Empowering Union and Intersection Types with Integrated Subtyping

FABIAN MUEHLBOECK, Cornell University, USA
ROSS TATE, Cornell University, USA

Union and intersection types are both simple and powerful but have seen limited adoption. The problem is that, so far, subtyping algorithms for type systems extended with union and intersections have typically been either unreliable or insufficiently expressive. We present a simple and composable framework for empowering union and intersection types so that they interact with the rest of the type system in an intuitive and yet still decidable manner. We demonstrate the utility of this framework by illustrating the impact it has made throughout the design of the Ceylon programming language developed by Red Hat.

CCS Concepts: • **Theory of computation → Type structures**; • **Software and its engineering → Compilers**; *Object oriented languages*; *Functional languages*; *Multiparadigm languages*;

Additional Key Words and Phrases: subtyping, unions, intersections, distributivity, decidability, extensibility

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

1 INTRODUCTION

In many languages, types can be interpreted as sets of values. In a language with subtyping, subtypes can often be interpreted as subsets, and union and intersection types—which can be interpreted as unions and intersections of sets—are an easy way to obtain joins and meets in the subtyping hierarchy. Thus union and intersection types have often been used in tasks like type inference [Gosling et al. 2005] and static analysis [Palsberg and Pavlopoulou 1998; Wells et al. 2002]. However, until recently they have rarely been exposed directly to programmers.

Part of the problem is that the set-based model suggests certain interactions of union and intersection types with each other and the rest of the type system, but these interactions have established themselves to be difficult to implement in ways that are both decidable and extensible. For example, one would expect unions and intersections to be distributive, but the standard syntax-directed subtyping rules for union and intersection types cannot recognize the following subtyping:

$$\tau \cap (\tau_1 \cup \tau_2) <: (\tau \cap \tau_1) \cup (\tau \cap \tau_2)$$

Pierce [1991] explored combining union and intersection types in the context of the Forsythe programming language [Reynolds 1988, 1997]. He gave rules for distributivity of intersections over both unions and functions, but he also relied on an explicit transitivity rule and thus had no clear decidable algorithm for subtyping, leaving this goal for future work. Decidability in this particular setting has been established in work on *minimal relevant logic* [Gochet et al. 2005; Routley and Meyer 1972; Viganò 2000] and on semantic subtyping [Frisch et al. 2002].

Authors' addresses: Fabian Muehlboeck, Department of Computer Science, Cornell University, Ithaca, NY, USA, fabianm@cs.cornell.edu; Ross Tate, Department of Computer Science, Cornell University, Ithaca, NY, USA, ross@cs.cornell.edu.

2475-1421/2018/11-ART112
https://doi.org/10.1145/3276482

However, the algorithms and proofs of these works are not easy to generalize and adapt into even slight extensions of the type systems. As an example, for nearly a decade the Scala team has investigated adding true union types to the language but has yet to make such an addition. The Dotty prototype in development for Scala does support union types [Rompf and Amin 2016; The Dotty Development Team 2015], but not in decidable manner, and it will probably be some more years until it matures to the point of supporting union types with "enough" decidability for practical purposes. Flow [Facebook 2014] and Pony [Clebsch et al. 2015] do have union and intersection types, but with very limited reasoning that is unable to recognize aspects such as distributivity. Typed Racket [Tobin-Hochstadt and Felleisen 2008], on the other hand, can recognize distributivity, but cannot recognize deeper properties such as the fact that a pair of union types is equivalent to a union of pair types. Julia only has union types but is capable of such deeper reasoning [Bezanson et al. 2017; Zappa Nardelli et al. 2018]. However, it is unknown whether Julia subtyping is decidable or even transitive, which are particularly important questions for Julia since subtyping is heavily used in its operational semantics, and which we discuss in more detail near the end of this paper. TypeScript [Bierman et al. 2014; Microsoft 2012] is best situated to take advantage of prior research due to its heavy use of structural subtyping, which has been the focus of semantic subtyping [Ancona and Corradi 2016; Castagna and Xu 2011; Frisch et al. 2002, 2008; Hosoya and Pierce 2003]. However, in order to accept common patterns in the JavaScript community, TypeScript chose to make its subtyping system intentionally unsound and intentionally intransitive [Microsoft 2018, Type Compatibility]. Lack of soundness means TypeScript can be optimistically aggressive with how it reasons about union and intersection types. Lack of transitivity means this aggressive reasoning can be inconsistent and hard to predict. For example, TypeScript will recognize that a particular value f belongs to a particular function type $(x : \tau_i) \Rightarrow \tau_o$ but then reject an invocation of f with an argument of type τ_i even when the required reasoning can be soundly conducted using the conservative techniques we describe here. Thus, even with the recent rise of union and intersection types in industry, there is still significant room for improvement, especially along the front of principled subtyping algorithms.

In this paper, we show how to extend reasoning about intersection types in a decidable and extensible manner. The resulting technique has already been adopted by Ceylon [King 2013], a Java-like industry language that, based on this work, was able to implement several type-system features on top of union and intersection types (Section 2). For example, Ceylon's type-checker can recognize that an intersection like `String` ∩ `Int` is equivalent to the bottom type `Nothing`, as `String` and `Int` have no common instances, and even uses this to implement pattern matching.

The key idea presented in this paper is that modifying the subtyping algorithm for naïve union and intersection types to *integrate* types as it recurses can significantly improve its ability to recognize desirable subtyping relationships (Section 4). Furthermore, provided this integration operation on types satisfies various requirements, this improved subtyping algorithm is mechanically verified in Coq [The Coq Development Team 1984] to be sound and complete with respect to an extended subtyping system for union and intersection types, one in which union and intersection types are empowered with useful interactions with other aspects of the type system. And by composing integration operators, we can repeatedly extend the system with more and more reasoning capabilities (Section 5). With this toolkit, we were able to apply our framework to the Ceylon programming language, which opted to fully integrate union and intersection types into its design due to the expressiveness and guarantees we were able to provide them with (Section 6).

While its original motivations came from challenges posed by the Ceylon team, our framework is language independent. In fact, as we detail the context of our framework (Section 3), we will use Forsythe's function types as our running example. We also close the paper with a broader discussion of the generality of our framework and its connections to other research and languages (Section 7).

2 MOTIVATION

Before discussing how our framework works, we first illustrate what our framework makes possible. We do so by demonstrating the impact it had on the Ceylon programming language. All of the features discussed in this section were features the Ceylon design team wanted to provide, but were only willing to do so if the features could be implemented in a simple and reliable manner. In each case, we realized the feature could naturally be encoded with union and intersections types *provided* union and intersection subtyping could be made sufficiently intelligent, and by encoding them all into subtyping we could make the features interact in a principled manner. Unfortunately the required intelligence was significantly beyond what the state of the art in union and intersection subtyping could achieve. Thus we created our framework to provide a unified algorithm for all of these features, enabling predictable, well-defined, and powerful interactions between them, by empowering union and intersection subtyping for Ceylon.

Principal Types. The Ceylon team wanted every expression to have a type that best describes it, a concept known as principal types (which should not be confused with principal type schemes [Damas and Milner 1982]) or as minimal types [Curien and Ghelli 1992]. This property makes it much easier to provide tooling for the language, such as efficient IDE support. However, principal types are not easy to ensure in a language with both subtype polymorphism and parametric polymorphism.

To see why, consider the following generic method:

```
Iterable<E> concat<E>(Iterable<E> first, Iterable<E> second)
```

It is safe to call this with an `Iterable<Integer>` and an `Iterable<Float>` as the first and second arguments. The reason is that `Iterable` is *covariant*, so both types are subtypes of `Iterable<Number>`. The problem is that deriving this common supertype, `Number`, is not always easy, especially since one needs the *least* common-supertype in order to provide the principal type. For example, in Java the least common-supertype of `Integer` and `Float` is actually an infinite type because both classes are `Comparable` with themselves. And to make matters worse, similar examples with *contravariant* generic classes and interfaces require computing the *greatest* common-*sub*type of two types.

Union and intersection types seem to make this problem trivial: the union type `Integer` ∪ `Float` is *by definition* the least common-supertype of `Integer` and `Float`, and the intersection of two types is by definition their greatest common-subtype. But while union and intersection types do easily solve the original problem, they introduce whole new problems.

To see why, suppose we have two type variables `A` and `B`. Consider what the resulting type of passing an `Iterable<A>` ∩ `Iterable<B>` to the following generic method should be:

```
E first<E>(Iterable<E> elems)
```

One valid type is `A`, since `Iterable<A>` ∩ `Iterable<B>` is by definition at least an `Iterable<A>`. But for that same reason, another valid type is `B`. Unfortunately, neither valid type is more precise than the other, so they both fail to be principal types.

The only possible principal type for this example is `A` ∩ `B`. But one must be careful. In C#, this would be unsound because C# permits *multiple*-instantiation inheritance. This means a class `Foo` can implement `Iterable<Integer>` using one set of methods and `Iterable<Float>` using another set. Consequently, a call to `first` with a `Foo` argument will necessarily pick one of the two implementations of `Iterable` and return a value that is either an `Integer` *or* a `Float` but not both.

On the other hand, Java and Ceylon both disallow multiple-instantiation inheritance. And due to their restrictions on inheritance, they each can safely admit the following subtyping rules:

$$C\langle\tau\rangle \cap C\langle\tau'\rangle <: C\langle\tau \cap \tau'\rangle \quad \text{when } C \text{ is covariant}$$
$$C\langle\tau\rangle \cap C\langle\tau'\rangle <: C\langle\tau \cup \tau'\rangle \quad \text{when } C \text{ is contravariant}$$

By using the first of these rules, one can show that `A∩B` is in fact the principal type of the expression in question. In general, these extensions to subtyping can be used to easily deduce principal types for many uses of generic methods (though not all, since such principal types do not always exist).

Disjointness and Null-Safety. Another feature Ceylon wanted to provide was the ability to recognize when intersections are uninhabitable. For example, `String` and `Integer` have no common instances, so the intersection `String ∩ Integer` is uninhabitable. More precisely, it is semantically equivalent to the bottom type `Nothing`. In Java, this type, if it existed, would be inhabited by `null`. Ceylon is null-safe, meaning types explicitly indicate whether or not they can be inhabited by `null`, so `Nothing` is in fact uninhabitable.

The utility of this feature is actually best demonstrated by its interaction with null-safety. In Ceylon, one uses the type `String?` to represent values that are either strings or `null`. This `?` operator is just a shorthand, and in fact `String?` simply represents the union type `String ∪ Null`.

Now suppose you have a list of `String?` and you want to fetch the non-null elements of the list. Ideally you could do so polymorphically using a generic method whose type argument can always be inferred, and in Ceylon this is achieved by the following generic method:

```
Iterable<E∩Object> filterNulls<E>(Iterable<E> elems)
```

Note that the return type is `Iterable<E∩Object>`. If we provide this method with a list of `String?`, the instantiation of the return type in full will be `Iterable<(String∪Null)∩Object>`. Since Ceylon is null-safe, every value of `Object` is necessarily not null, and every `String` is already an `Object`, so the type system should ideally be able to recognize that this type is equivalently just an `Iterable<String>`. It can do so by recognizing distributivity of intersections over unions as well as the following subtyping rule that is sound in any language with single-class inheritance:

$$C \cap C' <: \bot \quad \text{when both } C \text{ and } C' \text{ are classes and neither inherits the other}$$

Exhaustive Pattern Matching. The Ceylon team wanted to support algebraic data types with exhaustive pattern matching. They also wanted to support being able to branch based on the run-time type of a value. And ideally these related concepts could be handled uniformly.

The following is an example of how such a uniform treatment can be utilized:

```
switch (nums) { \\ nums has type Integer∪LinkedList<Integer>
 case (is Integer) { return nums; }
 case (is Nil) { return 0; }
 case (is Cons) { return nums.head; }
} \\ expected return type is Integer
```

Here `nums` is either an integer or a linked list of integers. The `LinkedList` class is declared with a clause "`of Nil, Cons`" that restricts it to have only two subclasses: `Nil` and `Cons`. Consequently, the type system should ideally be able to deduce that this pattern match is in fact exhaustive.

Rather than require a separate set of rules for exhaustiveness, our framework was able to make subtyping powerful enough to answer this question with a simple subtyping check. In particular, we just have to check if the type of `nums` is a subtype of the union of all the cases. In this example, we can prove this subtyping so long as we can recognize the following subtyping rule:

$$C\langle\tau\rangle <: C_1\langle\tau\rangle \cup \cdots \cup C_n\langle\tau\rangle \quad \text{when } C \text{ of } C_1, \ldots, C_n$$

Flow-Sensitivity. Lastly, the Ceylon design team wanted the language to be flow-sensitive. This would avoid the tedious and unreliable pattern of `instanceof` checks followed by casts. Naïvely, one could achieve this by simply changing the type of the variable to the type in the `instanceof`

(or `is`) check. However, this loses all other information about the variable that might have been in the context. So a better technique is to intersect the type of the variable with the type in the check.

The last case of the above pattern match illustrates how this typing rule, in combination with an improved subtyping system, successfully unifies the features we have discussed. In this case, the type of `nums` is refined to be

$$(\texttt{Integer} \cup \texttt{LinkedList<Integer>}) \cap \texttt{Cons<*>}$$

where `*` indicates the type argument is unknown since it was not checked in the cast. By distributivity, this is a subtype of

$$(\texttt{Integer} \cap \texttt{Cons<*>}) \cup (\texttt{LinkedList<Integer>} \cap \texttt{Cons<*>})$$

By disjointness, the left intersection is a subtype of $\bot$, which effectively eliminates this case of the union, leaving us with

$$\texttt{LinkedList<Integer>} \cap \texttt{Cons<*>}$$

Since `LinkedList` has an `of` clause, the left type is a subtype of the union of its cases, resulting in

$$(\texttt{Nil<Integer>} \cup \texttt{Cons<Integer>}) \cap \texttt{Cons<*>}$$

And by distributivity and disjointness again, we can eliminate the `Nil` case, leaving us with only the `Cons<Integer>` case. Consequently, we can determine that `nums` is specifically a `Cons<Integer>`, informing us that it indeed has a `head` field with the expected return type `Integer`.

The framework we developed is able to address each of these problems in a principled manner. Furthermore, the framework prescribes how to compose each of the individual solutions together into one coherent subtyping algorithm simultaneously supporting all the extensions, as needed by the above example. Next we describe when in general our framework can be applied, and how it achieves these results. As our running example, we will use the much simpler concept of function types and our application to Forsythe. But afterwards, we will revisit the above more elaborate collection of features desired by Ceylon.

3 FORMALIZING TRADITIONAL UNION AND INTERSECTION SUBTYPING

Our goal is to empower subtyping in type systems with union and intersection types. Thus we are looking at type systems where types have the following general form:

$$\tau ::= \bot \mid \top \mid \tau \cup \tau \mid \tau \cap \tau \mid \ell$$

The special types $\bot$ and $\top$ form the bottom and top of the subtyping hierarchy, respectively, while unions $\cup$ and intersections $\cap$ form joins and meets, respectively. Last, literals ℓ are the rest of the types in the particular type system at hand. For example, in an overly simplified functional language, literals would be defined mutually recursively with types τ as $\ell_{\text{Fun}} ::= \tau \to \tau$.

