The Duality of Subtyping

Bruno C. d. S. Oliveira
The University of Hong Kong, China
bruno@cs.hku.hk

Cui Shaobo
University of California San Diego, CA, USA
cuishaobo@gmail.com

Baber Rehman
The University of Hong Kong, China
brehman@cs.hku.hk

Abstract

Subtyping is a concept frequently encountered in many programming languages and calculi. Various forms of subtyping exist for different type system features, including intersection types, union types or bounded quantification. Normally these features are designed independently of each other, without exploiting obvious similarities (or dualities) between features.

This paper proposes a novel methodology for designing subtyping relations that exploits duality between features. At the core of our methodology is a generalization of subtyping relations, which we call *Duotyping*. Duotyping is parameterized by the mode of the relation. One of these modes is the usual subtyping, while another mode is supertyping (the dual of subtyping). Using the mode it is possible to generalize the usual rules of subtyping to account not only for the intended behaviour of one particular language construct, but also of its dual. Duotyping brings multiple benefits, including: shorter specifications and implementations, dual features that come essentially for free, as well as new proof techniques for various properties of subtyping. To evaluate a design based on Duotyping against traditional designs, we formalized various calculi with common OOP features (including union types, intersection types and bounded quantification) in Coq in both styles. Our results show that the metatheory when using Duotyping does not come at a significant cost: the metatheory with Duotyping has similar complexity and size compared to the metatheory for traditional designs. However, we discover new features as duals to well-known features. Furthermore, we also show that Duotyping can significantly simplify transitivity proofs for many of the calculi studied by us.

2012 ACM Subject Classification Software and its engineering → Object oriented languages

Keywords and phrases DuoTyping, OOP, Duality, Subtyping, Supertyping

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.29

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at
`https://doi.org/10.4230/DARTS.6.2.8`.
`https://github.com/baberrehman/coq-duotyping`

Funding Funded by Hong Kong Research Grant Council projects number 17210617 and 17209519.

Acknowledgements We thank the anonymous reviewers for their helpful comments.

1 Introduction

Subtyping is a concept frequently encountered in many programming languages and calculi. It is also a pervasive and fundamental feature in Object-Oriented Programming (OOP). Various forms of subtyping exist for different type system features, including *intersection types* [6], *union types* [6] or *bounded quantification* [15]. Modern OOP languages such as Scala [34], Ceylon [29], Flow [19] or TypeScript [12] all support the aforementioned type system features.

As programming languages evolve, new features are added. This requires that subtyping for these new features is developed and also integrated with existing features. However, the design and implementation of subtyping for new features is quite often non-trivial. There are several, well-documented issues in the literature. These include finding algorithmic forms for subtyping (for instance doing transitivity elimination) [43] or proving metatheoretical properties such as transitivity or narrowing [1]. Such issues occur, for instance, in some of the latest developments for OOP languages, such as the DOT calculi (which model the essence of Scala) [3]. One possible way to reduce the non-trivial amount of work needed to develop new features, would be if two related features could be develop at once with a coherent design. This paper explores a new methodology that enables such benefits.

Normally programming language features are designed independently of each other. However there are features that are closely related to each other, and can be viewed as *dual features*. Various programming language features are known to be dual in programming language theory. For instance sum and product types are well-known to be duals [14]. Similarly universal and existential quantification are dual concepts as well [9]. Moreover duality is a key concept in category theory [30] and many abstractions widely used in functional programming (such as Monads and CoMonads [44]) are also known to be duals.

In OOP type systems dual features are also common. For instance all OOP languages contain a *top type* (called `Object` in Java or `Any` in Scala), which is the supertype of all types. Many OOP languages also contain a *bottom type*, which is a subtype of all types. Top and bottom types can be viewed as dual features, mirroring the functionality of each other. *Intersection* and *union* types are another example of dual features. The intersection of two types `A` and `B` can be used to type a value that implements *both* `A` *and* `B`. The union of two types `A` and `B` can be used to type a value that implements *either* `A` *or* `B`.

Duality in OOP and subtyping is often only informally observed by humans. For instance, by simply understanding the behaviour of the features and observing their complementary roles, as we just did in the previous paragraph. At best duality is more precisely observed by looking at the rules for the language constructs and their duals and observing a certain symmetry between those rules. However existing formalisms and language designs for type systems and subtyping relations do not directly incorporate duality. Unfortunately this means that an opportunity to exploit obvious similarities between features is lost.

This paper proposes a novel methodology for designing subtyping relations that exploits duality between features directly in the formalism. At the core of our methodology is a generalization of subtyping relations, which we call *Duotyping*. Duotyping is parameterized by the mode of the relation. One of these modes is the usual subtyping, while another mode is supertyping (the dual of subtyping). Using the mode it is possible to generalize the usual rules of subtyping to account not only for the intended behaviour of one particular language construct, but also of its dual. This means that the behaviour of the language construct and its dual is modelled by a single, common set of rules. In turn this ensures that the behaviour of the two features is modelled consistently. Moreover it also enables various theorems/properties of subtyping to be generalized to account for the dual features. Therefore, Duotyping offers similar benefits to the how duality is exploited in category theory. More concretely, Duotyping brings multiple benefits for the design of subtyping relations, which are discussed next.

Shorter specifications. When duality is exploited in specifications of subtyping it leads to shorter specifications because rules for dual features are shared. This also ensures a consistent design of the rules between the dual features directly in the formalism. Such consistency is not enforceable in traditional formulations of subtyping where the rules are designed separately, and thus their design is completely unconstrained with respect to the dual feature. A concrete example that illustrates shorter specifications is a traditional subtyping relation with top, bottom, union and intersection types, which would normally have 8 subtyping rules for those constructs. In a design with Duotyping we only need 5 subtyping rules. Basically we need only *half of the rules* (4 in this case) to model the feature-specific rules, plus an additional *duality rule* which is generic (and plays a similar role to *reflexivity* and *transitivity*).

"Buy" one feature get one feature for free! Duality can lead to the discovery of new features. While top and bottom types, or intersection and union types are well-known in the literature (and understood to be duals), other features in languages with subtyping do not have a known dual feature in the literature. This is partly because, when a language designer employs traditional formulations of subtyping, he/she is often only interested in the design of a feature (but not necessarily of its dual). Even for the case of union and intersection types, intersection types were developed first and the development of union types occurred years later. Because the dual feature is often also useful, the traditional way to design subtyping rules represents a loss of opportunity to get another language feature essentially for free.

One well-known example of a language feature that has been widely exploited in the literature, but its dual feature has received much less attention is *bounded quantification* [17]. Bounded quantification allows type variables to be defined with *upper bounds*. However *lower bounds* are also useful. One can think of universal quantification with lower bounds as a dual to universal quantification with upper bounds. The essence of (upper) bounded quantification is captured by the well-known $F_{<:}$ calculus [17]. However, as far as we know, there is no design that extends $F_{<:}$ with lower bounded quantification in the literature. Applying a Duotyping design to $F_{<:}$ gives us, naturally, the two features at once (lower and upper bounded quantification), as illustrated in our Section 4. Such generalization of bounded quantification is related to the recent form of universal quantification with type bounds employed in Scala and the DOT family of calculi [3]. However, while Scala's type bounds are more expressive than what we propose, they are also much more complex and are in fact one of the key complications in the type systems of languages like Scala. Most DOT calculi require a built-in transitivity rule in subtyping because it is not known how to eliminate transitivity. In contrast, the generalization of $F_{<:}$ proposed by us has a formulation of subtyping where transitivity can be proved as a separate lemma.

New proof techniques. Designs of subtyping with duality also enable new proof techniques that exploit such duality. For instance there are various theorems that can be stated for both a feature and its dual, instead of having separate theorems for both. Some of the properties of union and intersection types are examples of this. Moreover, Duotyping also enables new proof techniques to prove traditionally hard theorems such as transitivity. Surprisingly to us, for the vast majority of the calculi that we have applied Duotyping to, transitivity proofs have been considerably simpler than their corresponding traditional formulations due to the use of Duotyping!

Shorter implementations. Finally Duotyping also enables for shorter implementations. The benefits of shorter implementations are similar and follow from the benefits of shorter specifications. However there is a complicating factor when moving from a *relational specification* into an implementation: the *duality rule* is non-algorithmic. This is akin to what happens with transitivity, which is often also used in declarative formulations of subtyping.

Eliminating transitivity to obtain an algorithmic system can often be a non-trivial challenge (as illustrated, for instance, by the DOT family of calculi [3]). However, we show that there is a simple and generally applicable technique that can be used to move from a declarative formulation of Duotyping into an algorithmic version. This contrasts with transitivity, for which there is not a generally applicable transitivity elimination technique.

To evaluate a design based on Duotyping against traditional designs of subtyping, we formalized various calculi with common OOP features (including union types, intersection types and bounded quantification) in Coq in both styles. Our results show that the metatheory when using Duotyping has similar complexity and size compared to traditional designs. However, the Duotyping formalizations come with more features (for instance lower-bounded quantification) that dualize other well-known features (upper-bounded quantification). Finally, we also show that Duotyping can significantly simplify transitivity proofs for many of the calculi studied by us.

In summary, the contributions of this paper are:

- **Duotyping:** A new methodology for the design of subtyping relations exploiting duality.
- **A case study on Duotyping:** A comprehensive study of various existing type systems and features, which were redesigned to employ the Duotyping methodology. Our results show that in most systems the size of the metatheory without duality and with duality is comparable, while often transitivity proofs become simpler when employing duality.
- **$F_{<:}$ with lower bounded quantification:** We propose a new generalization of System $F_{<:}$, called $F_{k\diamond}$, which allows not only type variables to be quantified with upper bounds and lower bounds as well. While this system is weaker than Scala/DOT's type bounds, it nonetheless allows for simple transitivity proofs (which have been a significant challenge in calculi with type bounds [40]).
- **Mechanization in Coq:** All the systems in our case study have been formalized in the Coq theorem prover [8].

2 Overview

This section gives an overview of Duotyping. We show how to design subtyping relations employing Duotyping, and discuss the advantages of a design with Duotyping instead of a traditional subtyping formulation in more detail.

2.1 Subtyping with union and intersection types

To motivate the design of Duotyping relations we first consider a traditional subtyping relation with union and intersection types, as well as top and bottom types. We choose a system with union and intersection types because these features are nowadays common in various OOP languages, including Scala [34], TypeScript [12], Ceylon [29] or Flow [19]. Therefore union and intersection types are of practical interest. Furthermore union and intersection types are simple, intuitive and good for showing duality between concepts.

