable then so is $c : (\Gamma, x : A \vdash \Delta)$.

Proof:
Inversion is immediate by definition in the $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d^-$ type system, and follows by induction on the

given typing derivation in the $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ type system. For inversion on a typed μ-abstraction, we have the following cases on the last inference of the derivation (inversion on a typed $\tilde{\mu}$-abstraction is dual):

- $ActR$: Is given immediately from the premise.
- $\cup R$: Follows from generalized weakening (Proposition 5.2).
- $\cap R$: Can be rewritten into $\cap\alpha$ as follows:

$$\frac{\begin{array}{cc}\vdots\,\mathcal{D} & \vdots\,\mathcal{E} \\ \Gamma \vdash \mu\alpha.c : A \mid \Delta & \Gamma \vdash \mu\alpha.c : B \mid \Delta\end{array}}{\Gamma \vdash \mu\alpha.c : A \cap B \mid \Delta}\cap R \quad\Rightarrow\quad \frac{\dfrac{\begin{array}{cc}\vdots\,\mathcal{D}_{IH} & \vdots\,\mathcal{E}_{IH} \\ c : (\Gamma \vdash \alpha : A, \Delta) & c : (\Gamma \vdash \alpha : B, \Delta)\end{array}}{c : (\Gamma \vdash \alpha : A \cap B, \Delta)}\cap\alpha}{\Gamma \vdash \mu\alpha.c : A \cap B \mid \Delta}\,ActR$$

□

Proposition 6.4. (Preservation)
For every admissible discipline d, in the $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d^-$ type system:

(i) If $c \mapsto_{\mu_d\tilde{\mu}_d\beta_d\varsigma_d} c'$ and $c : (\Gamma \vdash \Delta)$ is derivable, then so is $c' : (\Gamma \vdash \Delta)$.

(ii) If $e \mapsto_{\varsigma_d} e'$ and $\Gamma \mid e : A \vdash \Delta$ is derivable, then so is $\Gamma \mid e' : A \vdash \Delta$.

Likewise, the same properties hold in the $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ type system when d is n or v.

Proof:
By cases on the rewriting rule applied. For a command $c : (\Gamma \vdash \Delta)$, type preservation follows by induction on the given derivation. For a derivation ending in any non-Cut inference rule ($\cup x$ and $\cap x$ in call-by-value or $\cap\alpha$ and $\cup\alpha$ in call-by-name), this follows directly from the inductive hypothesis. The base case, where the derivation ends with a Cut between a term and coterm of type A, has the following sub-cases:

- $\langle \mu\alpha.c \parallel E_d \rangle \mapsto_{\mu_d}$: By inversion on the typing derivation of the μ-abstraction (Proposition 6.3), we know that $c : (\Gamma \vdash \alpha : A, \Delta)$, so $c\,[E_d/\alpha] : (\Gamma \vdash \Delta)$ by substitution (Proposition 6.2).
- $\langle V_d \parallel \tilde{\mu}x.c \rangle \mapsto_{\tilde{\mu}_d}$: Dual to the μ_d case above.
- $\langle \lambda x.v \parallel V_d \cdot E_d \rangle \mapsto_{\beta_d}$: The derivation must conclude with a Cut followed by a chain of $\cap\alpha$ and $\cup x$ inferences. So rewrite the bottom-most Cut the sub-derivation by induction on the derivations of $\Gamma \vdash \lambda x.v : A \mid \Delta$ and $\Gamma \mid V_d \cdot E_d : A \vdash \Delta$ as follows (using Proposition 6.2 in the base case $\to R - \to L$):
  - $\to R - \to L$:

$$\frac{\dfrac{\begin{array}{c}\vdots\,\mathcal{D} \\ \Gamma, x : B \vdash v : C \mid \Delta\end{array}}{\Gamma \vdash \lambda x.v : B \to C \mid \Delta} \quad \dfrac{\begin{array}{cc}\vdots\,\mathcal{E}_1 & \vdots\,\mathcal{E}_2 \\ \Gamma \vdash V_d : B \mid \Delta & \Gamma \mid E_d : C \vdash \Delta\end{array}}{\Gamma \mid V_d \cdot E_d : B \to C \mid \Delta}}{\langle \lambda x.v \parallel V_d \cdot E_d \rangle : (\Gamma \vdash \Delta)} \quad\Rightarrow\quad \frac{\begin{array}{cc}\vdots\,\mathcal{D}\,[\mathcal{E}_1/x] & \vdots\,\mathcal{E}_2 \\ \Gamma \vdash v\,[V_d/x] : C \mid \Delta & \Gamma \mid E_d : C \vdash \Delta\end{array}}{\langle v\,[V_d/x] \parallel E_d \rangle : (\Gamma \vdash \Delta)}$$

– $\cap R - \cap L$:

$$\dfrac{\dfrac{\dfrac{\vdots\,\mathcal{D}_i}{\Gamma \vdash \lambda x.v : B_i \mid \Delta} \quad \dfrac{\vdots\,\mathcal{E}}{\Gamma \mid V_d \cdot E_d : B_i \vdash \Delta}}{\Gamma \vdash \lambda x.v : B_1 \cap B_2 \mid \Delta} \quad \dfrac{\vdots\,\mathcal{E}}{\Gamma \mid V_d \cdot E_d : B_1 \cap B_2 \vdash \Delta}}{\langle \lambda x.v \parallel V_d \cdot E_d \rangle : (\Gamma \vdash \Delta)} \Rightarrow \dfrac{\dfrac{\vdots\,\mathcal{D}_i}{\Gamma \vdash \lambda x.v : B_i \mid \Delta} \quad \dfrac{\vdots\,\mathcal{E}}{\Gamma \mid V_d \cdot E_d : B_i \vdash \Delta}}{\langle \lambda x.v \parallel V_d \cdot E_d \rangle : (\Gamma \vdash \Delta)}$$

– $\cup R - \cup L$: Dual to the $\cap R - \cap L$ case above.

– $\cap E - \mathit{Any}$:

$$\dfrac{\dfrac{\dfrac{\vdots\,\mathcal{D}}{\Gamma \vdash \lambda x.v : B_1 \cap B_2 \mid \Delta}}{\Gamma \vdash \lambda x.v : B_i \mid \Delta} \quad \dfrac{\vdots\,\mathcal{E}}{\Gamma \mid V_d \cdot E_d : B_i \vdash \Delta}}{\langle \lambda x.v \parallel V_d \cdot E_d \rangle : (\Gamma \vdash \Delta)} \Rightarrow \dfrac{\dfrac{\vdots\,\mathcal{D}}{\Gamma \vdash \lambda x.v : B_1 \cap B_2 \mid \Delta} \quad \dfrac{\vdots\,\mathcal{E}}{\Gamma \mid V_d \cdot E_d : B_i \vdash \Delta}}{\langle \lambda x.v \parallel V_d \cdot E_d \rangle : (\Gamma \vdash \Delta)}$$

– $\mathit{Any} - \cup E$: Dual to the $\cap E - \mathit{Any}$ case above.

Notice that in each case, either the result of rewriting the derivation is the right-hand side of the rule (in $\to R - \to L$) or is a cut of the same term and coterm at a different type where the derivation and type is smaller (in $\to R - \to L$, $\cap R - \cap L$, or $\cup R - \cup L$) the derivation has one fewer elimination rule (in $\cap E - \mathit{Any}$ and $\mathit{Any} - \cup E$). So this process must eventually terminate.

For a coterm $\Gamma \mid e : A \vdash \Delta$, type preservation of the only step $e \mapsto_{\varsigma_d} e'$ proceeds by induction on the given typing derivation. If the derivation ends in $\to L$, then the right-hand side can be typed by additionally using the ActL, ActR, VarR, VarL, and Cut rules. Otherwise, the derivation ends with either $\cap L$, $\cup L$, or $\cup E$: each follows from the inductive hypothesis, using the fact that the set of covalues is closed under reduction (for $\cup L$). □

Proposition 6.5. (Subject Reduction)
For every admissible discipline d, in the $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d^-$ type system:

(i) If $c \to_{\mu_d\tilde{\mu}_d\beta_d\varsigma_d} c'$ and $c : (\Gamma \vdash \Delta)$ is derivable, then so is $c' : (\Gamma \vdash \Delta)$.

(ii) If $v \to_{\mu_d\tilde{\mu}_d\beta_d\varsigma_d} v'$ and $\Gamma \vdash v : A \mid \Delta$ is derivable, then so is $\Gamma \vdash v' : A \mid \Delta$.

(iii) If $e \to_{\mu_d\tilde{\mu}_d\beta_d\varsigma_d} e'$ and $\Gamma \mid e : A \vdash \Delta$ is derivable, then so is $\Gamma \mid e : A \vdash \Delta$.

Likewise, the same properties hold in the $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ type system when d is n or v.

Proof:
By induction on the given derivation, using Proposition 6.4 in the base case where an operational rule ($\mapsto$) is applied. Note that in each other case, where a reduction is applied to a sub-expression, the same inference rule still applies to the rewritten expression because (a) the set of values and covalues are closed under reduction, and (b) no inference rule performs a substitution in its conclusion. For example, in the $\cup x$ rule for call-by-value, if the command c in the conclusion reduces as $c \to c'$, then the inductive hypothesis applies to both premises so that $\cup x$ infers the same type for c'. □

7. Uniform Proof of Strong Normalization — Soundness

In order to show that well-typed commands, terms, and coterms are strongly normalizing, we will extend the uniform proof of strong normalization from [1] to also account for intersection and union types. Interestingly, even though this previous work did not account for union and intersection types within the typed language, the proof technique for strong normalization relied on a model of types that was built on top of a notion of subtyping with corresponding union and intersections. There, the purpose of subtyping was for establishing a measure to construct the model of individual types as the fixed-point solution to a (monotonic) operation. Here, the corresponding model of union and intersections play an even more prominent role, since they are the basis for representing the syntactic union and intersection types in the language. In the end, we will find that the disciplined call-by-value and call-by-name type systems are sound with respect to strong normalization, whereas the type system for non-deterministic reduction needs further restrictions on where intersection types can be introduced on the right and where union types can be introduced on the left.

