Distributing Intersection and Union Types with Splits and Duality (Functional Pearl)

XUEJING HUANG, The University of Hong Kong, China
BRUNO C. D. S. OLIVEIRA, The University of Hong Kong, China

Subtyping with intersection and union types is nowadays common in many programming languages. From the perspective of logic, the subtyping problem is essentially the problem of determining *logical entailment*: does a logical statement follow from another one? Unfortunately, algorithms for deciding subtyping and logical entailment with intersections, unions and various distributivity laws can be highly non-trivial.

This functional pearl presents a novel algorithmic formulation for subtyping (and logical entailment) in the presence of various distributivity rules between intersections, unions and implications (i.e. function types). Unlike many existing algorithms which first normalize types and then apply a subtyping algorithm on the normalized types, our new subtyping algorithm works directly on source types. Our algorithm is based on two recent ideas: a generalization of subtyping based on the duality of language constructs called *duotyping*; and *splittable types*, which characterize types that decompose into two simpler types. We show that our algorithm is sound, complete and decidable with respect to a declarative formulation of subtyping based on the *minimal relevant logic* **B+**. Moreover, it leads to a simple and compact implementation in under 50 lines of functional code.

CCS Concepts: • **Theory of computation → Type theory**; • **Software and its engineering → Data types and structures**.

Additional Key Words and Phrases: intersection types, union types, subtyping, distributivity

ACM Reference Format:
Xuejing Huang and Bruno C. d. S. Oliveira. 2021. Distributing Intersection and Union Types with Splits and Duality (Functional Pearl). *Proc. ACM Program. Lang.* 5, ICFP, Article 89 (August 2021), 24 pages. https://doi.org/10.1145/3473594

1 INTRODUCTION

Intersection and union types [Barbanera et al. 1995; Coppo and Dezani-Ciancaglini 1980; Coppo et al. 1980] are nowadays common in many programming languages and calculi. Mainstream OOP programming languages that employ union and intersection types include the upcoming Scala 3 [team and community contributors 2020] (based on the DOT calculus [Amin et al. 2016]), TypeScript [Microsoft 2012], Ceylon [Redhat 2011] and Flow [Facebook 2014]. ℂDuce [Benzaken et al. 2003], a language for XML processing, also supports union and intersection types with a very expressive subtyping relation. The Julia language [Bezanson et al. 2017], while not supporting intersection types, supports union types as well as interesting distributivity rules between unions and tuples [Zappa Nardelli et al. 2018].

Subtyping relations for union and intersection types can vary in expressive power. Some subtyping relations include distributivity rules between intersection and/or union types and other

Authors' addresses: Xuejing Huang, The University of Hong Kong, China, xjhuang@cs.hku.hk; Bruno C. d. S. Oliveira, The University of Hong Kong, China, bruno@cs.hku.hk.

2475-1421/2021/8-ART89
https://doi.org/10.1145/3473594

constructs. Languages that have some form of distributivity rules include Ceylon, ℂDuce, Julia and Scala 3 (or Dotty). A typical example is the well-known rule in the subtyping relation of Barendregt, Coppo, and Dezani-Ciancaglini [1983] (BCD subtyping), which distributes intersections over arrows.

$$\text{S-distArrR}\quad \frac{}{(A \to B_1) \wedge (A \to B_2) \leq A \to B_1 \wedge B_2}$$

In this rule, the intersection of two function types with the same input A is a subtype of a function type whose input is also A, and the output is the intersection of the output types of the two functions. Moreover, intersection and union types can distribute over each other:

$$\text{S-distOr}\quad \frac{}{(A_1 \vee B) \wedge (A_2 \vee B) \leq (A_1 \wedge A_2) \vee B}$$

Subtyping relations with intersections and unions have deep connections to logic, which follow from the *Curry-Howard isomorphism* [Howard 1980]. Types can be interpreted as propositions: intersections are interpreted as conjunctions, unions as disjunctions and functions types as implications. Furthermore, from the perspective of logic, the subtyping problem is essentially the problem of determining *logical entailment*: does a logical statement follow from another one? Where in logic one may write $P \vdash Q$ for logical entailment, with subtyping one writes $P <: Q$ to denote that Q is a supertype of (or follows from) P. Naturally, algorithms for deciding logical entailment have applications to other areas, such as theorem proving [Stolze 2019]. One particular subtyping relation of practical interest for programming languages is closely related to the basic positive logic **B+** of Routley and Meyer [1972]. The connection to programming languages is due to van Bakel et al. [2000], who have shown a type assignment system that corresponds to the **B+** logic. Logic **B+** is also called the *minimal relevant logic* since it is the minimal (or the weakest) relevant logic system that is complete for the Routley-Meyer ternary relational semantics. In the minimal relevant logic **B+**, there are two axioms that can be interpreted as the two distributivity rules above. The subtyping relations used in Ceylon and ℂDuce, for instance, include all the rules of the minimal relevant logic.

Existing subtyping algorithms that deal with distributivity rules often depend on a normalization pre-procedure, which rewrites all types into a normal form (such as the disjunctive or conjunctive normal form). Only after this step, the normalized types are compared for subtyping. Normalization is used for instance by Frisch et al. [2008] and ℂDuce, by some *flow-typing algorithms* [Pearce 2013], by the integrated subtyping framework of Muehlboeck and Tate [2018], and by the Delta-calculus [Stolze 2019]. While normalization is a well-known approach, it has some drawbacks: there is a high space cost, and the two-step approach can be quite involved both in terms of implementation and metatheory.

This pearl presents a novel algorithmic formulation of a powerful subtyping relation with union and intersection types that is based on the minimal relevant logic. Unlike many normalization-based algorithms, our new subtyping algorithm works directly on source types. In other words, there is only one step in our algorithm without any pre-processing phase. Instead of deriving an algorithmic formulation from first principles, we employ two recent ideas. The first idea is to deal with distributivity rules with *splittable types* [Huang et al. 2021]. This idea was proposed recently for BCD subtyping, which is a subtyping relation with intersection types (but no union types) and the rule S-distArrR. In this pearl, we show that the idea of splittable types generalizes to union types and other distributivity rules, such as the rule S-distOr. The second idea is to employ *duotyping* [Oliveira et al. 2020], which provides a generalization of subtyping with a mode. This mode allows exploiting fundamental dualities between union and intersection types and their

subtyping rules. This leads to a consistent and symmetrical design for the rules, and also benefits both metatheory and implementation.

We take an incremental tutorial-style approach in the presentation of this paper, starting from a simple subtyping relation that is extended with distributivity rules using splittable types in Section 2. Then, the formulation for subtyping based on minimal relevant logic is presented in Section 3. Such formulation is derived straightforwardly from the duotyping formulation discussed in Section 4.2. All the results presented in this paper – including soundness/completeness, decidability and transitivity – are mechanically formalized in the Coq theorem prover. Furthermore, we discuss implementation considerations in detail and show that our algorithmic formulation leads to a simple and compact Haskell implementation under 50 lines of code in Section 5.

In this pearl, we present the concepts and ideas from the perspective of subtyping. Nevertheless, the algorithms shown here can also be used for deciding logical entailment of propositional formulas and can be applied to other domains, such as theorem proving.

2 CHALLENGES OF DISTRIBUTIVITY AND BACKGROUND

This section discusses the challenges of distributivity rules for subtyping algorithms, and then provides background on the idea of *splittable types* for addressing such challenges in the simpler setting of BCD subtyping [Barendregt et al. 1983]. We follow the convention that intersections have a higher precedence than unions, and arrows have the lowest precedence.

2.1 The Challenges of Distributivity for Algorithmic Subtyping

To design an algorithmic system for the declarative subtyping relation (discussed later in Section 3), the key challenge is to eliminate the transitivity rule. Before we move on to our algorithmic formulation of the subtyping relation, we discuss why distributivity is challenging. Namely, we first discuss how the explicit transitivity rule is avoided in a simple subtyping relation with intersection types and no distributivity, and discuss where this approach fails when a distributive law is included.

Conventional subtyping with intersection types. We start with a simple, and standard algorithmic formulation of subtyping with intersections, shown at the right of Figure 1. The subtyping formulation on the left is an equivalent system whose rules are a subset of the declarative subtyping that we will study later.

In the declarative system, rule S-AND, rule S-ANDL and rule S-ANDR are about intersections. Rule S-TOP states that the top type ($\top$) is the upper bound of all types. The remaining rules are standard. According to rule S-AND, a common subtype of two types is also a subtype of their intersection. An intersection itself is a subtype of another type B if one of the types in the intersection is a subtype of B, by rule S-ANDL and rule S-ANDR (with the help of rule S-TRANS). The rules are straightforward, especially when the subtyping judgement is interpreted as set inclusion and the intersection of two types is, therefore, the intersection of two sets. From the coercive subtyping point of view, a term of an intersection type $A \wedge B$ can be directly converted into a term of A or a term of B.

Compared with the declarative rules, the algorithmic system has no explicit transitivity rule, and the reflexivity rule is specialized to the primitive type *Int*. Reflexivity is straightforward to obtain for any form of types, including the top type, function types, and intersection types. Among the three rules for intersections, only rule CS-ANDL and rule CS-ANDR are changed, which can be viewed as the rule S-ANDL and the rule S-ANDR with transitivity built-in.

In this case, such an algorithmic formulation follows from a common strategy for transitivity elimination: pushing transitivity into other rules.