The types used for the subtyping relation include the top type $\top$, the bottom type $\bot$, integer types Int, function types $A \to B$, intersection types $A \wedge B$ and union types $A \vee B$:

Types	A, B	$::=$	$\top \mid \bot \mid \mathsf{Int} \mid A \to B \mid A \wedge B \mid A \vee B$

$\boxed{A <: B}$ *(Traditional Subtyping)*

$$\frac{}{A <: \top}\ \textsc{ts-top} \qquad \frac{}{\bot <: A}\ \textsc{ts-btm} \qquad \frac{}{\mathsf{Int} <: \mathsf{Int}}\ \textsc{ts-int} \qquad \frac{B_1 <: A_1 \qquad A_2 <: B_2}{A_1 \to A_2 <: B_1 \to B_2}\ \textsc{ts-arrow}$$

$$\frac{A <: A_1 \qquad A <: A_2}{A <: A_1 \wedge A_2}\ \textsc{ts-andA} \qquad \frac{A_1 <: A}{A_1 \wedge A_2 <: A}\ \textsc{ts-andB} \qquad \frac{A_2 <: A}{A_1 \wedge A_2 <: A}\ \textsc{ts-andC}$$

$$\frac{A_1 <: A \qquad A_2 <: A}{A_1 \vee A_2 <: A}\ \textsc{ts-orA} \qquad \frac{A <: A_1}{A <: A_1 \vee A_2}\ \textsc{ts-orB} \qquad \frac{A <: A_2}{A <: A_1 \vee A_2}\ \textsc{ts-orC}$$

Figure 1 Subtyping for union and intersection types.

Traditional Subtyping. A simple subtyping relation accounting for union and intersection types is given in Figure 1. Rule TS-TOP defines that every type is a subtype of $\top$, and Rule TS-BTM states that every type is a supertype of $\bot$. Rule TS-INT is for integers, and states that Int is a subtype of itself. Rule TS-ARROW is the traditional subtyping rule for function types. Rules TS-ANDA, TS-ANDB, and TS-ANDC are subtyping rules for intersection types. Rules TS-ORA, TS-ORB, and TS-ORC are subtyping rules for union types. The rules that we employ here are quite common for systems with union and intersection types. For instance they are the same rules used in various DOT-calculi [3] (which model the essence of Scala). For simplicity we do not account for distributivity rules, which also appear in some type systems and calculi [7, 11, 47].

2.2 Subtyping Specifications using Duotyping

In the subtyping relation presented in Figure 1, it is quite obvious that many rules look alike. Some rules are essentially a "mirror image" of other rules. The rules for top and bottom types are an example of this. Another example are the rules TS-ANDB and TS-ORB. Although informally humans can easily observe the similarity between many of the rules, this similarity/duality is not expressed directly in the formalism. For example, there is nothing preventing us from designing rules that are not duals. Duotyping aims at capturing duality in the rules themselves, and expressing duality as part of the formalism, rather than just leaving duality informally observable by humans. This can prevent, for instance, designing rules for dual concepts that do not really dualize. Therefore Duotyping can enforce consistency of dual rule designs.

To illustrate how Duotyping rules are designed and relate to the traditional subtyping rules, lets refactor the traditional rules in a few basic steps. Firstly, lets assume that we have a second relation $A :> B$ that captures the *supertyping* between a type A and B. Supertyping is nothing but the subtyping relation with its arguments flipped. So, the rules of supertyping could be simply obtained by taking all the rules in Figure 1 and deriving corresponding rules where all the arguments are flipped around. We skip that boring definition here. With both supertyping and subtyping, the top and bottom rules can be presented as follows:

$$\frac{}{A <: \top}\ \textsc{ts-top} \qquad \frac{}{A :> \bot}\ \textsc{tsp-btm}$$

Similarly the rules rule TS-ANDB and rule TS-ORB, can be presented as:

$$\frac{A_1 \ <:\ A}{A_1 \wedge A_2 \ <:\ A}\ \textsc{ts-andb} \qquad \frac{A_1 \ :>\ A}{A_1 \vee A_2 \ :>\ A}\ \textsc{tsp-orb}$$

This simple refactoring shows that the only difference between dual rules is the relation itself, and the (dual) language constructs. Apart from that everything else is the same.

Duotyping. With Duotyping we can provide a single unified rule, which captures the two distinct subtyping rules, instead. The Duotyping relation is parameterized by a mode $\Diamond$:

$$\text{Mode} \quad \Diamond \quad ::= \quad <: \mid :>$$

which can be subtyping (<:) or supertyping (:>). Thus the Duotyping relation is of the form:

$$A \Diamond B$$

The mode $\Diamond$ is a (third) *parameter* of the relation (besides A and B). With this mode in place, we can readily capture the two refactored rules for supertyping of bottom types and subtyping of top types as two Duotyping rules. However this still requires us to write two distinct rules. To unify those rules into a single one, we introduce a function $\rceil\Diamond\lceil$ that chooses the right bound depending on the mode being used:

$$\rceil <: \lceil \ = \ \top$$
$$\rceil :> \lceil \ = \ \bot$$

If the mode is subtyping the upper bound of the relation is the top type, otherwise it is the bottom type. With $\rceil\Diamond\lceil$ we can then write a single unified rule that captures the upper bounds of subtyping and supertyping, and which generalizes both rule TS-TOP and rule TSP-BTM:

$$\frac{}{A \Diamond \rceil\Diamond\lceil}\ \textsc{gds-topbtm}$$

The Duality rule. The Duotyping rule above captures the 2 rules that were refactored above. However there are 4 rules in total for top and bottom types (two for subtyping and two for supertyping). The two missing rules are:

$$\frac{}{\bot \ <:\ A}\ \textsc{ts-btm} \qquad \frac{}{\top \ :>\ A}\ \textsc{tsp-top}$$

To capture these missing rules, the Duotyping relation includes a special duality rule:

$$\frac{B \overline{\Diamond} A}{A \Diamond B}\ \textsc{gds-dual}$$

which simply inverts the mode and flips the arguments of the relation. The definition of $\overline{\Diamond}$ is, unsurprisingly:

$$\overline{<:} \ = :>$$
$$\overline{:>} \ = <:$$

With the duality rule it is clear that the two missing rules are now derivable from the Duotyping rule for bounds and the duality rule. In essence this is the overall idea of the design of Duotyping rules.

$\boxed{A \lozenge B}$ *(Declarative* Duotyping*)*

$$\frac{}{A \lozenge \rceil \lozenge \lceil}\ \textsc{gds-topbtm} \qquad \frac{}{\mathsf{Int} \lozenge \mathsf{Int}}\ \textsc{gds-int} \qquad \frac{A_1 \overline{\lozenge} A_2 \qquad B_1 \lozenge B_2}{A_1 \to B_1 \lozenge A_2 \to B_2}\ \textsc{gds-arrow}$$

$$\frac{A \lozenge C}{(A \lozenge_? B) \lozenge C}\ \textsc{gds-Left} \qquad \frac{B \lozenge C}{(A \lozenge_? B) \lozenge C}\ \textsc{gds-Right} \qquad \frac{A \lozenge B \qquad A \lozenge C}{A \lozenge (B \lozenge_? C)}\ \textsc{gds-both}$$

$$\frac{B \overline{\lozenge} A}{A \lozenge B}\ \textsc{gds-dual}$$

Figure 2 The Duotyping relation for a calculus with union and intersection types.

Complete set of rules. Figure 2 shows the complete version of declarative Duotyping rules for a system with union and intersection types. Rule GDS-TOPBTM defines the rule bounds (which generalizes the rules for top and bottom types). Rule GDS-INT is a simple rule for integers. Int is subtype and supertype of Int. Rule GDS-ARROW is an interesting case. In the first premise $A \overline{\lozenge} B$ we invert the mode instead of flipping the arguments of the relation, as done in rule TS-ARROW. One side-effect of this change is that it keeps the rule fully *covariant*, which contrasts with subtyping relations where for arrow types we need contravariance for subtyping of the inputs. This apparently innocent change has important consequences and plays a fundamental role to simplify transitivity proofs as we shall see in Section 2.5.

Rules GDS-LEFT, GDS-RIGHT, and GDS-BOTH each generalize two rules in the traditional formulation of subtyping. Rule GDS-LEFT generalizes rules TS-ANDB and TS-ORB. Rule GDS-RIGHT generalizes rules TS-ANDC and TS-ORC. Rule GDS-BOTH generalizes rules TS-ANDA and TS-ORA. In the three rules an operation $A \lozenge_? B$ is used:

$$\begin{aligned} A <:_? B &= A \wedge B \\ A :>_? B &= A \vee B \end{aligned}$$

This operation is used to choose between intersection or union types depending on the mode. If the Duotyping mode is subtyping then we get a rule for intersection types, otherwise we get a dual rule for union types.

Uniform and dual rules. In the context of Duotyping it is useful to distinguish between two different kinds of rules: uniform rules and dual rules.

Uniform rules are those that are essentially the same for supertyping and subtyping. Rules GDS-INT and GDS-ARROW are uniform rules. In those rules the arguments of the relation are exactly the same no matter which mode is being used (subtyping or supertyping).

Dual rules are those that employ dual constructs, like the rules for top and bottom or the rules for union and intersections. Rules GDS-TOPBTM, GDS-LEFT, GDS-RIGHT, and GDS-BOTH are dual rules. The interesting point in these rules is that they use different (dual) constructs depending on the mode. For example, when instantiated with subtyping and supertyping, respectively, the rule GDS-TOPBTM results in:

$$\frac{}{A <: \top} \qquad \frac{}{A :> \bot}$$

2.3 Implementations using Duotyping

Figure 2 showed the declarative Duotyping rules for a calculus with union and intersection types. All the rules are syntax directed, except for the duality rule (rule GDS-DUAL). This rule flips the mode and arguments to generate a formulation using the dual mode: i.e. it flips subtyping to provide the equivalent supertyping formulation and vice versa. A benefit of using a formulation with the duality rule is that it enables a short specification of Duotyping. Unfortunately the duality rule is *not algorithmic*, because the duality rule can always be applied indefinitely. In other words naively translating the rules into an program would easily result in a non-terminating procedure. Therefore to obtain an algorithmic formulation some additional work is needed.

Fortunately, for declarative formulations of Duotyping, there is a simple technique that can be used to obtain an algorithmic formulation. A key observation is that Duotyping only needs to be flipped (with the duality rule) *at most one time*. Flipping the relation two or more times simply gets us back to the starting point. To capture this idea we can use a (boolean) flag that keeps track of whether the procedure has already employed the duality rule or not.

To make such an idea concrete, Figure 3 shows Haskell code that implements a procedure `duo` for determining Duotyping for two types. The code is based on the rules in Figure 2, but it uses a boolean flag to prevent the dual rule (the second to last case in `duo`) from being applied indefinitely. The boolean is true in the initial call or recursive calls to structurally smaller arguments. If the algorithm fails for the first five cases (which are basically a direct translation of the rules GDS-TOPBTM, GDS-INT, GDS-ARROW, GDS-LEFT, and GDS-RIGHT), then the algorithm simply flips the boolean flag, mode and arguments to run over a dual formulation. This is the second to last line of the algorithm.

For example, if the algorithm is called with the mode set to subtyping and it is not able to find any matching case with the first 5 rules, then it flips the boolean flag to `False`, subtyping to supertyping and the arguments to check the equivalent supertyping formulation. If again it fails to find a matching rule, `False` will be returned and the algorithm will terminate. This illustrates that it is enough to flip the boolean flag once to exploit Duotyping. In all our Coq formulations of Duotyping we have developed an alternative algorithmic formulation of Duotyping which uses an extra boolean flag and is shown to be sound and complete to the declarative formulations with the duality rule. In short there is an easy, general and provably *sound* and *complete* way to implement algorithms based on the idea of Duotyping, while at the same time retaining the benefits of reuse of the logic for rules for dual constructs.