The proof that follows is parameterized by a choice of discipline, d. At first, the initial discussion of pre-candidates in Section 7.1 is not affected by the choice of d. But in the following Sections 7.2, 7.3, and 7.4, we assume that d is a deterministic admissible discipline (like v or n) in order to prove that well-typed expressions of $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ are strongly normalizing. Then in Section 7.5, 7.6, and 7.7, we generalize to allow for d to be non-deterministic, but still admissible, disciplines (like u). As it turns out, the $\cap R$ and $\cup L$ rules are *too strong* for the non-deterministic u-reduction rules even with the (co)value restriction (which is trivialized by u), but the further restrictions imposed $\overline{\lambda}\mu\tilde{\mu}\cap\cup_u^-$ recover soundness in the face of non-determinism.

7.1. Pre-candidates

By now, a standard approach to proving strong normalization is with some variant of the reducibility candidates method, wherein first a domain of well-behaved candidate objects is established which fully encompasses the interpretation of types into their semantic meaning. Since we will be discussing multiple versions of *candidates*, it's useful to begin one step earlier at the domain of *pre-candidates*, which outlines the overall shape of candidates but stops short of enforcing the crucial properties that ensure they are well-behaved. Before getting into proper candidates, we can already explore some useful general properties about the domain of pre-candidates as a whole.

In the λ-calculus, types only describe terms. But in the $\overline{\lambda}\mu\tilde{\mu}$ sequent calculus, types describe *both* terms and coterms. For this reason, pre-candidates are dual objects containing both a term side and a coterm side, which is expressive enough to encompass both roles of types in the sequent calculus.

Definition 7.1. (Pre-candidate)
A *pre-candidate* is a pair $\mathbb{A} = (\mathbb{A}^+, \mathbb{A}^-)$ of a set of strongly normalizing terms ($\mathbb{A}^+$) and a set of strongly normalizing coterms ($\mathbb{A}^-$). As notation on a pre-candidate $\mathbb{A}$, we write $\mathbb{A}^+$ for the first component of $\mathbb{A}$ (the set of terms), $\mathbb{A}^-$ for the second component of $\mathbb{A}$ (the set of coterms), $v \in \mathbb{A}$ as shorthand for $v \in \mathbb{A}^+$ and $e \in \mathbb{A}$ as shorthand for $e \in \mathbb{A}^-$. Two important pre-candidates the *biggest* pre-candidate $\mathbb{W}$ of all strongly normalizing terms and coterms, and the pre-candidate $\mathbb{V}$ of

only strongly normalizing values and covalues.

$$\mathbb{W} \triangleq (\{v \mid v \text{ is strongly normalizing}\}, \{e \mid e \text{ is strongly normalizing}\})$$
$$\mathbb{V} \triangleq (\{v \in \mathbb{W} \mid v \text{ is a } d\text{-value}\}, \{e \in \mathbb{W} \mid e \text{ is a } d\text{-covalue}\})$$

The $\overline{\lambda}\mu\tilde{\mu}$ sequent calculus has more than just terms and coterms; it also has commands which represent execution states. What role do they play in the domain of pre-candidates? They are the key variable in the most fundamental operation on pre-candidates: orthogonality. The idea behind orthogonality is to take a set of especially well-behaved commands—for example, the set of all strongly normalizing commands—and use those commands as a test for generating well-behaved partners. That is, given some coterms as a set of observations, orthogonality can give back all terms that are strongly normalizing under each of those observations. Or dually, given some terms as a set of results, we can give back all coterms that are strongly normalizing when given each those results.

Definition 7.2. (Orthogonality)
Given *any* set of commands $\mathbb{P}$, the *orthogonality* operation is defined on any set of terms ($\mathbb{A}^+$) and set of coterms ($\mathbb{A}^-$) as:

$$\mathbb{A}^{+\mathbb{P}} \triangleq \{e \in \mathbb{W} \mid \forall v \in \mathbb{A}^+, \langle v \parallel e\rangle \in \mathbb{P}\} \qquad \mathbb{A}^{-\mathbb{P}} \triangleq \{v \in \mathbb{W} \mid \forall e \in \mathbb{A}^-, \langle v \parallel e\rangle \in \mathbb{P}\}$$

Orthogonality is lifted to operate on pre-candidates $\mathbb{A}$ by flipping its two sides as follows:

$$(\mathbb{A}^+, \mathbb{A}^-)^{\mathbb{P}} \triangleq (\mathbb{A}^{-\mathbb{P}}, \mathbb{A}^{+\mathbb{P}})$$

The most important application of orthogonality is with respect to the set of all strongly normalizing commands, which we write as $\perp\!\!\!\perp$:

$$\perp\!\!\!\perp \triangleq \{c \mid c \text{ is strongly normalizing}\}$$

The use of orthogonality with the above set of strongly normalizing commands is written $\mathbb{A}^{\perp\!\!\!\perp}$.

Since pre-candidates are effectively two-sided objects, there is more than one way to relate them compared with just a single set, since the two sides can be either in agreement or opposed to one another. The first relation is just straightforward containment where both sides are treated the same which we call *refinement*, which corresponds to the ordinary subset relation. The second relation has the two sides contrary to one another which we call *subtyping*, since it corresponds to the notion of behavioral subtyping on candidates. The idea behind subtyping $\mathbb{A} <: \mathbb{B}$ is that every value of $\mathbb{A}$ is also a valid value of $\mathbb{B}$, but also dually every observation of $\mathbb{B}$ can be used on values of $\mathbb{A}$.

Definition 7.3. (Subtyping and Refinement)
The *refinement* ($\sqsubseteq$) and *subtyping* ($\leq$) orders on pre-candidates $\mathbb{A} = (\mathbb{A}^+, \mathbb{A}^-)$ and $\mathbb{B} = (\mathbb{B}^+, \mathbb{B}^-)$ is:

$$\mathbb{A} \sqsubseteq \mathbb{B} \triangleq (\mathbb{A}^+ \subseteq \mathbb{B}^+) \wedge (\mathbb{A}^- \subseteq \mathbb{B}^-)$$
$$\mathbb{A} \leq \mathbb{B} \triangleq (\mathbb{A}^+ \subseteq \mathbb{B}^+) \wedge (\mathbb{A}^- \supseteq \mathbb{B}^-)$$

When $\mathbb{A} \sqsubseteq \mathbb{B}$ we say "$\mathbb{A}$ *refines* $\mathbb{B}$" (dually, "$\mathbb{B}$ *extends* $\mathbb{A}$") and when $\mathbb{A} \leq \mathbb{B}$ we say that "$\mathbb{A}$ is a *subtype* of $\mathbb{B}$" (dually, "$\mathbb{B}$ is a *supertype* of $\mathbb{A}$"). Note that refinement between pre-candidates says that one is wholly contained within the other, whereas subtyping order is inverted on negative side of pre-candidates. Both orders form separate lattices which come with their own notions of union and intersections (written $\sqcup$ and $\sqcap$ for refinement and $\vee$ and $\wedge$ for subtyping), defined as:

$$\mathbb{A} \sqcup \mathbb{B} \triangleq (\mathbb{A}^+ \cup \mathbb{B}^+, \mathbb{A}^- \cup \mathbb{B}^-) \qquad \mathbb{A} \sqcap \mathbb{B} \triangleq (\mathbb{A}^+ \cap \mathbb{B}^+, \mathbb{A}^- \cap \mathbb{B}^-)$$
$$\mathbb{A} \vee \mathbb{B} \triangleq (\mathbb{A}^+ \cup \mathbb{B}^+, \mathbb{A}^- \cap \mathbb{B}^-) \qquad \mathbb{A} \wedge \mathbb{B} \triangleq (\mathbb{A}^+ \cap \mathbb{B}^+, \mathbb{A}^- \cup \mathbb{B}^-)$$

Besides the obvious connection between subtyping of pre-candidates and subtyping of types, which is an important aspect of intersection and union types in practice, one reason motivating the two separate orderings is that they each have a very different relationship with the fundamental orthogonality operation. In particular, refinement order exhibits the following usual standard properties that arise in the study of *biorthogonality*, whereas subtyping order is stable under orthogonality.

Proposition 7.4. The following standard ordering properties of orthogonality hold:

(i) *Double orthogonal introduction*: $\mathbb{A} \sqsubseteq \mathbb{A}^{\perp\!\!\perp\perp}$

(ii) *Triple orthogonal elimination*: $\mathbb{A}^{\perp\!\!\perp\perp\perp} = \mathbb{A}^{\perp}$

(iii) *Contrapositive* (*a.k.a antitonicity*): If $\mathbb{A} \sqsubseteq \mathbb{B}$ then $\mathbb{B}^{\perp} \sqsubseteq \mathbb{A}^{\perp}$

(iv) *Monotonicity*: If $\mathbb{A} \leq \mathbb{B}$ then $\mathbb{A}^{\perp} \leq \mathbb{B}^{\perp}$

Furthermore, the following standard De Morgan properties hold:

(i) $(\mathbb{A} \sqcup \mathbb{B})^{\perp} = \mathbb{A}^{\perp} \sqcap \mathbb{B}^{\perp}$

(ii) $(\mathbb{A} \sqcap \mathbb{B})^{\perp} \sqsupseteq \mathbb{A}^{\perp} \sqcup \mathbb{B}^{\perp}$

Note that, on the one hand, the refinement properties of orthogonality above all correspond exactly to properties of negation in intuitionistic logic. On the other hand, orthogonality is a monotonic operation with respect to subtyping (in contrast with the antitone behavior of refinement). This fact about monotonicity is crucial to the uniform model of strong normalization [1], which generalizes the symmetric candidates methodology [15] by revealing its hidden use of subtyping.

7.2. Reducibility candidates

For our first, simpler, approach to candidate semantics for types, we will look to the *reducibility candidates* and *biorthogonality* methods which are appropriate for deterministic evaluation (*e.g.,* like the call-by-value v and call-by-name n disciplines). The main key of any reducibility candidates proof is a form of *expansion* lemma, which says that things which step to something good must have been good to begin with (*i.e.,* "goodness" is closed under $\mapsto$ expansion). For our specific setting, this expansion lemma takes the following form, which holds under the assumption that the chosen discipline d is deterministic.

Proposition 7.5. (Expansion)
For all deterministic d, (co)terms $v, e \in \mathbb{W}$ (*i.e.,* v and e are strongly normalizing), and commands c, c', if $c \mapsto_{\mu_d \tilde{\mu}_d \beta_d <_d} c' \in \perp\!\!\!\perp$ (*i.e.,* c steps to c' which is strongly normalizing) then $c \in \perp\!\!\!\perp$.