Adding distributivity: the simple approach to transitivity elimination fails. The rules in Figure 1 are quite standard and employed in several subtyping relations. Now, let us assume that we are

$$\textit{Type} \qquad A, B, C ::= \textit{Int} \mid A \to B \mid A \wedge B \mid \top$$

$\boxed{A \leq B}$ *(Declarative Subtyping)*

$$\textsc{S-refl}\ \frac{}{A \leq A} \qquad \textsc{S-trans}\ \frac{A \leq B \quad B \leq C}{A \leq C} \qquad \textsc{S-top}\ \frac{}{A \leq \top}$$

$$\textsc{S-arrow}\ \frac{B_1 \leq A_1 \quad A_2 \leq B_2}{A_1 \to A_2 \leq B_1 \to B_2} \qquad \textsc{S-and}\ \frac{A \leq B_1 \quad A \leq B_2}{A \leq B_1 \wedge B_2}$$

$$\textsc{S-andL}\ \frac{}{A_1 \wedge A_2 \leq A_1} \qquad \textsc{S-andR}\ \frac{}{A_1 \wedge A_2 \leq A_2}$$

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

$$\textsc{CS-int}\ \frac{}{\textit{Int} <: \textit{Int}} \qquad \textsc{CS-arrow}\ \frac{B_1 <: A_1 \quad A_2 <: B_2}{A_1 \to A_2 <: B_1 \to B_2}$$

$$\textsc{CS-top}\ \frac{}{A <: \top} \qquad \textsc{CS-and}\ \frac{A <: B_1 \quad A <: B_2}{A <: B_1 \wedge B_2}$$

$$\textsc{CS-andL}\ \frac{A_1 <: B}{A_1 \wedge A_2 <: B} \qquad \textsc{CS-andR}\ \frac{A_2 <: B}{A_1 \wedge A_2 <: B}$$

Fig. 1. The conventional algorithmic subtyping with intersections (without distributivity), compared with the declarative rules.

going to add the distributivity of arrows over intersections to the declarative system.

$$\textsc{S-distArrR}\ \frac{}{(A \to B_1) \wedge (A \to B_2) \leq A \to B_1 \wedge B_2}$$

Following the same method, it seems that we can achieve the goal of obtaining an algorithmic formulation by directly pushing transitivity into the rules. That is, by adding the following two rules.

$$\textsc{CS-distArrR-sub}\ \frac{C <: (A \to B_1) \wedge (A \to B_2)}{C <: A \to (B_1 \wedge B_2)} \qquad \textsc{CS-distArrR-super}\ \frac{A \to (B_1 \wedge B_2) <: C}{(A \to B_1) \wedge (A \to B_2) <: C}$$

Rule CS-distArrR-sub enables, for instance, arrows with a different input type to unify first, and then applying distributivity on the result:

$$\textsc{CS-distArrR-sub}\ \frac{\textsc{CS-and}\ \dfrac{\textsc{CS-andL}\ \dfrac{\textsc{CS-arrow}\ \dfrac{\textsc{CS-andL}\ \dfrac{A_1 <: A_1}{A_1 \wedge A_2 <: A_1} \quad B_1 <: B_1}{A_1 \to B_1 <: A_1 \wedge A_2 \to B_1}}{(A_1 \to B_1) \wedge (A_2 \to B_2) <: A_1 \wedge A_2 \to B_1} \quad \textsc{CS-andR}\ \dfrac{\ldots}{\ldots <: A_1 \wedge A_2 \to B_2}}{(A_1 \to B_1) \wedge (A_2 \to B_2) <: (A_1 \wedge A_2 \to B_1) \wedge (A_1 \wedge A_2 \to B_2)}}{(A_1 \to B_1) \wedge (A_2 \to B_2) <: A_1 \wedge A_2 \to B_1 \wedge B_2}$$

For simplification, we end at reflexivity in the derivation above. Meanwhile, rule CS-distArrR-super can apply distributivity to nested arrows:

$$\textsc{CS-distArrR-super}\ \dfrac{\textsc{CS-arrow}\ \dfrac{A <: A \qquad \dfrac{}{B \to C_1 \wedge C_2 <: B \to C_1 \wedge C_2}\ \textsc{CS-distArrR-super}}{A \to (B \to C_1) \wedge (B \to C_2) <: A \to B \to C_1 \wedge C_2}}{(A \to B \to C_1) \wedge (A \to B \to C_2) <: A \to B \to C_1 \wedge C_2}$$

Unfortunately, that is not enough. Transitivity can extend to both sides of the distributivity rule together. Consequently, the declarative system (Figure 4 extended by rule S-distArrR) can apply distributivity to nested arrows with different input types, as the following example shows.

$$\begin{array}{ll}
(A_1 \to B \to C_1) \wedge (A_2 \to B \to C_2) & \\
\le (A_1 \wedge A_2 \to B \to C_1) \wedge (A_1 \wedge A_2 \to B \to C_2) & \text{by rule S-and, rule S-arrow, and other rules} \\
\le A_1 \wedge A_2 \to (B \to C_1) \wedge (B \to C_2) & \text{by rule S-distArrR} \\
\le A_1 \wedge A_2 \to B \to C_1 \wedge C_2 & \text{by rule S-arrow with rule S-distArrR}
\end{array}$$

An attempt at providing an equivalent algorithmic formulation with rule CS-distArrR-sub and rule CS-distArrR-super fails to accept such subtyping statement. The derived result only matches rule CS-andL and rule CS-andR, but both of them drop part of the subtype, making it impossible to reach the result. In other words, we cannot extend the system directly in this way, without losing expressive power.

2.2 Background: Ordinary Types and Splittable Types for BCD Subtyping

Luckily there are other ways for obtaining transitivity elimination while adding distributivity. Here we review the idea of splittable types for BCD subtyping [Huang et al. 2021]. The rules presented in Figure 2 are equivalent to the declarative rules for the conventional subtyping in Figure 1 plus the distributivity rule (rule S-distArrR). For simplicity of presentation, we ignore the peculiar rule $\top \le \top \to \top$ in BCD subtyping here.

Ordinary types. Conventionally, types which are not intersections are called *ordinary types*. Originally, all arrow types are considered as ordinary types [Davies and Pfenning 2000]. However, in the presence of rule S-distArrR, function types like $A \to B_1 \wedge B_2$ may behave as if they were an intersection type. $A \to B_1 \wedge B_2 \le (A \to B_1) \wedge (A \to B_2)$ is already derivable without distributivity. Adding rule S-distArrR then makes the two types isomorphic. That is to say, it breaks the boundary of intersection rules and rules for ordinary types. In other words, the arrow type $A \to B_1 \wedge B_2$ can be converted into the intersection type $(A \to B_1) \wedge (A \to B_2)$ without losing any information. Since intersections are not ordinary types, it makes sense not to consider such (intersection-isomorphic) arrow types as ordinary types either, and restrict the arrow rule to ordinary arrow types. This leads to the definition of ordinary types presented in Figure 2, where A° denotes that type A is ordinary.

Splittable types. The type splitting relation $B \lhd A \rhd C$ denotes that the type in the middle can be split into two. Naturally, an intersection type can be decomposed into two parts: $A \lhd A \wedge B \rhd B$ by rule BSp-and in Figure 2. The relation extends the decomposition of intersections to types that are equivalent to intersections via distributivity rules. Specifically, according to rule S-distArrR, we know that an arrow type can split if its result type can (rule BSp-arrow). Such types are called splittable types. Every type is either ordinary or splittable and cannot be both at the same time. The outputs of the split operation $B \lhd A \rhd C$ represent an intersection type $B \wedge C$ that is isomorphic to A, therefore type splitting is always lossless.

$\boxed{A^{\circ}}$ *(Ordinary Types for BCD)*

$$\textsc{BO-top}\ \frac{}{\top^{\circ}} \qquad \textsc{BO-int}\ \frac{}{Int^{\circ}} \qquad \textsc{BO-arrow}\ \frac{B^{\circ}}{(A \to B)^{\circ}}$$

$\boxed{B \lhd A \rhd C}$ *(Splittable Types for BCD)*

$$\textsc{BSp-and}\ \frac{}{A \lhd A \land B \rhd B} \qquad \textsc{BSp-arrow}\ \frac{C \lhd B \rhd D}{A \to C \lhd A \to B \rhd A \to D}$$

$\boxed{A <:_a B}$ *(Modular BCD Subtyping)*

$$\textsc{BS-int}\ \frac{}{Int <:_a Int} \qquad \textsc{BS-top}\ \frac{}{A <:_a \top} \qquad \textsc{BS-arrow}\ \frac{B_1 \to B_2{}^{\circ} \quad B_1 <:_a A_1 \quad A_2 <:_a B_2}{A_1 \to A_2 <:_a B_1 \to B_2} \qquad \textsc{BS-and}\ \frac{B_1 \lhd B \rhd B_2 \quad A <:_a B_1 \quad A <:_a B_2}{A <:_a B}$$

$$\textsc{BS-andL}\ \frac{B^{\circ} \quad A_1 <:_a B}{A_1 \land A_2 <:_a B} \qquad \textsc{BS-andR}\ \frac{B^{\circ} \quad A_2 <:_a B}{A_1 \land A_2 <:_a B}$$

Fig. 2. The algorithmic BCD subtyping with intersection types and distributivity.

The modular BCD subtyping algorithm. The main idea for the algorithmic formulation of subtyping, shown at the bottom of Figure 2, is that the right-hand side type B keeps splitting until it becomes ordinary. When it splits, rule BS-AND is applied, which works in the same way as rule CS-AND when B is an intersection type. The most interesting case is when B is a splittable function type, for example, $B := B_1 \to B_2 \land B_3$. Type B can be split into $B_1 \to B_2$ and $B_1 \to B_3$. Therefore, the premises of $A <:_a B$ are $A <:_a B_1 \to B_2$ and $A <:_a B_1 \to B_3$, or equivalently, $A <:_a (B_1 \to B_2) \land (B_1 \to B_3)$. Thus we are able to conclude $A <:_a B$ with a combination of rule S-TRANS and rule S-DISTARRR in declarative subtyping. That is to say, while rule BS-AND combines rule S-TRANS and rule S-AND, it also takes rule S-DISTARRR into consideration implicitly. The ordinary-type conditions in gray eliminate some overlapping between the rules: we can see that when B is splittable, only rule BS-AND can be applied, since rule BS-ANDL and rule BS-ANDR require B to be ordinary. However, dropping such conditions does not alter the expressive power: it leads to an equivalent system (but with more overlapping).