2.4 Discovering new features

Duotyping can provide interesting extra features essentially for free. For example, the hallmark feature of the well-known $F_{<:}$ calculus (a polymorphic calculus with subtyping) [15] is *bounded quantification*, which is a feature used in most modern OOP languages (such as Scala or Java). In $F_{<:}$, bounded quantification allows type variables to be defined with *upper bounds*. For example, the following Scala program illustrates the use of such upper bounds:

```scala
class Person {
  def name: String = "person"
}

class Student extends Person {
  override def name: String = "student"
  def id: String = "id"
}
```

```
data Op = And | Or
data Typ = TInt | TArrow Typ Typ | TOp Op Typ Typ | TBot | TTop
data Mode = Sub | Sup

duo :: Bool -> Mode -> Typ -> Typ -> Bool
duo f m TInt TInt                       = True
duo f m _ b | b == mode_to_sub m        = True
duo f m (TArrow a b) (TArrow c d)       =
   duo True (flip m) a c && duo True m b d
duo f m (TOp op a b) c | choose m == op  = duo True m a c || duo True m b c
duo f m a (TOp op b c) | choose m == op  = duo True m a b && duo True m a c
duo True m a b                          = duo False (flip m) b a
duo _ _ _ _                             = False
```

Figure 3 Haskell code for implementing an algorithmic formulation of Duotyping rules.

```scala
class StudentsCollection[S <: Student](obj: S) {
  def student: S = obj
}
```

The Scala program shown above uses the upper bounds for the class *StudentsCollection* written as S <: Student. This upper bound restricts *StudentsCollection* to be instantiated with *Student* and its subtypes. Since the upper bound is *Student*, any class that is supertype of *Student* like *Person* cannot be instantiated in *StudentsCollection*.
However *lower bounds* are also useful, and indeed the Scala language allows them (though Java does not). One example of a program with lower bounds in Scala is:

```scala
class GraduateStudent extends  Student {
  def degree: String = "graduate degree"
}

class ResearchStudent extends GraduateStudent {
  override def degree: String = "research degree"
}

class CollectionExcludingResearchStudents[S >: GraduateStudent](obj: S) {
  def student: S = obj
}
```

In contrast to the upper bounds, the Scala program shown above uses the lower bounds for the class *CollectionExcludingResearchStudents* written as S >: GraduateStudent. This lower bound restricts *CollectionExcludingResearchStudents* to be instantiated with *GraduateStudent* and its supertypes. Since the lower bound is *GraduateStudent*. Any class that is a subtype of *GraduateStudent* (such as *ResearchStudent*) cannot be instantiated in *CollectionExcludingResearchStudents*. But any supertype of *GraduateStudent* like *Student* and *Person* (including *GraduateStudent*) can be instantiated in *CollectionExcludingResearchStudents*.

One can think of universal quantification with lower bounds as a dual to universal quantification with upper bounds. While there is no extension of $F_{<:}$ that we know of that presents universal quantification with lower bounds in the literature, applying a Duotyping design to $F_{<:}$ gives us, naturally, the two features at once (lower and upper bounded quantification).

Bounded quantification in $F_{<:}$. The traditional subtyping rule of System kernel $F_{<:}$ with upper bounded quantification is:

$$\frac{\Gamma, X\ <:\ A \vdash B\ <:\ C}{\Gamma \vdash (\forall X <: A.B)\ <:\ (\forall X <: A.C)}\ \textsc{ts-forallkfs}$$

In the premise of this rule, we add the type variable X to the context with an upper bound A. If under the extended context the bodies of the universal quantifier (B and C) are in a subtyping relation then the universal quantifiers are also in a subtyping relation.

To add lower bounded quantification the obvious idea is to add a second rule:

$$\frac{\Gamma, X\ :>\ A \vdash B\ <:\ C}{\Gamma \vdash (\forall X :> A.B)\ <:\ (\forall X :> A.C)}\ \textsc{ts-forallkfsb}$$

However this alone is not quite right because the environment is also extended with a lower bound ($X >: A$), which does not exist in $F_{<:}$ contexts. Therefore some additional care is also needed for the variable cases of $F_{<:}$ extended with lower bounded quantification. When an upper bounded constraint is found in the environment, the variable case needs to deal with the upper bound appropriately. Since there are two rules dealing with the variable case in $F_{<:}$, one possible approach is to add two more rules for dealing with upper bounds:

$$\frac{X{:>}A \in \Gamma \qquad \Gamma \vdash B\ <:\ A}{\Gamma \vdash B\ <:\ X}\ \textsc{ts-TVarb} \qquad \frac{X{:>}A \in \Gamma}{\Gamma \vdash X\ <:\ X}\ \textsc{ts-ReflTvar}$$

However such a design feels a little unsatisfactory. We need a total of 6 rules to fully deal with lower and upper bounded quantification (instead of 3 rules in $F_{<:}$). At the same time the rules are nearly identical, differing only on the kind of bounds that is used. Furthermore the metatheory of $F_{<:}$ also needs to be significantly changed. In particular *narrowing* has to be adapted to account for the lower bounds and *transitivity* has to be extended with several new cases. Since both *narrowing* and *transitivity* proofs for $F_{<:}$ are non-trivial, this extension is also non-trivial and adds further complexity to already complex proofs.

A variant of Kernel $F_{<:}$ with Duotyping. We now reconsider the design of Kernel $F_{<:}$ from scratch employing the Duotyping methodology. In the subtyping rule for universal quantification, it is important to note that the subtyping relation between two universal quantifiers in the conclusion is the same as the relation between types B and C in premise. Similarly, the (subtyping/supertyping) bounds of type variable X in the conclusion are the same as the bounds of type variable X in premise. In a design with Duotyping, we would like to generalize the two uses of subtyping. Therefore, we can design a single unified rule with the help of two modes:

$$\frac{\Gamma, X \lozenge_1 A \vdash B \lozenge_2 C}{\Gamma \vdash (\forall X \lozenge_1 A.B) \lozenge_2 (\forall X \lozenge_1 A.C)}\ \textsc{gs-forallkfs}$$

Section 4.1 explains the Duotyping rules of our Duotyping kernel $F_{<:}$ variant with union and intersection types ($F^{\wedge\vee}_{k\lozenge}$) in detail. Rule GS-FORALLKFS is the interesting case, capturing both upper and lower bounded quantification in an elegant way. This rule states that if in a well-formed context, type variable X has a $\lozenge_1$ relation with type A and if type B has $\lozenge_2$

$\boxed{A \lozenge B}$ *(Algorithmic Duotyping)*

$$\frac{}{A \lozenge \rceil \lozenge \lceil}\ \textsc{gs-TopBtmA} \qquad \frac{}{\rceil \overline{\lozenge} \lceil \lozenge A}\ \textsc{gs-TopBtmB} \qquad \frac{}{\mathsf{Int} \lozenge \mathsf{Int}}\ \textsc{gs-Int}$$

$$\frac{A_1 \overline{\lozenge} A_2 \qquad B_1 \lozenge B_2}{A_1 \to B_1 \lozenge A_2 \to B_2}\ \textsc{gs-Arrow}$$

Figure 4 The Duotyping relation for simply typed lambda calculus calculus.

relation with type C, then the universal quantification with body B has a $\lozenge_2$ relation with the universal quantification with body C. Correspondingly there are also two Duotyping rules for variables:

$$\frac{X \lozenge A \in \Gamma \qquad \Gamma \vdash A \lozenge B}{\Gamma \vdash X \lozenge B}\ \textsc{gs-TVarA} \qquad \frac{X \lozenge_1 A \in \Gamma}{\Gamma \vdash X \lozenge_2 X}\ \textsc{gs-ReflTVar}$$

In short, the design of a variant of $F_{<:}$ with Duotyping leads to a system that naturally accounts for both upper and lower bounded quantification. Moreover, the metatheory, and in particular the proofs of *narrowing* and *transitivity* are not more complex than the corresponding original $F_{<:}$ proofs. In fact the proof of transitivity is significantly simpler, because Duotyping enables novel proof techniques as we discuss next.

2.5 New proof techniques

Transitivity proofs are usually a challenge for systems with subtyping. This is partly because subtyping relations often need to deal with some *contravariance*. For instance, the rule TS-ARROW (in Figure 1) is contravariant on the input types. Such contravariance causes problems in certain proofs, including transitivity. To illustrate the issue more concretely, let's distill the essence of the problem by considering a simple lambda calculus with subtyping called $\lambda_{<:}$, where the types are:

Types	A, B	$::=$	$\top \mid \mathsf{Int} \mid A \to B$

and the subtyping rules for those types are just the relevant subset of the rules in Figure 1. The transitivity proof for this simple calculus is:

▶ **Lemma 1** ($\lambda_{<:}$ Transitivity). *If $A <: B$ and $B <: C$ then $A <: C$.*

Proof. By induction on type B.

- Case $\top$ and case Int are trivial to prove by destructing the hypothesis in context.
- Case $B_1 \to B_2$ requires inversion of the two hypotheses to discover that A can only be a function type, while C is either a function type or $\top$. ◀

In the arrow case, we need to invert both hypotheses to discover more information about A and C. For this very simply language this double inversion is not too problematic, but as the language of types grows and the subtyping relation becomes more complicated, such inversions become significantly harder to deal with.

At this point one may wonder if the transitivity proof could be done using a different inductive argument to start with, and thus avoid the double inversions. After all there are various other possible choices. Perhaps the most obvious choice is to try induction on the subtyping relation itself ($A <: B$), rather than on type B. However this does not work because of the contravariance for arrow types, which renders one of induction hypothesis in the arrow case useless (and thus do not allow the case to be proved). Other alternative choices for an inductive argument (such as type A or C) do not work for similar reasons.

Developing metatheory with Duotyping. In order to develop metatheory with Duotyping it is convenient to use an equivalent formulation of Duotyping that eliminates the duality rule (which is non-algorithmic and makes inversions more difficult). For $\lambda_{\Diamond}$, which is a Duotyping version of $\lambda_{<:}$, this would lead to the set of rules in Figure 4. This alternative algorithmic version eliminates the duality rule. Rules GS-TOPBTMA, GS-INT, and GS-ARROW are similar to the rules we discussed in Section 2.2. Rule GS-TOPBTMB is the dual rule of rule GS-TOPBTMA. With Rule GS-TOPBTMB, the duality rule is unnecessary.

Transitivity with Duotyping. Now we turn our attention to the proof of transitivity:

▶ **Lemma 2** ($\lambda_{\Diamond}$ Transitivity). *If $A \Diamond B$ and $B \Diamond C$ then $A \Diamond C$.*

Proof. By induction on $A \Diamond B$.

- All cases are trivial to prove by destructing $\Diamond$ and inversion of the second hypothesis ($B \Diamond C$). ◀

Transitivity of systems with Duotyping can often be proved by induction on the subtyping relation itself. This has the nice advantage that all the cases essentially become trivial to prove (for $\lambda_{\Diamond}$) and only a single inversion is needed for arrow types. A key reason why such approach works in the formulation with Duotyping is that we can keep case for arrow types covariant. Instead we only flip the mode. Another important observation is that when we prove a transitivity lemma with Duotyping we are, in fact, proving two lemmas simultaneously: one lemma for transitivity of subtyping, and another one for transitivity of supertyping. When we use the induction hypothesis we have access to both lemmas (by choosing the appropriate mode).