However, disciplined reduction has a serious consequence on this form of expansion. For example, consider the challenge of justifying in the call-by-value v discipline that a $\tilde{\mu}$-abstraction $\tilde{\mu}x.c$ is a coterm of some $\mathbb{A}$. We would like to argue behaviorally about how $\tilde{\mu}x.c$ behaves when paired with terms of $\mathbb{A}$. That is, we might know that for any $V_v \in \mathbb{A}$,

$$\langle V_v \parallel \tilde{\mu}x.c \rangle \mapsto_{\tilde{\mu}_v} c\,[V_v/x] \in \perp\!\!\!\perp$$

or in other words, $\tilde{\mu}x.c$ forms a strongly-normalizing command after one step when paired with any v-value of $\mathbb{A}$. But not every term is a v-value, so we can't use $\tilde{\mu}_v$-reduction to say anything about the other terms of $\mathbb{A}$! For this reason, we need to consider the (co)value restriction as part of the definition of reducibility candidates.

Definition 7.6. ((Co)value Restriction)
The *(co)value restriction* on a pre-candidate $\mathbb{A}$ is defined as: $\mathbb{A}^v \triangleq \mathbb{A} \sqcap \mathbb{V}$. Note that $\mathbb{A}^v \sqsubseteq \mathbb{A}$.

Definition 7.7. (Reducibility Candidate)
A *reducibility candidate* is a pre-candidate $\mathbb{A}$ such that $\mathbb{A}^{v\perp\!\!\!\perp} \sqsubseteq \mathbb{A} \sqsubseteq \mathbb{A}^{\perp\!\!\!\perp}$. In other words, a reducibility candidate $\mathbb{A}$ is any pre-candidate such that the following two properties hold:

(i) *Soundness* ($\mathbb{A} \sqsubseteq \mathbb{A}^{\perp\!\!\!\perp}$): For all $v, e \in \mathbb{A}$, the command $\langle v \parallel e \rangle$ is strongly normalizing.

(ii) *Completeness* ($\mathbb{A}^{v\perp\!\!\!\perp} \sqsubseteq \mathbb{A}$): If v is strongly normalizing as well as $\langle v \parallel E \rangle$ for all $E \in \mathbb{A}$, then $v \in \mathbb{A}$. Dually, if e is strongly normalizing as well as $\langle V \parallel e \rangle$ for all $V \in \mathbb{A}$, then $e \in \mathbb{A}$.

Intuitively, the soundness property of reducibility candidates ensures that they don't have too many (co)terms, which justifies that the Cut rule is sound, whereas the completeness property ensures that there are enough (co)terms, which justifies the generic $ActR$ and $ActL$ rules of every type along with type-specific inference rules. The fact that the completeness requirement of reducibility candidates ($\mathbb{A}^{v\perp\!\!\!\perp} \sqsubseteq \mathbb{A}$) only requires checking potential members against the *values* or *covalues* of that candidate means that the operational semantics is enough to justify membership of complex computations by expansion. Returning back to the above example, if we know that $\mathbb{A}$ is a reducibility candidate and $c\,[V/x] \in \perp\!\!\!\perp$ for any $V \in \mathbb{A}$, then we can conclude from expansion (Proposition 7.5) and completeness (Definition 7.7) that $\tilde{\mu}x.c \in \mathbb{A}$.

It turns out there is another equivalent definition of reducibility candidates, based on the standard properties of orthogonality from Proposition 7.4: reducibility candidates are exactly the fixed points of (co)value-restricted orthogonality. Fixed points play a prominent role later in Section 7.5.

Proposition 7.8. $\mathbb{A}$ is a reducibility candidate if and only if $\mathbb{A} = \mathbb{A}^{v\perp\!\!\!\perp}$.

So far, we have said *what* reducibility candidates are, but not yet *how* to make one of them. Here are two dual ways (one Positively-oriented and the other Negatively-oriented) to construct a

reducibility candidate from a set of values (C) or covalues (O), along with a common operation $\mathcal{R}(-)$ for generating a complete reducibility candidate from a sound pre-candidate $\mathbb{A}$ of (co)values.

$$Pos(C) \triangleq (C, C^{\perp\!\!\!\perp v})^{\perp\!\!\!\perp v} \qquad Neg(O) \triangleq (O^{\perp\!\!\!\perp v}, O)^{\perp\!\!\!\perp v} \qquad \mathcal{R}(\mathbb{A}) \triangleq \mathbb{A}^{\perp\!\!\!\perp}$$

Proposition 7.9. For any set of values C, covalues O, and pre-candidate $\mathbb{A} = \mathbb{A}^{\perp\!\!\!\perp v}$,

(i) $Pos(C) = Pos(C)^{\perp\!\!\!\perp v}$ and $C \subseteq Pos(C)^+$,

(ii) $Neg(O) = Neg(O)^{\perp\!\!\!\perp v}$ and $O \subseteq Neg(O)^-$, and

(iii) $\mathcal{R}(\mathbb{A})$ is a reducibility candidate such that $\mathcal{R}(\mathbb{A})^v = \mathbb{A}$.

7.3. Intersection and union candidates

Now that we have reducibility candidates which are capable of modeling potential types, we also need to determine what is the intersection and union of two reducibility candidates. A good place to start are the corresponding operations $\wedge$ and $\vee$ in the subtyping lattice. However, these two operations are not good enough: since $\wedge$ and $\vee$ are defined on pre-candidates, they are capable of combining any two reducibility candidates, but we only get *pre-candidate* back without any additional assurances. With intersections, the intersection $\mathbb{A} \wedge \mathbb{B}$ between two reducibility candidates $\mathbb{A}$ and $\mathbb{B}$ may not be another reducibility candidate, but there may be some other suitable reducibility candidate $\mathbb{C} \leq \mathbb{A} \wedge \mathbb{B}$ which has fewer terms and more coterms than $\mathbb{A} \wedge \mathbb{B}$.

For example, let $\mathbb{E}$ and $\mathbb{O}$ be the reducibility candidates corresponding to even and odd numbers, respectively. The subtyping intersection $\mathbb{E} \wedge \mathbb{O}$ will then contain all the terms which behave like *both* an even and an odd number, and all the coterms which are capable of accepting *either* an even or an odd number. But being both an even and an odd number is vacuous! So it would be safe to include many other sensible (*i.e.,* strongly normalizing) coterms which expect other inputs, like strings, since there is no way for the term to return any result at all. This is a case where just $\mathbb{E} \wedge \mathbb{O}$ is not complete enough to be a reducibility candidate, because it is missing some extra coterms that do not come from either $\mathbb{E}$ or $\mathbb{O}$.

So the true intersection and union operations for reducibility candidates is based on $\wedge$ and $\vee$, but needs to do some extra work to complete the definition and include *everything* that's safe given that some options may be eliminated. A second place to look is to the Pos and Neg construction of candidates from the previous section. As it turns out, either will do: intersection and union candidates can be seen as either positively- or negatively-oriented, both of which are equivalent to the completion of the base $\wedge$ and $\vee$ operations of the subtyping lattice.

Proposition 7.10. For all reducibility candidates $\mathbb{A}$ and $\mathbb{B}$,

$$\mathbb{A} \wedge \mathbb{B} \sqsubseteq (\mathbb{A} \wedge \mathbb{B})^{\perp\!\!\!\perp} \qquad (\mathbb{A} \wedge \mathbb{B})^v \sqsubseteq (\mathbb{A} \wedge \mathbb{B})^{v\perp\!\!\!\perp v} = Neg((\mathbb{A} \wedge \mathbb{B})^{v-}) = Pos((\mathbb{A} \wedge \mathbb{B})^{v+})$$
$$\mathbb{A} \vee \mathbb{B} \sqsubseteq (\mathbb{A} \vee \mathbb{B})^{\perp\!\!\!\perp} \qquad (\mathbb{A} \vee \mathbb{B})^v \sqsubseteq (\mathbb{A} \vee \mathbb{B})^{v\perp\!\!\!\perp v} = Pos((\mathbb{A} \vee \mathbb{B})^{v+}) = Neg((\mathbb{A} \vee \mathbb{B})^{v-})$$

Proof:
First, note that $\mathbb{A} \wedge \mathbb{B} \sqsubseteq (\mathbb{A} \wedge \mathbb{B})^{\perp\!\!\!\perp}$ by the term and coterm components of the De Morgan Laws (Proposition 7.4) and the fact that $\mathbb{A}$ and $\mathbb{B}$ are fixed points of $-^{\perp\!\!\!\perp}$ (Proposition 7.8) as follows:

$$(\mathbb{A} \wedge \mathbb{B})^{+} = \mathbb{A}^{+} \cap \mathbb{B}^{+} = \mathbb{A}^{\perp\!\!\!\perp +} \cap \mathbb{B}^{\perp\!\!\!\perp +} = \mathbb{A}^{-\perp\!\!\!\perp} \cap \mathbb{B}^{-\perp\!\!\!\perp} = (\mathbb{A}^{-} \cup \mathbb{B}^{-})^{\perp\!\!\!\perp} = (\mathbb{A} \wedge \mathbb{B})^{\perp\!\!\!\perp}$$
$$(\mathbb{A} \wedge \mathbb{B})^{-} = \mathbb{A}^{-} \cup \mathbb{B}^{-} = \mathbb{A}^{\perp\!\!\!\perp -} \cap \mathbb{B}^{\perp\!\!\!\perp -} = \mathbb{A}^{+\perp\!\!\!\perp} \cap \mathbb{B}^{+\perp\!\!\!\perp} \subseteq (\mathbb{A}^{+} \cup \mathbb{B}^{+})^{\perp\!\!\!\perp} = (\mathbb{A} \wedge \mathbb{B})^{\perp\!\!\!\perp}$$