The previous failed example can now be derived, as its right-hand side type, although not matched by rule S-DISTARRR, is captured by type splitting. Due to space limitations, we omit the type $(A_1 \to B \to C_1) \land (A_2 \to B \to C_2)$, which is unchanged across the application of rule BS-AND, in its premises. We also employ the rules without the ordinary-type conditions in the derivation for simplification. The main derivation is:

$$\textsc{BS-and}\ \frac{\textsc{BS-andL}\ \dfrac{\textsc{BS-arrow}\ \dfrac{\ldots}{A_1 \to B \to C_1 <:_a A_1 \land A_2 \to B \to C_1}}{\ldots <:_a A_1 \land A_2 \to B \to C_1} \qquad D \qquad \textsc{BS-andR}\ \dfrac{\textsc{BS-arrow}\ \dfrac{\ldots}{A_2 \to B \to C_2 <:_a A_1 \land A_2 \to B \to C_2}}{\ldots <:_a A_1 \land A_2 \to B \to C_2}}{(A_1 \to B \to C_1) \land (A_2 \to B \to C_2) <:_a A_1 \land A_2 \to B \to C_1 \land C_2}$$

```
-- ordinary type
ordinary :: Type → Bool
ordinary a = split a == Nothing

-- split type
split :: Type → Maybe (Type, Type)
split (TAnd a b) = Just (a, b)                          -- Bsp-and
split (TArrow a b)                                      -- Bsp-arrow
  | Just (b1, b2) <- split b
  = Just (TArrow a b1, TArrow a b2)
split _ = Nothing

-- subtyping
checkSub :: Type → Type → Bool
checkSub TInt TInt = True                               -- BS-int
checkSub _ TTop    = True                               -- BS-top
checkSub a b                                            -- BS-and
  | Just (b1, b2) <- split b
  = checkSub a b1 && checkSub a b2
checkSub (TAnd a1 a2) b                                 -- BS-andL BS-andR
  = checkSub a1 b || checkSub a2 b
checkSub (TArrow a1 a2) (TArrow b1 b2)                  -- BS-arrow
  = checkSub b1 a1 && checkSub a2 b2
checkSub _ _ = False
```

Fig. 3. Haskell implementation of BCD subtyping.

The missing subderivation D for type splitting is:

$$\textsc{BSp-arrow}\ \frac{\textsc{BSp-arrow}\ \dfrac{\textsc{BSp-and}\ \dfrac{}{C_1 \lhd C_1 \wedge C_2 \rhd C_2}}{B \to C_1 \lhd B \to C_1 \wedge C_2 \rhd B \to C_2}}{A_1 \wedge A_2 \to B \to C_1 \lhd A_1 \wedge A_2 \to B \to C_1 \wedge C_2 \rhd A_1 \wedge A_2 \to B \to C_2}$$

2.3 Implementation

Finally, we present an Haskell implementation of the rules in Figure 3. We model types as:

```
data Type = TInt | TTop | TArrow Type Type | TAnd Type Type
```

There are four constructors for *Int*, ⊤, arrow types, and intersection types respectively. A type is either ordinary or splittable. The `split` function in Figure 3 is based on the definition of the type splitting relation. It returns the split results if the input type is splittable. The `ordinary` function makes use of the fact that the set of ordinary types is complementary to the set of splittable types. It can also be implemented by analyzing the form of the type.

The `checkSub` function takes two types and decides whether the first input is a subtype of the second one. The first two cases correspond to rule BS-int and rule BS-top. Then the case corresponding to rule BS-and handles all cases of which the second input is splittable. Since the code executes sequentially, the second input is guaranteed to be ordinary after that. Following are the cases corresponding to rule BS-andL and rule BS-andR. When the first input is an intersection

$\boxed{A \leq B}$ *(Declarative Subtyping Extension)*

$$\textsc{S-bot}\ \frac{}{\bot \leq A} \qquad \textsc{S-or}\ \frac{A_1 \leq B \qquad A_2 \leq B}{A_1 \vee A_2 \leq B} \qquad \textsc{S-orL}\ \frac{}{B_1 \leq B_1 \vee B_2} \qquad \textsc{S-orR}\ \frac{}{B_2 \leq B_1 \vee B_2}$$

$$\textsc{S-distArrR}\ \frac{}{(A \to B_1) \wedge (A \to B_2) \leq A \to B_1 \wedge B_2} \qquad \textsc{S-distArrR-rev}\ \frac{}{A \to B_1 \wedge B_2 \leq (A \to B_1) \wedge (A \to B_2)}$$

$$\textsc{S-distArrL}\ \frac{}{(A_1 \to B) \wedge (A_2 \to B) \leq A_1 \vee A_2 \to B} \qquad \textsc{S-distArrL-rev}\ \frac{}{A_1 \vee A_2 \to B \leq (A_1 \to B) \wedge (A_2 \to B)}$$

$$\textsc{S-distOr}\ \frac{}{(A_1 \vee B) \wedge (A_2 \vee B) \leq (A_1 \wedge A_2) \vee B} \qquad \textsc{S-distAnd}\ \frac{}{(A_1 \vee A_2) \wedge B \leq (A_1 \wedge B) \vee (A_2 \wedge B)}$$

Fig. 4. Declarative subtyping rules (extends the left part of Figure 1).

type, it is necessary to try both rules before returning **False**. In the end, both types must be arrow types and must satisfy rule BS-arrow if the subtyping holds.

Roadmap. For the rest of the paper we will see how the ideas of splittable types can be extended into a more complex setting with union types and additional distributivity rules. Then, using another technique called duotyping, we will exploit the fundamental dualities between intersection and union types to further unify the rules in the system. Our ultimate goal is the implementation given in Section 5, which exploits splittable types and duality to obtain a compact functional implementation of a subtyping algorithm for the setting with union types and additional distributivity rules. Along the way, we show various results regarding the metatheory of the system, including soundness, completeness and decidability.

3 SUBTYPING BASED ON MINIMAL RELEVANT LOGIC

In this section, we show two equivalent subtyping relations (one declarative and another algorithmic) for a variant of minimal relevant logic subtyping [Routley and Meyer 1972; van Bakel et al. 2000]. The main novelty over the BCD subtyping relation presented in Section 2 are the addition of union types, and extra distributivity rules. We show that the idea of splittable types smoothly extends to deal with those features. The algorithmic formulation is derived from a formulation based on duotyping that will be presented in Section 4, but the presentation in this section is understandable independently of duotyping.

3.1 Declarative subtyping

Denoted by meta-variables A and B, types include the integer type Int, function types $A \to B$, intersection $(A \wedge B)$ and union $(A \vee B)$ types, as well as the top $(\top)$ and bottom $(\bot)$ types.

$$Type \qquad A, B ::= Int \mid A \to B \mid A \wedge B \mid A \vee B \mid \top \mid \bot$$

The declarative subtyping rules in Figure 4 extend the rules on the left of Figure 1. We describe the new rules next.

(1) Rule S-bot, similarly to rule S-top, defines the bottom type as the lowermost bound among types. $\bot$ has no inhabited values. In other words, it is the empty type from the set-theoretic

view of types. In a system where arrow types are interpreted as logical implications, the bottom type can be used to encode negation as $A \to \bot$.

(2) Rule S-OR, rule S-ORL, and rule S-ORR define basic subtyping for unions, similarly to the three intersection rules. In a language with union types, a term of type A can be transformed into any union type containing A. From the point of view of proof theory, having the proof of either A or B is enough to construct a proof of $A \vee B$. That is to say, a union type is a supertype of its components. Moreover, a union type is a subtype of some type if both its components are subtypes of that type.

(3) The remaining six rules are related to the distributivity of intersections and unions over other constructs. Rule S-DISTARRR is part of the BCD subtyping (discussed in Section 2). Along with rule S-DISTARRL, the two rules distribute arrows over intersections and unions, respectively. These two rules have two corresponding reversed rules (rule S-DISTARRR-REV and rule S-DISTARRL-REV). Note that the latter two rules are not necessary since they can be derived from other rules, but we present them here because in Section 4.1 they will play a role in our reformulation of the subtyping relation using *duotyping* [Oliveira et al. 2020]. Rule S-DISTARRR-REV and rule S-DISTARRL-REV, in combination with the previous two rules, illustrate that the two types in the subtyping relation are isomorphic (i.e. they are subtypes of each other).

(4) The interaction between intersection types and union types is described by the rule S-DISTOR and the rule S-DISTAND. They can distribute over each other. The reversed rules are derivable and therefore omitted. To be noted, it would not affect the whole system to drop one of the two rules (either rule S-DISTOR or rule S-DISTAND): in the presence of one of the two rules, the other rule can be derived by the other subtyping rules.

Remark. It is should be noted that, in combination with transitivity, more general subtyping rules become derivable. For instance, an intersection of any two arrow types has a supertype that is a combination of them:

$$\frac{}{(A_1 \to A_2) \wedge (B_1 \to B_2) \leq A_1 \wedge B_1 \to A_2 \wedge B_2}\ \text{S-DISTARR-GEN}$$

In other words, rule S-DISTARR-GEN is not restricted to types that share the same input type (as the rule S-DISTARRR). Similarly, the subtype of an intersection of two arrow types can be obtained from an arrow type that takes the union of the input types of the arrow types:

$$\frac{}{A_1 \vee B_1 \to A_2 \wedge B_2 \leq (A_1 \to A_2) \wedge (B_1 \to B_2)}\ \text{S-DISTARR-REV-GEN}$$

3.2 Algorithmic Subtyping: Adding Union Types and More Distributivity

Now we are ready to move on to the design of algorithmic subtyping for Figure 4. We first cover the extended definitions of ordinary and splittable types here, in which the initial definition is revised, and a dual version is defined for union types.