The proof of the transitivity lemma by induction on the Duotyping relation can scale up to more complex subtyping/Duotyping relations. This includes subtyping relations with advanced features such as intersection types, union types, parametric polymorphism and bounded quantification. All of these can follow the same strategy (induction on the Duotyping relation) to simplify the transitivity proof, as we shall see in Section 5.

3 The $\lambda_{\Diamond}^{\wedge\vee}$ calculus

In Section 2 we gave an overview and discussed advantages of using the Duotyping relation. In this section we introduce a lambda calculus with union and intersection types that is based on Duotyping. We aim at showing that developing calculi and metatheory using Duotyping is simple, requiring only a few small adaptations compared with more traditional formulations based on subtyping. Our main aim is to show type soundness (subject-reduction and preservation) for $\lambda_{\Diamond}^{\wedge\vee}$.

3.1 Syntax and Duotyping

Syntax. Figure 5 shows the syntax of the calculus. The types for $\lambda_{\Diamond}^{\wedge\vee}$ were already introduced in Section 2. Terms include all the constructs for the lambda calculus (variables x, functions $\lambda x : A.\, e$ and applications $e_1\, e_2$) and integers (n). Values are a subset of terms, consisting of

Types	A, B	$::=$	$\top \mid \bot \mid \mathsf{Int} \mid A \to B \mid A \wedge B \mid A \vee B$
Terms	e	$::=$	$x \mid n \mid \lambda x : A.\, e \mid e_1\, e_2$
Values	v	$::=$	$n \mid \lambda x : A.\, e$
Context	Γ	$::=$	$\bullet \mid \Gamma, x : A$
Mode	$\Diamond$	$::=$	$<: \mid :>$

$\boxed{A \Diamond B}$ *(Algorithmic Duotyping)*

$$\frac{A \Diamond C}{(A \Diamond_? B) \Diamond C}\ \textsc{gs-leftA} \qquad \frac{A \Diamond B}{A \Diamond (B\ \overline{\Diamond}_?\ C)}\ \textsc{gs-leftB} \qquad \frac{B \Diamond C}{(A \Diamond_? B) \Diamond C}\ \textsc{gs-rightA}$$

$$\frac{A \Diamond C}{A \Diamond (B\ \overline{\Diamond}_?\ C)}\ \textsc{gs-rightB} \qquad \frac{A \Diamond B \qquad A \Diamond C}{A \Diamond (B \Diamond_? C)}\ \textsc{gs-bothA} \qquad \frac{A \Diamond C \qquad B \Diamond C}{(A\ \overline{\Diamond}_?\ B) \Diamond C}\ \textsc{gs-bothB}$$

Figure 5 Syntax and Duotyping relation for union and intersection types.

abstractions and integers only. The mode $\Diamond$ is used to choose the mode of the relation: it can be either subtyping ($<:$) or supertyping ($:>$). Typing contexts Γ are standard and used to track the types of the variables in a program. Finally, a well-formedness relation $\Gamma \vdash \mathbf{ok}$ ensures that typing contexts are well-formed.

Duotyping for $\lambda_{\Diamond}^{\wedge\vee}$. The Duotyping rules for $\lambda_{\Diamond}^{\wedge\vee}$ were already partly presented in Figure 2. In addition to the rules in the $\lambda_{\Diamond}$, we also need extra rules for union and intersection types. These extra rules are presented in Figure 5. Rules GS-LEFTA, GS-RIGHTA, and GS-BOTHA are also similar to the rules GDS-LEFT, GDS-RIGHT, and GDS-BOTH presented in Figure 2. Since we eliminate the *duality rule* in the algorithmic version, we add dual subtyping rules. Rules GS-LEFTB, GS-RIGHTB, and GS-BOTHB are the dual versions of rules GS-LEFTA, GS-RIGHTA, and GS-BOTHA respectively. This formulation is shown to be sound and complete with respect to the formulation with the duality rule in Figure 2. As explained in Section 2 this variant of the rules makes some proofs easier, thus we employ it here. The Duotyping relation is reflexive and transitive:

▶ **Theorem 3** (Reflexivity). *$A \Diamond A$.*

Proof. By induction on type A. Reflexivity is trivial to prove by applying subtyping rules. ◀

▶ **Theorem 4** (Transitivity). *If $A \Diamond B$ and $B \Diamond C$ then $A \Diamond C$.*

Proof. By induction on subtyping relation.

- Cases rule GS-TOPBTMA, rule GS-INT, rule GS-LEFTA, rule GS-RIGHTA and rule GS-BOTHA are trivial to prove.
- Case rule GS-TOPBTMB requires an additional Lemma 5.
- Case rule GS-ARROW requires induction on hypothesis and subtyping rules.
- Cases rule GS-LEFTB and rule GS-RIGHTB requires an additional Lemma 6 to be applied on hypothesis in context.
- Case rule GS-BOTHB requires induction on the hypothesis. This case also requires rule GS-LEFTB, rule GS-RIGHTB, and rule GS-BOTHA subtyping rules. ◀

We used the following auxiliary lemmas to prove transitivity.

$\boxed{\Gamma \vdash e : A}$ *(Typing)*

$$\frac{\Gamma \vdash \mathbf{ok} \qquad x : A \in \Gamma}{\Gamma \vdash x : A}\ \textsc{g-var} \qquad \frac{\Gamma \vdash \mathbf{ok}}{\Gamma \vdash n : \mathsf{Int}}\ \textsc{g-int} \qquad \frac{\Gamma, x : A_1 \vdash e_2 : A_2}{\Gamma \vdash \lambda x : A_1.\, e_2 : A_1 \to A_2}\ \textsc{g-abs}$$

$$\frac{\Gamma \vdash e_1 : A_1 \to A_2 \qquad \Gamma \vdash e_2 : A_1}{\Gamma \vdash e_1\ e_2 : A_2}\ \textsc{g-app} \qquad \frac{\Gamma \vdash e : B \qquad B <: A}{\Gamma \vdash e : A}\ \textsc{g-sub}$$

$\boxed{e_1 \longrightarrow e_2}$ *(Reduction)*

$$\frac{}{(\lambda x : A_1.\, e_1)\ v_2 \longrightarrow [x \mapsto v_2]e_1}\ \textsc{gred-AppAbs} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\ e \longrightarrow e_1'\ e}\ \textsc{gred-Fun} \qquad \frac{e_1 \longrightarrow e_1'}{v\ e_1 \longrightarrow v\ e_1'}\ \textsc{gred-Arg}$$

Figure 6 Typing and reduction for $\lambda_{\Diamond}^{\wedge\vee}$.

▶ **Lemma 5** (Bound Selection). *If* $\rceil\Diamond\lceil\Diamond B$ *then* $A\Diamond B$.

This lemma captures the upper and lower bounds with respect to relation between two types. If the mode is subtyping, then it states that any type that is supertype of $\top$ is supertype of all the other types. If the mode is supertyping, then it states that any type that is subtype of $\bot$ is subtype of all the other types. In essence the lemma generalizes the following two lemmas (defined directly over subtyping and supertyping):

- If $\top <: B$ then $A <: B$
- If $\bot :> B$ then $A :> B$

▶ **Lemma 6** (Inversion for rule GDS-Both). *If* $C \Diamond (A \Diamond_? B)$ *then* $(C \Diamond A)$ *and* $(C \Diamond B)$.

This lemma captures the relation between types with respect to the duality of union and intersection types. It is the general form of two lemmas:

▶ **Lemma 7** (Inversion for Union types). *If* $(A \vee B) <: C$ *then* $(A <: C)$ *and* $(B <: C)$.

▶ **Lemma 8** (Inversion for Intersection types). *If* $C <: (A \wedge B)$ *then* $(C <: A)$ *and* $(C <: B)$.

Finally there is also a *duality lemma*, which complements reflexivity and transitivity:

▶ **Lemma 9** (Duality). $A\Diamond B = B\overline{\Diamond}A$.

This lemma captures the essence of duality, and enables us to switch the mode of the relation by flipping the arguments as well. Furthermore, the duality lemma plays a crucial role when proving soundness and completeness with respect to the declarative version of Duotyping, which has duality as an axiom instead. All of these lemmas are used in later proofs for type soundness.

3.2 Semantics and type soundness

Typing. The first part of Figure 6 presents the typing rules of $\lambda_{\Diamond}^{\wedge\vee}$. The rules are standard. Note that rule G-SUB is the subsumption rule: if an expression e has type B and B is a subtype of A then e has type A. Noteworthy, $B <: A$ is the Duotyping relation being used with the subtyping mode.

Reduction. At the bottom of Figure 6 we show the reduction rules of $\lambda^{\wedge\vee}_{\Diamond}$. Again, the reduction rules are standard. Rule GRED-APPABS is the usual beta-reduction rule, which substitutes a value v_2 for x in the lambda body e_1. Rule GRED-FUN and rule GRED-ARG are the standard call-by-value rules for applications.

Type soundness. The proof for type soundness relies on the usual preservation and progress lemmas:

▶ **Lemma 10** (Type Preservation). *If* $\Gamma \vdash e : A$ *and* $e \longrightarrow e'$ *then:* $\Gamma \vdash e' : A$.

Proof. By induction on the typing relation and with the help of Lemma 9. ◀

▶ **Lemma 11** (Progress). *If* $\Gamma \vdash e : A$ *then:*
1. *either e is a value.*
2. *or e can take a step to e'.*

Proof. By induction on the typing relation. ◀

3.3 Summary and Comparison

Besides $\lambda^{\wedge\vee}_{\Diamond}$, which employs the Duotyping relation, we have also formalized a lambda calculus with union and intersection types using the traditional subtyping relation ($\lambda^{\wedge\vee}_{<:}$). Most of the metatheory is similar with a great deal of theorems being almost the same. The main differences are in the metatheory for subtyping which has to be generalized. For example both reflexivity and transitivity have to be generalized to operate in the Duotyping relation instead. The formalization with Duotyping only has two additional lemmas (the duality lemma and the bound selection lemma), which have no counterparts with subtyping. The number of lines of code for the formalization of $\lambda^{\wedge\vee}_{<:}$ is 596 whereas for $\lambda^{\wedge\vee}_{\Diamond}$ is 630. The total number of lemmas required for $\lambda^{\wedge\vee}_{<:}$ are 23 and 25 for $\lambda^{\wedge\vee}_{\Diamond}$. Following two lemmas in $\lambda^{\wedge\vee}_{<:}$ are captured as one lemma in $\lambda^{\wedge\vee}_{\Diamond}$ (Lemma 6):

Inversion for Union Types. This lemma is already stated as Lemma 7: it is the inversion of the subtyping rule for the union types in the traditional subtyping relation. The lemma states that if the union of two types A and B is the subtype of a type C, then both types A and B are subtypes of type C.

Inversion for Intersection Types. This lemma, which corresponds to Lemma 8, is the inversion of the subtyping rule for the intersection types with the traditional subtyping relation. It states that if a type C is the subtype of the intersection of two types A and B, then the type C is a subtype of both types A and B.