Next, observe that $(\mathbb{A} \wedge \mathbb{B})^{v\perp\!\!\!\perp v} = \mathit{Neg}((\mathbb{A} \wedge \mathbb{B})^{v-}) = \mathit{Pos}((\mathbb{A} \wedge \mathbb{B})^{v+})$ from the additional facts that and $\mathbb{B}$ are fixed points of both $-^{v\perp\!\!\!\perp}$ (Proposition 7.8), and $\mathit{Pos}(C)$ and $\mathit{Neg}(O)$ are fixed points of $-^{\perp\!\!\!\perp v}$ (Proposition 7.9), like so:

$$\begin{aligned}(\mathbb{A} \wedge \mathbb{B})^{v\perp\!\!\!\perp v} &= ((\mathbb{A}^{-} \cup \mathbb{B}^{-})^{v\perp\!\!\!\perp v}, (\mathbb{A}^{+} \cap \mathbb{B}^{+})^{v\perp\!\!\!\perp v}) = ((\mathbb{A}^{-} \cup \mathbb{B}^{-})^{v\perp\!\!\!\perp v}, (\mathbb{A}^{-v\perp\!\!\!\perp} \cap \mathbb{B}^{-v\perp\!\!\!\perp})^{v\perp\!\!\!\perp v}) \\ &= ((\mathbb{A}^{v-} \cup \mathbb{B}^{v-})^{\perp\!\!\!\perp v}, (\mathbb{A}^{v-} \cup \mathbb{B}^{v-})^{\perp\!\!\!\perp v\perp\!\!\!\perp v}) = \mathit{Neg}((\mathbb{A} \wedge \mathbb{B})^{v-})\end{aligned}$$

$$\begin{aligned}\mathit{Neg}((\mathbb{A} \wedge \mathbb{B})^{v-}) &= \mathit{Neg}((\mathbb{A} \wedge \mathbb{B})^{v-})^{\perp\!\!\!\perp v} = ((\mathbb{A}^{v-} \cup \mathbb{B}^{v-})^{\perp\!\!\!\perp v}, (\mathbb{A}^{v-} \cup \mathbb{B}^{v-})^{\perp\!\!\!\perp v\perp\!\!\!\perp v})^{\perp\!\!\!\perp v} \\ &= (\mathbb{A}^{v\perp\!\!\!\perp v+} \cap \mathbb{B}^{v\perp\!\!\!\perp v+}, (\mathbb{A}^{v\perp\!\!\!\perp v+} \cap \mathbb{B}^{v\perp\!\!\!\perp v+})^{\perp\!\!\!\perp v})^{\perp\!\!\!\perp v} = (\mathbb{A}^{v+} \cap \mathbb{B}^{v+}, (\mathbb{A}^{v+} \cap \mathbb{B}^{v+})^{\perp\!\!\!\perp v})^{\perp\!\!\!\perp v} \\ &= ((\mathbb{A}^{v+} \cap \mathbb{B}^{v+})^{\perp\!\!\!\perp v\perp\!\!\!\perp v}, (\mathbb{A}^{v+} \cap \mathbb{B}^{v+})^{\perp\!\!\!\perp v}) = \mathit{Pos}((\mathbb{A} \wedge \mathbb{B})^{v+})\end{aligned}$$

The calculations for unions are dual to the above. □

Now we have the core pre-candidate of (co)values that describe the intersection and union of two candidates, including all newly-valid responses that ignore options which were eliminated. We can complete them in the usual way by applying a final orthogonal operation, which includes any safe and sensible non-(co)values based on that core.

Proposition 7.11. For all reducibility candidates $\mathbb{A}$ and $\mathbb{B}$, with respect to $\leq$ order,

(i) $(\mathbb{A} \wedge \mathbb{B})^{v\perp\!\!\!\perp v\perp\!\!\!\perp}$ is the greatest reducibility candidate lower bound of $\mathbb{A}$ and $\mathbb{B}$, and

(ii) $(\mathbb{A} \vee \mathbb{B})^{v\perp\!\!\!\perp v\perp\!\!\!\perp}$ is the least reducibility candidate upper bound of $\mathbb{A}$ and $\mathbb{B}$.

Proof:
First, note that $(\mathbb{A} \wedge \mathbb{B})^{v\perp\!\!\!\perp v\perp\!\!\!\perp} = \mathcal{R}((\mathbb{A} \wedge \mathbb{B})^{v\perp\!\!\!\perp v})$ is indeed a reducibility candidate due to Propositions 7.10 and 7.9. Second, by monotonicity (Proposition 7.4) and the fact that $\mathbb{A}$ and $\mathbb{B}$ are fixed points of $-^{v\perp\!\!\!\perp}$ (Proposition 7.8), we have from $\mathbb{A} \wedge \mathbb{B} \leq \mathbb{A}$ and $\mathbb{A} \wedge \mathbb{B} \leq \mathbb{B}$:

$$(\mathbb{A} \wedge \mathbb{B})^{v\perp\!\!\!\perp v\perp\!\!\!\perp} \leq \mathbb{A}^{v\perp\!\!\!\perp v\perp\!\!\!\perp} = \mathbb{A} \qquad\qquad (\mathbb{A} \wedge \mathbb{B})^{v\perp\!\!\!\perp v\perp\!\!\!\perp} \leq \mathbb{B}^{v\perp\!\!\!\perp v\perp\!\!\!\perp} = \mathbb{B}$$

so that $(\mathbb{A} \wedge \mathbb{B})^{v\perp\!\!\!\perp v\perp\!\!\!\perp} \leq \mathbb{A} \wedge \mathbb{B}$, since $\mathbb{A} \wedge \mathbb{B}$ is the greatest lower bound of $\mathbb{A}$ and $\mathbb{B}$. Third, suppose that there is some other reducibility candidate $\mathbb{C}$ that is also a lower bound of $\mathbb{A}$ and $\mathbb{B}$. It follows that $\mathbb{C} \leq \mathbb{A} \wedge \mathbb{B}$, and thus

$$\mathbb{C} = \mathbb{C}^{v\perp\!\!\!\perp v\perp\!\!\!\perp} \leq (\mathbb{A} \wedge \mathbb{B})^{v\perp\!\!\!\perp v\perp\!\!\!\perp}$$

by monotonicity again. Therefore, $(\mathbb{A} \wedge \mathbb{B})^{v\perp\!\!\!\perp v\perp\!\!\!\perp}$ must be the greatest lower bound of $\mathbb{A}$ and $\mathbb{B}$ that is also a reducibility candidate (*i.e.,* a fixed point of $-^{v\perp\!\!\!\perp}$).

The fact that $(\mathbb{A} \vee \mathbb{B})^{v\perp\!\!\!\perp v\perp\!\!\!\perp}$ is the least reducibility candidate upper bound of $\mathbb{A}$ and $\mathbb{B}$ is symmetric to the above fact about intersections. □

7.4. A uniform model of deterministic types

We now have established enough of a framework to interpret syntactic types as reducibility candidates and typing judgements as logical statements. At this point, these following definitions are standard, as per the approaches of logical relations and biorthogonality. The interpretation of types (which is parameterized by a mapping ρ from atomic propositions p to candidates) is defined as:

$$\mathbb{A} \cdot \mathbb{B} \triangleq \{V_d \cdot E_d \mid V_d \in \mathbb{A}, E_d \in \mathbb{B}\}$$

$$\begin{aligned} [\![p]\!]_\rho &\triangleq \rho(p) & [\![A \to B]\!]_\rho &\triangleq \mathcal{R}(Neg([\![A]\!]_\rho \cdot [\![B]\!]_\rho)) \\ [\![A \cap B]\!]_\rho &\triangleq ([\![A]\!]_\rho \wedge [\![B]\!]_\rho)^{v \perp\!\!\!\perp v \perp\!\!\!\perp} & [\![A \cup B]\!]_\rho &\triangleq ([\![A]\!]_\rho \vee [\![B]\!]_\rho)^{v \perp\!\!\!\perp v \perp\!\!\!\perp} \end{aligned}$$

Typing judgements are then interpreted as a statement about membership of a command, term, or coterm in the candidate corresponding to a type. To handle the environments Γ and Δ, we quantify over all suitable substitutions of values and covalues for the free variables and covariables. In particular, we make use of a simultaneous substitution $V_1/x_1, \ldots V_n/x_n, E_1/\alpha_1, \ldots E_m/\alpha_m$ which we range over with the metavariable σ, and write $c[\sigma]$, *etc.,* , to denote the application of the substitution σ to c. $Subst$ is the set of all such simultaneous substitutions σ.

$$\begin{aligned} [\![\Gamma \vdash \Delta]\!]_\rho &\triangleq \{\sigma \in Subst \mid \forall x{:}A \in \Gamma, x\,[\sigma] \in [\![A]\!]_\rho\} \\ &\quad \cap \{\sigma \in Subst \mid \forall \alpha{:}A \in \Delta, \alpha\,[\sigma] \in [\![A]\!]_\rho\} \\ [\![c : (\Gamma \vdash \Delta)]\!]_\rho &\triangleq \forall \sigma \in [\![\Gamma \vdash \Delta]\!]_\rho . c\,[\sigma] \in \perp\!\!\!\perp \\ [\![\Gamma \vdash v : A \mid \Delta]\!]_\rho &\triangleq \forall \sigma \in [\![\Gamma \vdash \Delta]\!]_\rho . v\,[\sigma] \in [\![A]\!]_\rho \\ [\![\Gamma \mid e : A \vdash \Delta]\!]_\rho &\triangleq \forall \sigma \in [\![\Gamma \vdash \Delta]\!]_\rho . e\,[\sigma] \in [\![A]\!]_\rho \end{aligned}$$

Proposition 7.12. (Deterministic Soundness)
For every admissible deterministic discipline d, if a judgement J is derivable in $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$, then $[\![J]\!]_\rho$ is true for any ρ.

Proposition 7.13. (Deterministic Strong Normalization)
For every admissible deterministic discipline d, every well-typed command, term, and coterm of $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ is strongly normalizing.

7.5. Symmetric candidates

So far in the previous three sections, we have only considered the possibility where d is a deterministic discipline. This assumption showed up in the key expansion property (Proposition 7.5), which only works for a deterministic operational semantics. But in the non-deterministic case, we could have a critical pair of steps $c_1 \leftarrowtail \langle v \parallel e \rangle \rightarrowtail c_2$ where just because we know that c_2 is strongly normalizing, that doesn't mean that c_1 must *also* be strongly normalizing. Since the expansion property is not guaranteed for non-deterministic reduction (like when $d = u$), we need another approach. Instead of reducibility candidates, here we will consider a strictly more general notion of symmetric candidates.