Subtyping systems for union and intersection types follow certain patterns, both in how they are declaratively specified and in how they are algorithmically decided. Even the proofs that the algorithm is sound and complete with respect to the specification exhibit common reductive patterns. In order to improve these preexisting subtyping systems, we need to understand these patterns. In the following, we illustrate these patterns, and at the same time we formalize them so that we may formally describe the requirements and guarantees of our framework.

3.1 Declarative Subtyping

Traditionally, subtyping with union and intersection types is specified as in Figure 1. In the start of the first row, subtyping is specified to be reflexive and transitive. In the second row are the declarative subtyping rules for union and intersection types modeling their set-theoretic intuitions. Lastly, at the end of the first row is the subtyping rule for literals. It is formalized abstractly by

$$\frac{}{\tau <: \tau} \qquad \frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3} \qquad \frac{r \in \mathcal{D} \qquad \forall \langle \tau^{\leftarrow}, \tau^{\rightarrow} \rangle \in \mathcal{P}_r.\ \ \tau^{\leftarrow} <: \tau^{\rightarrow}}{\ell_r^{\leftarrow} <: \ell_r^{\rightarrow}}$$

$$\frac{}{\tau <: \top} \qquad \frac{}{\bot <: \tau} \qquad \frac{}{\tau_i <: \tau_1 \cup \tau_2} \qquad \frac{}{\tau_1 \cap \tau_2 <: \tau_i} \qquad \frac{\tau_1 <: \tau \quad \tau_2 <: \tau}{\tau_1 \cup \tau_2 <: \tau} \qquad \frac{\tau <: \tau_1 \quad \tau <: \tau_2}{\tau <: \tau_1 \cap \tau_2}$$

Fig. 1. Declarative Subtyping

assuming a (usually infinite) set of declarative literal subtyping rules $\mathcal{D}$. For each rule $r \in \mathcal{D}$, there is a left literal $\ell_r^{\leftarrow}$ (the subtype), a right literal $\ell_r^{\rightarrow}$ (the supertype), and a set of premises $\mathcal{P}_r \subseteq \tau \times \tau$. Each premise has two types that must be subtypes for the rule to be applicable. The literal subtyping rule in Figure 1 indicates that two literals are subtypes if there is a rule in $\mathcal{D}$ that concludes with those two literals and for which subtyping can be shown to hold for every premise.

As an example, consider our toy functional language whose literals ℓ_{Fun} are function types. Subtyping on functions is contravariant with respect to the parameter type and covariant with respect to the return type. Since there are an infinite number of potential parameter and return types, there are an infinite number of rules formulating variance of function types. However, this can be expressed concisely by just two rule *schemata* using *metavariables*, with rules resulting from instantiating the metavariables. It is common to conflate rules and rule schemata, and we ourselves do so in this paper for the sake of simplicity, letting the reader apply context and intuition to disambiguate where necessary rather than constantly overwhelm the reader with disambiguations. As such the two rules (technically rule schemata) for variance of function types are the following:

$$\frac{\tau_i' <: \tau_i}{\tau_i \to \tau_o <: \tau_i' \to \tau_o} \qquad \frac{\tau_o <: \tau_o'}{\tau_i \to \tau_o <: \tau_i \to \tau_o'}$$

We can encode these rules in our framework by defining $\mathcal{D}_{\text{Fun}}$ to be comprised of two cases parametrized by the possible instantiations of the relevant metavariables:

$$\mathcal{D}_{\text{Fun}} ::= \text{Contra}(\tau_i, \tau_i', \tau_o) \mid \text{Co}(\tau_i, \tau_o, \tau_o')$$

The components of these rules are then defined as follows:

$r \in \mathcal{D}_{\text{Fun}}$	$\ell_r^{\leftarrow}$	$\ell_r^{\rightarrow}$	$\mathcal{P}_r$
$\text{Contra}(\tau_i, \tau_i', \tau_o)$	$\tau_i \to \tau_o$	$\tau_i' \to \tau_o$	$\{\langle \tau_i', \tau_i \rangle\}$
$\text{Co}(\tau_i, \tau_o, \tau_o')$	$\tau_i \to \tau_o$	$\tau_i \to \tau_o'$	$\{\langle \tau_o, \tau_o' \rangle\}$

Thus, $\mathcal{D}_{\text{Fun}}$ is the infinite set of all possible instantiations of the two rule schemata given above, and the metafunctions $\ell^{\rightarrow}$, $\ell^{\leftarrow}$, and $\mathcal{P}$ access the relevant parts of these instantiations.

Encodings aside, the advantage of the declarative subtyping rules is that one can quickly assess that they correctly capture the intended features. For one, subtyping is transitive by definition. But this same transitivity rule is often also the greatest problem with the declarative subtyping rules. The reason is that it is not *syntax-directed*. In trying to determine whether one type is a subtype of another, there are an infinite number of possible middle types that could be used to exploit transitivity. This means one cannot effectively search through the space of possible declarative subtyping proofs as a subtyping algorithm. Next we discuss the standard solution to this problem: design a new set of subtyping rules that are syntax-directed, and then prove this syntax-directed subtyping system to be equivalent to the declarative subtyping system.

$$\frac{r \in \mathcal{R} \qquad \forall \langle \tau^{\leftarrow}, \tau^{\rightarrow} \rangle \in \mathcal{P}_r.\ \ \tau^{\leftarrow} \leq \tau^{\rightarrow}}{\ell_r^{\leftarrow} \leq \ell_r^{\rightarrow}}$$

$$\frac{}{\tau \leq \top} \qquad \frac{}{\bot \leq \tau} \qquad \frac{\tau \leq \tau_i}{\tau \leq \tau_1 \cup \tau_2} \qquad \frac{\tau_i \leq \tau}{\tau_1 \cap \tau_2 \leq \tau} \qquad \frac{\tau_1 \leq \tau \quad \tau_2 \leq \tau}{\tau_1 \cup \tau_2 \leq \tau} \qquad \frac{\tau \leq \tau_1 \quad \tau \leq \tau_2}{\tau \leq \tau_1 \cap \tau_2}$$

Fig. 2. Reductive Subtyping

3.2 Reductive Subtyping

There are many ways to develop subtyping algorithms and prove them correct. For reasons that should become clear once we discuss the proof of correctness, we use the term *reductive* to refer to the particularly common—even textbook [Pierce 2002]—technique we formalize here. The syntax-directed reductive subtyping rules for union and intersection types are shown in Figure 2. The rules for reflexivity and transitivity have been removed, and a number of the rules specific to union and intersection types have been adjusted to conceptually directly incorporate the removed rules. And as before, the rest of subtyping consists of rules concerning specifically literals. However, in order to incorporate the removed rules into the literal rules, the abstract literal rule ranges over a different set $\mathcal{R}$ of *reductive* rules for literals.

For example, for our toy functional language, there is only one reductive subtyping rule on functions, combining the two declarative subtyping rules into one:

$$\textsc{Rule}\qquad \frac{\tau_i' \leq \tau_i \qquad \tau_o \leq \tau_o'}{\tau_i \to \tau_o \leq \tau_i' \to \tau_o'}$$

$$\textsc{Encoding}\qquad \begin{array}{r|lll} r \in \mathcal{R}_{\text{Fun}} & \ell_r^{\leftarrow} & \ell_r^{\rightarrow} & \mathcal{P}_r \\ \hline \mathsf{Fun}(\tau_i, \tau_i', \tau_o, \tau_o') & \tau_i \to \tau_o & \tau_i' \to \tau_o' & \{\langle \tau_i', \tau_i \rangle, \langle \tau_o, \tau_o' \rangle\} \end{array}$$

3.3 Proof Search as an Algorithm

The intent is to derive an algorithm from this new set of rules. Given a potential subtyping, proof search considers each rule that could conclude with that subtyping and recursively checks whether the premises of the rule hold. However, for this process to terminate, the reductive rules must satisfy two important properties.

3.3.1 Syntax-Directedness. Proof search must consider every rule that can apply to the potential subtyping at hand. For this to be possible, there must be a finite number of such rules. Furthermore, the process must consider every premise of the rules, so there must be a finite number of premises for each rule. These properties together are what make a system syntax-directed. Clearly they hold for the standard reductive subtyping rules, i.e. the non-literal rules for unions and intersections. Thus one only needs to show they also hold for the custom reductive literal rules.

Requirement 1 (Syntax-Directedness) *For every pair of literals $\ell^{\leftarrow}$ and $\ell^{\rightarrow}$, the set of applicable reductive literal subtyping rules $\{r \in \mathcal{R} \mid \ell_r^{\leftarrow} = \ell^{\leftarrow} \wedge \ell_r^{\rightarrow} = \ell^{\rightarrow}\}$ must be computable and finite. For every reductive literal subtyping rule r in $\mathcal{R}$, the set of premises $\mathcal{P}_r$ must be computable and finite.*

3.3.2 Well-Foundedness. Proof search is recursive, and so one must ensure that this recursion terminates. For the standard reductive rules, one can easily see that the combined syntactic height of the two types being investigated always decreases. Consequently, every path of recursion is guaranteed to always eventually reach a point in which the two types being compared are both literals. So once again termination comes down to a property of the custom reductive literal rules.

Requirement 2 (Well-Foundedness) *There exists a function m from pairs of types to some set M with a well-founded relation $\prec$ satisfying the following inequalities:*

$$\forall \tau_1^{\leftarrow}, \tau_1^{\rightarrow}, \tau_2^{\leftarrow}, \tau_2^{\rightarrow}. \quad \tau_1^{\leftarrow} \ll \tau_2^{\leftarrow} \ \wedge \ \tau_1^{\rightarrow} \ll \tau_2^{\rightarrow} \implies m(\tau_1^{\leftarrow}, \tau_1^{\rightarrow}) \preceq m(\tau_2^{\leftarrow}, \tau_2^{\rightarrow})$$
$$\forall \tau^{\leftarrow}, \tau^{\rightarrow}, r \in \mathcal{R}. \quad \langle \tau^{\leftarrow}, \tau^{\rightarrow} \rangle \in \mathcal{P}_r \implies m(\tau^{\leftarrow}, \tau^{\rightarrow}) \prec m(\ell_r^{\leftarrow}, \ell_r^{\rightarrow})$$

where $\ll$ is the weakest reflexive relation satisfying $\forall i, \tau_1, \tau_2.\ \tau_i \ll \tau_1 \cap \tau_2 \ \wedge \ \tau_i \ll \tau_1 \cup \tau_2$

Typically the ordering $\preceq$ has joins and a bottom element, which are then used to define m on union and intersection types. So the only aspect of the measure function m that is actually interesting is its definition on pairs of literals. This is the case for our toy functional language, where the measure space is $\mathbb{N}$ with $<$. In this case, m is defined on literals as follows:

$$m(\tau_i^{\leftarrow} \to \tau_o^{\leftarrow}, \tau_i^{\rightarrow} \to \tau_o^{\rightarrow}) = 1 + \max(m(\tau_i^{\rightarrow}, \tau_i^{\leftarrow}), m(\tau_o^{\leftarrow}, \tau_o^{\rightarrow}))$$

3.4 Equivalence of Declarative and Reductive Subtyping

Syntax-directedness and well-foundedness of the reductive rules ensure that proof search prescribes a proper algorithm. However, that algorithm is only guaranteed to be sound and complete with respect to *reductive* subtyping, whereas the goal is to have a sound and complete algorithm for *declarative* subtyping. The standard is to bridge this gap by proving that the two subtyping systems are in fact equivalent, making a decision procedure for one also a decision procedure for the other. This proof is typically comprised of three main components.

3.4.1 Reflexivity. The first and easiest step is to prove that reductive subtyping is reflexive, since it has no rule declaring it as such. Interestingly, although well-foundedness was originally intended to guarantee termination of proof search, one can repurpose it to simplify our proof of reflexivity. In short, if for every τ one can apply reductive rules to the goal $\tau \leq \tau$ in order to reduce the problem to other goals of the form $\tau' \leq \tau'$, well-foundedness informs us that recursively applying this goal-reduction procedure is guaranteed to terminate.

As an example, suppose τ is of the form $\tau_1 \cap \tau_2$. Then the following shows how one can reduce the goal of proving $\tau_1 \cap \tau_2 \leq \tau_1 \cap \tau_2$ to the goals $\tau_1 \leq \tau_1$ and $\tau_2 \leq \tau_2$:

$$\dfrac{\dfrac{\tau_1 \leq \tau_1}{\tau_1 \cap \tau_2 \leq \tau_1} \qquad \dfrac{\tau_2 \leq \tau_2}{\tau_1 \cap \tau_2 \leq \tau_2}}{\tau_1 \cap \tau_2 \leq \tau_1 \cap \tau_2}$$

Similar reductions can be done for $\bot$, $\top$, and unions, leaving only reflexivity of literals:

Requirement 3 (Literal Reflexivity) $\forall \ell.\ \exists r \in \mathcal{R}.\ \ell_r^{\leftarrow} = \ell = \ell_r^{\rightarrow} \ \wedge \ \forall \langle \tau^{\leftarrow}, \tau^{\rightarrow} \rangle \in \mathcal{P}_r.\ \tau^{\leftarrow} = \tau^{\rightarrow}$

For our toy functional language, the reflexivity rule for the literal $\tau_i \to \tau_o$ is $\mathsf{Fun}(\tau_i, \tau_i, \tau_o, \tau_o)$.

3.4.2 Rule Conversion. Besides reflexivity, which we just discussed, and transitivity, which we will discuss next, there is a tight correspondence between the declarative rules and the reductive rules. The second step of the equivalence proof is to demonstrate this correspondence by converting each rule of one system into a combination of the rules of the other system.

For example, consider the reductive rule and its conversion into declarative rules below:

$$\dfrac{\tau \leq \tau_i}{\tau \leq \tau_1 \cup \tau_2} \quad \xmapsto{\text{reductive-to-declarative conversion}} \quad \dfrac{\tau <: \tau_i \qquad \overline{\tau_i <: \tau_1 \cup \tau_2}}{\tau <: \tau_1 \cup \tau_2}$$

Note that whereas the reductive rule had one premise of the form $\tau \leq \tau_i$, the declarative proof has one assumption of the form $\tau <: \tau_i$.

$$\left(\begin{array}{c}\text{All rules in Figure 1,}\\ \text{replacing } \tau^{\leftarrow} <: \tau^{\rightarrow} \text{ with } \mathcal{P} \vdash \tau^{\leftarrow} <: \tau^{\rightarrow}\end{array}\right) \qquad \frac{\langle \tau^{\leftarrow}, \tau^{\rightarrow} \rangle \in \mathcal{P}}{\mathcal{P} \vdash \tau^{\leftarrow} <: \tau^{\rightarrow}}$$

Fig. 3. Declarative Subtyping with Assumptions

$$\left(\begin{array}{c}\text{All rules in Figure 2,}\\ \text{replacing } \tau^{\leftarrow} \leq \tau^{\rightarrow} \text{ with } \mathcal{P} \vdash \tau^{\leftarrow} \leq \tau^{\rightarrow}\end{array}\right) \qquad \frac{\langle \tau^{\leftarrow}, \tau^{\rightarrow} \rangle \in \mathcal{P}}{\mathcal{P} \vdash \tau^{\leftarrow} \leq \tau^{\rightarrow}}$$

Fig. 4. Reductive Subtyping with Assumptions

To formalize this correspondence between premises in the rule and assumptions in the converted proof, Figure 3 defines proofs of declarative subtyping with assumptions.