4 The $F^{\wedge\vee}_{k\Diamond}$ calculus

In Section 3 we introduced a simple calculus with union and intersection types using Duotyping. This section extends that calculus with bounded quantification based on kernel $F_{<:}$. This new variant also employs Duotyping and is called $F^{\wedge\vee}_{k\Diamond}$. The main aim of this section is to show that sometimes we can get interesting and novel dual features come for free. In addition to upper bounded quantification of $F_{<:}$, System $F^{\wedge\vee}_{k\Diamond}$ provides lower bounded quantification as well. Additionally, we also show the type soundness of $F^{\wedge\vee}_{k\Diamond}$.

Types	A, B	$::=$	$\top \mid \bot \mid \mathsf{Int} \mid A \to B \mid A \wedge B \mid A \vee B \mid X \mid \forall (X \Diamond A).B$
Terms	e	$::=$	$x \mid n \mid \lambda x : A.\, e \mid e_1\, e_2 \mid \Lambda(X \Diamond A).e \mid e\, A$
Values	v	$::=$	$n \mid \lambda x : A.\, e \mid \Lambda(X \Diamond A).e$
Context	Γ	$::=$	$\bullet \mid \Gamma, x : A \mid \Gamma, X \Diamond A$
Mode	$\Diamond$	$::=$	$<: \mid :>$

$\boxed{\Gamma \vdash A \Diamond B}$ *($F_{k\Diamond}^{\wedge\vee}$ Duotyping)*

$$\frac{\Gamma \vdash \mathbf{ok} \qquad X \Diamond_1 A \in \Gamma}{\Gamma \vdash X \Diamond_2 X}\ \textsc{gs-ReflTVarA} \qquad \frac{X \Diamond A \in \Gamma \qquad \Gamma \vdash A \Diamond B}{\Gamma \vdash X \Diamond B}\ \textsc{gs-TVarA}$$

$$\frac{X \Diamond A \in \Gamma \qquad \Gamma \vdash B\, \overline{\Diamond}\, A}{\Gamma \vdash B\, \overline{\Diamond}\, X}\ \textsc{gs-TVarB} \qquad \frac{\Gamma, X \Diamond_1 A \vdash B \Diamond_2 C}{\Gamma \vdash (\forall X \Diamond_1 A.B) \Diamond_2 (\forall X \Diamond_1 A.C)}\ \textsc{gs-forallkfs}$$

Figure 7 Syntax and additional rules for Duotyping in $F_{k\Diamond}^{\wedge\vee}$.

4.1 Syntax and Duotyping

Syntax. Figure 7 shows the syntax of the calculus $F_{k\Diamond}^{\wedge\vee}$. Types $\top$, $\bot$, Int, $A \to B$, $A \wedge B$, $A \vee B$ are already introduced in Section 2. Type variable X and a universal quantifier on type variables $\forall(X \Diamond A).B$ are the two additional types in $F_{k\Diamond}^{\wedge\vee}$. Terms x, n, $\lambda x : A.\, e$, $e_1\, e_2$ are already discussed in Section 3.1. Type abstraction $\Lambda(X \Diamond A).e$ and type application $e\, A$ are two additional terms in $F_{k\Diamond}^{\wedge\vee}$. Values are a subset of terms, consisting of term abstraction, type abstraction and integers.

Duotyping for $F_{k\Diamond}^{\wedge\vee}$. Duotyping rules for a calculus with union and intersection types are presented in Figure 4. $F_{k\Diamond}^{\wedge\vee}$ has two significant differences in its Duotyping rules in comparison to Figure 4, which are presented in Figure 7. The first one is the addition of a typing context in the Duotyping rules. This is important to ensure that type variables are bound. Thus, Duotyping for $F_{k\Diamond}^{\wedge\vee}$ is now of the form $\Gamma \vdash A \Diamond B$. The second difference is that there are four more rules, three of them (rules GS-ReflTVarA, GS-TVarA, and GS-forallkfs) were already explained in Section 2.4. Rule GS-TVarB is the dual of rule GS-TVarA. We introduce this rule to eliminate the *duality rule*.

The Duotyping relation for $F_{k\Diamond}^{\wedge\vee}$ is reflexive and transitive as well:

▶ **Theorem 12** (Reflexivity). $\Gamma \vdash A \Diamond A$.

Proof. By induction on type A. ◀

▶ **Theorem 13** (Transitivity). *If* $\Gamma \vdash A \Diamond B$ *and* $\Gamma \vdash B \Diamond C$ *then* $\Gamma \vdash A \Diamond C$.

Proof. By induction on $\Gamma \vdash A \Diamond B$.

- Cases rule GS-topbtmA, rule GS-topbtmB, rule GS-int, rule GS-ReflTVar, rule GS-TVarA, rule GS-leftA, rule GS-rightA, rule GS-bothA are trivial to prove.
- Case rule GS-arrow is proved using the induction hypotheses.
- Case rule GS-TVarB can be proved using Lemma 16.
- Case rule GS-forallkfs is proved using the induction hypotheses.
- Case rule GS-leftB can be proved using an additional Lemma 15.
- Case rule GS-rightB also uses Lemma 15.
- Case rule GS-bothB is proved using the induction hypotheses. ◀

The auxiliary lemmas for transitivity are described next and are essentially the same as in Section 3.1.

▶ **Lemma 14** (Bound Selection). *If* $\Gamma \vdash \rceil \Diamond \lceil \Diamond B$ *then* $\Gamma \vdash A \Diamond B$.

▶ **Lemma 15** (Inversion for rule GDS-Both). *If* $\Gamma \vdash C \ \Diamond \ (A \ \Diamond_? \ B)$ *then* $\Gamma \vdash (C \ \Diamond \ A)$ *and* $(C \ \Diamond \ B)$.

There is also a *duality lemma*:

▶ **Lemma 16** (Duality). $\Gamma \vdash A \Diamond B \ = \Gamma \vdash B \overline{\Diamond} A$.

Finally, We also proved weakening and the narrowing lemmas for Duotyping calculus. Here we briefly compare the narrowing lemma for $F^{\wedge\vee}_{k<:}$ and $F^{\wedge\vee}_{k\Diamond}$:

▶ **Lemma 17** ($F^{\wedge\vee}_{k<:}$ Narrowing Lemma). *If* $\Gamma \vdash A \ <: \ B$ *and* $\Gamma, X <: B, \Gamma_1 \vdash C <: D$ *then* $\Gamma, X <: A, \Gamma_1 \vdash C <: D$

▶ **Lemma 18** ($F^{\wedge\vee}_{k\Diamond}$ Narrowing Lemma). *If* $\Gamma \vdash A \Diamond_1 B$ *and* $\Gamma, X \Diamond_1 B, \Gamma_1 \vdash C \Diamond_2 D$ *then* $\Gamma, X \Diamond_1 A, \Gamma_1 \vdash C \Diamond_2 D$

Lemma 17 exploits only the subtyping relation while Lemma 18 exploits our Duotyping relation. Lemma 18 illustrates how lower and upper bounds are captured under a unified mode relation in narrowing. Like the transitivity statement using a Duotyping formulation, one can think of the Duotyping narrowing lemma as actually two distinct lemmas: one for narrowing of upper bounds and another for narrowing of lower bounds. Also, it is important to note that Lemma 18 is using two modes $\Diamond_1$ and $\Diamond_2$. $\Diamond_1$ is the relation between types A, B and the type variable X. Whereas, $\Diamond_2$ is the relation between type C and type D. Those two relations do not need to be the same.

4.2 Semantics and type soundness

Typing. The first part of Figure 8 presents the typing rules of $F^{\wedge\vee}_{k\Diamond}$. The first five rules are standard and are already explained in Section 2.1. Rules G-TABS and G-TAPP are the two additional rules in $F^{\wedge\vee}_{k\Diamond}$. Rule G-TABS is similar to the standard rule for type abstractions in $F_{<:}$ except that it generalizes the subtyping bound to a $\Diamond$ bound, which could either be subtyping or supertyping. Rule G-APP again differs from the rule for type applications in $F_{<:}$ by using a $\Diamond$ bound instead of just a subtyping bound. These two rules rules are noteworthy because they also illustrate an advantage of using Duotyping in the typing relation. Without Duotyping we would need multiple typing rules to capture different variations of the bounds.

Reduction. The last part of Figure 8 presents the reduction rules of our calculus. Again, reduction rules are standard except for the rule GRED-TAPPTABS. In rule GRED-TAPPTABS the duality relation captures both upper and the lower bounds. Rule GRED-TFUN is the standard reduction rule for the type applications.

Type Soundness. We proved the type soundness for our calculus. All the proofs are formalized in Coq theorem prover.

▶ **Lemma 19** (Type Preservation). *If* $\Gamma \vdash e : A$ *and* $e \longrightarrow e'$ *then:* $\Gamma \vdash e' : A$.

Proof. By induction on the typing relation.
- Case rules G-VAR, G-INT, G-ABS, G-TABS, and G-SUBS are trivial to solve.
- Case rule G-APP uses Theorem 12 and Lemma 16.
- Case rule G-TAPP uses Theorem 12. ◀

$\boxed{\Gamma \vdash e : A}$ *(Typing)*

$$\frac{\Gamma \vdash \mathbf{ok} \qquad x : A \in \Gamma}{\Gamma \vdash x : A}\ \textsc{G-var} \qquad \frac{\Gamma \vdash \mathbf{ok}}{\Gamma \vdash n : \mathsf{Int}}\ \textsc{G-int} \qquad \frac{\Gamma, x : A_1 \vdash e_2 : A_2}{\Gamma \vdash \lambda x : A_1.\, e_2 : A_1 \to A_2}\ \textsc{G-abs}$$

$$\frac{\Gamma \vdash e_1 : A_1 \to A_2 \qquad \Gamma \vdash e_2 : A_1}{\Gamma \vdash e_1\ e_2 : A_2}\ \textsc{G-app} \qquad \frac{\Gamma \vdash e : B \qquad \Gamma \vdash B\ <:\ A}{\Gamma \vdash e : A}\ \textsc{G-subs}$$

$$\frac{\Gamma, X \Diamond A \vdash e : B}{\Gamma \vdash \Lambda X \Diamond A.\, e : \forall(X \Diamond A).B}\ \textsc{G-tabs} \qquad \frac{\Gamma \vdash e : \forall(X \Diamond A).B \qquad \Gamma \vdash C \Diamond A}{\Gamma \vdash e\, C : [X \mapsto C]B}\ \textsc{G-tapp}$$

$\boxed{e_1 \longrightarrow e_2}$ *(Reduction)*

$$\frac{}{(\lambda x : A_1.\, e_1)\ v_2 \longrightarrow [x \mapsto v_2]e_1}\ \textsc{gred-AppAbs} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\ e \longrightarrow e_1'\ e}\ \textsc{gred-Fun} \qquad \frac{e_1 \longrightarrow e_1'}{v\ e_1 \longrightarrow v\ e_1'}\ \textsc{gred-Arg}$$

$$\frac{}{(\Lambda X \Diamond A.\, e_1)\ B \longrightarrow [X \mapsto B]e_1}\ \textsc{gred-TAppTAbs} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\ A \longrightarrow e_1'\ A}\ \textsc{gred-TFun}$$

Figure 8 Typing and reduction of the duotyped kernel $F_{<:}$.