The first step on the road to symmetric candidates is to generalize the orthogonality operation, referred to here as pre-orthogonality. Originally in Definition 7.2, orthogonality applied the same predicate represented by the set $\mathbb{P}$ of commands to both the term and coterm sides of a pre-candidate. This is a symmetric sort of operation which treats both sides the same. However, the key insight into building a symmetric candidate is, oddly enough, to be *non*-symmetric in the treatment of the two sides with two different predicates $\mathbb{P}$ and $\mathbb{Q}$. Only at the very end will the symmetry be restored.

Definition 7.14. (Pre-orthogonality)
Since pre-candidates are two-sided objects, we can generalize this operation to *pre-orthogonality* on pre-candidates, we use two different sets of commands ($\mathbb{P}$ and $\mathbb{Q}$) for each side as follows:

$$(\mathbb{A}^+, \mathbb{A}^-)^{(\mathbb{P},\mathbb{Q})} \triangleq (\mathbb{A}^{-\mathbb{P}}, \mathbb{A}^{+\mathbb{Q}})$$

So that orthogonality on pre-candidates, $\mathbb{A}^{\mathbb{P}}$, is the thing same as symmetric pre-orthogonality, $\mathbb{A}^{(\mathbb{P},\mathbb{P})}$.

Since pre-orthogonality is a strict generalization of plain orthogonality not all of the standard properties (namely, double orthogonal introduction and triple orthogonal elimination) necessarily hold. However, many of them carry over to the more general setting, including the important antitonicity and monotonicity relationship with refinement and subtyping.

Proposition 7.15. Pre-orthogonality maintains the following orders:

(i) *Antitonicity*: If $\mathbb{A} \sqsubseteq \mathbb{B}$ then $\mathbb{B}^{(\mathbb{P},\mathbb{Q})} \sqsubseteq \mathbb{A}^{(\mathbb{P},\mathbb{Q})}$.

(ii) *Monotonicity*: If $\mathbb{A} \leq \mathbb{B}$ then $\mathbb{A}^{(\mathbb{P},\mathbb{Q})} \leq \mathbb{B}^{(\mathbb{P},\mathbb{Q})}$.

Furthermore, pre-orthogonality satisfies the following De Morgan properties:

(i) $(\mathbb{A} \sqcup \mathbb{B})^{(\mathbb{P},\mathbb{Q})} = \mathbb{A}^{(\mathbb{P},\mathbb{Q})} \sqcap \mathbb{B}^{(\mathbb{P},\mathbb{Q})}$

(ii) $(\mathbb{A} \sqcap \mathbb{B})^{(\mathbb{P},\mathbb{Q})} \sqsupseteq \mathbb{A}^{(\mathbb{P},\mathbb{Q})} \sqcup \mathbb{B}^{(\mathbb{P},\mathbb{Q})}$

Using the more general notion of pre-orthogonality, we can define a *saturation* operation which ensures that all the necessary (co)terms are in a pre-candidate as dictated by the typing rules, but which may be overaggressive and include *too many* (co)terms with non-sound (*i.e.,* non-normalizing) interactions with one another.

Definition 7.16. (Saturation)
We can generalize the set of strongly normalizing commands ($\perp\!\!\!\perp$) to include potentially strongly normalizing commands as follows:

$$\perp\!\!\!\perp^{R?} \triangleq \perp\!\!\!\perp \cup \{c \mid (\exists c'.c \mapsto c' \in \perp\!\!\!\perp) \wedge (\nexists c.c \mapsto_R c' \notin \perp\!\!\!\perp)\}$$

In addition to commands that must be normalizing *now*, $\perp\!\!\!\perp^{R?}$ also includes commands which (1) are normalizing *after* one R-step and (2) cannot step to a non-normalizing command. Note that by choosing a different R, we can stipulate that only positive, negative or neutral steps are allowed to be used, or any combination thereof.

The *Saturation* operation on any pre-candidate $\mathbb{A}$ is then defined as $\mathbb{A}^s \triangleq \mathbb{A}^{v(\perp\!\!\!\perp^{\mu_d?}, \perp\!\!\!\perp^{\tilde{\mu}_d \varsigma_d?})}$. Expanding the definitions, the saturation of a pre-candidate $\mathbb{A}$ is:

$$\mathbb{A}^{s+} \triangleq \{v \in \mathbb{W} \mid \forall E \in \mathbb{A}, \langle v \parallel E\rangle \in \perp\!\!\!\perp \vee (\langle v \parallel E\rangle \mapsto c \in \perp\!\!\!\perp \wedge (\langle v \parallel E\rangle \mapsto_{\mu_d} c' \implies c' \in \perp\!\!\!\perp))\}$$
$$\mathbb{A}^{s-} \triangleq \{e \in \mathbb{W} \mid \forall V \in \mathbb{A}, \langle V \parallel e\rangle \in \perp\!\!\!\perp \vee (\langle V \parallel e\rangle \mapsto c \in \perp\!\!\!\perp \wedge (\langle V \parallel e\rangle \mapsto_{\tilde{\mu}_d \varsigma_d} c' \implies c' \in \perp\!\!\!\perp))\}$$

The notion of saturation lets us give an alternative definition for *candidate* which has a more generous lower bound. That is, $\mathbb{A}^s$ extends $\mathbb{A}^{v\perp\!\!\!\perp}$, which means that it is easier to show that a (co)term is in $\mathbb{A}^s$ than in $\mathbb{A}^{v\perp\!\!\!\perp}$. The difference is that with saturation, we only need to justify a (co)term by its own behavior, and can ignore the behavior of its partner. For example, it is enough to know that $\langle V \parallel \tilde{\mu}x.c\rangle \mapsto_{\tilde{\mu}_d} c\,[V/x] \in \perp\!\!\!\perp$ even if it's possible that $\langle V \parallel \tilde{\mu}x.c\rangle \mapsto_{\mu_d} c' \notin \perp\!\!\!\perp$ as well.

Definition 7.17. (Symmetric Candidate)
A *symmetric candidate* is any pre-candidate $\mathbb{A}$ such that $\mathbb{A}^s \sqsubseteq \mathbb{A} \sqsubseteq \mathbb{A}^{\perp\!\!\!\perp}$. In other words, a symmetric candidate $\mathbb{A}$ is a pre-candidate satisfying the following three properties:

(i) *Soundness* ($\mathbb{A} \sqsubseteq \mathbb{A}^{\perp\!\!\!\perp}$): For all $v, e \in \mathbb{A}$, the command $\langle v \parallel e\rangle$ is strongly normalizing.

(ii) *Completeness* ($\mathbb{A}^s \sqsubseteq \mathbb{A}$): If v is strongly normalizing and for any $E \in \mathbb{A}$, there is a strongly normalizing c such that $\langle v \parallel E\rangle \mapsto^?_{\mu_d \beta_d} c$, then $v \in \mathbb{A}$. Dually, if e is strongly normalizing and for any $V \in \mathbb{A}$, there is a strongly normalizing c such that $\langle V \parallel e\rangle \mapsto^?_{\tilde{\mu}_d \varsigma_d \beta_d} c$, then $e \in \mathbb{A}$.

As with reducibility candidates, symmetric candidates can also be equivalently rephrased as the fixed point of their completeness operation $-^s$.

Proposition 7.18. $\mathbb{A}$ is a symmetric candidate if and only if $\mathbb{A} = \mathbb{A}^s$.

However, due to the potential for nondeterminacy and the extra leniency of saturation, symmetric candidates are much more difficult to construct. The first step in the construction is to isolate the terms and coterms which still behave deterministically with respect to strong normalization ($\perp\!\!\!\perp$). Terms like $\lambda x.v$ and $V \cdot E$ can only participate in at most one operational step, so they can serve as part of the initial core of (co)values that determine a candidate.

Definition 7.19. (Deterministic Normalization)
The set of commands for whom head reduction deterministically results in only normalizing or only non-normalizing commands is:

$$\perp\!\!\!\perp^d \triangleq \{c \mid \forall c_1, c_2.(c \mapsto c_1, c_2) \implies (c_1 \in \perp\!\!\!\perp \iff c_2 \in \perp\!\!\!\perp)\}$$

The pre-candidate $\mathbb{D}$ of *deterministically normalizing* terms and coterms is then $\mathbb{W}^{\perp\!\!\!\perp^d}$. Expanding the definitions, $\mathbb{D}$ consists of the following set of terms and coterms:

$$\mathbb{D}^+ \triangleq \{v \in \mathbb{W}^+ \mid \forall e \in \mathbb{W}^-, (\langle v \parallel e\rangle \mapsto c_1, c_2) \implies (c_1 \in \perp\!\!\!\perp \iff c_2 \in \perp\!\!\!\perp)\}$$
$$\mathbb{D}^- \triangleq \{e \in \mathbb{W}^- \mid \forall v \in \mathbb{W}^+, (\langle v \parallel e\rangle \mapsto c_1, c_2) \implies (c_1 \in \perp\!\!\!\perp \iff c_2 \in \perp\!\!\!\perp)\}$$

We write $\mathbb{A}^d$ as shorthand for $\mathbb{A} \sqcap \mathbb{D}$.

The positively- and negatively-oriented constructions must be further refined to only include *deterministically normalizing (co)values*. The bigger challenge is to complete the candidate to include all sensible terms and coterms. Just applying a final orthogonal, as in $\mathcal{R}(-)$, is no longer enough, since that doesn't yield a fixed point of saturation. Instead, we can rely on the Knaster-Tarsky fixed point theorem to provide such a solution, since saturation is monotonic with respect to subtyping. Since we are dealing with a complete lattice of subtyping, we are guaranteed both a largest and a smallest such solution with respect to subtyping. Note that this means the *smallest* solution contains the least terms and the *most* coterms, and dually for the *largest*. These definitions are given as follows:

$$Pos_d(C) \triangleq (C, C^{\perp\!\!\!\perp dv})^{\perp\!\!\!\perp dv} \qquad \mathcal{S}_\bot(\mathbb{C}) \triangleq \bigwedge\{\mathbb{A} \mid \mathbb{A} \geq \mathbb{C} \sqcup \mathbb{A}^s\}$$
$$Neg_d(O) \triangleq (O^{\perp\!\!\!\perp dv}, O)^{\perp\!\!\!\perp dv} \qquad \mathcal{S}_\top(\mathbb{C}) \triangleq \bigvee\{\mathbb{A} \mid \mathbb{A} \leq \mathbb{C} \sqcup \mathbb{A}^s\}$$

Proposition 7.20. For any set of values C, set of covalues O, and pre-candidate $\mathbb{A} = \mathbb{A}^{\perp\!\!\!\perp dv}$:

(i) $Pos_d(C) = Pos_d(C)^{\perp\!\!\!\perp dv}$ and $C \subseteq Pos_d(C)^+$,

(ii) $Neg_d(O) = Neg_d(O)^{\perp\!\!\!\perp dv}$ and $O \subseteq Neg_d(O)^-$, and

(iii) there exists at least one symmetric candidate extending $\mathbb{A}$, where $\mathcal{S}_\bot(\mathbb{A})$ is the smallest one and $\mathcal{S}_\top(\mathbb{A})$ is the largest one (with respect to $\leq$).