Using this new definition, we can formalize the above conversion as follows:

$$\frac{\tau \leq \tau_i}{\tau \leq \tau_1 \cup \tau_2} \quad \xmapsto{\text{reductive-to-declarative conversion}} \quad \frac{\dfrac{\langle \tau, \tau_i \rangle \in \{\langle \tau, \tau_i \rangle\}}{\{\langle \tau, \tau_i \rangle\} \vdash \tau <: \tau_i} \qquad \dfrac{}{\{\langle \tau, \tau_i \rangle\} \vdash \tau_i <: \tau_1 \cup \tau_2}}{\{\langle \tau, \tau_i \rangle\} \vdash \tau <: \tau_1 \cup \tau_2}$$

One can derive similar conversions for the remaining standard reductive rules as well. All that remains are the custom reductive literal rules.

Requirement 4 (Reductive-to-Declarative Literal Conversion) $\quad \forall r \in \mathcal{R}.\ \mathcal{P}_r \vdash \ell_r^{\leftarrow} <: \ell_r^{\rightarrow}$

For our toy functional language, the conversion for the rule $\mathsf{Fun}(\tau_i, \tau_i', \tau_o, \tau_o')$ is the following:

$$\frac{\tau_i' \leq \tau_i \qquad \tau_o \leq \tau_o'}{\tau_i \to \tau_o \leq \tau_i' \to \tau_o'} \quad \xmapsto{\text{reductive-to-declarative conversion}} \quad \frac{\dfrac{\tau_i' <: \tau_i}{\tau_i \to \tau_o <: \tau_i' \to \tau_o} \qquad \dfrac{\tau_o <: \tau_o'}{\tau_i' \to \tau_o <: \tau_i' \to \tau_o'}}{\tau_i \to \tau_o <: \tau_i' \to \tau_o'}$$

These conversions enable us to inductively translate proofs of reductive subtyping into proofs of declarative subtyping. For the other direction, one needs the following, using proofs of reductive subtyping with assumptions as defined in Figure 4:

Requirement 5 (Declarative-to-Reductive Literal Conversion) $\quad \forall r \in \mathcal{D}.\ \mathcal{P}_r \vdash \ell_r^{\leftarrow} \leq \ell_r^{\rightarrow}$

With this and reflexivity, one can inductively translate proofs of declarative subtyping into reductive subtyping *provided* one can show reductive subtyping is transitive.

3.4.3 Transitivity. The last step to proving equivalence is transitivity, and also the most difficult step. Fortunately, one can once again repurpose well-foundedness to prove transitivity by reducing goals, just as was done for reflexivity. But whereas before goals were always of the form $\tau \leq \tau$, now they are of the form $\tau \leq \tau''$ where we have a proof of $\tau \leq \tau'$ and a proof of $\tau' \leq \tau''$ for some τ'. The key is to do a case analysis on the two proofs, taking advantage of the fact that τ' occurs in the conclusions of both proofs in order to eliminate a number of combinations of cases.

As an example, suppose the last steps of the two proofs are respectively as follows:

$$\textsc{Left}\ \frac{\tau \leq \tau_1' \qquad \tau \leq \tau_2'}{\tau \leq \tau_1' \cap \tau_2'} \qquad \frac{\tau_1' \leq \tau''}{\tau_1' \cap \tau_2' \leq \tau''}\ \textsc{Right}$$

Then this implies we have a proof of $\tau \leq \tau_1'$ and a proof of $\tau_1' \leq \tau''$. One can recursively reduce these two smaller proofs to get a proof of $\tau \leq \tau''$.

One can develop similar reductions for all the other possible cases in which either one of the last steps of the proofs is a standard reductive rule. And in every recursive reduction, the size of the proofs being reduced is strictly smaller. This ensures that reduction always eventually arrives

at the case where both of the last steps are custom reductive literal rules. Here one can use well-foundedness to guarantee termination *provided* one can find a custom reductive literal rule to apply. Thus the largest step typically involved in proving equivalence is showing the following property.

Requirement 6 (Literal Transitivity) *For every pair of rules $r^{\leftarrow}$ and $r^{\rightarrow}$ in $\mathcal{R}$ such that $\ell^{\rightarrow}_{r^{\leftarrow}} = \ell^{\leftarrow}_{r^{\rightarrow}}$, there exists a rule r in $\mathcal{R}$ such that $\ell^{\leftarrow}_{r} = \ell^{\leftarrow}_{r^{\leftarrow}}$ and $\ell^{\rightarrow}_{r} = \ell^{\rightarrow}_{r^{\rightarrow}}$ and the following holds:*

$$\forall \langle \tau^{\leftarrow}, \tau^{\rightarrow} \rangle \in \mathcal{P}_r.\ \exists \tau.\ \ (\mathcal{P}_{r^{\leftarrow}} \vdash \tau^{\leftarrow} \leq \tau\ \wedge\ \mathcal{P}_{r^{\rightarrow}} \vdash \tau \leq \tau^{\rightarrow})\ \ \vee\ \ (\mathcal{P}_{r^{\rightarrow}} \vdash \tau^{\leftarrow} \leq \tau\ \wedge\ \mathcal{P}_{r^{\leftarrow}} \vdash \tau \leq \tau^{\rightarrow})$$

The disjunction in the above requirement corresponds to the notion of variance in subtyping. Conceptually, a *covariant* premise of a rule is one in which the above will be proven using the left case of the disjunction, whereas a *contraviariant* premise uses the right case. The only difference is which assumptions the left-hand and right-hand subtyping proofs can make. In the covariant case, the proof of the left/right-hand subtyping can assume the premises of the same-handed rule, whereas in the contravariant case the proof of the left/right-hand subtyping can assume the properties of the opposite-handed rule. This is best illustrated by our toy functional language.

Suppose $r^{\leftarrow}$ is $\mathsf{Fun}(\tau_i, \tau_i', \tau_o, \tau_o')$ and $r^{\rightarrow}$ is $\mathsf{Fun}(\tau_i', \tau_i'', \tau_o', \tau_o'')$, in combination concluding with $\tau_i \to \tau_o \leq \tau_i' \to \tau_o' \leq \tau_i'' \to \tau_o''$. Then r is $\mathsf{Fun}(\tau_i, \tau_i'', \tau_o, \tau_o'')$, concluding with $\tau_i \to \tau_o \leq \tau_i'' \to \tau_o''$ as required. The following shows how the premises of r are proven with appropriate assumptions:

$$\frac{\langle \tau_i'', \tau_i' \rangle \in \mathcal{P}_{r^{\rightarrow}}}{\mathcal{P}_{r^{\rightarrow}} \vdash \tau_i'' \leq \tau_i'} \quad \wedge \quad \frac{\langle \tau_i', \tau_i \rangle \in \mathcal{P}_{r^{\leftarrow}}}{\mathcal{P}_{r^{\leftarrow}} \vdash \tau_i' \leq \tau_i} \qquad \frac{\langle \tau_o, \tau_o' \rangle \in \mathcal{P}_{r^{\leftarrow}}}{\mathcal{P}_{r^{\leftarrow}} \vdash \tau_o \leq \tau_o'} \quad \wedge \quad \frac{\langle \tau_o', \tau_o'' \rangle \in \mathcal{P}_{r^{\rightarrow}}}{\mathcal{P}_{r^{\rightarrow}} \vdash \tau_o' \leq \tau_o''}$$

Notice that the left-hand subtyping proof for *input* types uses the assumptions of the right-hand rule, whereas the left-hand subtyping proof for *output* types uses the assumptions of the left-hand rule. That is, the premise for input types is proven contravariantly, whereas the premise for output types is proven covariantly, which reflects the fact that function types are contravariant with respect to their input types and covariant with respect to their output types.

With this final requirement, one can prove that reductive subtyping is transitive. This was the final gap between declarative subtyping and reductive subtyping. Consequently, one finally knows that declarative subtyping and reductive subtyping are equivalent, and that proof search for reductive subtyping is a sound and complete algorithm for declarative subtyping. We formalize this general pattern in developing algorithms for subtyping systems with the following theorem.

Theorem (Decidability of Declarative Subtyping) *For every set of literals ℓ, set of declarative rules $\mathcal{D}$, and set of reductive rules $\mathcal{R}$ satisfying Requirements 1 through 6, proof search for reductive subtyping as defined in Figure 2 is a decision procedure for declarative subtyping as defined in Figure 1.*

Proof. Mechanically proved in Coq as outlined above [Muehlboeck and Tate 2018]. □

4 EMPOWERING UNIONS AND INTERSECTIONS

The previous section discussed typical decidable type systems with union and intersection types. Unfortunately, these typical systems fail to recognize many useful subtyping relations. Distributivity is one example, but more interesting are the relations specific to the particular literals at hand.

For example, consider our toy functional language. Suppose we have determined that a particular expression always produces a value of type τ_1 when given an integer. Suppose we have also determined that that same expression always produces a value of type τ_2 when given an integer. For the λ-calculus, Pierce recognized that, in this situation, that expression will always output values belonging to both τ_1 and τ_2 when given an integer. Thus, for Forsythe [Reynolds 1988, 1997] he proposed adding the following axiom to the subtyping rules [Pierce 1989]:

$$\frac{}{(\tau \to \tau_1') \cap (\tau \to \tau_2') <: \tau \to (\tau_1' \cap \tau_2')}$$

$$\left(\begin{array}{c}\text{All rules in Figure 1,}\\ \text{replacing } \tau^{\leftarrow} <: \tau^{\rightarrow} \text{ with } \tau^{\leftarrow} <:_{\mathcal{E}} \tau^{\rightarrow}\end{array}\right)$$

$$\frac{}{\tau \cap (\tau_1 \cup \tau_2) <:_{\mathcal{E}} (\tau \cap \tau_1) \cup (\tau \cap \tau_2)}$$

$$\frac{\langle \tau^{\leftarrow}, \tau^{\rightarrow} \rangle \in \mathcal{E}}{\tau^{\leftarrow} <:_{\mathcal{E}} \tau^{\rightarrow}}$$

Fig. 5. Extended Subtyping

τ	$\mathrm{DNF}_c(\tau)$ where $c : \vec{\ell} \to \tau$
$\bot$	$\bot$
$\tau_1 \cup \tau_2$	$\mathrm{DNF}_c(\tau_1) \cup \mathrm{DNF}_c(\tau_2)$
$\top$	$c([])$
$\tau_1 \cap \tau_2$	$\mathrm{DNF}_{\lambda \vec{\ell}_1 .\, \mathrm{DNF}_{\lambda \vec{\ell}_2 . c(\vec{\ell}_1 + \vec{\ell}_2)}(\tau_2)}(\tau_1)$
ℓ	$c([\ell])$

Fig. 6. Definition of DNF_c

The goal of our framework is to develop decision procedures for declarative subtyping systems empowered with such axioms (and distributivity). That is, given a set $\mathcal{E} \subseteq \tau \times \tau$, we aim to provide a decision procedure for *extended* subtyping as defined in Figure 5. Furthermore, since the declarative subtyping system being extended had already been proven decidable, we want to reuse as much of that algorithm and proof work as possible, only requiring the designer to do the work specific to the extensions at hand. Since distributivity is an extension of arbitrary union and intersection types, we tackle that extension first. Afterwards, we will consider extensions incorporating literals, such as the extension $\mathcal{E}_{\mathrm{Out}}$ defined as $\{\langle (\tau \to \tau_1') \cap (\tau \to \tau_2'), \tau \to (\tau_1' \cap \tau_2') \rangle \mid \tau, \tau_1', \tau_2'\}$, which corresponds to the above extension to Forsythe proposed by Pierce.

4.1 Distributivity

Observe that the right-hand type of the distributivity axiom in Figure 5 is simply the result of distributing the intersection in the left-hand type over the union in the left-hand type. That is, there is an operation we can apply to the left-hand type to arrive at the right-hand type. The high-level strategy of our *integrated*-subtyping technique is to transform the left-hand type in order to *integrate* the axiom into the type itself. In the case of distributivity, this integrating operator simply maps a type to its disjunctive normal form, effectively distributing all its intersections over its unions. Prior works have employed a similar strategy [Aiken and Wimmers 1993; Ancona and Corradi 2016; Frisch et al. 2008; Jones et al. 2007; Pierce 1991; Reynolds 1988], but here we demonstrate that this can be employed in both a principled and extensible manner.

We formalize this operator in Figure 6 using a continuation-style implementation. The intermediate continuations effectively collect the literals to be intersected together, calling the original continuation c after all the literals have been collected. By instantiating c with the operator $\bigcap$ that simply intersects the literals together, $\mathrm{DNF}_{\cap}$ maps each type to its disjunctive normal form.

The operator $\mathrm{DNF}_{\cap}$ exhibits three important properties. First, $\mathrm{DNF}_{\cap}(\tau \cap (\tau_1 \cup \tau_2))$ is declaratively/reductively a subtype of $(\tau \cap \tau_1) \cup (\tau \cap \tau_2)$. This illustrates that $\mathrm{DNF}_{\cap}$ integrates the distributivity axiom into the left-hand type, which will ensure that integrated subtyping is distributive. Second, $\mathrm{DNF}_{\cap}(\tau)$ is declaratively/reductively a subtype of τ, which will ensure that integrated subtyping is still reflexive. Third, whenever τ is in disjunctive normal form, say by being in the image of $\mathrm{DNF}_{\cap}$, one can show that τ is declaratively/reductively a subtype of $\mathrm{DNF}_{\cap}(\tau')$ whenever τ is declaratively/reductively a subtype of τ', which will ensure that integrated subtyping is still transitive. As a consequence of these properties, integrated subtyping will be equivalent to extended subtyping. But before we properly define integrated subtyping, we want to consider how to integrate extensions beyond distributivity as well.

4.2 Intersectors

As we distribute intersections over unions, we have the opportunity to do more with the resulting union of intersections. In particular, DNF accepts any operator mapping a list of literals to a type.

$$\frac{\mathrm{DNF}_{\sqcap}(\tau) \leq^{\sqcap} \tau'}{\tau \leq_{\sqcap} \tau'} \qquad \frac{\begin{pmatrix}\text{All rules in Figure 2 except literal subtyping,}\\ \text{replacing } \tau^{\leftarrow} \leq \tau^{\rightarrow} \text{ with } \tau^{\leftarrow} \leq^{\sqcap} \tau^{\rightarrow}\end{pmatrix} \quad r \in \mathcal{R} \qquad \forall \langle \tau^{\leftarrow}, \tau^{\rightarrow} \rangle \in \mathcal{P}_r.\ \ \mathrm{DNF}_{\sqcap}(\tau^{\leftarrow}) \leq^{\sqcap} \tau^{\rightarrow}}{\ell_r^{\leftarrow} \leq^{\sqcap} \ell_r^{\rightarrow}}$$

Fig. 7. Integrated Subtyping

Instead of simply intersecting the literals together, this operator could incorporate domain-specific reasoning about the literals. For example, the operator could check whether two literals in the list are disjoint classes, in which case it could simply return $\bot$. This provides a means to integrate more extensions into the type as we transform it.