▶ **Lemma 20** (Progress). *If* $\Gamma \vdash e : A$ *then:*

1. *either* e *is value.*
2. *or* e *can take step to* e'.

Proof. By induction on the typing relation.

- Case rules G-VAR, G-INT, G-ABS, G-TABS, and G-SUBS are trivial to solve.
- Case rule G-APP requires canonical forms.
- Case rule G-TAPP requires canonical forms. ◀

4.3 Summary and Comparison

Besides $F^{\wedge\vee}_{k\Diamond}$, which employs the Duotyping relation, we have also formalized a calculus $F^{\wedge\vee}_{k<:}$: an extension of kernel $F_{<:}$ (only with upper bounded quantification) with union and intersection types using the traditional subtyping relation. The essential differences are similar to what we already discussed in Section 3.3. The formalization with Duotyping only has two additional lemmas (the duality lemma and the bound selection lemma), besides a few minor auxiliary lemmas. The number of lines for proof for the formalization of $F^{\wedge\vee}_{k<:}$ is 1648 whereas for $F^{\wedge\vee}_{k\Diamond}$ is 1770. The total lemmas required for $F^{\wedge\vee}_{k<:}$ are 74 and 81 for $F^{\wedge\vee}_{k\Diamond}$. We emphasize that one significant difference between $F^{\wedge\vee}_{k<:}$ and $F^{\wedge\vee}_{k\Diamond}$ is the additional lower bounded quantification provided by $F^{\wedge\vee}_{k\Diamond}$. This is an extra feature which comes essentially for free with Duotyping.

5 A Case Study on Duotyping

In this section we present an empirical case study, which we conducted to validate some of the benefits of Duotyping. Overall, the results of our case study indicate that: Duotyping does allow for compact specifications; the complexity of developing formalization with Duotyping is comparable to similar developments using traditional subtyping relations; transitivity proofs are often significantly simpler; and Duotyping is a generally applicable technique.

5.1 Case Study

We formalized a number of different calculi using Duotyping. All the proofs and metatheory are mechanically checked by the Coq theorem prover. We also formalized a few traditional subtyping systems for comparison. Table 1 shows a brief overview of various systems that we formalized. $\lambda_{<:}$, $\lambda_{<:}^{\wedge\vee}$, $F_{k<:}$, $F_{k<:}^{\wedge\vee}$ and $F_{F<:}$ are the traditional subtyping systems. The Coq formalizations for the traditional subtyping systems are based on existing Coq formalizations from the locally nameless representation with cofinite quantification tutorial and repository (`https://www.chargueraud.org/softs/ln/`) by Charguéraud [18]. The formalizations of $\lambda_{\Diamond}$, $\lambda_{\Diamond}^{\wedge\vee}$, $F_{k\Diamond}$, $F_{k\Diamond}^{\wedge\vee}$ and $F_{F\Diamond}$ are their respective Duotyping formulations, and modify the original ones with traditional subtyping. Subscript $<:$ represents a calculus with traditional subtyping whereas $\Diamond$ represents a calculus with Duotyping. Superscript $\wedge\vee$ is the notation for a system with intersection and union types. Subscript k corresponds to the kernel version of a variant of $F_{<:}$, while subscript F corresponds to the corresponding full version. We also formalized a simple polymorphic system without bounded quantification using Duotyping. We have two Duotyping variants for this polymorphic type system without bounded quantification. One without union and intersection types ($F_{\Diamond}$) and another with union and intersection types ($F_{\Diamond}^{\wedge\vee}$).

In Table 1, the last column (Transitivity) summarizes the proof technique used in each system to prove transitivity. Recall the transitivity lemma (using the Duotyping formulation):

▶ **Theorem 21** (Transitivity). *If $A\Diamond B$ and $B\Diamond C$ then $A\Diamond C$.*

Induction on the middle type means induction on type B (or well-formed type B for polymorphic systems), whereas induction on the Duotyping relation means induction on $A\Diamond B$.

Research Questions. Section 1 discussed benefits of using Duotyping. This section attempts to quantify some of these benefits. More concretely, we answer the following questions in this section:

- Does Duotyping provide shorter specifications?
- Does Duotyping increase the complexity of the formalization and metatheory of the language?
- Does Duotyping make transitivity proofs simpler?
- Is Duotyping a generally applicable technique?

We follow an empirical approach to answer these questions and address each question in a separate (sub)section. Obviously a precise measure for complexity/simplicity is hard to obtain. We use SLOC for the formalization and proofs as an approximation. All the formalizations are written in the same Coq style to ensure that the comparisons are fair.

5.2 Does Duotyping provide shorter specifications?

This section answers our first question. In short our case study seems to support this conclusion. The declarative Duotyping rules of all the systems that we formalized are shown in Table 2. Please note that the formulation also contains the *duality rule*. $\lambda_{\Diamond}$ has the basic set of Duotyping rules. These rules are common in all of the systems. $\lambda_{\Diamond}^{\wedge\vee}$ has the subtyping rules for intersection types and union types in addition to the rules from $\lambda_{\Diamond}$. $F_{\Diamond}$ contains two more rules (rules GDS-ReflTvarP and GDS-forallfsp) in addition to the rules from $\lambda_{\Diamond}$. $F_{\Diamond}^{\wedge\vee}$ has all the rules from $\lambda_{\Diamond}$, $\lambda_{\Diamond}^{\wedge\vee}$ and $F_{\Diamond}$. $F_{k\Diamond}$ has three additional

Table 1 Description of all systems.

Name	Description	SLOC	Transitivity
$\lambda_{<:}$	STLC with subtyping	537	By induction on the middle type.
$\lambda_{\Diamond}$	STLC with Duotyping	583	By induction on the Duotyping relation.
$\lambda^{\wedge\vee}_{<:}$	STLC with subtyping, union types and intersection types	595	By induction on the middle type.
$\lambda^{\wedge\vee}_{\Diamond}$	STLC with Duotyping, union types and intersection types	623	By induction on the Duotyping relation.
$F_{\Diamond}$	Simple polymorphic system with Duotyping and without bounded quantification	1466	By induction on the Duotyping relation.
$F^{\wedge\vee}_{\Diamond}$	Simple polymorphic system with Duotyping, union types and intersection types and without bounded quantification	1546	By induction on the Duotyping relation.
$F_{k<:}$	System $F_{<:}$ kernel	1542	By induction on the (well-formed) middle type.
$F_{k\Diamond}$	System $F_{<:}$ kernel with Duotyping	1579	By induction on the Duotyping relation.
$F^{\wedge\vee}_{k<:}$	System $F_{<:}$ kernel with subtyping, union types and intersection types	1648	By induction on the (well-formed) middle type.
$F^{\wedge\vee}_{k\Diamond}$	System $F_{<:}$ kernel with Duotyping, union types and intersection types	1770	By induction on the Duotyping relation.
$F_{F<:}$	System full $F_{<:}$	1518	By induction on the (well-formed) middle type.
$F_{F\Diamond}$	System full $F_{<:}$ with Duotyping	1786	By induction on the (well-formed) middle type.

subtyping rules GDS-REFLTVAR, GDS-TVAR, and GDS-FORALLKFS in addition to the rules from $\lambda_{\Diamond}$. $F^{\wedge\vee}_{k\Diamond}$ has all the rules from $\lambda_{\Diamond}$, $\lambda^{\wedge\vee}_{\Diamond}$, and $F_{k\Diamond}$. $F_{F\Diamond}$ has an additional subtyping rule GDS-FORALLFFS.

Comparison with systems using traditional subtyping. Table 3 shows the number of rules and features for different calculi formulated with subtyping and Duotyping. In our formulation, $\lambda_{<:}$ has 3 types $\top$, Int, *and* $A \to B$. This requires 3 subtyping rules to capture the subtyping relation of these 3 types. If we wanted to support the $\bot$ type in $\lambda_{<:}$ we would need to add 1 more subtyping rule. In the table we express the extra rules required for extra features as (+n), where n is the number of extra rules. Duotyping supports $\bot$ for free by exploiting the dual nature of $\top$ with the help of *duality rule*. Systems with more rules follow the same approach for traditional systems i.e more types require more subtyping rules. If we wanted to support the $\bot$ type in $\lambda^{\wedge\vee}_{<:}$ we also need 1 additional rule. To further extend our discussion to the polymorphic systems with bounded quantification, we would need 4 additional rules in $F_{k<:}$ (1 for $\bot$ type and 3 for lower bounded quantification). Similarly we would need 4 additional rules to support lower bounds and lower bounded quantification in $F^{\wedge\vee}_{k<:}$.

In summary, in the systems that we compared Duotyping has a similar number of rules to systems with subtyping, but it comes with extra features. If we wanted to add those features to systems with traditional subtyping, then that would generally result in more rules for the traditional versions compared to Duotyping. This would also have an impact in the SLOC of the metatheory, increasing the metatheory for those systems considerably.

Table 2 Declarative Duotyping rules of all systems.

Name	Duotyping Rules
$\lambda_\Diamond$	$\boxed{A \Diamond B}$ *($\lambda_\Diamond$ Duotyping)* $\dfrac{}{A \Diamond \rceil \Diamond \lceil}$ GDS-TOPBTM $\dfrac{}{\mathsf{Int} \Diamond \mathsf{Int}}$ GDS-INT $\dfrac{A_1 \overline{\Diamond} A_2 \qquad B_1 \Diamond B_2}{A_1 \to B_1 \Diamond A_2 \to B_2}$ GDS-ARROW $\dfrac{B \overline{\Diamond} A}{A \Diamond B}$ GDS-DUAL
$\lambda_\Diamond^{\wedge\vee}$	$\boxed{A \Diamond B}$ *($\lambda_\Diamond^{\wedge\vee}$ Duotyping plus all rules from $\lambda_\Diamond$)* $\dfrac{A \Diamond C}{(A \Diamond_? B) \Diamond C}$ GDS-LEFT $\dfrac{B \Diamond C}{(A \Diamond_? B) \Diamond C}$ GDS-RIGHT $\dfrac{A \Diamond B \qquad A \Diamond C}{A \Diamond (B \Diamond_? C)}$ GDS-BOTH
$F_\Diamond$	$\boxed{A \Diamond B}$ *($F_\Diamond$ Duotyping plus all rules from $\lambda_\Diamond$)* $\dfrac{}{X \Diamond X}$ GDS-REFLTVARP $\dfrac{A \Diamond B}{(\forall X.A) \Diamond (\forall X.B)}$ GDS-FORALLFSP
$F_\Diamond^{\wedge\vee}$	$\boxed{A \Diamond B}$ *($F_\Diamond^{\wedge\vee}$ Duotyping plus all rules from $\lambda_\Diamond$, $\lambda_\Diamond^{\wedge\vee}$ and $F_\Diamond$)*
$F_{k\Diamond}$	$\boxed{\Gamma \vdash A \Diamond B}$ *($F_{k\Diamond}$ Duotyping plus all rules from $\lambda_\Diamond$)* $\dfrac{\Gamma \vdash \mathbf{ok} \qquad X \Diamond_1 A \in \Gamma}{\Gamma \vdash X \Diamond_2 X}$ GDS-REFLTVAR $\dfrac{X \Diamond A \in \Gamma \qquad \Gamma \vdash A \Diamond B}{\Gamma \vdash X \Diamond B}$ GDS-TVAR $\dfrac{\Gamma, X \Diamond_1 A \vdash B \Diamond_2 C}{\Gamma \vdash (\forall X \Diamond_1 A.B) \Diamond_2 (\forall X \Diamond_1 A.C)}$ GDS-FORALLKFS
$F_{k\Diamond}^{\wedge\vee}$	$\boxed{\Gamma \vdash A \Diamond B}$ *($F_{k\Diamond}^{\wedge\vee}$ Duotyping plus all rules from $\lambda_\Diamond$, $\lambda_\Diamond^{\wedge\vee}$ and $F_{k\Diamond}$)*
$F_{F\Diamond}$	$\boxed{\Gamma \vdash A \Diamond B}$ *($F_{F\Diamond}$ Duotyping plus all rules from $F_{k\Diamond}$ excluding rule* GS-FORALLKFS *and union/intersection rules)* $\dfrac{\Gamma \vdash A \ \lvert \Diamond_1 \rvert_{\Diamond_2} \ B \qquad \Gamma, X \Diamond_1 (A \ \widetilde{\Diamond} \ B) \vdash A_1 \Diamond_2 B_1}{\Gamma \vdash (\forall X \Diamond_1 A.A_1) \Diamond_2 (\forall X \Diamond_1 B.B_1)}$ GDS-FORALLFFS

Table 3 Comparing the features and number of rules with subtyping and Duotyping.