Proof:
Parts (i) and (ii) are analogous to Proposition 7.9, and part (iii) follows from Proposition 7.18 and the fact that $\mathbb{C} = \mathbb{C}^{\perp\!\!\!\perp dv}$ and $\mathbb{A} = \mathbb{C} \sqcup \mathbb{A}^s$ implies $\mathbb{A} = \mathbb{A}^s$. □

We've alluded to the fact that symmetric candidates "generalize" reducibility candidates. This is well-known to be true in the weak sense that symmetric candidates are powerful enough to capture fundamentally non-deterministic operational semantics, whereas reducibility candidates and biorthogonality only apply to a deterministic system. However, it is also true in the much stronger sense that the notions of symmetric candidate and reducibility candidate are the same for deterministic systems, such that the two constructions are exactly equal.

Proposition 7.21. Every symmetric candidate is a reducibility candidate. Furthermore, for any deterministic discipline d, every reducibility candidate is a symmetric candidate and, for any $\mathbb{C} = \mathbb{C}^{\perp\!\!\!\perp dv}$, $\mathcal{S}_\bot(\mathbb{C}) = \mathcal{S}_\top(\mathbb{C}) = \mathcal{R}(\mathbb{C})$ is the unique reducibility candidate extending $\mathbb{C}$.

7.6. The symmetric lattice of subtyping

Symmetric candidates are more powerful, in that they allow us to model the strong normalization of non-deterministic systems. However, the consequence of this power is that they are much more difficult to pin down; in contrast to the definite construction of reducibility candidates, symmetric candidates are built as *merely* the solution to some recursive equation, and allow for some arbitrary choice of which solution to use. This has a serious impact on the status of intersections and unions of symmetric candidates: we cannot just give a definite description in terms of simpler operations.

Even still, because we are just dealing with a monotonic operation (saturation) on a complete lattice (subtyping), they are still guaranteed to exist.

Proposition 7.22. The set of symmetric candidates forms a lattice with respect to subtyping, *i.e.,* every two symmetric candidates $\mathbb{A}$ and $\mathbb{B}$ has a least upper bound supertype, written $\mathbb{A} \curlyvee \mathbb{B}$, and a greatest lower bound subtype, written $\mathbb{A} \curlywedge \mathbb{B}$, which are both symmetric candidates.

Proof:
Symmetric candidates are exactly the fixed points of the saturation operation $-^s$ on pre-candidates (Proposition 7.18), which is monotonic with respect to the subtyping order of pre-candidates [30]. □

Note that (by Proposition 7.21), if the chosen discipline d is deterministic then

$$\mathbb{A} \curlywedge \mathbb{B} = (\mathbb{A} \wedge \mathbb{B})^{v\perp\!\!\!\perp v\perp\!\!\!\perp} \qquad\qquad \mathbb{A} \curlyvee \mathbb{B} = (\mathbb{A} \vee \mathbb{B})^{v\perp\!\!\!\perp v\perp\!\!\!\perp}$$

Even though we have a less specific definition of intersection and union of symmetric candidates, we can still give a useful bound on how far away from the naïve $\wedge$ and $\vee$ they can be.

Proposition 7.23. For all symmetric candidates $\mathbb{A}$ and $\mathbb{B}$,

$$\mathcal{S}_\bot(\mathbb{A}^{dv} \wedge \mathbb{B}^{dv}) \leq \mathbb{A} \curlywedge \mathbb{B} \qquad\qquad \mathbb{A} \curlyvee \mathbb{B} \geq \mathcal{S}_\top(\mathbb{A}^{dv} \vee \mathbb{B}^{dv})$$

Proof:
First, note that because $\mathbb{A} = \mathbb{A}^s$ and $\mathbb{B} = \mathbb{B}^s$,

$$(\mathbb{A}^{dv} \sqcap \mathbb{B}^{dv}) \sqcup \mathbb{A}^s = \mathbb{A} \sqsubseteq (\mathbb{A}^{dv} \sqcup \mathbb{B}^{dv}) \sqcup \mathbb{A}^s \qquad (\mathbb{A}^{dv} \sqcap \mathbb{B}^{dv}) \sqcup \mathbb{B}^s = \mathbb{B} \sqsubseteq (\mathbb{A}^{dv} \sqcup \mathbb{B}^{dv}) \sqcup \mathbb{B}^s$$
$$(\mathbb{A}^{dv} \wedge \mathbb{B}^{dv}) \sqcup \mathbb{A}^s \leq \mathbb{A} \leq (\mathbb{A}^{dv} \vee \mathbb{B}^{dv}) \sqcup \mathbb{A}^s \qquad (\mathbb{A}^{dv} \wedge \mathbb{B}^{dv}) \sqcup \mathbb{B}^s \leq \mathbb{B} \leq (\mathbb{A}^{dv} \vee \mathbb{B}^{dv}) \sqcup \mathbb{B}^s$$

and so by the definition of $\mathcal{S}_\bot(-)$ and $\mathcal{S}_\top(-)$,

$$\mathcal{S}_\bot(\mathbb{A}^{dv} \wedge \mathbb{B}^{dv}) \leq \mathbb{A} \leq \mathcal{S}_\top(\mathbb{A}^{dv} \vee \mathbb{B}^{dv}) \qquad \mathcal{S}_\bot(\mathbb{A}^{dv} \wedge \mathbb{B}^{dv}) \leq \mathbb{B} \leq \mathcal{S}_\top(\mathbb{A}^{dv} \vee \mathbb{B}^{dv})$$
$$\mathcal{S}_\bot(\mathbb{A}^{dv} \wedge \mathbb{B}^{dv}) \leq \mathbb{A} \wedge \mathbb{B} \qquad \mathbb{A} \vee \mathbb{B} \leq \mathcal{S}_\top(\mathbb{A}^{dv} \vee \mathbb{B}^{dv})$$

Therefore, because both $\mathcal{S}_\top(\mathbb{A}^{dv} \vee \mathbb{B}^{dv})$ and $\mathcal{S}_\top(\mathbb{A}^{dv} \vee \mathbb{B}^{dv})$ are fixed points of $-^s$, and because $\mathbb{A} \curlywedge \mathbb{B}$ is defined to be the *largest* such fixed point subtype of $\mathbb{A} \wedge \mathbb{B}$ and $\mathbb{A} \curlyvee \mathbb{B}$ the *smallest* such fixed point supertype of $\mathbb{A} \vee \mathbb{B}$, it must be that $\mathcal{S}_\bot(\mathbb{A}^{dv} \wedge \mathbb{B}^{dv}) \leq \mathbb{A} \curlywedge \mathbb{B}$ and $\mathbb{A} \curlyvee \mathbb{B} \leq \mathcal{S}_\top(\mathbb{A}^{dv} \vee \mathbb{B}^{dv})$. □

As a result, it must be the case that every deterministic value in $\mathbb{A}^{dv+} \cap \mathbb{B}^{dv+}$ is included in $\mathbb{A} \curlywedge \mathbb{B}$, and every deterministic covalue in $\mathbb{A}^{dv-} \cap \mathbb{B}^{dv-}$ is included in $\mathbb{A} \curlyvee \mathbb{B}$.

7.7. A uniform model of non-deterministic types

We can now alternatively interpret syntactic types as symmetric candidates to handle non-deterministic operational semantics. This is a small variation on the previous model given in Section 7.4. The interpretation of types as symmetric candidates is:

$$\llbracket p \rrbracket_\rho \triangleq \rho(p) \qquad \llbracket A \to B \rrbracket_\rho \triangleq \mathcal{S}_\top(Neg(\llbracket A \rrbracket_\rho \cdot \llbracket B \rrbracket_\rho))$$
$$\llbracket A \cap B \rrbracket_\rho \triangleq \llbracket A \rrbracket_\rho \curlywedge \llbracket B \rrbracket_\rho \qquad \llbracket A \cup B \rrbracket_\rho \triangleq \llbracket A \rrbracket_\rho \curlyvee \llbracket B \rrbracket_\rho$$

And the interpretation of typing judgements are the same as before. Note that, since reducibility candidates are the same thing as symmetric candidates for deterministic disciplines like v and n, the above definition is identical to the one in Section 7.4 in this special case. However, for a non-deterministic discipline like u, the two models are quite different. In particular, this model allows us to prove soundness for non-deterministic substitution disciplines, but the cost of this extra generality is that it only covers the smaller typing system $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d^-$. In this sense, symmetric candidates subsumes reducibility candidates, in the sense that the more informative reducibility candidate model can be fully recovered. But the deterministic model based on reducibility candidates is still vital for reasoning about more expressive type systems like $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d$ when d happens to be deterministic.

Proposition 7.24. (Non-Deterministic Soundness)
For every admissible discipline d (even non-deterministic ones), if the judgement J is derivable in $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d^-$, then $\llbracket J \rrbracket_\rho$ is true for any ρ.

Proposition 7.25. (Non-Deterministic Strong Normalization)
For every admissible discipline d (even non-deterministic ones), every well-typed command, term, and coterm of $\overline{\lambda}\mu\tilde{\mu}\cap\cup_d^-$ is strongly normalizing.