Intersection-ordinary and intersection-splittable types. With union types and the bottom type taken into consideration, we need to revise the previous definitions. Firstly we rename them as *intersection-ordinary types* (A°, top of Figure 5) and *intersection-splittable types* ($B \lhd A \rhd C$, in the middle of Figure 5). Originally, "ordinary" was used to describe the lack of "intersections", and "splittable" meant a type that has two parts connected by an intersection. Now we make this explicit in the names. Intersection-ordinary types include (non-splittable) union types, but exclude intersection types at the top level (although intersection types can appear in some nested positions

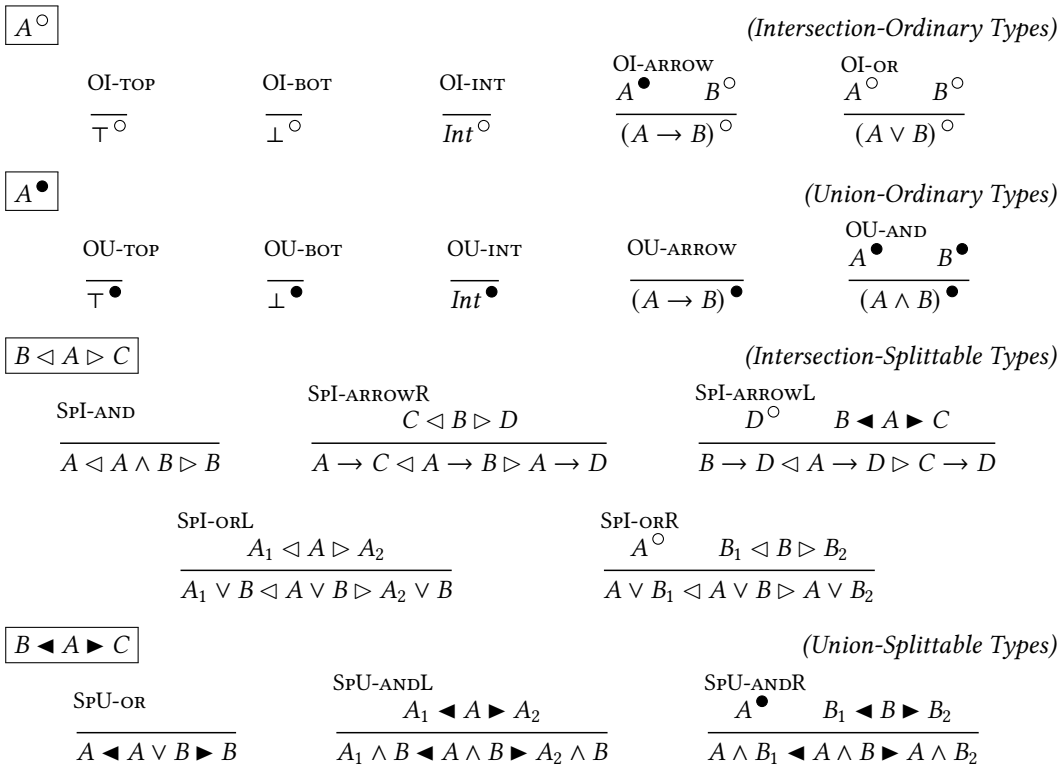

Fig. 5. Ordinary and splittable types.

inside the types). In contrast, intersection-splittable types include top-level intersections and some union types. Mainly, there are four changes in the new definition.

(1) The bottom type is ordinary like the top type.
(2) Due to the distributivity of union over intersections (rule S-DISTOR and rule S-DISTAND), some unions are also isomorphic to intersections, for example the type $(A_1 \wedge A_2) \vee B$ is isomorphic to $(A_1 \vee B) \wedge (A_2 \vee B)$. That is to say, the former can be split into $A_1 \vee B$ and $A_2 \vee B$. Splitting the union type $A \vee B$, first tries to split A by rule SPI-ORL, and only moves to B if A cannot be split (rule SPI-ORR). Thus only unions whose components are both ordinary can be treated as ordinary types (rule OU-AND).
(3) Rule SPI-ARROWL is brought by rule S-DISTARRL and rule S-DISTARRL-REV: if its input type is a union, an arrow type can be converted into an intersection type, and therefore it is not ordinary. In the ordinary rule for arrow types, the new version has more restrictions: the input type of an intersection-ordinary arrow type must not be a union-like type, i.e. it is *a union-ordinary type* (Figure 5).
(4) To be noted, a side condition D° is added in rule SPI-ARROWL to ensure that the relation can be used as a deterministic function, like in rule SPI-ORR. Although we prioritize rule SPI-ORL and rule SPI-ARROWR here, we believe some other arrangements are also feasible.

Union-ordinary and union-splittable types. Figure 5 also presents the definition of union-ordinary types ($A^{\bullet}$) and union-splittable types ($B \blacktriangleleft A \blacktriangleright C$). A union-splittable type is isomorphic to the

union of its split results, and union-ordinary types are those that cannot be split. Such a definition is almost the dual of the intersection- rules: just exchange intersection and union, and switch intersection- and union- judgments.

The key difference is that no arrow types are union-splittable and they are all union-ordinary. Correspondingly, splitting union types lacks arrow related rules, and rule OU-arrow has no premise. The source of the difference is that both of the two distributivity arrow rules in the declarative system (rule S-distArrR and rule S-distArrL) relate arrow types with intersection types but not union types. To have an exact dual, the union-ordinary and union-splittable types definitions would lead to the following two rules:

S-distArrR-union

$$\frac{}{A \to B_1 \vee B_2 \leq (A \to B_1) \vee (A \to B_2)}$$

S-distArrL-union

$$\frac{}{A_1 \wedge A_2 \to B \leq (A_1 \to B) \vee (A_2 \to B)}$$

However, such rules do not lead to valid coercions from the coercive subtyping point of view where unions are interpreted as sums and intersections are interpreted as products. If we added these two rules, then some arrow types could be union-splittable.

While a type must be either intersection- (union-) ordinary or intersection- (union-) splittable, the two sets of ordinary (and splittable) definitions are overlapping. The following table presents some examples of each of the four kinds of types:

	Intersection-	Union-
Ordinary types	Int°, $\top^{\circ}$, $\bot^{\circ}$	$Int^{\bullet}$, $\top^{\bullet}$, $\bot^{\bullet}$
	$Bool \wedge String \to Int \vee Char^{\circ}$	$Bool \wedge String \to Int \vee Char^{\bullet}$
	$Int \vee Char^{\circ}$	$Int \wedge Char^{\bullet}$
	~~$A \wedge B^{\circ}$~~	~~$A \vee B^{\bullet}$~~
	~~$Int \to A \wedge B^{\circ}$~~	$Int \to A \vee B^{\bullet}$
Splittable types	$A \lhd A \wedge B \rhd B$	$A \blacktriangleleft A \vee B \blacktriangleright B$
	$A \vee Int \lhd (A \wedge B) \vee Int \rhd B \vee Int$	$A \wedge B \blacktriangleleft (A \wedge B) \vee Int \blacktriangleright Int$
	$A \vee B \lhd (A \vee B) \wedge Int \rhd Int$	$A \wedge Int \blacktriangleleft (A \vee B) \wedge Int \blacktriangleright B \wedge Int$
	$Int \to A \lhd Int \to A \wedge B \rhd Int \to B$	~~$Int \to A \blacktriangleleft Int \to A \vee B \blacktriangleright Int \to B$~~

The examples using a ~~strikeout~~ font represent negative examples: that is types that do not conform to the definition. For instance, in the first cell, $Int \vee Char$ is intersection-ordinary, while $A \wedge B$ is not. Types that are ordinary from both perspectives include Int, $\top$, $\bot$, and all intersection-ordinary arrow types. Such arrow types can contain intersections in negative positions, or unions in positive positions, like $Bool \wedge String \to Int \vee Char$. In contrast, only some union types and intersection types are both intersection- or union- splittable, as demonstrated by the second and third lines in the cell of the splittable types. Once we have $A_1 \lhd A \rhd A_2$ and $B_1 \blacktriangleleft A \blacktriangleright B_2$, we know that A is isomorphic to $A_1 \wedge A_2$ and $B_1 \vee B_2$. Via the subtyping rules in Figure 4, we can obtain $A_i \leq B_j$ $(i, j = 1, 2)$. The last examples for splittable types correspond to the last examples for ordinary types. They emphasize the dissymmetry between intersection-splittable types and union-splittable types, while other examples highlight the symmetric part.

Algorithmic subtyping. Compared to the modular BCD subtyping in Figure 2, we have a rule AS-bot for the bottom type, and three more rules for union-splittable types (rule AS-or, rule AS-orL, and rule AS-orR) in our algorithmic system in Figure 6. Similarly to the modular BCD subtyping, the three distributivity rules in the declarative system in Figure 4 (rule S-distArrR, rule S-distArrL,

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

$$\textsc{AS-int}\ \frac{}{Int <:_a Int} \qquad \textsc{AS-top}\ \frac{}{A <:_a \top} \qquad \textsc{AS-bot}\ \frac{}{\bot <:_a A}$$

$$\textsc{AS-arrow}\ \frac{(A_1 \to A_2)^{\circ} \quad (B_1 \to B_2)^{\circ} \quad B_1 <:_a A_1 \quad A_2 <:_a B_2}{A_1 \to A_2 <:_a B_1 \to B_2}$$

$$\textsc{AS-and}\ \frac{B_1 \lhd B \rhd B_2 \quad A <:_a B_1 \quad A <:_a B_2}{A <:_a B} \qquad \textsc{AS-andL}\ \frac{B^{\circ} \quad A_1 \lhd A \rhd A_2 \quad A_1 <:_a B}{A <:_a B}$$

$$\textsc{AS-andR}\ \frac{B^{\circ} \quad A_1 \lhd A \rhd A_2 \quad A_2 <:_a B}{A <:_a B}$$

$$\textsc{AS-or}\ \frac{A^{\circ} \quad B^{\circ} \quad A_1 \blacktriangleleft A \blacktriangleright A_2 \quad A_1 <:_a B \quad A_2 <:_a B}{A <:_a B} \qquad \textsc{AS-orL}\ \frac{A^{\circ\bullet} \quad B^{\circ} \quad B_1 \blacktriangleleft B \blacktriangleright B_2 \quad A <:_a B_1}{A <:_a B}$$

$$\textsc{AS-orR}\ \frac{A^{\circ\bullet} \quad B^{\circ} \quad B_1 \blacktriangleleft B \blacktriangleright B_2 \quad A <:_a B_2}{A <:_a B}$$

Fig. 6. Algorithmic subtyping rules.

and rule S-distOr), are covered with the rule AS-and by splitting the supertype. Besides this, rule AS-andL and rule AS-andR generalize the subtype to an intersection-splittable type, while the modular BCD subtyping uses an intersection type. That is because rule AS-arrow is restricted to only handle ordinary types with the two intersection-ordinary premises, and intersection-splittable types need help from other rules. In rule AS-orL and rule AS-orR, we use $A^{\circ\bullet}$ to denote both A° and $A^{\bullet}$. Compared to the rules for intersection-splittable types, the three rules have additional restrictions on types to avoid overlapping with them. These gray-highlighted premises divide rules into groups and make an order among them, except for rule AS-top and rule AS-bot, which can still overlap with other rules. Such gray conditions help to implement an algorithm with less backtracking, but we can remove these premises and the subtyping system would remain equivalent in terms of expressive power.