System	Subtyping rules count	System	Duotyping rules count	Duotyping extra features
$\lambda_{<:}$	3 (+1)	$\lambda_{\Diamond}$	4	lower bounds in $\lambda_{\Diamond}$
$\lambda_{<:}^{\wedge\vee}$	9 (+1)	$\lambda_{\Diamond}^{\wedge\vee}$	7	lower bounds in $\lambda_{\Diamond}^{\wedge\vee}$
$F_{k<:}$	5 (+4)	$F_{k\Diamond}$	7	lower bounds and lower bounded quantification in $F_{k\Diamond}$
$F_{k<:}^{\wedge\vee}$	11 (+4)	$F_{k\Diamond}^{\wedge\vee}$	10	lower bounds and lower bounded quantification in $F_{k\Diamond}^{\wedge\vee}$

Table 4 SLOC of traditional subtyping and Duotyping systems.

Subtyping System	SLOC	Duotyping System	SLOC
$\lambda_{<:}$	537	$\lambda_{\Diamond}$	583
$\lambda_{<:}^{\wedge\vee}$	595	$\lambda_{\Diamond}^{\wedge\vee}$	623
$F_{k<:}$	1542	$F_{k\Diamond}$	1579
$F_{k<:}^{\wedge\vee}$	1648	$F_{k\Diamond}^{\wedge\vee}$	1770

5.3 Does Duotyping increase the complexity of the formalization and metatheory of the language?

At first, one may think that Duotyping increases the complexity of formalization and metatheory of the language, since it provides interesting extra features and generalizations normally come at a cost. Interestingly, Duotyping does not add significant extra complexity in the formalization and metatheory of the language. Table 4 shows the SLOC for formalizations using traditional subtyping and Duotyping systems. The lines of code for $\lambda_{<:}^{\wedge\vee}$ are 595 and the lines of code for $\lambda_{\Diamond}^{\wedge\vee}$ are 623. Similarly, the lines of code for $F_{k<:}^{\wedge\vee}$ are 1648 and 1770 for $F_{k\Diamond}^{\wedge\vee}$. Although SLOC for Duotyping systems are slightly more than traditional subtyping systems, the Duotyping systems come with extra features. Nevertheless the mechanization effort is roughly the same for version with and without Duotyping. Also, as illustrated in Sections 3 and 4, the vast majority of the lemmas/metatheory for calculi with Duotyping are similar to traditional systems with subtyping.

5.4 Does Duotyping make transitivity proofs simpler?

Transitivity is often the most difficult property to prove in the metatheory of a language with subtyping. Table 1 highlights a brief comparison between the techniques for the transitivity proof of various systems. Transitivity of systems with Duotyping is generally proved by induction on the Duotyping relation. One exception is $F_{F\Diamond}$ where induction on the Duotyping does not work. As discussed in Section 2.5 Duotyping allows us to simplify the transitivity proof by using a different inductive argument.

Table 5 shows the SLOC for transitivity proofs of various systems. The SLOC for $\lambda_{<:}$ transitivity proof are 7 and the SLOC for $\lambda_{\Diamond}$ transitivity proof are 4. Similarly, the SLOC for $F_{k<:}^{\wedge\vee}$ transitivity proof are 38 and 18 for the transitivity proof of $F_{k\Diamond}^{\wedge\vee}$. This evaluation shows that Duotyping always allows us to reduce the size of the transitivity proof. Again, it is important to note that Duotyping also provides extra features of lower bound and lower bounded quantification. Despite these additional features in Duotyping systems, their transitivity proofs are shorter than the traditional systems with subtyping.

Table 5 SLOC for transitivity proofs.

Subtyping System	Transitivity SLOC	Duotyping System	Transitivity SLOC
$\lambda_{<:}$	7	$\lambda_{\Diamond}$	4
$\lambda_{<:}^{\wedge\vee}$	13	$\lambda_{\Diamond}^{\wedge\vee}$	11
$F_{k<:}$	26	$F_{k\Diamond}$	13
$F_{k<:}^{\wedge\vee}$	38	$F_{k\Diamond}^{\wedge\vee}$	18

However we could not employ this proof technique in our Duotyping version of full $F_{<:}$ ($F_{F\Diamond}$). The problem is related to *narrowing*, which in $F_{F\Diamond}$ is closely coupled with *transitivity*. Despite that we could still apply the technique to most systems with Duotyping, and even for $F_{F\Diamond}$ we can still prove transitivity using the same technique as in the traditional $F_{<:}$ (i.e. using the middle type as the inductive argument).

5.5 Is Duotyping a generally applicable technique?

Our case studies indicate that Duotyping is generally an applicable technique. In all the systems that we have tried to use Duotyping, we have managed to successfully apply it. Furthermore we believe that Duotyping can be essentially applied to any system with a traditional subtyping relation. The most complex system where we have employed Duotyping is $F_{F\Diamond}$. In $F_{F\Diamond}$ universal quantification allows Duotyping between the bounds, generalizing the universal quantification presented in Section 4. Rule GDS-FORALLFFS in $F_{F\Diamond}$ employs two operations $|\Diamond_1|_{\Diamond_2}$ and $A \mathrel{\widetilde{\Diamond}} B$:

$$
\begin{array}{lrcl}
|\Diamond_1|_{\Diamond_2} & & & \\
 & |<:|_{\Diamond_2} & = & \overline{\Diamond}_2 \\
 & |:>|_{\Diamond_2} & = & \Diamond_2 \\
A \mathrel{\widetilde{\Diamond}} B & & & \\
 & A \mathrel{\widetilde{<:}} B = & & B \\
 & A \mathrel{\widetilde{:>}} B = & & A
\end{array}
$$

$|\Diamond_1|_{\Diamond_2}$ takes two modes $\Diamond_1$ and $\Diamond_2$ as input, and flips $\Diamond_2$ if $\Diamond_1$ is subtyping, otherwise it returns $\Diamond_2$. This operation chooses the mode to check the relationship between the bounds of the two universal quantifiers being compared for Duotyping. The second operation $A \mathrel{\widetilde{\Diamond}} B$ selects the bounds to use in the environment when checking the Duotyping of the bodies of the universal quantifiers. It takes a mode $\Diamond$ and two types A B as inputs, and returns the second type if the mode is subtyping, otherwise it returns the first type.

6 Related Work

Apart from informally observing duality of type system features, as far as we known, formally exploiting duality in subtyping relations has not been investigated in the past. However there is plenty of work on uses of duality in programming language theory. Furthermore there is related work on type systems that exploit various generalizations for added expressive power or economy in metatheory and implementation. We discuss these next.

6.1 Duality in Logic and Programming Language Theory

In type theory [5] and/or category theory [30, 14] duality occurs in various forms. For instance, the duality between sum and product types is well-known in both type and category theory. Properties about such types often explicitly acknowledge duality. Many properties about sum types are presented as dual properties of corresponding properties on product types and vice-versa. Our Lemma 6 is an example of a property that applies to both union and/or intersection types. In this property duality is not only acknowledged, but directly exploited in the lemma itself to provide a generalized property that can be specialized to one construct and its dual. Various other dualities between constructs are known and exploited in various ways in type and/or category theory. For example, existential and universal quantification can be captured by an encoding by one through the other. The type $\exists\alpha.\ A$ can be encoded as $\forall\beta.(\forall\alpha.\ A \to \beta) \to \beta$, which requires a kind of CPS translation [20] of the corresponding terms. Similar encodings exist for sums and products.

In the field of *proof-theoretic semantics* [25] and in *natural deduction* the concept of *harmony* is used to describe introduction and elimination rules that are in some sense dual. For instance, the usual rules for introduction and elimination of conjunction are in perfect harmony. The inversion principles by Prawitz [38] are a general procedure to associate to any arbitrary collection of introduction rules a specific collection of elimination rules. The elimination rules are in harmony with the given collection of introduction rules. Prawitz inversion principles attempt to capture harmony in a more precise way, directly expressing it formally. Therefore inversion principles have similar considerations to Duotyping in terms of expressing some form of duality directly in a formalism. However inversion principles focus on introduction and/or elimination rules, while Duotyping is focused on subtyping. Nevertheless in future work we are interested in exploiting the use of duality in the typing relation more. We believe that the notion of harmony and inversion principles could be quite helpful in such work.

Double-line rules [21] are deduction rules that can be read both from top to bottom (as usual) and also from bottom to top. In other words they express two standard (dual) deduction rules in a single double-line rule. Like Duotyping, double-line rules aim at expressing a form of duality in a single rule. Unlike Duotyping, double-line rules are concerned with (dual) rules where the premises and conclusions of one rule become the conclusions and premises of the other rule, respectively.

Bernardi et al. [10] explain duality relations in the context in *session types*. Binary session types have two endpoints connected through one communication channel. In session types, connected endpoints should have a dual relation in their session types. The duality relation in session types is related to types and may have various interpretations. In contrast Duotyping is about subtyping (or supertyping).

The duality between data and codata is well-known in programming language theory [14]. Data types and codata types are duals in the sense that data types are defined in terms of constructors while codata types are defined in terms of destructors. More recently, such duality has been exploited in language design [35, 13] to provide an automatic way to switch between programs defined on datatypes and equivalent programs defined on codata types. The use of duality in this line of work is quite different from ours.

6.2 Generalizations in Type Systems and Type Theory

Pure type systems (PTSs) [45, 31, 2, 28, 41, 48] capture a generalization of various type systems ($F, F^{\omega}, \lambda\mathrm{P}$). Typing rules of multiple type systems are expressed in pure type systems via parameterization. PTSs are parameterized by three sets: a set of sorts; a set of axioms;

and a set of rules. Concrete type systems (such as System F), are recovered with concrete instantiations of those sets. Pure type systems with subtyping [48] are a variant of pure type systems that captures a family of type systems with subtyping. This variant captures only the upper bounds. It does not provide subtyping generalization with both upper and the lower bounded quantification like our Duotyping generalizations of $F_{<:}$. Pure subtype systems [26] is a family of calculi based on subtyping only (and without a typing relation). This system eliminates the need of typing and presents an alternative to typing using subtyping only. Pure subtype systems support upper bounded quantification, but no lower bounded quantification.