8. Related Work

Intersection types were introduced in the λ-calculus in the late 1970s in the work of Coppo and Dezani [4], Pottinger [5], and Sallé [6], in order to overcome the limitations of the simply typed lambda calculus. In the following years, they became a powerful tool for completely characterizing strong normalization and other kinds of normalization properties at a syntactic level. As a consequence, typability with intersection types is undecidable. The question of inhabitation with intersection types was open for a long time untill it was proven to be undecidable by Urzyczyn [31]. However, the inhabitation problem becomes decidable for non-idempotent intersection types, as shown by Buccarelli *et al.* [32]. In semantics, filter models based on intersection types, by Barendregt *et al.* [33], provide completeness of type assignment. In programming languages, intersection types were promoted by Reynolds [34] and Pierce [35]. Later, call-by-value languages with intersection and union types were developed by Dunfield and Pfenning [36, 37]. In logic, it is well-known that intersection types do not correspond to intuitionistic conjunction, as shown by Hindley [38]. As opposed to proof-theoretic connectives, intersection types have been interpreted as a proof-functional connective *a.k.a*

strong conjunction by Pottinger [5], Lopez-Escobar [39] and Mints [40]. An extensive survey on the theory of intersection types is given by Barendregt *et al.* [41].

Union types came into the picture in the 1990s from MacQueen *et al.*, Pierce [8] and Barbanera *et al.* [9]. They were meant to be a straightforward extension of intersection type systems, since the combination of intersection and union types together preserve the completeness properties regarding strong normalization and filter models. However, it turned out that subject reduction failed in the proposed type system with both intersection and union types, *i.e.,* types are not preserved under reduction. The question of the logical meaning of intersection and union types has spurred on the study of many interesting topics including relevance of intersection and union types by Dezani *et al.* [42], intersection logic with parallel derivations by Ronchi Della Rocca and Roversi [43] and intersection types à la Church by Liquori and Ronchi Della Rocca [44], among others. Recently, intersection and union types à la Church were used with relevant implication and dependent types in order to build a prototype theorem prover with ad hoc—instead of parametric—polymorphism by Honsell *et al.* [11].

In the classical setting, intersection and union types were introduced in sequent calculi based on Curien and Herbelin's [14] (Herbelin [45]) $\overline{\lambda}\mu\tilde{\mu}$ by Dougherty *et al.* [46, 47], and van Bakel [48]. The failure of subject reduction was originally present in the early works and was investigated later on. The system $\mathcal{M}^{\cap}$ of Dougherty *et al.* [47], equipped with subtyping, is closely related to the $\overline{\lambda}\mu\tilde{\mu}\cap\cup_n$ and $\overline{\lambda}\mu\tilde{\mu}\cap\cup_n^-$ systems presented in this paper when extended with subtyping (as shown in Section 4.5). Intersection types were introduced to Parigot's [49] classical natural deduction $\lambda\mu$-calculus by van Bakel et al. [50] and further developed by de Liguoro [51] for the approximation theorem of the $\Lambda\mu$-calculus, a variant of $\lambda\mu$-calculus. A translation of intersection and union types in $\lambda\mu$-calculus was given by Kikuchi and Sakurai [52].

Non-idempotent intersection types in λ-calculus, wherein the type A is distinct from $A \cap A$, were introduced by Bernadet and Lengrand [53] as a suitable way to prove termination directly instead of employing the usual reducibility method. Kesner and Vial [54] then built on this approach in the classical $\lambda\mu$-calculus.

More recent investigations of intersection types were done in the lambda calculus with records by Bessai *et al.* [55] and in a polarized calculus by Tsukada and Nakazawa [56], among others, giving good evidence that intersection and union types have been a perennial source of insight for theory and practice for more than half a century.

9. Conclusion

We showed how duality helps bring out the expressiveness of intersection and union types, and how discipline helps tame them. We looked at the connection between computation (the dynamic behavior of a program) and typing (the static specification of a program) for implicit types like intersections and unions, and how the harmony between these dynamic and static facets gives us a calculus with desirable properties. The harmony is most resounding for the dual call-by-value and call-by-name disciplines, which achieve two perfectly dual typed calculi of intersection and union types, $\overline{\lambda}\mu\tilde{\mu}\cap\cup_v$ and $\overline{\lambda}\mu\tilde{\mu}\cap\cup_n$, which are sound, complete, and type safe. When considering the classical non-deterministic calculus, we end up with two different type systems of interest: the full type system $\overline{\lambda}\mu\tilde{\mu}\cap\cup_u$ which is complete, and the simplified type system $\overline{\lambda}\mu\tilde{\mu}\cap\cup_u^-$ which is sound and type safe.

Evaluation impacts the substitution principle of the calculus. In the same way, evaluation also impacts implicit types like intersections and unions. In the setting of the sequent calculus, the disciplined type system can be expressed in a generic way, which is the same for every evaluation strategy, by only depending on a (co)value restriction determined by that evaluation strategy. This (co)value restriction is essential for scaling up the foundational sequent or lambda calculus to a more full-fledged programming language, in which there are both intersection and union types, as well as the potential for computational effects like recursion and exceptions.

Acknowledgment We would like to thank the reviewers for detailed comments and suggestions.

References

[1] Downen P, Johnson-Freyd P, Ariola ZM. Uniform Strong Normalization for Multi-Discipline Calculi. In: Rewriting Logic and its Applications. 2018 pp. 205–225. doi:10.1007/978-3-319-99840-4_12.

[2] Church A. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 1940. **5**:56–68. doi:10.2307/2266170.

[3] Barendregt H. Introduction to generalized type systems. *Journal of Functional Programming*, 1991. **1**(2):125–154.

[4] Coppo M, Dezani-Ciancaglini M. A new type assignment for λ-terms. *Arch. Math. Log.*, 1978. **19**(1):139–156. doi:10.1007/BF02011875.

[5] Pottinger G. A type assignment for the strongly normalizable λ-terms. In: Seldin JP, Hindley JR (eds.), To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, pp. 561–577. Academic Press, London, 1980.

[6] Sallé P. Une extension de la théorie des types en lambda-calcul. In: Ausiello G, Böhm C (eds.), Fifth International Conference on Automata, Languages and Programming, volume 62 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978 pp. 398–410.

[7] MacQueen DB, Plotkin GD, Sethi R. An Ideal Model for Recursive Polymorphic Types. *Information and Control*, 1986. **71**(1/2):95–130. doi:10.1016/S0019-9958(86)80019-5.

[8] Pierce BC. Programming with Intersection Types, Union Types, and Polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, 1991.

[9] Barbanera F, Dezani-Ciancaglini M, de'Liguoro U. Intersection and Union Types: Syntax and Semantics. *Inf. Comput.*, 1995. **119**(2):202–230. doi:10.1006/inco.1995.1086.

[10] Liquori L, Stolze C. Personal communication, 2019.

[11] Honsell F, Liquori L, Stolze C, Scagnetto I. The Delta-Framework. In: 38th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2018, December 11-13, 2018, Ahmedabad, India. 2018 pp. 37:1–37:21. doi:10.4230/LIPIcs.FSTTCS.2018.37.

[12] Tofte M. Type Inference for Polymorphic References. *Inf. Comput.*, 1990. **89**(1):1–34. doi:10.1016/0890-5401(90)90018-D.

[13] Harper R, Lillibridge M. ML with callcc is unsound. *Message to the TYPES mailing list*, July 1991.

[14] Curien PL, Herbelin H. The duality of computation. In: Odersky M, Wadler P (eds.), Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000. ACM. ISBN 1-58113-202-6, 2000 pp. 233–243. doi:10.1145/351240.351262.

[15] Barbanera F, Berardi S. A Symmetric Lambda Calculus for "Classical" Program Extraction. In: Proceedings of the International Conference on Theoretical Aspects of Computer Software, TACS '94. Springer-Verlag, London, UK, UK. ISBN 3-540-57887-0, 1994 pp. 495–515.

[16] Wadler P. Call-by-value is dual to call-by-name. *SIGPLAN Notices*, 2003. **38**(9):189–201. doi:10.1145/944746.944723.

[17] Downen P, Ariola ZM. A tutorial on computational classical logic and the sequent calculus. *J. Funct. Program.*, 2018. **28**:e3. doi:10.1017/S0956796818000023.

[18] Downen P. Sequent Calculus: A Logic and a Language for Computation and Duality. Ph.D. thesis, University of Oregon, 2017.

[19] Herbelin H, Zimmermann S. An Operational Account of Call-By-Value Minimal and Classical λ-Calculus in "Natural Deduction" Form. In: Curien PL (ed.), Typed Lambda Calculi and Applications: 9th International Conference, TLCA 2009. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-642-02273-9, 2009 pp. 142–156. doi:10.1007/978-3-642-02273-9_12.

[20] Carraro A, Guerrieri G. A Semantical and Operational Account of Call-by-Value Solvability. In: Foundations of Software Science and Computation Structures - 17th International Conference, FOSSACS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings. 2014 pp. 103–118. doi:10.1007/978-3-642-54830-7_7.

[21] Sabry A, Felleisen M. Reasoning About Programs in Continuation-Passing Style. *Lisp and Symbolic Computation*, 1993. **6**(3-4):289–360. doi:10.1007/BF01019462.

[22] Andreoli J. Logic Programming with Focusing Proofs in Linear Logic. *J. Log. Comput.*, 1992. **2**(3):297–347. doi:10.1093/logcom/2.3.297.

[23] Laurent O. Étude de la polarisation en logique. Thèse de doctorat, Université de la Méditerranée - Aix-Marseille II, 2002.

[24] Girard J. A New Constructive Logic: Classical Logic. *Mathematical Structures in Computer Science*, 1991. **1**(3):255–296. doi:10.1017/S0960129500001328.

[25] Downen P, Ariola ZM. The Duality of Construction. In: Shao Z (ed.), 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*. Springer. ISBN 978-3-642-54832-1, 2014 pp. 249–269. doi:10.1007/978-3-642-54833-8_14.

[26] Giannini P, Ronchi Della Rocca S. Characterization of typings in polymorphic type discipline. In: Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988. IEEE Computer Society. ISBN 0-8186-0853-6, 1988 pp. 61–70. doi:10.1109/LICS.1988.5101.

[27] Urzyczyn P. Type Reconstruction in F_ω. *Mathematical Structures in Computer Science*, 1997. **7**(4):329–358. doi:10.1017/S0960129597002302.

[28] Milner R. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.*, 1978. **17**(3):348–375. doi:10.1016/0022-0000(78)90014-4.

[29] Wright AK, Felleisen M. A Syntactic Approach to Type Soundness. *Inf. Comput.*, 1994. **115**(1):38–94. doi:10.1006/inco.1994.1093.