Example. Let us demonstrate the algorithmic formulation with an example, where we assume type A and type B are not splittable and omit the subderivations for ordinary types.

$$\textsc{AS-and}\ \frac{\textsc{AS-orR}\ \dfrac{\textsc{AS-int}\ \dfrac{}{Int <:_a Int} \quad D_3}{Int <:_a A \vee Int} \quad D_2}{(A \vee B) \wedge Int <:_a A \vee Int} \quad D_1 \quad \textsc{AS-andL}\ \dfrac{D_2 \quad \textsc{AS-or}\ \dfrac{\ldots}{A \vee B <:_a A \vee B}}{(A \vee B) \wedge Int <:_a A \vee B}}{(A \vee B) \wedge Int <:_a A \vee (Int \wedge B)}$$

The missing subderivation D_1 for type splitting is:

$$\textsc{SpI-orR}\ \frac{A^{\circ} \quad \textsc{SpI-and}\ \dfrac{}{Int \lhd Int \wedge B \rhd B}}{A \vee Int \lhd A \vee (Int \wedge B) \rhd A \vee B}$$

Subderivations D_2 and D_3 are:

$$\frac{}{A \vee B \lhd (A \vee B) \wedge \mathit{Int} \rhd \mathit{Int}}\ \textsc{SpI-and} \qquad \frac{}{A \blacktriangleleft A \vee \mathit{Int} \blacktriangleright \mathit{Int}}\ \textsc{SpU-or}$$

In the next section, we will see the duality of intersection and union types more clearly.

4 DUOTYPING BASED ON MINIMAL RELEVANT LOGIC

The algorithmic subtyping relation in Section 3.2 was not designed from first principles. Instead, it is derived (in a straightforward way) from another definition that exploits the duality between unions and intersections (as well as top and bottom). The approach that exploits duality is so-called *duotyping* [Oliveira et al. 2020]. The key idea is to generalize the subtyping relation with a third argument, which is the mode of the relation (subtyping or supertyping). Then many dual rules can be expressed as a single rule in the duotyping relation, and the dual rules are ensured (by construction) to be designed in a consistent way. In turn, this leads to an implementation approach that can exploit duotyping and modes, to reduce the number of cases, definitions and code. Our implementation in Section 5 will use duotyping.

This section shows the duotyping definitions from which the subtyping relations in Section 3 were derived, and discusses the respective metatheory, including transitivity as well as various soundness and completeness theorems among the different relations. In this paper, we opted to present the traditional formulation of subtyping in Section 3 first because the formulation of duotyping, while more compact, is also more abstract and can be harder to grasp than the more concrete subtyping formulation. We first reformulate declarative subtyping in terms of duotyping, and then exploit the duotyping structure found in the declarative formulation to design the algorithmic formulation.

4.1 Declarative Duotyping

Before developing the algorithmic version of duotyping we first show a duotyping version of declarative subtyping. Duotyping is particularly useful when the types in the language have multiple dual constructs, which is precisely the case here: we have both intersections and unions, as well as top and bottom types. Thus, understanding the subtyping relation from the point of view of duotyping can shed new light over the various rules used in subtyping, and give us some hints regarding the design of an algorithmic version.

The key idea of duotyping. The key idea in duotyping is to have a generalization of the subtyping relation that takes an extra mode as an argument:

$$\mathit{Mode} \qquad \diamond ::= \prec \mid \succ$$

The mode can either be subtyping ($\prec$) or supertyping ($\succ$). A duotyping judgement $A \diamond B$ can therefore be interpreted as both A is a subtype of B ($A \prec B$), and A is a supertype of B ($A \succ B$), depending on which mode is chosen. This extra mode is helpful to generalize dual rules for dual constructs.

Auxiliary functions. Before we dive into the duotyping rules, some auxiliary definitions need to be introduced. At the top of Figure 7, we restate some of the auxiliary functions as described in the original duotyping work [Oliveira et al. 2020]. For example, to combine the conventional subtyping rules for top and bottom (rule S-TOP and rule S-BOT), a function $\rceil\diamond\lceil$ parameterized by the mode $\diamond$ is necessary. The intersection and union constructors are also dual, and they can be selected using the function $(A \diamond_? B)$. Finally, there is a flip function to invert the mode.

$$\rceil \prec \lceil \;=\; \top \qquad (A \prec_? B) \;=\; A \wedge B \qquad \overline{\prec} \;=\; \succ$$
$$\rceil \succ \lceil \;=\; \bot \qquad (A \succ_? B) \;=\; A \vee B \qquad \overline{\succ} \;=\; \prec$$

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

$$\text{D-DUAL}\ \frac{B \,\overline{\diamond}\, A}{A \diamond B} \qquad \text{D-REFL}\ \frac{}{A \diamond A} \qquad \text{D-TRANS}\ \frac{A \diamond B \qquad B \diamond C}{A \diamond C} \qquad \text{D-BOUND}\ \frac{}{A \diamond \rceil \diamond \lceil} \qquad \text{D-ARROW}\ \frac{A \,\overline{\diamond}\, B \qquad C \diamond D}{A \to C \diamond B \to D}$$

$$\text{D-AND}\ \frac{A \diamond B \qquad A \diamond C}{A \diamond (B \diamond_? C)} \qquad \text{D-ANDL}\ \frac{}{(A \diamond_? B) \diamond A} \qquad \text{D-ANDR}\ \frac{}{(A \diamond_? B) \diamond B} \qquad \text{D-DISTARRR}\ \frac{}{(A \to B) \wedge (A \to C) \diamond A \to B \wedge C}$$

$$\text{D-DISTARRL}\ \frac{}{(A \to C) \wedge (B \to C) \diamond A \vee B \to C} \qquad \text{D-DISTOR}\ \frac{}{((A_1 \,\overline{\diamond}_?\, B) \diamond_? (A_2 \,\overline{\diamond}_?\, B)) \diamond ((A_1 \diamond_? A_2) \,\overline{\diamond}_?\, B)}$$

Fig. 7. Declarative duotyping and auxiliary functions (top).

Declarative duotyping. With these helper functions, the declarative duotyping relation is defined in the bottom part of Figure 7. For instance, using $\rceil \diamond \lceil$ we can capture the two subtyping rules for top and bottom with the following duotyping rule:

$$\text{D-BOUND}\ \frac{}{A \diamond \rceil \diamond \lceil}$$

Some rules (rule D-REFL, rule D-TRANS and rule D-ARROW) are direct generalizations from the original subtyping rules in Figure 1 and Figure 4, since those rules are reversible (i.e. they work in both modes). One unique rule in the duotyping formulation is the duality rule (rule D-DUAL), which transforms a subtyping judgement to a supertyping one, or vice versa. The remaining rules match with two original subtyping rules: rule D-AND, rule D-ANDL, and rule D-ANDR unify the three intersection rules and three union rules (rule S-AND, rule S-ANDL, rule S-ANDR, rule S-OR, rule S-ORL, and rule S-ORR). Rule D-DISTARRR and rule D-DISTARRL are for distributing arrows over unions and intersections. As mentioned in Section 3, their dual rules are absent, therefore cannot be further unified. The rule D-DISTOR is the generalization of rule S-DISTOR and rule S-DISTAND. Compared to the rules in Figure 4, the duotyping version is pleasingly more compact (11 rules versus 17 rules). More importantly, the dual relationship between various rules in Figure 4 is now made explicit in the formalism.

4.2 Algorithmic Duotyping

The rules for the algorithmic duotyping system are presented in Figure 8.

Ordinary and splittable types. By parametrizing over the mode $\diamond$, the two ordinary definitions in Figure 5 can be merged. Instantiated $\diamond$ by subtyping, $A^{\prec}$ defines the intersection-ordinary relation, which is the same relation as the one on the top of Figure 5. Its supertyping dual $A^{\succ}$, unsurprisingly, matches with the previous union-ordinary definition. A similar unification applies to splittable types. $B \lhd A_{\prec} \rhd C$ is for intersection-splittable types, while $B \lhd A_{\succ} \rhd C$ is for union-splittable ones. Only arrow-related rules (rule O-ARROWI, rule O-ARROWU, rule SP-ARROWR, rule SP-ARROWL)

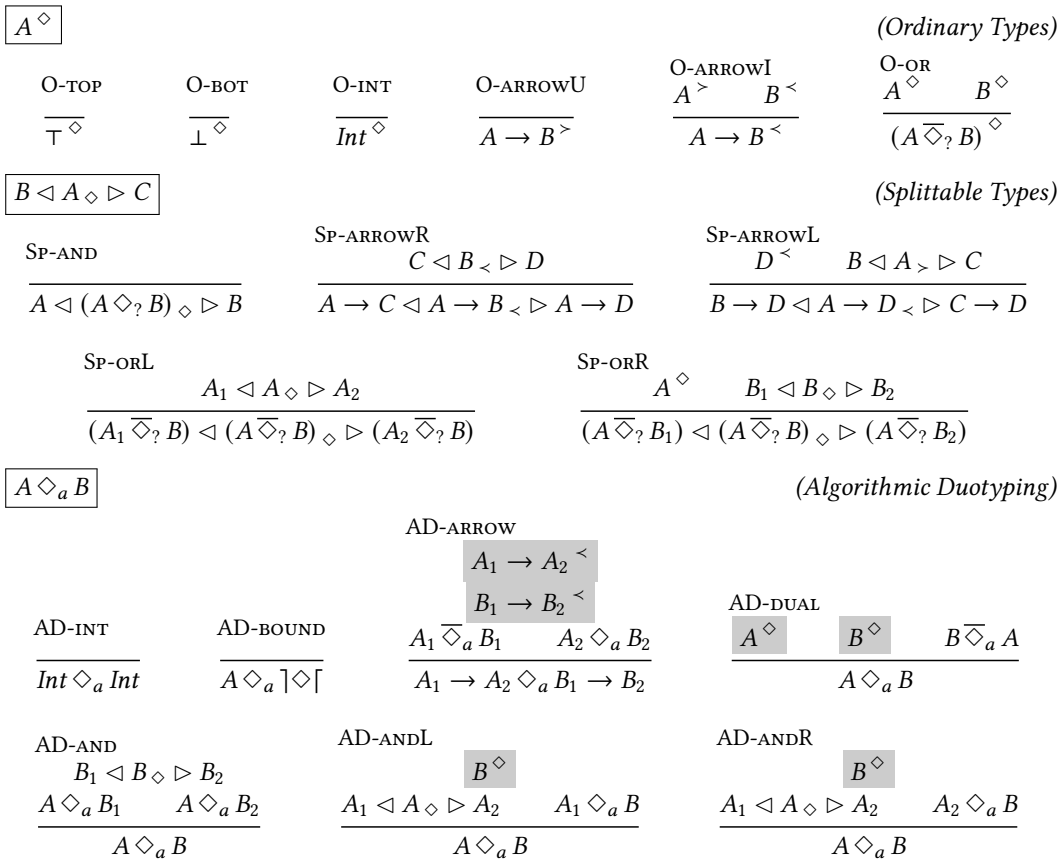

Fig. 8. Algorithmic duotyping.

are specific to a particular mode (subtyping or supertyping), but otherwise, all the other rules are generic on the mode.