Modal Type Theory. Modal type theory [33] is an extension of type theory which provides type rules using modalities. Modal type theory can represent a proposition as types which may be proved based upon the deduction rules in a given context. Modal type theory also employs modes, for instance *possibility* and *necessity* [42, 33]. There are many type systems that use modes to generalize typing relations. One can view Duotyping as a simple instance of a relation with a mode. In Duotyping the mode is either subtyping or supertyping.

Bi-directional type checking. Bi-directional type checking [37, 23] also employs a mode, but in the typing relation instead. Bi-directional type checking is a common technique, used in implementations of programming languages, that can eliminate redundant type annotations. Bi-directional type-checking is also employed is several type systems, especially those where full type inference is undecidable [37, 24]. In such cases only partial inference methods are feasible in practice, which means that some type annotations are necessary. Bi-directional type checking is useful in such cases, allowing the type information to be easily propagated without requiring further (redundant) annotations. The modes in bi-directional type-checking are checking or synthesis. Checking checks a given term against a given type, whereas the synthesis infers the type based upon the available information in the context.

Unified Subtyping. Unified subtyping [46] is a technique that can be used in dependently typed systems supporting unified syntax to model typing and subtyping in a single relation. The single unified subtyping relation generalizes both typing and subtyping. Like Duotyping, unified subtyping can also help reducing language metatheory and duplication. However unified subtyping is orthogonal to Duotyping and does not exploit duality of features. We believe that both techniques can complement each other.

Bounded quantification and generalizations. System $F_{<:}$ [16] is extensively studied due to its feature of bounded quantification. F-bounded quantification [15] is a generalization of bounded quantification to handle recursive types. Although we are not aware of an extension of $F_{<:}$ with lower bounded quantification, such notion has appeared before in some calculi. For instance, Igarashi and Viroli [27] have pointed out correspondence between use-site variance and existential types and, in order to capture contravariance, they introduced lower-bounded existential types.

One generalization of $F_{<:}$ is studied by Amin and Rompf [4], which formalizes *type bounds* in Scala. Type bounds is an interesting feature in Scala as elaborated by the following code (code extended from Section 2.4):

```
class TypeBoundsCollection[S >: GraduateStudent <: Student](obj: S) {
  def student: S = obj
}
```

While in our variants of $F_{<:}$ we support either lower bounded quantification or upper bounded quantification (but not both at once), Scala's type bounds allow both upper and lower bounds at once. This is clearly more expressive than what we have, but it comes with its own problems. Formalisms with Scala-like type bounds often need to include a transitivity axiom (and thus are non-algorithmic) and they have to deal with the bad bounds problem. In contrast our simpler extension of type bounds is comparable in complexity to $F_{<:}$'s upper bounded quantification, and there is a set of algorithmic subtyping rules without a built-in transitivity axiom.

Intersection and Union Types. Intersection and union types [6, 22, 32, 36] are getting significant attention in recent years, and are used in several modern programming languages (including Scala, Flow or TypeScript). Reynolds [39] was the first to promote the use of intersection types in programming languages. Later on, Pierce [36] studied intersection types, union types and polymorphism combined in a typed λ-calculus. Recently, Muehlboeck and Tate [32] presented a generalized formulation of calculi with union and intersection types. They demonstrated it with the help of Ceylon programming language [29]. Dunfield [22] presented an expressive calculus with a merge operator and unrestricted intersection types with union types. We exploit the duality of union and intersection types to illustrate Duotyping. Our Duotyping calculi manages to capture the six common rules for unions and intersections using three rules only (plus the duality rule), which provides a simple illustrative example of the use of duality.

7 Conclusion

In this paper, we have presented a generalization of subtyping using a relation parameterized by a mode. We call this relation Duotyping. Duotyping allows formalizations of subtyping to exploit duality between features directly in the formalism, and provides multiple benefits over traditional subtyping relation. It shortens subtyping specifications, provides dual features essentially for free and simplifies the transitivity proof in many calculi. An example of an extra dual feature that is obtained for free, with a Duotyping design, is lower bounded quantification. Lower bounded quantification arises naturally when we apply the Duotyping to conventional $F_{<:}$ type systems. To validate the benefits of Duotyping, we have conducted an empirical evaluation, implemented multiple calculi using both a traditional subtyping relation and a Duotyping formulation, and compared the resulting formalizations.

Duality has been studied extensively in several contexts in the past. However, as far as we know, our work is the first to study duality in the context of subtyping. Future work on Duotyping includes applying the Duotyping methodology to several other forms of subtyping that are of practical interest. We are particularly interested to apply Duotyping to systems that include a feature, but have no obvious dual feature. We hope to discover potentially new features that are useful for programming. Furthermore, we hope to scale the approach so that it can be used in real programming language implementations, leading to more compact and consistent implementations of subtyping. Another domain for future work on Duotyping is the typing relation. Although we already have some simple cases where Duotyping also provides benefits for typing (see Section 4), we acknowledge that introduction and elimination rules for some (dual) constructs introduce new challenges. For instance, more complex introduction and elimination forms for union types and intersection types can be quite different [22]. Further exploration is needed to see how much Duotyping can help in typing relations for calculi with such features. More generally, Duotyping promotes the use of duality in language design. We envision that in the future new language designs can exploit duality to ensure consistency between various language constructs, by exploiting techniques similar to Duotyping.

References

1 Andreas Abel and Dulma Rodriguez. Syntactic metatheory of higher-order subtyping. In *International Workshop on Computer Science Logic*, pages 446–460. Springer, 2008.

2 Robin Adams. Pure type systems with judgemental equality. *Journal of Functional Programming*, 16(2):219–246, 2006.

3 Nada Amin, Adriaan Moors, and Martin Odersky. Dependent object types. In *19th International Workshop on Foundations of Object-Oriented Languages*, 2012.

4 Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, 2017.

5 Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Academic Press, Inc., 1986.

6 Franco Barbanera, Mariangiola Dezaniciancaglini, and Ugo Deliguoro. Intersection and union types: syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.

7 Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment 1. *The journal of symbolic logic*, 48(4):931–940, 1983.

8 Bruno Barras, Samuel Boutin, Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The coq proof assistant reference manual: Version 6.1, 1997.

9 Jon Barwise and Robin Cooper. Generalized quantifiers and natural language. In *Philosophy, language, and artificial intelligence*, pages 241–301. Springer, 1981.

10 Giovanni Bernardi, Ornela Dardha, Simon J Gay, and Dimitrios Kouzapas. On duality relations for session types. In *International Symposium on Trustworthy Global Computing*, pages 51–66. Springer, 2014.

11 Jan Bessai, Boris Düdder, Andrej Dudenhefner, Tzu-Chun Chen, and Ugo de'Liguoro. Typing classes and mixins with intersection types. *arXiv preprint*, 2015. `arXiv:1503.04911`.

12 Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.

13 David Binder, Julian Jabs, Ingo Skupin, and Klaus Ostermann. Decomposition diversity with symmetric data and codata. *Proceedings of the ACM on Programming Languages*, 4(POPL):30, 2019.

14 Richard Bird and Oege de Moor. *The Algebra of Programming*. Prentice-Hall, 1996. URL: `http://www.cs.ox.ac.uk/publications/books/algebra/`.

15 Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA*, volume 89, pages 273–280, 1989.

16 Luca Cardelli, Simone Martini, John C Mitchell, and Andre Scedrov. An extension of system f with subtyping. *Information and Computation*, 109(1-2):4–56, 1994.

17 Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985.

18 Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 2011.

19 Avik Chaudhuri. Flow: a static type checker for javascript. *SPLASH-I In Systems, Programming, Languages and Applications: Software for Humanity*, 2015.

20 Oliver Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

21 Kosta Došen. Logical constants as punctuation marks. *Notre Dame J. Formal Logic*, 30(3):362–381, June 1989. `doi:10.1305/ndjfl/1093635154`.

22 Joshua Dunfield. Elaborating intersection and union types. *Journal of Functional Programming*, 24(2-3):133–165, 2014.

23 Joshua Dunfield and Neel Krishnaswami. Bidirectional typing. *arXiv preprint*, 2019. `arXiv:1908.05839`.

24 Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ICFP*, 2013.

25 Gerhard Gentzen. Untersuchungen über das logische Schliessen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1934.

26 DeLesley S. Hutchins. Pure subtype systems. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2010, 2010.

27 Atsushi Igarashi and Mirko Viroli. On variance-based subtyping for parametric types. In *ecoop*, pages 441–469, Malaga, Spain, June 2002. sv. To appear in ACM Transactions on Programming Languages and Systems.

28 LSV Jutting. Typing in pure type systems. *Information and Computation*, 105(1):30–41, 1993.

29 Gavin King. The ceylon language specification, version 1.0, 2013.

30 Saunders Mac Lane. *Categories for the Working Mathematician.* Number 5 in Graduate Texts in Mathematics. Springer, 2 edition, 1998.

31 James McKinna and Robert Pollack. Pure type systems formalized. In *International Conference on Typed Lambda Calculi and Applications*, pages 289–305. Springer, 1993.

32 Fabian Muehlboeck and Ross Tate. Empowering union and intersection types with integrated subtyping. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):112, 2018.

33 Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic (TOCL)*, 9(3):23, 2008.

34 Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language, 2004.

35 Klaus Ostermann and Julian Jabs. Dualizing generalized algebraic data types by matrix transposition. In *European Symposium on Programming*, pages 60–85. Springer, 2018.

36 Benjamin C Pierce. Programming with intersection types, union types, and polymorphism, 2002.

37 Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1), 2000.

38 Dag Prawitz. Proofs and the meaning and completeness of the logical constants. In *Essays on Mathematical and Philosophical Logic. Synthese Library (Studies in Epistemology, Logic, Methodology, and Philosophy of Science)*, 1979.

39 John C Reynolds. Preliminary design of the programming language forsythe, 1988.

40 Tiark Rompf and Nada Amin. Type soundness for dependent object types (dot). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, 2016.

41 Paula Severi and Erik Poll. Pure type systems with definitions. In *International Symposium on Logical Foundations of Computer Science*, pages 316–328. Springer, 1994.

42 Alex K Simpson. The proof theory and semantics of intuitionistic modal logic, 1994.

43 Martin Steffen and Benjamin Pierce. Higher-order subtyping, 1994.

44 Tarmo Uustalu and Varmo Vene. Comonadic notions of computation. *Electronic Notes in Theoretical Computer Science*, 203(5):263–284, 2008.

45 LS van Benthem Jutting, James McKinna, and Robert Pollack. Checking algorithms for pure type systems. In *International Workshop on Types for Proofs and Programs*, pages 19–61. Springer, 1993.

46 Yanpeng Yang and Bruno C. d. S. Oliveira. Unifying typing and subtyping. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):47, 2017.

47 Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. Julia subtyping: a rational reconstruction. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):113, 2018.

48 Jan Zwanenburg. Pure type systems with subtyping. In *International Conference on Typed Lambda Calculi and Applications*, pages 381–396. Springer, 1999.