We call such an operator an *intersector*, and we denote intersectors using $\bigsqcap$. One simple case is where the intersector is simply $\bigcap$, which has the effect of only adding distributivity. But since the truly interesting expressivity comes from the domain-specific extensions $\mathcal{E}$, we use the following requirement to formalize the intent that the intersector $\bigsqcap$ combines with DNF to form an *integrator* $\mathrm{DNF}_{\sqcap}$ that integrates those extensions into the types.

Requirement 7 (Intersector Completeness) $\quad \forall \langle \tau^{\leftarrow}, \tau^{\rightarrow} \rangle \in \mathcal{E}.\ \ \mathrm{DNF}_{\sqcap}(\tau^{\leftarrow}) <: \tau^{\rightarrow}$

One easy way to integrate all extensions into an intersector is simply to define the intersector so that it always outputs $\bot$. Such an intersector is clearly undesirable because it conceptually is doing too much; that is, it is integrating more information into the type than can be deduced from the extensions. As such, we also impose the following requirement that ensures the intersector integrates nothing more than what the extensions can deduce.

Requirement 8 (Intersector Soundness) $\quad \forall \vec{\ell}.\ \ \bigcap \vec{\ell} <:_{\mathcal{E}} \bigsqcap \vec{\ell}$

As an example, consider the following intersector $\bigsqcap_{\mathrm{Out}}$ for our running Forsythe extension $\mathcal{E}_{\mathrm{Out}}$:

$$\bigsqcap_{\mathrm{Out}} \vec{\ell} = \bigcap_{\varnothing \subset \mathcal{L} \subseteq \vec{\ell}} \left(\bigcap\nolimits_{(\tau_i \to \tau_o) \in \mathcal{L}} \tau_i \right) \to \left(\bigcap\nolimits_{(\tau_i \to \tau_o) \in \mathcal{L}} \tau_o \right)$$

To see why this works, consider the case in which there are two function-type literals $\tau_i \to \tau_o$ and $\tau_i' \to \tau_o'$. In this case, the result of $\bigsqcap_{\mathrm{Out}}$ is $(\tau_i \to \tau_o) \cap (\tau_i' \to \tau_o') \cap (\tau_i \cap \tau_i' \to \tau_o \cap \tau_o')$. If τ_i equals τ_i' so that we can refer to both as, say, τ, then the last component of this intersection is clearly equivalent to $\tau \to (\tau_o \cap \tau_o')$, thereby integrating the extension $\mathcal{E}_{\mathrm{Out}}$. Note that variance of function types makes $\bigsqcap_{\mathrm{Out}}$ still sound with respect to $\mathcal{E}_{\mathrm{Out}}$ even when τ_i does not equal τ_i'.

4.3 Integrated Subtyping

Now that we have an intersector of literals $\bigsqcap$ that corresponds to the intended extension $\mathcal{E}$, we define our integrated subtyping rules in Figure 7, incorporating the intersector by *integrating* left-hand types using $\mathrm{DNF}_{\sqcap}$. The integrated subtyping rules essentially describe the same algorithm as the reductive subtyping rules with just two small changes. The first is that integrated subtyping $\leq_{\sqcap}$ starts by integrating the left-hand type and then calling the recursive procedure described by $\leq^{\sqcap}$. The second is that, when this procedure recurses in the case of literals, it again integrates the left-hand type. The effect of this is that the left-hand type is always *integrated* at critical points where the procedure recurses, meaning it is essentially in the image of $\mathrm{DNF}_{\sqcap}$.

Lemma 1 (Integrated Soundness) *Assuming hitherto requirements:*

$$\forall \tau^{\leftarrow}, \tau^{\rightarrow}.\ \ \tau^{\leftarrow} \leq_{\sqcap} \tau^{\rightarrow} \implies \tau^{\leftarrow} <:_{\mathcal{E}} \tau^{\rightarrow}$$

Soundness of integrated subtyping with respect to extended subtyping follows from simply adapting the proof of soundness of reductive typing to incorporate Requirement 8 whenever integrated subtyping integrates the left-hand type. Completeness, on the other hand, is much more challenging to achieve. Requirement 7 is just one of many steps to this effect. In the following, we first ensure that integrated subtyping is still decidable, and then the remainder of the section establishes completeness so that we can use the decision procedure for integrated subtyping as a decision procedure for extended subtyping. As our running example, we will demonstrate that these steps produce a decision procedure for our toy functional language extended with $\mathcal{E}_{\mathsf{Out}}$.

4.4 Decidability

Integrated subtyping is syntax-directed for the same reason as reductive subtyping. Thus we can decide integrated subtyping by recursively searching the proof space *provided* we can guarantee its recursion terminates. We already have a well-founded measure for reductive subtyping, and we can repurpose it to prove termination of integrated subtyping provided the intersector preserves it.

Requirement 9 (Measure Preservation) $\forall \tau^{\leftarrow}, \tau^{\rightarrow}.\ \ m(\mathrm{DNF}_{\sqcap}(\tau^{\leftarrow}), \tau^{\rightarrow}) \leq m(\tau^{\leftarrow}, \tau^{\rightarrow})$

In the common case where m is defined on unions and intersections using joins, this amounts to only ensuring that $m(\bigsqcap \vec{\ell}, \tau^{\rightarrow})$ is always less than or equal to $m(\bigcap \vec{\ell}, \tau^{\rightarrow})$. For $\bigsqcap_{\mathsf{Out}}$ this holds because the measures of both sides are in fact always equal. That is, $\bigsqcap_{\mathsf{Out}}$ preserves termination for our toy functional language because it does not increase the nesting depth of function types.

Lemma 2 (Integrated Decidability) *Assuming hitherto requirements, the relation $\leq_{\sqcap}$ is decidable.*

4.5 Integrating

Clearly the literal intersector $\bigsqcap$ and the corresponding type integrator $\mathrm{DNF}_{\sqcap}$ are the key new components of integrated subtyping. The high-level intuition of how they work is that the integrator $\mathrm{DNF}_{\sqcap}$ forms a comonad on subtyping. Integrated subtyping then is *conceptually* akin to the Kleisli category of this comonad. However, integrated subtyping *recursively* integrates the comonad into its rules and its corresponding proof-search algorithm. As such, we cannot simply reuse standard theorems from category theory to achieve our results, and much of the challenge lies in addressing this recursive integration of $\mathrm{DNF}_{\sqcap}$ into the subtyping system. But we can still use these theorems as inspiration in establishing the equivalence of integrated and extended subtyping.

4.5.1 Dereliction. Intuitively, the intersector $\bigsqcap$ makes the type more precise. That is, $\bigsqcap$ should add information to the type, not remove information, as formalized by the following requirement.

Requirement 10 (Literal Dereliction) $\forall \vec{\ell}.\ \ \bigsqcap \vec{\ell} \leq \bigcap \vec{\ell}$

This trivially holds for $\bigsqcap_{\mathsf{Out}}$ since the resulting intersection always contains the input literals.

Lemma 3 (Dereliction) *Assuming hitherto requirements:* $\forall \tau^{\leftarrow}, \tau^{\rightarrow}.\ \tau^{\leftarrow} \leq \tau^{\rightarrow} \implies \mathrm{DNF}_{\sqcap}(\tau^{\leftarrow}) \leq \tau^{\rightarrow}$

4.5.2 Intersected. Just like $\mathrm{DNF}_{\cap}$ produces types in disjunctive normal form, we need some property that describes the key trait of the image of the intersector $\bigsqcap$. Let ϕ be some such property of lists of literals. We say a list of literals is *intersected* if it satisfies ϕ. We extend this predicate to arbitrary types using the dnf_{ϕ} predicate defined in Figure 8, which informally states that a type is *integrated* if it is a union of intersections of *intersected* lists of literals.

Just like the goal of $\mathrm{DNF}_{\cap}$ is to produce types that are in disjunctive normal form, the goal of $\mathrm{DNF}_{\sqcap}$ is to produce integrated types. This is ensured by the following requirement.

Requirement 11 (Intersector Integrated) $\forall \vec{\ell}.\ \ \mathrm{dnf}_{\phi}(\bigsqcap \vec{\ell})$

τ	$\mathrm{dnf}_\phi(\tau)$ where $\phi \subseteq \vec{\ell}$	$\mathrm{dnf}^\cap_\phi(\tau)$ where $\phi \subseteq \vec{\ell}$
$\bot$	**true**	**false**
$\tau_1 \cup \tau_2$	$\mathrm{dnf}_\phi(\tau_1) \wedge \mathrm{dnf}_\phi(\tau_2)$	**false**
$\top$	$\mathrm{dnf}^\cap_\phi(\top)$	$\phi([])$
$\tau_1 \cap \tau_2$	$\mathrm{dnf}^\cap_\phi(\tau_1 \cap \tau_2)$	$\mathrm{dnf}^\cap_{\lambda\vec{\ell}_1.\,\mathrm{dnf}^\cap_{\lambda\vec{\ell}_2.\,\phi(\vec{\ell}_1 +\!\!+ \vec{\ell}_2)}(\tau_2)}(\tau_1)$
ℓ	$\mathrm{dnf}^\cap_\phi(\ell)$	$\phi([\ell])$

Fig. 8. Definition of dnf_ϕ

For our running example, we can define ϕ_{Out} to hold for a list of function-type literals when, given any two function types $\tau_1 \to \tau_1'$ and $\tau_2 \to \tau_2'$ in the list, the combined function-type literal $(\tau_1 \cap \tau_2) \to (\tau_1' \cap \tau_2')$ is also in the list up to syntactic equivalence, meaning the only differences are resolved by associativity, commutativity, and idempotence of intersection types.

Lemma 4 (Integrator Integrated) *Assuming hitherto requirements:* $\forall \tau.\ \mathrm{dnf}_\phi(\mathrm{DNF}_\sqcap(\tau))$

4.5.3 Promotion. For our purposes, the critical property of disjunctive normal form is that, whenever a type τ is in disjunctive normal form and is a declarative/reductive subtype of some other type τ', then τ is furthermore a declarative/reductive subtype of $\mathrm{DNF}_\cap(\tau')$. This property further substantiates the intuition that disjunctive normal form and $\mathrm{DNF}_\cap$ integrate distributivity directly into types. We want there to be a similar relationship between integrated types and the integrator $\mathrm{DNF}_\sqcap$. This relationship should extend from a similar relationship between intersected literals and the intersector $\bigsqcap$. The following requirement formalizes that relationship.

Requirement 12 (Literal Promotion) *For every two lists of literals $\vec{\ell}$ and $\vec{\ell}'$ and list of reductive literal subtyping rules $\vec{r} \subseteq \mathcal{R}$ such that*

$$\forall \ell' \in \vec{\ell}'.\ \exists r \in \vec{r}.\ \ell_r^{\rightarrow} = \ell' \qquad \wedge \qquad \forall r \in \vec{r}.\ \ell_r^{\leftarrow} \in \vec{\ell}$$

if $\phi(\vec{\ell})$ holds then there exists a list of reductive literal subtyping rules $\vec{r}_\sqcap \subseteq \mathcal{R}$ such that $\bigsqcap \vec{\ell}'$ is of the form $\cdots \cup \left(\bigcap_{r_\sqcap \in \vec{r}_\sqcap} \ell_{r_\sqcap}^{\rightarrow}\right) \cup \ldots$ and for all reductive literal subtyping rules $r_\sqcap$ in $\vec{r}_\sqcap$

$$\ell_{r_\sqcap}^{\leftarrow} \in \vec{\ell} \qquad \wedge \qquad \forall \langle \tau^{\leftarrow}, \tau^{\rightarrow} \rangle \in \mathcal{P}_{r_\sqcap}.\ \left(\bigcup_{r \in \vec{r}} \mathcal{P}_r\right) \vdash \tau^{\leftarrow} \leq \tau^{\rightarrow}$$

The premise of Requirement 12 indicates we have some list of rules ($\vec{r}$) that can be used to prove that one intersection of literals ($\bigcap \vec{\ell}$) is a subtype of another ($\bigcap \vec{\ell}'$). The requirement, then, is essentially that if $\vec{\ell}$ is furthermore *intersected* (i.e. $\phi(\vec{\ell})$ holds) then it should be possible to prove from the combined premises of $\vec{r}$ that $\bigcap \vec{\ell}$ is furthermore a subtype of $\bigsqcap \vec{\ell}'$. More specifically, the requirement is that there is a list of rules ($\vec{r}_\sqcap$) that proves that $\bigcap \vec{\ell}$ is a subtype of $\bigsqcap \vec{\ell}'$ and whose premises can be proven from the premises of $\vec{r}$. In a sense, Requirement 12 states that there is a proof-theoretic analog to intersectors $\bigsqcap$ in that the list of rules $\vec{r}_\sqcap$ is the "promoted" analog of $\vec{r}$.

For our running example, suppose the list of rules $\vec{r}$ consists of the rules r_1 and r_2, which are $\mathsf{Fun}(\tau_i^1, \tau_i^{1\prime}, \tau_o^1, \tau_o^{1\prime})$ and $\mathsf{Fun}(\tau_i^2, \tau_i^{2\prime}, \tau_o^2, \tau_o^{2\prime})$ respectively. This means the list of literals $\vec{\ell}'$ is $\tau_i^{1\prime} \to \tau_o^{1\prime}$ and $\tau_i^{2\prime} \to \tau_o^{2\prime}$, and it means the list of literals $\vec{\ell}$ contains at least $\tau_i^1 \to \tau_o^1$ and $\tau_i^2 \to \tau_o^2$. The result of applying the intersector $\bigsqcap_{\mathsf{Out}}$ to $\vec{\ell}'$ is then

$$(\tau_i^{1\prime} \to \tau_o^{1\prime}) \cap (\tau_i^{2\prime} \to \tau_o^{2\prime}) \cap (\tau_i^{1\prime} \cap \tau_i^{2\prime} \to \tau_o^{1\prime} \cap \tau_o^{2\prime})$$

This intersection is comprised of three literals, which means we need three "promoted" rules $\vec{r}_\sqcap$. In this case the first two rules are simply r_1 and r_2, whose premises are trivially provable from

the premises of $\vec{r}$. For the last rule, we need a rule that proves that some literal in $\vec{\ell}$ is a subtype of $\tau_i^{1\prime} \cap \tau_i^{2\prime} \to \tau_o^{1\prime} \cap \tau_o^{2\prime}$. In general, such a rule does not necessarily exist, but here we are allowed to assume that $\vec{\ell}$ is *intersected*, meaning it satisfies ϕ_{Out}. Since we know $\vec{\ell}$ contains at least $\tau_i^1 \to \tau_o^1$ and $\tau_i^2 \to \tau_o^2$, by the definition of ϕ_{Out} we know $\vec{\ell}$ must also contain the literal $(\tau_i^1 \cap \tau_i^2) \to (\tau_o^1 \cap \tau_o^2)$ (or something syntactically equivalent). Thus we can use $\mathsf{Fun}(\tau_i^1 \cap \tau_i^2, \tau_i^{1\prime} \cap \tau_i^{2\prime}, \tau_o^1 \cap \tau_o^2, \tau_o^{1\prime} \cap \tau_o^{2\prime})$ (or something syntactically equivalent) as our final "promoted" rule, since the fact that $\tau_i^{1\prime} \cap \tau_i^{2\prime}$ is a subtype of $\tau_i^1 \cap \tau_i^2$ and that $\tau_o^1 \cap \tau_o^2$ is a subtype of $\tau_o^{1\prime} \cap \tau_o^{2\prime}$ is easily proven from the *combined* premises of $\vec{r}$ (which are $\langle \tau_i^{1\prime}, \tau_i^1 \rangle$, $\langle \tau_o^1, \tau_o^{1\prime} \rangle$, $\langle \tau_i^{2\prime}, \tau_i^2 \rangle$, and $\langle \tau_o^2, \tau_o^{2\prime} \rangle$).