Duotyping. In total there are 7 rules in the algorithmic duotyping system, compared to 10 in the subtyping system in Figure 6.

Duotyping reorganizes and unifies the rules in a more abstract style. All rules, except rule AD-INT, rule AD-ARROW and rule AD-DUAL unify a pair of subtyping rules. In essence one of the pairs is simply the result of flipping arguments and the mode. For instance, rule AS-TOP flips rule AS-BOT (and vice-versa), rule AS-ANDL flips rule AS-ORL (and vice-versa) and so on. While the definition of type splitting is extended, the subtyping rules are very similar to the modular BCD subtyping rules in Figure 2. Besides the generalization of mode, the key difference is that the left-hand side type in rule AD-ANDL and rule AD-ANDR is splittable rather than being restricted to intersections. As a consequence, rule AD-ARROW has an additional condition.

Strictly speaking, the inclusion of the duality rule (rule AD-DUAL) means that the system is not fully algorithmic. A naive implementation could apply the duality rule indefinitely, thus resulting in a non-terminating function. However, the set of rules with the duality rule is morally algorithmic. As illustrated by Oliveira et al. [2020], in implementation we can use a boolean flag to prevent

repeated flipping with a case similar to the duality rule. We will see this approach in Section 5 when we illustrate our Haskell implementation.

From duotyping to subtyping: deriving the subtyping rules. As we have mentioned, the algorithmic subtyping rules can be derived from the duotyping formulation by specializing the mode. Take rule AD-andL as an example:

$$\text{AD-andL-sub}\quad \frac{B^{\prec} \qquad A_1 \lhd A_{\prec} \rhd A_2 \qquad A_1 \prec_a B}{A \prec_a B} \qquad \text{AD-andL-super}\quad \frac{B^{\succ} \qquad A_1 \lhd A_{\succ} \rhd A_2 \qquad A_1 \succ_a B}{A \succ_a B}$$

The rules above are the result of specializing rule AD-andL to subtyping and supertyping, respectively. To make them compatible with the subtyping style, all judgments in supertyping mode need to be flipped via the duality rule. After that, each instances agree with rule AS-andL and rule AS-orL respectively. Note that rule AS-orL has more restrictions ($A^{\prec}$ and $B^{\prec}$) than rule AD-andL-super. These two conditions are essentially the ordinary conditions that arise from the use of the duality rule to convert supertyping into subtyping.

The order of rules. Except for rule AD-int and rule AD-bound, other rules have duotyping judgements as premises, which means the algorithm will search further after it matches its goal with the rule. Without the gray-highlighted conditions, it is possible for one duotyping judgement to satisfy multiple rules at the same time, i.e. rule AD-and and rule AD-dual for $A_1 \vee A_2 \prec_a B_1 \wedge B_2$. Thus we use the ordinary-type conditions in gray to make these recursive rules disjoint and sort them in the following order: 1) rule AD-and; 2) rule AD-andL and rule AD-andR; 3) rule AD-dual and rule AD-arrow. Then a subtyping (or supertyping) judgement cannot match later rules once it satisfies the conditions of one rule (regardless of the satisfaction of the subtyping premises). For example, rule AD-dual cannot be used to flip $A_1 \vee A_2 \prec_a B$ unless $B^{\prec}$. In that case, rule AD-and will be applied to the flipped goal $B \succ_a A_1 \vee A_2$, as $A_1 \lhd A_1 \vee A_{2\,\succ} \rhd A_2$. To justify the order, we prove the following lemmas:

Lemma 4.1 (Inversions on Splittable Types). *Assuming $A \diamond_a B$,*

- *if $B_1 \lhd B_{\diamond} \rhd B_2$ then $A \diamond_a B_1$ and $A \diamond_a B_2$.*
- *if $A_1 \lhd A_{\diamond} \rhd A_2$ and $B^{\diamond}$ then $A_1 \diamond_a B_1$ or $A_2 \diamond_a B_2$.*
- *if $A_1 \lhd A_{\overline{\diamond}} \rhd A_2$ then $A_1 \diamond_a B$ and $A_2 \diamond_a B$.*
- *if $A^{\overline{\diamond}}$ and $B_1 \lhd B_{\overline{\diamond}} \rhd B_2$ then $A \diamond_a B_1$ or $A \diamond_a B_2$*

An inversion lemma tells us that it is safe to prioritize certain rule in some cases. The first one is for rule AD-and: if a subtyping (or supertyping) judgement holds, and its right-hand side type is splittable under the mode, then it must satisfy rule AD-and. The second one is for rule AD-andL and rule AD-andR. It has an extra condition $B^{\diamond}$, which means the judgement does not meet the conditions of rule AD-and. The next two are for the rule AD-dual, with it we can unfold the duotyping rules by mirroring rule AD-and, rule AD-andL and rule AD-andR with extra conditions. For instance, the dual rule of rule AD-and would be:

$$\text{AD-unfold-or}\quad \frac{\begin{array}{c} A^{\diamond} \\ B^{\diamond} \qquad A_1 \lhd A_{\overline{\diamond}} \rhd A_2 \\ A_1 \diamond_a B \qquad A_2 \diamond_a B \end{array}}{A \diamond_a B}$$

In short, there are two principles: rule AD-and before rule AD-andL and rule AD-andR; the dual of rule AD-and before the dual of rule AD-andL and rule AD-andR. Since the order we choose

obeys the principle, we can prove the duotyping system with the gray conditions is equivalent to the system without such conditions. The same applies to the derived subtyping system (Figure 6). Violating the principles may lead to false-negative results (i.e. an incomplete implementation with respect to the algorithmic specification). For example, if an algorithm tries rule AD-andL and rule AD-andR before rule AD-and, it will reject $Int \wedge Char \prec_a Int \wedge Char$ because both rule AD-andL and rule AD-andR fail.

4.3 Metatheory

Here we present some theorems that connect the two systems in subtyping style (introduced in Section 3) with the two systems in duotyping style.

Theorem 4.2 (Equivalence of Declarative Systems). *For any type A B,*

- *if* $A \diamond B$, *then* $\diamond = \prec$ *and* $A \leq B$, *or* $\diamond = \succ$ *and* $B \leq A$.
- *if* $A \leq B$, *then* $A \prec B$ *and* $B \succ A$.

The first property in the above theorem states the declarative duotyping system (Figure 7) is sound with respect to the declarative subtyping (Figure 4), no matter whether the judgement is in subtyping or supertyping mode. The second property is the completeness of the duotyping system.

After proving that the declarative subtyping relation is equally transformed into duotyping, we show that the algorithmic duotyping system can be mapped into the subtyping in Figure 6 as well. Firstly, the definitions of ordinary and splittable types are equivalent to the intersection- and union-ordinary types and splittable types defined in Figure 5.

Lemma 4.3 (Equivalence of Ordinary and Splittable Types). *For any type A,*

- A° *if and only if* $A^{\prec}$.
- $A^{\bullet}$ *if and only if* $A^{\succ}$.
- $B_1 \lhd A \rhd B_2$ *if and only if* $B_1 \lhd A_{\prec} \rhd B_2$.
- $B_1 \blacktriangleleft A \blacktriangleright B_2$ *if and only if* $B_1 \lhd A_{\succ} \rhd B_2$.

Then the soundness and completeness of the algorithmic subtyping (Figure 6) regarding to the algorithmic duotyping (Figure 8) can be established.

Theorem 4.4 (Equivalence of the Algorithmic Systems). *For any type A B,*

- *if* $A <:_a B$, *then* $A \prec_a B$ *and* $B \succ_a A$.
- *if* $A \diamond_a B$, *then* $\diamond = \prec$ *and* $A <:_a B$, *or* $\diamond = \succ$ *and* $B <:_a A$.

Properties for the algorithmic system. With a set of duotyping rules, one can reason about not only subtyping and supertyping, but also two modes together, which helps to unify theorems and proofs. Here we build the theorems on the two duotyping systems. Thanks to the equivalence between subtyping and duotyping systems, these theorems justify the algorithmic subtyping system (Figure 6) with respect to the declarative subtyping system (Figure 4) as well.

One of the key properties that validate the algorithmic system is the equivalence to the declarative system.

Theorem 4.5 (Soundness and Completeness of Algorithmic Duotyping). $A \diamond_a B$ *if and only if* $A \diamond B$.

To establish it, reflexivity and transitivity are a must.

Theorem 4.6 (Reflexivity of the Algorithmic Duotyping). $A \diamond_a A$.

During the proof we need to consider whether a type can be split or not. The process relies on two facts: First, types can be divided into ordinary types and splittable ones under any mode. Second, type splitting produces unique results.

Table 1. Summary of the proof scripts.

File	SLOC	Description
TypeSize.v	55	Defines the size of type for induction measures.
Definitions.v	397	Contains definitions for all relations. It is generated by the tool Ott [Sewell et al. 2007].
Duotyping.v	966	Contains Lemma 4.1, Theorem 4.5, Theorem 4.6, Lemma 4.7, Lemma 4.8, Lemma 4.9, and Theorem 4.10.
Equivalence.v	159	Relates the two declarative systems and the two algorithmic systems, respectively. It contains Theorem 4.2, Lemma 4.3, and Theorem 4.4.
Subtyping.v	663	Contains some lemmas about the two subtyping systems. Three of them are used in the proof of Theorem 4.4
DistAnd.v	28	Justifies one statement in the paper. It shows that the rule S-DISTAND (in Figure 4) is omittable.
DistSubtyping.v	832	A stand-alone file, which contains the two subtyping systems in Section 3, as well as related proofs that algorithmic subtyping (Figure 6) is decidable and equivalent to the declarative system (Figure 4).
Total	3,100	(2,240 excluding the last two files)

LEMMA 4.7 (TYPES ARE EITHER ORDINARY OR SPLITTABLE). *For any type A,*

- $A^{\diamond}$ *or* $B \lhd A_{\diamond} \rhd C$ *for some type* B *and* C.
- $A^{\diamond}$ *and* $B \lhd A_{\diamond} \rhd C$ *cannot both hold.*

LEMMA 4.8 (DETERMINISM OF TYPE SPLITTING). *If* $A_1 \lhd A_{\diamond} \rhd A_2$ *and* $B_1 \lhd A_{\diamond} \rhd B_2$ *then* $A_1 = A_2$ *and* $B_1 = B_2$.