[30] Johnson-Freyd P. Properties of Sequent-Calculus-Based Languages. Ph.D. thesis, University of Oregon, 2018.

[31] Urzyczyn P. The Emptiness Problem for Intersection Types. In: Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994. IEEE Computer Society. ISBN 0-8186-6310-3, 1994 pp. 300–309. doi:10.1109/LICS.1994.316059.

[32] Bucciarelli A, Kesner D, Ronchi Della Rocca S. Inhabitation for Non-idempotent Intersection Types. *Logical Methods in Computer Science*, 2018. **14**(3). doi:10.23638/LMCS-14(3:7)2018.

[33] Barendregt HP, Coppo M, Dezani-Ciancaglini M. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 1983. **48**(4):931–940 (1984).

[34] Reynolds JC. Design of the Programming Language Forsythe. Report CMU–CS–96–146, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1996.

[35] Pierce BC. Intersection Types and Bounded Polymorphism. In: Bezem M, Groote JF (eds.), Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings, volume 664 of *Lecture Notes in Computer Science*. Springer. ISBN 3-540-56517-5, 1993 pp. 346–360. doi:10.1007/BFb0037117.

[36] Dunfield J, Pfenning F. Type Assignment for Intersections and Unions in Call-by-Value Languages. In: Gordon AD (ed.), Foundations of Software Science and Computational Structures, 6th International Conference, FOSSACS 2003 Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings, volume 2620 of *Lecture Notes in Computer Science*. Springer. ISBN 3-540-00897-7, 2003 pp. 250–266. doi:10.1007/3-540-36576-1_16.

[37] Dunfield J. Elaborating intersection and union types. In: Thiemann P, Findler RB (eds.), ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012. ACM. ISBN 978-1-4503-1054-3, 2012 pp. 17–28. doi:10.1145/2364527.2364534.

[38] Hindley JR. Coppo-Dezani Types do not Correspond to Propositional Logic. *Theor. Comput. Sci.*, 1984. **28**:235–236. doi:10.1016/0304-3975(83)90074-9.

[39] Lopez-Escobar EGK. Proof functional connectives. In: Di Prisco CA (ed.), Methods in Mathematical Logic, volume 1130 of *Lecture Notes in Mathematics*. Springer-Verlag, 1985 p. 208–221.

[40] Mints G. The Completeness of Provable Realizability. *Notre Dame Journal of Formal Logic*, 1989. **30**(3):420–441. doi:10.1305/ndjfl/1093635158.

[41] Barendregt HP, Dekkers W, Statman R. Lambda Calculus with Types. Perspectives in logic. Cambridge University Press, 2013. ISBN 978-0-521-76614-2. URL `http://www.cambridge.org/de/academic/subjects/mathematics/logic-categories-and-sets/lambda-calculus-types`.

[42] Dezani-Ciancaglini M, Ghilezan S, Venneri B. The "Relevance" of Intersection and Union Types. *Notre Dame Journal of Formal Logic*, 1997. **38**(2):246–269. doi:10.1305/ndjfl/1039724889.

[43] Ronchi Della Rocca S, Roversi L. Intersection Logic. In: Fribourg L (ed.), Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, volume 2142 of *Lecture Notes in Computer Science*. Springer, 2001 pp. 414–428. doi:10.1007/3-540-44802-0_29.

[44] Liquori L, Ronchi Della Rocca S. Intersection-types à la Church. *Inf. Comput.*, 2007. **205**(9):1371–1386. doi:10.1016/j.ic.2007.03.005.

[45] Herbelin H. C'est maitenant q'on calcul, au coer de la dualité. Habilitation. Habilitation à diriger les reserches, Université Paris 11, 2005. URL `http://pauillac.inria.fr/~herbelin/habilitation/memoire.ps`.

[46] Dougherty DJ, Ghilezan S, Lescanne P. Characterizing strong normalization in a language with control operators. In: Moggi E, Warren DS (eds.), Proceedings of the 6th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 24-26 August 2004, Verona, Italy. ACM. ISBN 1-58113-819-9, 2004 pp. 155–166. doi:10.1145/1013963.1013982.

[47] Dougherty DJ, Ghilezan S, Lescanne P. Characterizing strong normalization in the Curien-Herbelin symmetric lambda calculus: Extending the Coppo-Dezani heritage. *Theor. Comput. Sci.*, 2008. **398**(1-3):114–128. doi:10.1016/j.tcs.2008.01.022.

[48] van Bakel S. Completeness and partial soundness results for intersection and union typing for $\overline{\lambda}\mu\tilde{\mu}$. *Ann. Pure Appl. Logic*, 2010. **161**(11):1400–1430. doi:10.1016/j.apal.2010.04.010.

[49] Parigot M. Strong Normalization for Second Order Classical Natural Deduction. In: Proceedings of the Eighth Annual Symposium on Logic in Computer Science (LICS '93), Montreal, Canada, June 19-23, 1993. IEEE Computer Society. ISBN 0-8186-3140-6, 1993 pp. 39–46. doi:10.1109/LICS.1993.287602.

[50] van Bakel S, Barbanera F, de'Liguoro U. Intersection Types for the lambda-mu Calculus. *Logical Methods in Computer Science*, 2018. **14**(1). doi:10.23638/LMCS-14(1:2)2018.

[51] de'Liguoro U. The approximation theorem for the $\Lambda\mu$-calculus. *Mathematical Structures in Computer Science*, 2017. **27**(5):560–580. doi:10.1017/S0960129515000286.

[52] Kikuchi K, Sakurai T. A Translation of Intersection and Union Types for the $\lambda\mu$-Calculus. In: Garrigue J (ed.), Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings, volume 8858 of *Lecture Notes in Computer Science*. Springer. ISBN 978-3-319-12735-4, 2014 pp. 120–139. doi:10.1007/978-3-319-12736-1_7.

[53] Bernadet A, Lengrand S. Non-idempotent intersection types and strong normalisation. *Logical Methods in Computer Science*, 2013. **9**(4). doi:10.2168/LMCS-9(4:3)2013.

[54] Kesner D, Vial P. Types as Resources for Classical Natural Deduction. Submitted, University Paris Diderot, 2018.

[55] Bessai J, Chen T, Dudenhefner A, Düdder B, de'Liguoro U, Rehof J. Mixin Composition Synthesis based on Intersection Types. *CoRR*, 2017. **abs/1712.06906**. `1712.06906`.

[56] Tsukada T, Nakazawa K. Intersection and Union Type Assignment and Polarised $\overline{\lambda}\mu\tilde{\mu}$. Draft, University of Tokyo, 2018.

A. Binding and Substitution

The standard definition of free (FV) and bound (BV) variables and covariables for $\overline{\lambda}\mu\tilde{\mu}$ are as follows:

$$
\begin{aligned}
FV(\langle v \parallel e \rangle) &= FV(v) \cup FV(e) \\
FV(x) &= \{x\} & FV(\alpha) &= \{\alpha\} \\
FV(\mu\alpha.c) &= FV(c) - \{\alpha\} & FV(\tilde{\mu}x.c) &= FV(c) - \{x\} \\
FV(\lambda x.v) &= FV(v) - \{x\} & FV(v \cdot e) &= FV(v) \cup FV(e)
\end{aligned}
$$

$$
\begin{aligned}
BV(\langle v \parallel e \rangle) &= BV(v) \cup BV(e) \\
BV(x) &= \{\} & BV(\alpha) &= \{\} \\
BV(\mu\alpha.c) &= BV(c) \cup \{\alpha\} & BV(\tilde{\mu}x.c) &= BV(c) \cup \{x\} \\
BV(\lambda x.v) &= BV(v) \cup \{x\} & BV(v \cdot e) &= BV(v) \cup BV(e)
\end{aligned}
$$

The standard definition of capture-avoiding substitution of terms for variables and coterms for covariables is:

$$
\begin{aligned}
\langle v' \parallel e' \rangle [v/x] &= \langle v' [v/x] \parallel e' [v/x] \rangle \\
x [v/x] &= v \\
y [v/x] &= y && (\text{if } x \neq y) \\
(\mu\beta.c) [v/x] &= \mu\beta.(c [v/x]) && (\text{if } \beta \notin FV(v)) \\
(\lambda y.v') [v/x] &= \lambda y.(v' [v/x]) && (\text{if } y \notin FV(v), x \neq y) \\
\beta [v/x] &= \beta \\
(\tilde{\mu}y.c) [v/x] &= \tilde{\mu}y.(c [v/x]) && (\text{if } y \notin FV(v), x \neq y) \\
(v' \cdot e') [v/x] &= (v' [v/x]) \cdot (e' [v/x])
\end{aligned}
$$

$$
\begin{aligned}
\langle v' \parallel e' \rangle [e/\alpha] &= \langle v' [e/\alpha] \parallel e' [e/\alpha] \rangle \\
x [e/\alpha] &= x \\
(\mu\beta.c) [e/\alpha] &= \mu\beta.(c [e/\alpha]) && (\text{if } \beta \notin FV(e), \alpha \neq \beta) \\
(\lambda y.v') [e/\alpha] &= \lambda y.(v' [e/x]) && (\text{if } y \notin FV(e)) \\
\alpha [e/\alpha] &= e \\
\beta [e/\alpha] &= \beta && (\text{if } \alpha \neq \beta) \\
(\tilde{\mu}y.c) [e/\alpha] &= \tilde{\mu}y.(c [e/\alpha]) && (\text{if } y \notin FV(e)) \\
(v' \cdot e') [e/\alpha] &= (v' [e/\alpha]) \cdot (e' [e/\alpha])
\end{aligned}
$$

The standard definition of α-equivalence in the $\overline{\lambda}\mu\tilde{\mu}$-calculus includes the following axioms:

$$
\begin{aligned}
\mu\alpha.c &= \mu\beta.(c [\beta/\alpha]) && (\text{if } \beta \notin FV(c)) \\
\tilde{\mu}x.c &= \tilde{\mu}y.(c [y/x]) && (\text{if } y \notin FV(c)) \\
\lambda x.v &= \lambda y.(v [y/x]) && (\text{if } y \notin FV(v))
\end{aligned}
$$

along with rules for compatibility (*i.e.,* for any context C, if $v =_\alpha v'$ then $C[v] =_\alpha C[v']$, and similar for coterms and commands).