Lemma 5 (Promotion) *Assuming hitherto requirements:*

$$\forall \tau^{\leftarrow}, \tau^{\rightarrow}.\ \ \mathrm{dnf}_{\phi}(\tau^{\leftarrow}) \ \ \wedge \ \ \tau^{\leftarrow} \leq \tau^{\rightarrow} \implies \tau^{\leftarrow} \leq \mathrm{DNF}_{\sqcap}(\tau^{\rightarrow})$$

Promotion for reductive subtyping is easily proven by induction. However, promotion for integrated subtyping is not nearly so simple. The problem is that literal promotion makes use of reductive proofs with assumptions. While such proofs can easily be converted into reductive subtyping (without assumptions) when reductive subtyping holds for each of the assumptions, the same is not true for integrated subtyping because its subtyping rule for literals differs. In order to bridge this difference, one first needs to prove that integrated subtyping is monotonic with respect to reductive subtyping, and the proof of this involves a complex mix of transitivity reduction, promotion of reductive subtyping, and "big-step" well-founded coinduction alternating with "small-step" induction. Fortunately, our framework abstracts away all this complex proof machinery from the concerns of its users, providing the following convenient lemmas.

Lemma 6 (Integrated Monotonicity) *Assuming hitherto requirements:*

$$\forall \tau_1, \tau_2, \tau_3, \tau_4.\ \ \tau_1 \leq \tau_2 \ \ \wedge \ \ \tau_2 \leq_{\sqcap} \tau_3 \ \ \wedge \ \ \tau_3 \leq \tau_4 \implies \tau_1 \leq_{\sqcap} \tau_4$$

Lemma 7 (Integrated Assumptions) *Assuming hitherto requirements:*

$$\forall \mathcal{P}, \tau^{\leftarrow}, \tau^{\rightarrow}.\ \ \mathcal{P} \vdash \tau^{\leftarrow} \leq \tau^{\rightarrow} \implies \left(\forall \langle \tau_p^{\leftarrow}, \tau_p^{\rightarrow} \rangle \in \mathcal{P}.\ \ \tau_p^{\leftarrow} \leq_{\sqcap} \tau_p^{\rightarrow} \right) \implies \tau^{\leftarrow} \leq_{\sqcap} \tau^{\rightarrow}$$

Lemma 8 (Integrated Promotion) *Assuming hitherto requirements:*

$$\forall \tau^{\leftarrow}, \tau^{\rightarrow}.\ \ \tau^{\leftarrow} \leq_{\sqcap} \tau^{\rightarrow} \implies \tau^{\leftarrow} \leq_{\sqcap} \mathrm{DNF}_{\sqcap}(\tau^{\rightarrow})$$

4.6 Equivalence of Extended and Integrated Subtyping

We have now established the essential comonad-like properties of $\mathrm{DNF}_{\sqcap}$. Now we move on to utilizing these properties to prove that integrated subtyping admits the rules of extended subtyping that were removed to achieve decidability.

4.6.1 Reflexivity. Reflexivity is a corollary of the prior lemmas. In particular, a reductive proof with assumptions in which the assumption set $\mathcal{P}$ is empty directly corresponds to reductive subtyping, so Lemma 7 indicates that reductive subtyping implies integrated subtyping. As a consequence, reflexivity of reductive subtyping implies reflexivity of integrated subtyping.

Lemma 9 (Integrated Reflexivity) *Assuming hitherto requirements:* $\forall \tau.\ \tau \leq_{\sqcap} \tau$.

4.6.2 Rule Conversion. Integrated subtyping relies upon the reductive literal subtyping rules $\mathcal{R}$, whereas extended subtyping relies upon the declarative literal subtyping rules $\mathcal{D}$. For our proof of equivalence of declarative and reductive subtyping, we already required a conversion of declarative literal rules into reductive subtyping proofs with assumptions (Requirement 5), and for our proof of integrated promotion, we already demonstrated that reductive proofs with assumptions can be converted into integrated subtypings under appropriate assumptions (Lemma 7). As a consequence,

we can simply reuse the required conversions of literal rules to prove that integrated subtyping also admits the declarative literal rules.

Lemma 10 (Declarative-to-Integrated Literal Conversion) *Assuming hitherto requirements:*

$$\forall r \in \mathcal{D}.\ \ (\forall \langle \tau^{\leftarrow}, \tau^{\rightarrow} \rangle \in \mathcal{P}_r.\ \ \tau^{\leftarrow} \leq_{\sqcap} \tau^{\rightarrow}) \implies \ell_r^{\leftarrow} \leq_{\sqcap} \ell_r^{\rightarrow}$$

4.6.3 Transitivity. The final step is to prove transitivity of subtyping. As with reductive subtyping, conceptually this is done by recursively reducing the left-hand proof and right-hand proof, eventually making progress towards a combined subtyping proof, which in turn ensures termination due to well-foundedness of integrated subtyping. However, there is a key difference here that arises when reducing applications of literal rules, which we illustrate using our running example.

Suppose the left-hand proof is an application of $\mathsf{Fun}(\tau_i, \tau_i', \tau_o, \tau_o')$ and the right-hand proof is an application of $\mathsf{Fun}(\tau_i', \tau_i'', \tau_o', \tau_o'')$, meaning the integrated proofs being reduced appear as follows:

$$\dfrac{\dfrac{\vdots}{\mathrm{DNF}_{\sqcap}(\tau_i') \leq^{\sqcap} \tau_i} \quad \dfrac{\vdots}{\mathrm{DNF}_{\sqcap}(\tau_o) \leq^{\sqcap} \tau_o'}}{\tau_i \to \tau_i' \leq^{\sqcap} \tau_o \to \tau_o'} \qquad \dfrac{\dfrac{\vdots}{\mathrm{DNF}_{\sqcap}(\tau_i'') \leq^{\sqcap} \tau_i'} \quad \dfrac{\vdots}{\mathrm{DNF}_{\sqcap}(\tau_o') \leq^{\sqcap} \tau_o''}}{\tau_i' \to \tau_i'' \leq^{\sqcap} \tau_o' \to \tau_o''}$$

Notice there is a mismatch between the conclusions of the subproofs: the left-hand types are *integrated* but the right-hand types are not. But we can resolve this mismatch by *promoting* the appropriate proofs so that we can recursively reduce the transitive pairs of proofs for i and for o. Thus we can make progress towards proving $\tau_i \to \tau_i'' \leq^{\sqcap} \tau_o \to \tau_o''$ by applying $\mathsf{Fun}(\tau_i, \tau_i'', \tau_o, \tau_o'')$.

While this intuition is the essence of how transitivity is proven, the actual proof once again involves a complex mix of transitivity reduction, promotion of integrated subtyping, and "big-step" well-founded coinduction alternating with "small-step" induction. In particular, integrated subtyping is not itself reductive—transitivity is not simply achieved by showing that each cutpoint reduces. While that is a step of the proof, the much more challenging step is recursively incorporating the integrator into transitivity, with promotion essentially integrating the intended axioms into the subtyping proofs themselves just like the integrator does with types. Thus integrated subtyping is proof-theoretically more powerful than reductive subtyping, making it likely *necessary* for these more advanced subtyping systems. Fortunately, our framework abstracts away this complex proof machinery, providing the following convenient lemmas and the main theorem of this paper.

Lemma 11 (Integrated Transitivity) *Assuming hitherto requirements:*

$$\forall \tau_1, \tau_2, \tau_3.\ \ \tau_1 \leq_{\sqcap} \tau_2 \ \wedge\ \tau_2 \leq_{\sqcap} \tau_3 \implies \tau_1 \leq_{\sqcap} \tau_3$$

Lemma 12 (Integrated Completeness) *Assuming hitherto requirements:*

$$\forall \tau^{\leftarrow}, \tau^{\rightarrow}.\ \ \tau^{\leftarrow} <:_{\mathcal{E}} \tau^{\rightarrow} \implies \tau^{\leftarrow} \leq_{\sqcap} \tau^{\rightarrow}$$

Theorem (Decidability of Extended Subtyping) *For every set of literals ℓ, set of declarative rules $\mathcal{D}$, set of reductive rules $\mathcal{R}$, set of extensions $\mathcal{E}$, intersector of literals $\sqcap$, and intersected predicate ϕ satisfying Requirements 1 through 12, proof search for integrated subtyping as defined in Figure 7 is a decision procedure for extended subtyping as defined in Figure 5.*

Proof. Mechanically proved in Coq [Muehlboeck and Tate 2018], along with a proof that the proof search can be safely optimized by using a rule "prioritization". This means that when proof search encounters a situation in which multiple rules are applicable, it does not need to search the rules that have lower "priority" than some other applicable rule. In this case, the high-priority rules are the left union and bottom rules, the mid-priority rules are all the right rules, and the low-priority rules are the left intersection rules and the custom literal rules. □

Thus, since we already illustrated that $\bigsqcap_{\text{Out}}$ satisfies the requirements with respect to $\mathcal{E}_{\text{Out}}$, this means proof search for $\leq_{\bigsqcap_{\text{Out}}}$ is a decision procedure for $<:_{\mathcal{E}_{\text{Out}}}$. But $\mathcal{E}_{\text{Out}}$ is only one of the extensions Pierce considered for Forsythe, one that he already proved decidable (without union types) [Pierce 1989]. Rather than redo this work each time one considers another extension, next we discuss how one can develop each extension separately and then easily compose the extensions together with just one more simple proof. However, for the sake of space, we will be illustrating composability with our main application, Ceylon, rather than our toy functional language.

5 COMPOSABILITY

So far, we have had only a relatively simple integrator as an example. As we discussed in Section 2, Ceylon uses union and intersection types to build several type-system features, which requires more involved type integrators. We would like to be able to specify them separately and then only later compose them into a single master type integrator. In this section, we discuss how to do this.

Suppose one has developed two intersectors $\bigsqcap_1$ and $\bigsqcap_2$ and respective intersected predicates ϕ_1 and ϕ_2 on the same set of literals ℓ and literal subtyping rules $\mathcal{R}$. One can compose them as follows:

$$\bigsqcap \vec{\ell} ::= \text{DNF}_{\sqcap_2}(\bigsqcap_1 \vec{\ell}) \qquad\qquad \phi(\vec{\ell}) ::= \phi_1(\vec{\ell}) \wedge \phi_2(\vec{\ell})$$

The only thing one needs to show for this composition to work as expected is that $\bigsqcap_2$ preserves ϕ_1.

Requirement 13 (Intersected Preservation) $\forall \vec{\ell}.\ \ \phi_1(\vec{\ell}) \Rightarrow \text{dnf}_{\phi_1}(\bigsqcap_2 \vec{\ell})$

Theorem (Intersector Composability) *For every set of literals ℓ, set of declarative rules $\mathcal{D}$, set of reductive rules $\mathcal{R}$, and sets of extensions $\mathcal{E}_1$ and $\mathcal{E}_2$ with corresponding intersectors of literals $\bigsqcap_1$ and $\bigsqcap_2$ and intersected predicates ϕ_1 and ϕ_2 each satisfying Requirements 1 through 12, satisfying Requirement 13 implies the composed intersector $\bigsqcap \vec{\ell} = \text{DNF}_{\sqcap_2}(\bigsqcap_1 \vec{\ell})$ and intersected predicate $\phi(\vec{\ell}) = \phi_1(\vec{\ell}) \wedge \phi_2(\vec{\ell})$ satisfy Requirements 1 through 12 with respect to the extension $\mathcal{E}_1 \cup \mathcal{E}_2$.*

Proof. Mechanically proved in Coq as outlined above [Muehlboeck and Tate 2018]. □

6 APPLICATION TO CEYLON

We have been using a toy language as our running illustrative example. In this section, we will show how to use integrated subtyping to uniformly implement various aspects of an industry language, namely the Ceylon language [King 2013] and its features that we discussed in Section 2.

6.1 Unempowered Ceylon

Ceylon is an object-oriented language, and its literals are class types. More specifically, Ceylon uses generics with declaration-site variance, so its class types are parameterized with some number of arguments, some of which are covariant and some of which are contravariant. For the sake of concision, we approximate this expressiveness with single-parameter class types employing *use*-site variance. As such, our literals will be of the form $C\langle \textbf{in}\ \tau_i\ \textbf{out}\ \tau_o \rangle$, with the **in** argument being the *contravariant* argument to the parameter, and the **out** argument being the *covariant* argument. As an example, the literal `MutableList`$\langle$**in** `Integer`$\langle$**in** $\bot$ **out** $\top\rangle$ **out** `Number`$\langle$**in** $\bot$ **out** $\top\rangle\rangle$ represents a mutable list that one can put integers *in*to and get numbers *out* of. The classes `Integer` and `Number` make no use of their type parameter, so in such cases we will use $C\langle\rangle$ as shorthand for $C\langle$**in** $\bot$ **out** $\top\rangle$. Thus the example literal will be written as `MutableList`$\langle$**in** `Integer`$\langle\rangle$ **out** `Number`$\langle\rangle\rangle$.