With reflexivity, the soundness of type splitting can be obtained directly. This suggests that the intersection (or union, according to the mode) of the splitting results is isomorphic to the original type.

LEMMA 4.9 (SOUNDNESS OF SPLITTING). *If* $B_1 \lhd A_{\diamond} \rhd B_2$ *then* $A \diamond_a (B_1 \diamond_? B_2)$ *and* $(B_1 \diamond_? B_2) \overline{\diamond}_a A$.

Compared with reflexivity, transitivity is straightforward since the ordinary conditions eliminate most overlapping.

THEOREM 4.10 (TRANSITIVITY OF THE ALGORITHMIC DUOTYPING). *If* $A \diamond_a B$ *and* $B \diamond_a C$ *then* $A \diamond_a C$.

Decidability is the other key property. Its proof replays the algorithm in duotyping style.

THEOREM 4.11 (DECIDABILITY OF THE ALGORITHMIC DUOTYPING). *It is decidable whether* $A \diamond_a B$.

4.4 Coq Formalization and Proof Statistics

All the lemmas and theorems are formalized and verified in the Coq proof assistant [Coq Development Team 2021]. We use *LibTactics.v* from the TLC Coq library [Charguéraud and Pottier [n.d.]], which defines a collection of general-purpose tactics. In the formalization, a variant of algorithmic duotyping in Figure 8 is formalized where the rule AD-DUAL is eliminated and dual rules are made explicit. The two variants (with and without dual rules) are proved to be equivalent in Coq, and some of the lemmas of algorithmic duotyping are proved using this variant. We also provide a stand-alone file for the two subtyping systems in Section 3. The proof scripts include 3,100 lines of code. Table 1 provides a brief summary of the files in the formalization, their number of source lines of code (SLOC), and a brief description of the content.

```
split :: Mode → Type → Maybe (Type, Type)
split MSub (TArrow a b)                          -- Sp-arrowR
  | Just (b1, b2) <- split MSub b
  = Just (TArrow a b1, TArrow a b2)
split MSub (TArrow a b)                          -- Sp-arrowL
  | Just (a1, a2) <- split MSuper a
  = Just (TArrow a1 b, TArrow a2 b)
split m (TOp m' a b)                             -- Sp-and
  | m == m'
  = Just (a, b)
split m (TOp m' a b)                             -- Sp-orL
  | Just (a1, a2) <- split m a
  = Just (TOp m' a1 b, TOp m' a2 b)
split m (TOp m' a b)                             -- Sp-orR
  | Just (b1, b2) <- split m b
  = Just (TOp m' a b1, TOp m' a b2)
split _ _ = Nothing
```

Fig. 9. Haskell implementation of splittable types in the algorithmic duotyping system

5 A FUNCTIONAL IMPLEMENTATION IN HASKELL

After working out through splittable types and duotyping, we now show the Haskell implementation of the algorithmic duotyping formulation presented in Figure 8. Our implementation exploits the extra ordinary-type conditions in the duotyping rules to avoid too much backtracking, and thus result in a more efficient implementation.

5.1 Abstract Syntax and Modes

The datatype definitions for the implementation are:

```
data Mode = MSub | MSuper
          deriving (Eq, Show)

data Type = TInt | TTop | TBot | TArrow Type Type | TOp Mode Type Type
          deriving (Eq, Show)
```

The datatype `Mode` models the two modes, which stand for the two directions of duotyping judgements: `MSub` for subtyping; and `MSuper` for supertyping. The datatype `Type` models the abstract syntax of types. The first four constructors directly correspond to the integer, top, bottom and arrow types. In the last constructor, we make use of the mode to unify intersection and union types. Specifically, `TOp MSub A B` means $A \wedge B$, and `TOp MSuper A B` means $A \vee B$.

5.2 Type Splitting

Figure 9 shows the implementation of type splitting. The type splitting function follows the formalization directly. The mode specifies whether it is for intersection- (`MSub`) or union- (`MSuper`) splittable types. The function splits the given type when possible and returns **Nothing** if the type is ordinary and cannot be split. The actual implementation makes an interesting use of *pattern guards* [Erwig and Peyton Jones 2000]. For instance, in the first case we have to analyse the result of `split MSub b` to decide whether to execute the code on the right-side of = or fail and move to the next case. If the pattern **Just** `(b1,b2)` does not match the result then we fail and move to the next case. Regarding the order of cases, we can divide them (denoted by the corresponding rules)

```
check :: Mode → Type → Type → Bool → Bool
check _ TInt TInt _ = True                                    -- AD-int
check m _ t _                                                 -- AD-bound
  | select m == t
  = True
check m a b _                                                 -- AD-and
  | Just (b1, b2) <- split m b
  = (check m a b1 False) && (check m a b2 False)
check m a b _                                                 -- AD-andL AD-andR
  | Just (a1, a2) <- split m a
  = (check m a1 b False) || (check m a2 b False)
check m a b False = check (flipmode m) b a True               -- AD-dual
check m (TArrow a1 a2) (TArrow b1 b2) _                       -- AD-arrow
  = (check (flipmode m) a1 b1 False) && (check m a2 b2 False)
check _ _ _ _ = False
```

Fig. 10. Haskell implementation of the duotyping checking algorithm

into three groups based on the form of the split type: 1) rule Sp-arrowR and rule Sp-arrowL; 2) rule Sp-and; 3) rule Sp-orL and rule Sp-orR. While the order inside the group is restricted by the rules, the order across groups does not matter. Actually, the precedence among rules is merely assigned to avoid non-determinism of the split result, and the order itself is insignificant.

5.3 Duotyping and Subtyping

Figure 10 shows the implementation of duotyping. The code uses auxilary functions for flipping modes and selecting $\top$ or $\bot$ by mode, which are trivial to implement:

```
flipmode :: Mode → Mode
flipmode MSub = MSuper
flipmode MSuper = MSub

select :: Mode → Type
select MSub   = TTop
select MSuper = TBot
```

The main function check takes two types and a mode as inputs (following the duotyping judgment $A \diamond_a B$), and one additional boolean flag. The output is a boolean which denotes if the judgement holds. The mode is flipped when it does not fit with the code corresponding to rule AD-int, rule AD-bound, rule AD-and, rule AD-andL, and rule AD-andR, according to rule AD-dual. In such a case we recheck the resulting judgement, which is equivalent to check the initial judgement by the dual of the above rules. A boolean flag is used to make sure such flipping only happens at most once for one judgement. This implementation approach for duotyping follows the approach proposed by Oliveira et al. [2020]. The arrow rule is the last one to be checked because flipping the goal does not affect it. At that point, we know that types on both sides are fully ordinary, and both of them are not *Int*. Thus, if rule AD-arrow fails, a negative result will be returned. Finally, to obtain a function that checks the subtyping of two types we can simply have:

```
sub :: Type → Type → Bool
sub a b = check MSub a b False
```

5.4 Eliminating Backtracking

Note that no backtracking is employed during the process, except for the rule AD-andL and rule AD-andR: both rules need to be considered if the first attempt fails. The lack of other forms of backtracking is justified by our duotyping rules with ordinary conditions, which follows Lemma 4.1. Rules that involve no recursion are put at the start of the function. Rule AD-bound only returns a positive result so it is always safe to prioritize it among overlapping rules. Rule AD-dual overlaps with rule AD-arrow, but in that case, after rule AD-dual, the flipped goal only matches with rule AD-arrow, which makes no difference to directly applying rule AD-dual. Meanwhile, with the current order that we use, the ordinary-type conditions are guaranteed by previous rules which handle splittable types, and therefore not appear in the code.

An alternative implementation is to check rule AD-bound and its dual (for $\top$ and $\bot$) and rule AD-arrow before the rules for splittable types (regardless of the ordinary conditions), which can potentially save some space and time. It can be justified by the following inversion lemma:

Lemma 5.1 (Rule AD-arrow Inversion). *If* $A_1 \to A_2 \diamond_a B_1 \to B_2$ *then* $A_1 \overline{\diamond}_a A_2$ *and* $B_1 \diamond_a B_2$.

If both types in a duotyping judgment are arrow types, their input types and output types must satisfy the duotyping relation respectively as required by the rule AD-arrow. That means it is safe to put the arrow case before others in the Haskell implementation.

6 RELATED WORK

This section discusses related work, focusing on subtyping algorithms with distributivity rules.

Origins of union and intersection types. Coppo et al. extended the lambda calculus à la Curry, and introduced intersection types [Coppo and Dezani-Ciancaglini 1980; Coppo et al. 1980, 1981]. Following them, Barendregt, Coppo, and Dezani-Ciancaglini [1983] proposed a subtyping relation for intersection types with distributivity, that became known as BCD subtyping. Later on, the line of work expanded to union types, and distributivity rules for union types were added by Barbanera et al. [1995]. The work by van Bakel et al. [2000] was found to be "a pleasant surprise": from the logic system **B+**, a type system with intersection and union can be designed directly, with its subtyping arising from the axioms and rules of **B+**, via the Curry-Howard isomorphism [Howard 1980]. The **B+** logic itself was invented in the 70s, by Routley and Meyer [1972], as the minimal one among positive relevant logics, to formalize propositional entailment. Its connectives include conjunctions, disjunctions, and implications, which correspond to intersections, unions, and arrows.

Union and intersection types for programming languages. The use of intersection types for practical programming goes back to Reynolds's Forsythe [Reynolds 1988, 1997]. Pierce [1991] extended this line of work to union types, as well as second-order polymorphism. He also invented a decision procedure for intersections and bounded variables [Pierce 2018], which lead to an algorithmic system for BCD subtyping [Bi et al. 2018]. This algorithm analyzes the supertype structurally, storing its input type for arrows in a queue. When it reaches primitive types or the top type, it then decomposes the subtype part, and pops the queue for comparison. The completeness proof with respect to BCD subtyping relies on an extra concept of "reflexive supertypes", and therefore is not trivial.