Figure 9 shows the declarative and reductive literal subtyping rules for Ceylon. They assume a given relation $C\langle\cdot\rangle <:: C'\langle\dot{\tau}\rangle$ indicating that class C with type parameter $\cdot$ directly inherits class C' with type argument $\dot{\tau}$, where the grammar for parameterized types $\dot{\tau}$ is the same as for types τ with a single additional case $\cdot$ representing the type parameter. Because use-site variance supplies

$$\frac{\tau_i' <: \tau_i \qquad \tau_o <: \tau_o'}{C\langle \mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o\rangle <: C\langle \mathbf{in}\ \tau_i'\ \mathbf{out}\ \tau_o'\rangle} \qquad \frac{C\langle\cdot\rangle <:: C'\langle\dot\tau\rangle}{C\langle \mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o\rangle <: C'\langle \mathbf{in}\ \dot\tau[\mathbf{in}\ \tau_o\ \mathbf{out}\ \tau_i]\ \mathbf{out}\ \dot\tau[\mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o]\rangle}$$

$$\frac{}{C\langle\cdot\rangle \leq:: C\langle\cdot\rangle} \qquad \frac{C\langle\cdot\rangle <:: C'\langle\dot\tau\rangle \quad C'\langle\cdot\rangle \leq:: C''\langle\dot\tau'\rangle}{C\langle\cdot\rangle \leq:: C''\langle\dot\tau'[\dot\tau]\rangle}$$

$$\frac{C\langle\cdot\rangle \leq:: C'\langle\dot\tau\rangle \quad \tau_i' \leq \dot\tau[\mathbf{in}\ \tau_o\ \mathbf{out}\ \tau_i] \quad \dot\tau[\mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o] \leq \tau_o'}{C\langle \mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o\rangle \leq C'\langle \mathbf{in}\ \tau_i'\ \mathbf{out}\ \tau_o'\rangle}$$

$\dot\tau$	$\dot\tau[\mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o]$
$\cdot$	τ_o
$\bot$	$\bot$
$\dot\tau_1 \cup \dot\tau_2$	$\dot\tau_1[\mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o] \cup \dot\tau_2[\mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o]$
$\top$	$\top$
$\dot\tau_1 \cap \dot\tau_2$	$\dot\tau_1[\mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o] \cap \dot\tau_2[\mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o]$
$C\langle \mathbf{in}\ \dot\tau_i\ \mathbf{out}\ \dot\tau_o\rangle$	$C\langle \mathbf{in}\ \dot\tau_i[\mathbf{in}\ \tau_o\ \mathbf{out}\ \tau_i]\ \mathbf{out}\ \dot\tau_o[\mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o]\rangle$

Fig. 9. Declarative and Reductive Literal Subtyping Rules for Ceylon (assuming Material-Shape Separation)

two type arguments to each type parameter, substitution $\dot\tau[\mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o]$ substitutes all occurrences of $\cdot$ in $\dot\tau$ with τ_i in contravariant positions and τ_o in covariant positions. We show the subtyping rules in inference-rule format above for readability, but of course these presentations could be reformatted as collections of literal subtyping rules $\mathcal{D}_{\text{Ceylon}}$ and $\mathcal{R}_{\text{Ceylon}}$.

Similar to how subtyping for functions worked in our toy functional language, we had to merge the two rules for inheritance and variance into a single rule for reductive subtyping. The reductive literal subtyping rule is clearly syntax-directed. Its use of the reflexive-transitive closure $\leq::$ of the inheritance relation $<::$ makes it satisfy the literal reflexivity and transitivity requirements. The declarative and reductive rules admit each other, leaving only the well-foundedness requirement. Greenman, Muehlboeck, and Tate [2014] proposed a *Material-Shape Separation* restriction on the inheritance relation that makes reductive subtyping well-founded in the sense of Requirement 2, which we adopt. All of this implies that declarative and reductive subtyping for Ceylon are equivalent and decidable by the Decidability of Declarative Subtyping Theorem.

But declarative and reductive subtyping are incapable of the reasoning needed to support the various features of Ceylon discussed in Section 2. The remainder of this section illustrates how to extend subtyping with the necessary new functionality. We first discuss each feature independently, then show to use our Intersector Composability Theorem to unify all these features into one coherent and principled subtyping system. As we proceed, we make use of the following shorthands:

$$\text{ClassOf}(C\langle \mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o\rangle) = C \qquad C\langle\rangle = C\langle \mathbf{in}\ \bot\ \mathbf{out}\ \top\rangle \qquad \frac{C\langle\cdot\rangle <:: C'\langle\dot\tau\rangle}{C <:: C'} \qquad \frac{C\langle\cdot\rangle \leq:: C'\langle\dot\tau\rangle}{C \leq:: C'}$$

6.2 Disjointness

Ceylon reasons about when intersections of two types are uninhabitable. We can add disjointness reasoning to a nominal type system using the extension in Figure 10. First, we assume we are given a decidable and sound, but not necessarily complete, relation between classes indicating when two classes are disjoint. We further assume this relation is symmetric and respects inheritance. Second, we extend subtyping to say that, whenever two classes are disjoint, any intersection of their instantiations is a subtype of $\bot$. Third, we say that a list of literals is intersected if it does not contain any disjoint classes, and we define our intersector to replace any list of literals containing disjoint classes with $\bot$. Lastly, we prove that this definition satisfies the requirements outlined in Section 4. These proofs are straightforward, so we omit them here and throughout this section.

Given a decidable relation C dsj C'

$$\frac{C \text{ dsj } C'}{C' \text{ dsj } C} \qquad \frac{C <:: C' \qquad C' \text{ dsj } C''}{C \text{ dsj } C''}$$

Extended Subtyping

$$\frac{C \text{ dsj } C'}{C\langle \mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o\rangle \cap C'\langle \mathbf{in}\ \tau_i'\ \mathbf{out}\ \tau_o'\rangle <: \bot}$$

Integrated Subtyping

$$\phi_{\text{dsj}}(\vec{\ell}) = \nexists \ell, \ell' \in \vec{\ell}.\ \text{ClassOf}(\ell) \text{ dsj } \text{ClassOf}(\ell') \qquad \bigcap_{\text{dsj}} \vec{\ell} = \begin{cases} \bigcap \vec{\ell} & \text{if } \phi_{\text{dsj}}(\vec{\ell}) \\ \bot & \text{otherwise} \end{cases}$$

Fig. 10. Disjointness Extension

6.3 Principal Instantiation

Given an intersection $C\langle \mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o\rangle \cap C\langle \mathbf{in}\ \tau_i'\ \mathbf{out}\ \tau_o'\rangle$, Ceylon can combine the covariant and contravariant type arguments to generate the *principal instantiation* $C\langle \mathbf{in}\ \tau_i \cup \tau_i'\ \mathbf{out}\ \tau_o \cap \tau_o'\rangle$ of class C for that intersection. Figure 11 formalizes this extension. The formalization actually assumes a hierarchy satisfying single-instantiation inheritance, but this is only because our simplification lacks the declaration-site variance necessary to express the more flexible concept of principal-instantiation inheritance [King 2013]. In either case, the extension to subtyping is the same: contravariant type arguments are combined with unions, and covariant type arguments are combined with intersections. Integrating, on the other hand, is much more complicated, though most of the complexity is just tedium. The intersector needs to scale to arbitrary-sized lists of literals, and more importantly it must deal with the fact that inheritance can make the contributors to a principal

Given

$$\frac{C\langle\cdot\rangle \leq:: C'\langle\dot{\tau}\rangle \qquad C\langle\cdot\rangle \leq:: C'\langle\dot{\tau}'\rangle}{\dot{\tau} = \dot{\tau}'}$$

Extended Subtyping

$$\frac{}{C\langle \mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o\rangle \cap C\langle \mathbf{in}\ \tau_i'\ \mathbf{out}\ \tau_o'\rangle <: C\langle \mathbf{in}\ \tau_i \cup \tau_i'\ \mathbf{out}\ \tau_o \cap \tau_o'\rangle}$$

Integrated Subtyping

$$\text{PrinIn}_C(\vec{\ell}) = \left\{ \dot{\tau}[\mathbf{in}\ \tau_o\ \mathbf{out}\ \tau_i] \;\middle|\; C'\langle \mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o\rangle \in \vec{\ell} \wedge C'\langle\cdot\rangle \leq:: C\langle\dot{\tau}\rangle \right\}$$

$$\text{PrinOut}_C(\vec{\ell}) = \left\{ \dot{\tau}[\mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o] \;\middle|\; C'\langle \mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o\rangle \in \vec{\ell} \wedge C'\langle\cdot\rangle \leq:: C\langle\dot{\tau}\rangle \right\}$$

$$\text{Cover}(\vec{\ell}) = \bigcup_{C \subseteq \{\text{ClassOf}(\ell) \mid \ell \in \vec{\ell}\}} \text{LCA}_{<::}(C) \qquad \text{(where LCA}_{<::} \text{ is least common-ancestors w.r.t. <::)}$$

$$\phi_{\text{PrinInst}}(\vec{\ell}) = \forall C \in \text{Cover}(\vec{\ell}).\ \exists \vec{\ell}', \vec{\tau}_i, \vec{\tau}_o.\ \left(\begin{array}{c} \vec{\ell} = \vec{\ell}' \cup \{C\langle \mathbf{in}\ \bigcup \vec{\tau}_i\ \mathbf{out}\ \bigcap \vec{\tau}_o\rangle\} \\ \wedge \\ \text{PrinIn}_C(\vec{\ell}') \subseteq \vec{\tau}_i \quad \wedge \quad \text{PrinOut}_C(\vec{\ell}') \subseteq \vec{\tau}_o \end{array} \right)$$

$$\bigcap_{\text{PrinInst}} \vec{\ell} = \bigcap_{C \in \text{Cover}(\vec{\ell})} C\langle \mathbf{in}\ \bigcup \text{PrinIn}_C(\vec{\ell})\ \mathbf{out}\ \bigcap \text{PrinOut}_C(\vec{\ell})\rangle$$

Fig. 11. Principal-Instantiation Extension

Given an operator cases(C) that, if **defined**, specifies a finite and computable set of classes

$$\frac{\text{cases}(C)\ \textbf{defined}}{\text{cases}(C) \neq \varnothing} \qquad \frac{C' \in \text{cases}(C)}{C' <:: C \quad \wedge \quad \text{cases}(C')\ \textbf{undefined}} \qquad \frac{\text{cases}(C)\ \textbf{defined} \qquad C'\langle\cdot\rangle <:: C\langle\dot{\tau}\rangle}{C' \in \text{cases}(C) \quad \wedge \quad \dot{\tau} = \cdot}$$

Extended Subtyping

$$\frac{\text{cases}(C)\ \textbf{defined}}{C\langle \textbf{in}\ \tau_i\ \textbf{out}\ \tau_o\rangle <: \bigcup_{C' \in \text{cases}(C)} C'\langle \textbf{in}\ \tau_i\ \textbf{out}\ \tau_o\rangle}$$

Integrated Subtyping

$$\text{expand}(C\langle \textbf{in}\ \tau_i\ \textbf{out}\ \tau_o\rangle) = \begin{cases} \bigcup_{C' \in \text{cases}(C)} C'\langle \textbf{in}\ \tau_i\ \textbf{out}\ \tau_o\rangle & \text{if cases}(C)\ \textbf{defined} \\ C\langle \textbf{in}\ \tau_i\ \textbf{out}\ \tau_o\rangle & \text{otherwise} \end{cases}$$

$$\phi_{\text{cases}}(\vec{\ell}) = \nexists \ell \in \vec{\ell}.\ \text{cases}(\text{ClassOf}(\ell))\ \textbf{defined} \qquad \bigsqcap_{\text{cases}} \vec{\ell} = \text{DNF}_{\cap}\left(\bigcap_{\ell \in \vec{\ell}} \text{expand}(\ell)\right)$$

Fig. 12. Enumerated-Cases Extension (assuming Material-Shape Separation for cases)

instantiation arise indirectly[1]. The Cover function is used to identify the set of classes that can have such an indirect contribution. In Ceylon, Cover needs to be more advanced to address other features of the language, but for our simplified language we can just use least common-ancestors.

We should remark that this feature can also be applied to plain function types with the following:

$$\bigsqcap(\tau_i \to \tau_o, \tau_i' \to \tau_o') = (\tau_i \cup \tau_i') \to (\tau_o \cap \tau_o')$$

Such an extension is actually strictly more expressive than that of Forsythe. However, while this this extension is sound for Ceylon due to Ceylon's nominal nature, it is in fact unsound for Forsythe due to Forsythe's structural nature. Thus it is interesting and important that our framework can express both Ceylon's permissive extension and Forsythe's restrictive extension precisely.

6.4 Classes with Enumerated Cases

Ceylon provides algebraic data types by allowing classes to explicitly enumerate all permitted subclasses if desired. For example, the `Nat` data type could be defined as follows:

```
abstract class Nat of Zero, Succ<Nat> { ... }
class Zero extends Nat { ... }
class Succ<out N extends Nat> extends Nat { ... }
```

Because an enumeration is provided, Ceylon makes `Nat`⟨⟩ a subtype of `Zero`⟨⟩ ∪ `Succ`⟨`Nat`⟨⟩⟩. The same could be done for a `LinkedList` class with `Nil` and `Cons` cases. This is particularly useful in the context of `switch` statements because it unifies the reasoning for union types and sum types.

We formalize this extension in Figure 12. Here, a class C is a class with enumerated cases if the function cases(C) is defined. If defined, this function returns a finite set of classes. Any class in that set must directly inherit from C and not itself be a class with enumerated cases (as a simplification to avoid reasoning about chains of expansions). Conversely, if a class has enumerated cases, then any class that directly inherits from it must be in the set of cases and must not change the type arguments (in order to avoid the complexities of generalized algebraic data types). Again, this formalization

[1]For example, if $C_\ell\langle\cdot\rangle <:: C\langle\cdot\rangle$ and $C_r\langle\cdot\rangle <:: C\langle\cdot\rangle$, then the type $C_\ell\langle \textbf{in}\ \tau_i\ \textbf{out}\ \tau_o\rangle \cap C_r\langle \textbf{in}\ \tau_i'\ \textbf{out}\ \tau_o'\rangle$ must be a subtype of C's principal instantiation $C\langle \textbf{in}\ \tau_i \cup \tau_i'\ \textbf{out}\ \tau_o \cap \tau_o'\rangle$ despite not *directly* mentioning class C.

Given	Extended Subtyping
$\dfrac{}{C \leq:: \texttt{Object} \quad \vee \quad C \leq:: \texttt{Null}}$	$\dfrac{}{\top <: \texttt{Object}\langle\rangle \cup \texttt{Null}\langle\rangle}$

Integrated Subtyping

$$\phi_{\text{ObjNull}}(\vec{\ell}) = \exists \texttt{C}\langle \textbf{in } \tau_i \textbf{ out } \tau_o\rangle \in \vec{\ell} \qquad \bigsqcap_{\text{ObjNull}} \vec{\ell} = \begin{cases} \bigcap \vec{\ell} & \text{if } \phi_{\text{ObjNull}}(\vec{\ell}) \\ (\texttt{Object}\langle\rangle \bigcap \vec{\ell}) \cup (\texttt{Null}\langle\rangle \bigcap \vec{\ell}) & \text{otherwise} \end{cases}$$

Fig. 13. Object-Null Extension

is overly restrictive due to the lack of declaration-site variance, bounded type parameters, and limited arity, but it still captures the key components. Note that the intersector needs to apply $\text{DNF}_\cap$ because the expansions of cases could introduce a union inside the intersection. This is fine, though, because one can easily show that $\text{DNF}_\cap$ will not reintroduce unintegrated intersections.

Lastly, we chose the particular example of `Nat` for a reason. Note that in our example $\texttt{Nat}\langle\rangle$ gets expanded to $\texttt{Zero}\langle\rangle \cup \texttt{Succ}\langle\texttt{Nat}\langle\rangle\rangle$, which in turn contains $\texttt{Nat}\langle\rangle$. Although not supported by the simplification presented here, this example illustrates another reason why it is important to integrate $\bigsqcap$ into the recursive algorithm itself, rather than eagerly expand a type recursively, since an eager of expansion of $\texttt{Nat}\langle\rangle$ would continue forever.

However, with integration comes the problem that the termination measure might increase. In the case of this extension, we convert class literals to literals of *sub*classes, which may have a higher termination measure, as is the case for the measure that Greenman, Muehlboeck, and Tate [2014] provided. We address this by assuming Material-Shape Separation *for cases*. By this we mean that all of the declared superclasses (besides C) of the cases of a class C are well-defined class types prior to the declaration of C. This enables the measure of C to incorporate the measures of its cases so that replacing C with the union of its cases does not increase the measure. Ceylon can furthermore adapt the "swap" measure from Tate, Leung, and Lerner [2011], which intuitively measures how often types can "swap sides" due to contravariance as subtyping recurses. One can easily show that material-shape separation ensures that this "swap" measure is also a well-founded measure for reductive subtyping. The effect of using the "swap" measure instead is that the intersector need only ensure that it does not introduce new forms of contravariance in order to ensure decidability of integrated subtyping. For example, because the class `Succ` is covariant, the type $\texttt{Nat}\langle\rangle$ can be expanded to $\texttt{Zero}\langle\rangle \cup \texttt{Succ}\langle\texttt{Nat}\langle\rangle\rangle$ even though the expanded type again refers to $\texttt{Nat}\langle\rangle$.

6.5 `Object` and `Null`

In Ceylon, every value belongs to either `Object` or `Null`. Consequently, `Object` and `Null` are essentially the enumerated cases of $\top$. Even though $\top$ is not itself a class, we can still express this using an integrator, as shown in Figure 13. For this, a list of literals is intersected if it contains a class literal. In our simplified calculus, this is equivalent to saying the intersection is nonempty, but in Ceylon literals might also be type variables (which will be discussed in Section 7.4). The intersector, then, checks this condition and, if it fails to hold, simply distributes the intersection of literals through the union $\texttt{Object}\langle\rangle \cup \texttt{Null}\langle\rangle$. Thus in particular the empty intersection $\top$ becomes $\texttt{Object}\langle\rangle \cup \texttt{Null}\langle\rangle$. Interestingly, the Ceylon team actually avoided adding this feature because $\top$ is essentially everywhere and it was not clear that the original informal reasoning behind our technique safely applied to such an omnipresent type. But now our formalized reasoning and mechanical verification confidently illustrate that this is in fact the easiest extension to verify.

Given a decidable predicate class(C)

$$\frac{C <:: C_1 \qquad C <:: C_2 \qquad \text{class}(C_1) \qquad \text{class}(C_2)}{\text{class}(C) \quad \wedge \quad C_1 = C_2} \qquad \frac{\text{cases}(C)\ \textbf{defined}}{\text{class}(C)}$$

Definition of Disjointness

$$\frac{C \leq:: \texttt{Object} \qquad C' \leq:: \texttt{Null}}{C \text{ dsj } C' \quad \wedge \quad C' \text{ dsj } C} \qquad \frac{\text{class}(C) \qquad \text{class}(C') \qquad \neg\, C \leq:: C' \qquad \neg\, C' \leq:: C}{C \text{ dsj } C'}$$

Integrated Subtyping

$$\bigsqcap_{\text{Ceylon}} = \bigsqcap_{\text{ObjNull}} ; \bigsqcap_{\text{cases}} ; \bigsqcap_{\text{dsj}} ; \bigsqcap_{\text{PrinInst}} \qquad \text{where } \left(\bigsqcap_1 ; \bigsqcap_2\right) \vec{\ell} = \text{DNF}_{\bigsqcap_2}\left(\bigsqcap_1 \vec{\ell}\right)$$

Fig. 14. Simplified-Ceylon Subtyping System (using the extensions in Figures 10 through 13)

6.6 Composing Features

Lastly, we compose the extensions together to develop the subtyping system and decision procedure for our simplified Ceylon in Figure 14. This composition assumes we have made an explicit separation of classes and interfaces. With this separation in place, we enforce single inheritance of classes, which we then employ to provide a specific disjointness relation for the disjointness extension. Furthermore, because we also assume only classes can have enumerated cases, we can prove that different cases are disjoint from each other. This is used to prove that principal instantiation does not reintroduce classes with enumerated cases into non-disjoint intersections. Consequently, the intersector for each extension can be shown to preserve the fact that intersections are intersected according to all the previously applied extensions, thereby satisfying Requirement 13 (Intersected Preservation). We chose to apply the extensions in the given order so as to ensure this property. Thus we have compositionally built a decidable subtyping system for Ceylon, with most of our proof effort focused specifically on the extensions rather than the previously established features of generics with variance due to our Decidability of Extended Subtyping Theorem.

7 VARIATIONS, GENERALIZATIONS, RELATED WORK, AND FUTURE WORK

The technique we have presented is actually a specialization to union and intersection types [Coppo and Dezani-Ciancaglini 1978; Coppo et al. 1979; Pottinger 1980; van Wijngaarden et al. 1975] of a more general framework we developed that is independent of union and intersection types. We chose to present this particular specialization because it abstracts away much of the complex underlying proof theory, making the technique more accessible while still being applicable to the original language the framework was invented for. Nonetheless it is important to convey some sense of the more general framework, especially as it pertains to existing research and languages.

7.1 Miminal Relevant Logic and Relaxing Requirements

Van Bakel, Dezani-Ciancaglini, de'Liguoro, and Motohama [2000] discovered that the propositions-as-types interpretation of **B+** [Routley and Meyer 1972], a logic known as minimal relevant logic, corresponds to a call-by-value λ-calculus with subtyping, union, and intersection types. In particular, it corresponds to our toy functional language extended with $\mathcal{E}_{\text{Out}}$ along with the following extension $\mathcal{E}_{\text{In}}$ that was also postulated by Pierce [1991] for Forsythe:

$$\overline{(\tau_1 \to \tau') \cap (\tau_2 \to \tau') <: (\tau_1 \cup \tau_2) \to \tau'}$$

Minimal relevant logic is known to be decidable [Gochet et al. 2005; Viganò 2000], which provides a subtyping algorithm for our toy functional language with both $\mathcal{E}_{\mathsf{Out}}$ and $\mathcal{E}_{\mathsf{In}}$ extensions. A slight variant of our technique can also implement this extension using the following intersector:

$$\bigsqcap_{\mathsf{InOut}} \vec{\ell} = \bigcap_{\varnothing \subset \mathcal{L} \subseteq \{L \mid \varnothing \subset L \subseteq \vec{\ell}\}} \left(\bigcap\nolimits_{L \in \mathcal{L}} \bigcup\nolimits_{(\tau_i \to \tau_o) \in L} \tau_i\right) \to \left(\bigcup\nolimits_{L \in \delta(\mathcal{L})} \bigcap\nolimits_{(\tau_i \to \tau_o) \in L} \tau_o\right)$$

where $\delta(\mathcal{L}) = \{\{\sigma(L) \mid L \in \mathcal{L}\} \mid \sigma \text{ is a function assigning to each } L \in \mathcal{L} \text{ an element } \ell \in L\}$

The necessary variations are two parts. First, minimal relevant logic has no $\bot$ type, so we have to remove that from the type system entirely. Second, because of the order in which the lemmas in Section 4.5.3 are proven, Requirement 12 (Literal Promotion) can be relaxed a bit. In particular, the required proofs with assumptions demonstrating literal promotion can actually be proofs with assumptions *and monotonicity*, meaning they can use the following rule in addition to all the rules in Figure 4 (replacing $\mathcal{P} \vdash \tau^{\leftarrow} \leq \tau^{\rightarrow}$ with $\mathcal{P} \vdash_{\leq} \tau^{\leftarrow} \leq \tau^{\rightarrow}$):

$$\frac{\tau_1 \leq \tau_2 \qquad \mathcal{P} \vdash_{\leq} \tau_2 \leq \tau_3 \qquad \tau_3 \leq \tau_4}{\mathcal{P} \vdash_{\leq} \tau_1 \leq \tau_4}$$

With these two variations, one can easily (in a relative sense) show that $\bigsqcap_{\mathsf{InOut}}$ satisfies the requirements of our framework relative to the $\mathcal{E}_{\mathsf{Out}}$ and $\mathcal{E}_{\mathsf{In}}$ extensions, and as such integrated subtyping $\leq_{\bigsqcap_{\mathsf{InOut}}}$ provides a decision procedure for extended subtyping $<:_{\mathcal{E}_{\mathsf{Out}} \cup \mathcal{E}_{\mathsf{In}}}$.

Furthermore, essentially the same proof of the requirements applies to the variant of $\bigsqcap_{\mathsf{InOut}}$ in which $\mathcal{L}$ is allowed to be empty. This variant implements the additional extension $\top <: \top \to \top$, which was postulated by Barbanera, Dezani-Ciancaglini, and de'Liguoro [1995], and which Liquori and Stolze [2017] recently proved decidable. Thus in all of the above variations on extensions, subtyping has already been proven decidable. However, each proof used a new algorithm and a customized proof technique. Our technique can handle them all in a unified manner, and at the same time the proof is easy to adapt to even further extensions.

7.2 Integrators: Beyond Union and Intersection Types

On that point, our generalized framework extends even beyond union and intersection types. In place of $\mathrm{DNF}_{\sqcap}$ and $\mathrm{dnf}_{\sqcap}$, one can use an arbitrary type integrator $\int$ and integrated predicate. In place of union and intersection subtyping rules, one can abstractly formulate all the subtyping rules much like we did for literal rules. This generalized framework soundly and completely extends subtyping algorithms with the ability to recognize additional axioms. The technique we presented is a specialization of that framework to distributive union and intersection types. Interestingly, by specializing we were able to relax some of the requirements imposed by our general framework, and we are working on integrating these insights back into the generalized framework.

7.3 Predicative Higher-Rank Polymorphism and Duality

Our framework can even be dualized, with the *right*-hand type being *co*integrated rather than the left-hand type being integrated, and this dual variant also has prior applications. In predicative higher-rank polymorphism, Jones, Vytiniotis, Weirich, and Shields [2007] transformed right-hand types into weak-prenex form in order to address a problem in earlier work by Odersky and Läufer [1996] that did not recognize the following distributivity law described by Mitchell [1988]: $\forall\alpha.\, \tau_1 \to \tau_2 <: \tau_1 \to \forall\alpha.\, \tau_2$ (when α is not free in τ_1). Reynolds' own algorithm for Forsythe (without unions) transforms the types *on both sides* into intersections of "simple types" [Reynolds 1988, 1997]. But the type transformer satisfies the requirements of a *co*integrator, and so our framework indicates the algorithm could be optimized to only transform the right-hand type.

$$\frac{}{\vdash_{\mathcal{E}} \cdot} \qquad \frac{\Theta \vdash_{<:\mathcal{E}} \tau_\ell \qquad \Theta \vdash_{<:\mathcal{E}} \tau_u \qquad \vdash_{\mathcal{E}} \Theta \qquad \Theta \vdash \tau_\ell <:_{\mathcal{E}} \tau_u}{\vdash_{\mathcal{E}} \Theta, \tau_l < \alpha < \tau_u}$$

$$\left(\begin{array}{c}\text{All rules in Figure 1,}\\ \text{replacing } \tau^{\leftarrow} <: \tau^{\rightarrow} \text{ with } \Theta \vdash \tau^{\leftarrow} <: \tau^{\rightarrow}\end{array}\right)$$

$$\frac{\tau_\ell < \alpha < \tau_u \in \Theta}{\Theta \vdash \tau_\ell <:_{\mathcal{E}} \alpha} \qquad \frac{\tau_\ell < \alpha < \tau_u \in \Theta}{\Theta \vdash \alpha <:_{\mathcal{E}} \tau_u}$$

Fig. 15. Extended Subtyping with Bounded Type Variables

7.4 Bounded Type Variables and Well-Formed Kind Contexts

Another aspect of subtyping we have not yet discussed is contexts. For example, languages with subtyping typically also have a kind context of bounded type variables. Ceylon itself is an example of such a language. We have proven that our technique extends to languages with type variables that can be both lower- and upper-bounded, assuming the variable-constraint hierarchy is well-founded in accordance with Material-Shape Separation [Greenman et al. 2014].

Although this has been only proven by hand, we present the extension of our framework to bounded type variables here both for completeness with respect to our primary application, Ceylon, and to illustrate an interesting subtlety in this extension. The grammar of types τ is extended with type variables α. Kind contexts Θ are simply lists of type-variable constraints $\tau < \alpha < \tau$. However, in order to ensure decidability, we must restrict these type-variable constraints in accordance with Material-Shape Separation [Greenman et al. 2014]. That is, the upper- and lower-bounds of a type variable α must not themselves refer to α. In order to formalize this, we use a type-validity judgement $\Theta \vdash_S \tau$ that is defined in the obvious manner from a literal-validity judgement $\Theta \vdash_S \ell$ specified by the particular type system at hand to indicate when all the free type variables in a literal ℓ are declared in Θ (where S is a parameter representing the subtyping relation).

Extended subtyping with bounded type variables is formalized in Figure 15. On the left, we define kind-context validity. There are two key aspects to note here. First, when adding a new constrained type variable α to a kind context Θ, one checks that the bounds are valid under Θ *without* α. This captures Material-Shape Separation as applied to type variables (although we do not assume the literals here represent classes). Second, one checks that the lower bound is *already* a subtype of the upper bound under Θ *without* the new constraint. This ensures that the new constraint does not indirectly introduce any subtypings between types that do not reference α. For example, under the kind context $\top < \alpha < \bot$, transitivity would imply that all types are subtypes of each other,

$$\frac{}{\vdash_{\sqcap} \cdot} \qquad \frac{\Theta \vdash_{\leq_\sqcap} \tau_\ell \qquad \Theta \vdash_{\leq_\sqcap} \tau_u \qquad \vdash_{\sqcap} \Theta \qquad \Theta \vdash \tau_\ell \leq_\sqcap \tau_u}{\vdash_{\sqcap} \Theta, \tau_l < \alpha < \tau_u}$$

$$\frac{\Theta \vdash \mathrm{DNF}^{\Theta}_{\sqcap}(\tau) \leq^{\sqcap} \tau'}{\Theta \vdash \tau \leq_\sqcap \tau'}$$

$$\left(\begin{array}{c}\text{All rules in Figure 2 except literal subtyping,}\\ \text{replacing } \tau^{\leftarrow} \leq \tau^{\rightarrow} \text{ with } \Theta \vdash \tau^{\leftarrow} \leq^{\sqcap} \tau^{\rightarrow}\end{array}\right)$$

$$\frac{\tau_\ell < \alpha < \tau_u \in \Theta \qquad \Theta \vdash \tau \leq^{\sqcap} \tau_\ell}{\Theta \vdash \tau \leq^{\sqcap} \alpha} \qquad \frac{}{\Theta \vdash \alpha \leq^{\sqcap} \alpha}$$

$$\frac{r \in \mathcal{R} \qquad \forall \langle \tau, \tau' \rangle \in \mathcal{P}_r.\ \ \Theta \vdash \mathrm{DNF}^{\Theta}_{\sqcap}(\tau) \leq^{\sqcap} \tau'}{\Theta \vdash \ell_r^{\leftarrow} \leq^{\sqcap} \ell_r^{\rightarrow}}$$

Fig. 16. Integrated Subtyping with Bounded Type Variables

$$\mathrm{DNF}^{\Theta}_{\sqcap}(\tau) = \mathrm{DNF}^{\Theta}_{\lambda\vec{\ell},\vec{\alpha}.\,\mathrm{DNF}_{\lambda\vec{\ell}'.(\bigcap\vec{\ell}')\cap(\bigcap\vec{\alpha})}(\bigsqcap\vec{\ell})}(\tau)$$

τ	$\mathrm{DNF}^{\Theta}_{c}(\tau)$ where $c : \vec{\ell} \times \vec{\alpha} \to \tau$
$\bot$	$\bot$
$\tau_1 \cup \tau_2$	$\mathrm{DNF}^{\Theta}_{c}(\tau_1) \cup \mathrm{DNF}^{\Theta}_{c}(\tau_2)$
$\top$	$c([],[])$
$\tau_1 \cap \tau_2$	$\mathrm{DNF}^{\Theta}_{\lambda\vec{\ell}_1,\vec{\alpha}_1.\,\mathrm{DNF}^{\Theta}_{\lambda\vec{\ell}_2,\vec{\alpha}_2.c(\vec{\ell}_1+\vec{\ell}_2,\vec{\alpha}_1+\vec{\alpha}_2)}(\tau_2)}(\tau_1)$
ℓ	$c([\ell],[])$
α	$\mathrm{DNF}^{\Theta}_{\lambda\vec{\ell},\vec{\alpha}.c(\vec{\ell},[\alpha]+\vec{\alpha})}(\tau_u)$ where $\tau_\ell < \alpha < \tau_u \in \Theta$

Fig. 17. Definition of $\mathrm{DNF}^{\Theta}_{\sqcap}$

even if none of the types involved mention α. On the right, we define extended subtyping with the standard declarative rules for bounded type variables.