The notion of splittable types was introduced in a modular formulation of BCD subtyping [Huang et al. 2021]. This formulation is a cut-free subtyping relation equivalent to BCD subtyping. The metatheory of this alternative formulation of BCD subtyping is pleasingly simple, and our work has shown that this idea can be extended to union types and additional distributivity rules.

Algorithms for minimal relevant logic subtyping. There are some algorithms designed for the provability problem of the minimal relevant logic [Gochet et al. 1995; Viganò 2000]. Such algorithms build a deduction system. But their formalization is quite different from conventional subtyping relations. Here we focus on algorithms designed for subtyping systems that are similar to the minimal relevant logic in expressiveness power. Unlike Pierce's decision procedure or modular BCD subtyping, most algorithms for minimal relevant logic subtyping require a pre-processing step on types. Typically this pre-processing step is some reduction of types into a normal form.

Muehlboeck and Tate [2018] proposed a composable algorithmic framework called integrated subtyping. Integrated subtyping is able to generate decision procedures for various systems with union and intersection types including BCD subtyping, and the one arising from minimal relevant logic. Their strategy is to transform types on the left-hand side to a normal form, which is the disjunctive normal form in the setting without distributivity over arrows. A function called intersector is used in the conversion, and it varies according to the subtyping rules.

Subtyping in the Delta-calculus [Stolze 2019] extends BCD subtyping with union types, and has rules similar to minimal relevant logic. This work provides an algorithm for deciding subtyping, which first rewrites types into some standard normal forms. After rewriting, the left-hand type is a union of intersections, while the right-hand type is an intersection of unions. The basic components in both types include type variables and arrow types rephrased into a normal form, which corresponds to our intersection-ordinary arrow types. Because all arrow types are naturally union-ordinary, these components are ordinary in both modes, and therefore the left-hand side type is an intersection of intersection-ordinary types, while the right-hand side type is a union of union-ordinary types. However, this method may not be able to scale if union arrow distributivity rules (rule S-distArrR-union and rule S-distArrL-union, discussed in Section 3.2) are to be added in the future. As discussed at the end of Section 4, the next step of the algorithm is similar to ours, except that it works on normalized types.

Frisch et al. [2008] extended *semantic subtyping* to intersection and union types for the ℂDuce project. Type constructors act as their corresponding set-theoretic operators. They provided an algorithm for the induced subtyping relation. Firstly they write types into a disjunctive normal form. Then they check whether the result of the left-hand type minus the right-hand type is an empty type by enumeration. Improved versions of the algorithm were implemented and run in the current implementation of ℂDuce [Frisch 2004]. Pearce followed their path and introduced a subtyping algorithm with a constructive proof for soundness and completeness [Pearce 2013].

7 CONCLUSION

This pearl shows a new algorithm for deciding subtyping (and logical entailment) in the presence of union types, intersection types and distributivity rules. Such algorithms are known to be challenging to implement and formalize. Most previous work has addressed similar problems using a pre-processing step to transform types into a normal form, before comparing types for subtyping. Here we present a new algorithm that directly compares source types for subtyping, without a pre-processing phase.

Splittable types [Huang et al. 2021] are key to our algorithm. While originally proposed for a calculus with intersection types only, our pearl illustrates that splittable types can scale up to systems with union types and additional distributivity rules. Moreover, *duotyping* [Oliveira et al. 2020] helped in designing a very symmetric formulation of algorithmic subtyping. One interesting aspect revealed by duotyping is that minimal relevant logic is not fully symmetric from the point of view of duality. As discussed in Section 3.2, minimal relevant logic lacks some "dual" axioms, making the subtyping rules not completely dual. This was a surprise to us. Although the absence of bottom types in minimal relevant logic created an obvious imbalance with respect to duality

(which is easy to correct), we only detected the later issue with the duotyping design. Nonetheless, duotyping was still helpful to organize many of the other rules and relations, and leads to an implementation that can exploit duality to avoid extra code for dual cases. Overall we believe that both splittable types and duotyping are helpful in the design of expressive subtyping relations, and hope that this work encourages further exploration and use of both ideas.

ACKNOWLEDGMENTS

We would like to thank Jinxu Zhao for his insightful discussions throughout the metatheory's development. We appreciate the valuable comments from Yaoda Zhou and the anonymous reviewers that improved the manuscript and the accompanying artifact. We are also grateful to Fabian Muehlboeck and Ross Tate for their detailed reply about their algorithm. This work has been sponsored by the Hong Kong Research Grant Council projects number 17209519 and 17209520.

REFERENCES

Nada Amin, Karl Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday* (2016), 249–272. https://doi.org/10.1007/978-3-319-30936-1_14

F. Barbanera, M. Dezaniciancaglini, and U. Deliguoro. 1995. Intersection and Union Types: Syntax and Semantics. *Information and Computation* 119, 2 (1995), 202–230. https://doi.org/10.1006/inco.1995.1086

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A Filter Lambda Model and the Completeness of Type Assignment. *The Journal of Symbolic Logic* 48, 4 (1983), 931–940. http://www.jstor.org/stable/2273659

Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. 2003. CDuce: An XML-Centric General-Purpose Language. (2003), 51–63. https://doi.org/10.1145/944705.944711

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (2017), 65–98. https://doi.org/10.1137/141000671

Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2018. The Essence of Nested Composition. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands (LIPIcs, Vol. 109)*, Todd D. Millstein (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:33. https://doi.org/10.4230/LIPIcs.ECOOP.2018.22

Arthur Charguéraud and François Pottier. [n.d.]. TLC: a non-constructive library for Coq. https://www.chargueraud.org/softs/tlc/.

M. Coppo and M. Dezani-Ciancaglini. 1980. An extension of the basic functionality theory for the λ-calculus. *Notre Dame Journal of Formal Logic* 21, 4 (1980), 685 – 693. https://doi.org/10.1305/ndjfl/1093883253

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. 1980. Principal Type Schemes and Lambda-calculus Semantics. (1980), 535–560. http://www.di.unito.it/~dezani/papers/CDV80.pdf

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. 1981. Functional Characters of Solvable Terms. *Math. Log. Q.* 27, 2-6 (1981), 45–58. https://doi.org/10.1002/malq.19810270205

The Coq Development Team. 2021. *The Coq Reference Manual, version 8.13.2.* Available electronically at https://coq.inria.fr/distrib/current/refman/.

Rowan Davies and Frank Pfenning. 2000. Intersection types and computational effects. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, 198–208. https://doi.org/10.1145/351240.351259

Martin Erwig and Simon Peyton Jones. 2000. Pattern Guards and Transformational Patterns. In *Haskell Workshop 2000* (haskell workshop 2000 ed.). https://www.microsoft.com/en-us/research/publication/pattern-guards-and-transformational-patterns/

Facebook. 2014. Flow. https://flow.org.

Alain Frisch. 2004. *Théorie, conception et réalisation d'un langage de programmation adapté à XML.* Ph.D. Dissertation. PhD thesis, Université Paris 7.

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic Subtyping: Dealing Set-Theoretically with Function, Union, Intersection, and Negation Types. *J. ACM* 55, 4, Article 19 (Sept. 2008), 64 pages. https://doi.org/10.1145/1391289.1391293

Paul Gochet, Pascal Gribomont, and Didier Rossetto. 1995. ALGORITHMS FOR RELEVANT LOGIC: Leo Apostel in memoriam. *Logique et Analyse* 38, 150/152 (1995), 329–346. http://www.jstor.org/stable/44084547

William A. Howard. 1980. The formulae-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, λ-calculus and Formalism*, J. Hindley and J. Seldin (Eds.). Academic Press, 479–490.

Xuejing Huang, Jinxu Zhao, and Bruno C. d. S. Oliveira. 2021. *Taming the Merge Operator: a Type-directed Operational Semantics Approach.* Technical Report TR-2021-01. Department of Computer Science, The University of Hong Kong. https://www.cs.hku.hk/data/techreps/document/TR-2021-01.pdf

Microsoft. 2012. TypeScript. https://www.typescriptlang.org.

Fabian Muehlboeck and Ross Tate. 2018. Empowering Union and Intersection Types with Integrated Subtyping. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 112 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276482

Bruno C. d. S. Oliveira, Cui Shaobo, and Baber Rehman. 2020. The Duality of Subtyping. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 29:1–29:29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.29

David J. Pearce. 2013. Sound and Complete Flow Typing with Unions, Intersections and Negations. In *Verification, Model Checking, and Abstract Interpretation*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 335–354.

Benjamin C. Pierce. 1991. *Programming with Intersection Types and Bounded Polymorphism.* Ph.D. Dissertation. Carnegie Mellon University.

Benjamin C. Pierce. 2018. A decision procedure for the subtype relation on intersection types with bounded variables. https://doi.org/10.1184/R1/6587339.v1

Redhat. 2011. Ceylon. https://ceylon-lang.org.

John C Reynolds. 1988. *Preliminary design of the programming language Forsythe.* Technical Report CMU-CS-88-159. Carnegie Mellon University.

John C Reynolds. 1997. Design of the Programming Language Forsythe. In *ALGOL-like languages*. Springer, 173–233.

Richard Routley and Robert K. Meyer. 1972. The Semantics of Entailment: III. *Journal of Philosophical Logic* 1, 2 (1972), 192–208. http://www.jstor.org/stable/30226036

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. 2007. Ott: Effective Tool Support for the Working Semanticist. *SIGPLAN Not.* 42, 9 (Oct. 2007), 1–12. https://doi.org/10.1145/1291220.1291155

Claude Stolze. 2019. *Combining union, intersection and dependent types in an explicitly typed lambda-calculus.* Ph.D. Dissertation. Université Côte d'Azur.

The Scala Center team and community contributors. 2020. Scala 3. https://dotty.epfl.ch.

Steffen van Bakel, Mariangiola Dezani-Ciancaglini, Ugo de' Liguoro, and Yoko Motohama. 2000. *The Minimal Relevant Logic and the Call-by-Value Lambda Calculus.* Technical Report TR-ARP-05-2000. The Australian National University. http://www.di.unito.it/~deligu/papers/vBDdLM-trANU00.pdf

Luca Viganò. 2000. An O(n log n)-Space Decision Procedure for the Relevance Logic B+. *Studia Logica: An International Journal for Symbolic Logic* 66, 3 (2000), 385–407. http://www.jstor.org/stable/20016237

Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. 2018. Julia Subtyping: A Rational Reconstruction. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 113 (Oct. 2018), 27 pages. https://doi.org/10.1145/3276483