Integrated subtyping with bounded type variables is formalized in Figure 16. It adds two of the standard reductive subtyping rules for bounded type variables. However, surprisingly, it is missing the rule for upper bounds of type variables. This is because we discovered that upper bounds were actually best achieved by using the integrator rather than a rule. The reason is that the integrator already needs to recurse through the upper bounds of type variables in order to fulfill its responsibilities. For example, while the type $\alpha \cap \texttt{Null}\langle\rangle$ might appear to be already integrated, if α has $\texttt{Object}\langle\rangle$ as its upper bound then the integrator needs to replace the entire type with $\bot$ in order to properly implement disjointness. Thus the definition of $\mathrm{DNF}^{\Theta}_{c}$, shown in Figure 17, recurses into the upper bounds of variables, collecting the list of literals *and* type variables that need to be intersected together. Finally, $\mathrm{DNF}^{\Theta}_{\sqcap}$ takes the collected literals, passes them to the intersector, and then adds the collected type variables to each intersection in the resulting sum of products. The consequence of this is that the upper bound of a type variable is already incorporated into the result of the integrator, making the upper-bound rule for type variables unnecessary.

With a few requirements formulating standard expectations about the literal-validity judgement $\Theta \vdash_S \ell$, one can prove that integrated subtyping with bounded type variables is both decidable and equivalent to extended subtyping with bounded type variables. However, we have yet to mechanically prove these facts. As for the more general framework, we have an abstract formulation of contexts for which kind contexts of bounded type variables are simply a special case.

7.5 Julia and Changing Kind Contexts

An even greater challenge we have yet to approach, though, is contexts that change as the algorithm recurses. For example, Julia [Bezanson et al. 2017] has a `UnionAll` type constructor that is essentially upper-and-lower-bounded existential quantification [Zappa Nardelli et al. 2018]. As the algorithm recurses, it unpacks these existential types when they are on the left-hand side, adding their type variables and respective bounds to the kind context. To make matters even more difficult, in order to determine how to pack existential types when they are on the right-hand side, the algorithm returns inferred constraints on the to-be-instantiated type variables. If extended to handle these issues, our framework could likely solve an important open problem for Julia. In particular, Julia subtyping attempts to support essentially the following axioms regarding its covariant tuple types:

$$\frac{}{(\tau_1 \cup \tau_1') \times \tau_2 <: (\tau_1 \times \tau_2) \cup (\tau_1' \times \tau_2)} \qquad \frac{}{(\exists \tau_\ell \leqslant \alpha \leqslant \tau_u.\,\tau_1) \times \tau_2 <: \exists \tau_\ell \leqslant \alpha \leqslant \tau_u.\,(\tau_1 \times \tau_2)}$$

However, largely due to these axioms, it is not yet known whether Julia subtyping is decidable or even transitive. Both those properties are especially important for Julia because Julia uses subtyping checks to resolve multimethod invocations, so decidability would ensure these run-time decisions would behave as expected, and transitivity would enable Julia to optimize its resolution strategies. These axioms can be implemented by integrating the left-hand type to pull unions and existentials out of tuples, so extending our framework to this setting of changing input and output contexts would enable us to prove for certain that Julia subtyping is both decidable and transitive.

7.6 Regular-Coinductive Subtyping

Another way changing input contexts are used in subtyping is to track which subtyping goals are currently in progress. This is useful for *regular* subtyping systems [Bonchi and Pous 2013; Brandt and Henglein 1998; Gesbert et al. 2015; Hosoya et al. 2000; Kozen et al. 1995], which are particularly common in domain-specific languages [Acay and Pfenning 2017; Ancona and Corradi 2016; Anderson et al. 2014; Henglein and Nielsen 2011; Jeannin et al. 2017] and are also used in C# [Hejlsberg et al. 2005; Kennedy and Pierce 2007; Viroli 2000]. Whereas well-founded subtyping rules ensure recursive proof search terminates, regular subtyping rules ensure the recursion eventually repeats itself. Thus tracking in-progress goals enables the recursive search to identify infinite loops, which are the only form of infinite recursion in regular systems, and terminate accordingly. Since our proofs for the most part simply rely on the recursion principle of proof search, it seems likely we could extend our framework to this setting. Ancona and Corradi [2016] recently achieved decidability for their subtyping system by transforming the types on both sides, and extending our framework might ensure that they could achieve decidability more efficiently by transforming only the left-hand type.

7.7 Semantic Subtyping

The work by Ancona and Corradi [2016] mentioned above has another important quality: it is an example of *semantic* subtyping [Frisch et al. 2002; Hosoya and Pierce 2003]. In semantic subtyping one constructs a set-theoretic model and an interpretation of types as sets in that model; two types are then considered to be subtypes whenever their corresponding sets in the model are subsets. The challenge, then, is to show that this subtype relation derived from the model is decidable for the type system at hand, and there has been impressive work achieving this for various type systems [Ancona and Corradi 2016; Castagna and Xu 2011; Dardha et al. 2013; Frisch et al. 2002, 2008; Hosoya and Pierce 2003]. In particular, the work on semantic subtyping provides powerful reasoning for union and intersection types due to their obvious correspondence to unions and intersections of sets, and is able to recognize subtypings such as the above example regarding Julia's union and covariant tuple types. In fact, the work on semantic subtyping can even reason about *negative* types [Aiken and Wimmers 1993; Frisch et al. 2002], where the type $\neg\tau$ represents the set of values *not* represented by τ. However, we found that semantic subtyping was ill-suited to our setting for both methodological and technological reasons.

On the methodological side, semantic subtyping is fundamentally tied to a model. But in a setting such as Ceylon, that model is constantly changing. It changes because the exact design of the Ceylon language changes (especially in the early stages). It changes because the exact design of various Ceylon libraries change, with fields and methods of classes and interfaces regularly being added, removed, or modified as projects evolve. So with semantic subtyping, the set of programs that are accepted would change with each of these changes. This is to be expected—the problem, though, is in *how* this set of programs changes. For example, adding new functionality to the language can cause once-valid programs to become invalid because the new functionality happens to change the model in such a way that types whose interpretations in the model used to coincide no longer do.

Similarly, adding a new method to a class can make that class no longer equivalent to another class and consequently programs whose validity unwittingly relied on that equivalence unexpectedly become invalid. And unfortunately one cannot simply limit the reasoning provided by semantic subtyping because the proofs that the derived subtype system is well-behaved (e.g. transitive) are themselves derived from the set-theoretic nature of the model and do not easily adapt to such limitations. Thus we found we needed a declarative system so that we could explicitly state which rules we support now *and commit to supporting in the future.*

On the technological side, we encountered issues with nominality, generics, and decidability. Ceylon is a nominal object-oriented language. It uses nominality as a tool for modularity. For example, one can add a method to a class and, because doing has no impact on the name of this class, know that this will not cause previously-valid programs to become invalid. Unfortunately the work on semantic subtyping has often overlooked or oversimplified the nominal aspects of subtyping. For example, although Dardha, Gorla, and Varacca [2013] provide a nominal semantic model for subtyping, their model assumes a closed world and so will treat interfaces without common implementing classes as disjoint even when there is no (declared) guarantee disallowing developers from adding such common implementing classes in the future. Furthermore their encoding does not extend to reified generics with variance. Putting nominality aside, a more fundamental issue is that generics enable developers to write classes whose corresponding structural types are *not* regular. Regularity has been a fundamental assumption of the semantic-subtyping literature, and generics violate that assumption. In fact, this violation likely ensures that semantic subtyping is undecidable for the resulting model. Thus languages with generics *must* limit which subtypings they recognize. The declarative methodology enables designers to pick which subtypings they believe are most important, and our framework provides a tool for quickly assessing which extensions can be implemented reliably.

8 CONCLUSION

We presented a general formulation of languages with union and intersection types that can be adapted to many standard programming languages. We showed a way to soundly extend the subtyping systems and algorithms of these languages in a systematic way that preserves key properties like transitivity and decidability. We demonstrated the application of this approach to Ceylon, and concretely illustrated how to model a range of type-system extensions, which we could scale to due to a composability theorem. Integrated subtyping is a relatively simple yet quite powerful method for extending algorithms with new functionality while preserving the underlying principles. The extensibility of this approach will hopefully empower union and intersection types to become integral parts of many major programming languages.

ACKNOWLEDGMENTS

We are extremely grateful to the Ceylon language-design team for their ongoing collaboration that led to this work. We thank Benjamin Pierce for offering historical perspective on union and intersection types and Forsythe, Andrew Kent for providing details about union and intersections of contracts in Typed Racket, and Julia Belyakova, Benjamin Chung, Artem Pelenitsyn, Jan Vitek, and Francesco Zappa Nardelli for their insights into and discussion regarding the Julia language. Lastly, we greatly appreciate the substantial time and effort that the reviewers, the artifact-evaluation committee, and the Cornell Programming Languages Discussion Group volunteered to greatly improve the quality of this work.

This material is based upon work supported by the NSF under grant CCF-1350182. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

Coşku Acay and Frank Pfenning. 2017. Intersections and Unions of Session Types. *Electronic Proceedings in Theoretical Computer Science* 242, ITRS (2017), 4–19.

Alexander Aiken and Edward L. Wimmers. 1993. Type Inclusion Constraints and Type Inference. In *FPCA*. ACM, New York, NY, USA, 31–41.

Davide Ancona and Andrea Corradi. 2016. Semantic Subtyping for Imperative Object-oriented Languages. In *OOPSLA*. ACM, New York, NY, USA, 568–587.

Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *POPL*. ACM, New York, NY, USA, 113–126.

Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. 1995. Intersection and Union Types: Syntax and Semantics. *Information and Computation* 119, 2 (1995), 202–230.

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.

Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP*. Springer Berlin Heidelberg, Berlin, Heidelberg, DE, 257–281.

Filippo Bonchi and Damien Pous. 2013. Checking NFA Equivalence with Bisimulations up to Congruence. In *POPL*. ACM, New York, NY, USA, 457–468.

Michael Brandt and Fritz Henglein. 1998. Coinductive Axiomatization of Recursive Type Equality and Subtyping. *Fundamenta Informaticae* 33, 4 (1998), 309–338.

Giuseppe Castagna and Zhiwu Xu. 2011. Set-Theoretic Foundation of Parametric Polymorphism and Subtyping. In *ICFP*. ACM, New York, NY, USA, 94–106.

Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny Capabilities for Safe, Fast Actors. In *AGERE!* ACM, New York, NY, USA, 1–12.

Mario Coppo and Mariangiola Dezani-Ciancaglini. 1978. A New Type Assignment for λ-Terms. *Archiv für Mathematische Logik und Grundlagenforschung* 19, 1 (1978), 139–156.

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Patrick Sallé. 1979. Functional Characterization of Some Semantic Equalities Inside λ-Calculus. *Automata, Languages and Programming* 71 (1979), 133–146.

Pierre-Louis Curien and Giorgio Ghelli. 1992. Coherence of Subsumption, Minimum Typing and Type-Checking in F$\leq$. *Mathematical Structures in Computer Science* 2, 1 (1992), 55—-91.

Luís Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *POPL*. ACM, New York, NY, USA, 207–212.

Ornela Dardha, Daniele Gorla, and Daniele Varacca. 2013. Semantic Subtyping for Objects and Classes. In *MOODS/FORTE*. Springer Berlin Heidelberg, Berlin, Heidelberg, DEU, 66–82.

Facebook. 2014. The Flow Static Type Checker Documentation.

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2002. Semantic Subtyping. In *LICS*. IEEE, Washington, DC, USA, 137–146.

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic Subtyping: Dealing Set-Theoretically with Function, Union, Intersection, and Negation Types. *Journal of the ACM* 55, 4, Article 19 (2008), 64 pages.

Nils Gesbert, Pierre Genevès, and Nabil Layaïda. 2015. A Logical Approach to Deciding Semantic Subtyping. *TOPLAS* 38, 1, Article 3 (2015), 31 pages.

Paul Gochet, Pascal Gribomont, and Didier Rossetto. 2005. Algorithms for Relevant Logic. In *Logic, Thought and Action*. Springer Netherlands, Dordrecht, NLD, 479–496.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. 2005. *The Java™ Language Specification, 3rd Edition*. Addison-Wesley Professional, Boston, MA, USA.

Ben Greenman, Fabian Muehlboeck, and Ross Tate. 2014. Getting F-Bounded Polymorphism into Shape. In *PLDI*. ACM, New York, NY, USA, 89–99.

Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. 2005. *C# Language Specification 2.0*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Fritz Henglein and Lasse Nielsen. 2011. Regular Expression Containment: Coinductive Axiomatization and Computational Interpretation. In *POPL*. ACM, New York, NY, USA, 385–398.

Haruo Hosoya and Benjamin C. Pierce. 2003. XDuce: A Statically Typed XML Processing Language. *ACM Transactions on Internet Technology* 3, 2 (2003), 117–148.

Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. 2000. Regular Expression Types for XML. In *ICFP*. ACM, New York, NY, USA, 11–22.

Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. 2017. CoCaml: Functional Programming with Regular Coinductive Types. *Fundamenta Informaticae* 150, 3/4 (2017), 347–377.

Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical Type Inference for Arbitrary-Rank Types. *Journal of Functional Programming* 17, 1 (2007), 1–82.

Andrew J. Kennedy and Benjamin C. Pierce. 2007. On Decidability of Nominal Subtyping with Variance. In *FOOL*. ACM, New York, NY, USA, Article 5, 12 pages.

Gavin King. 2013. The Ceylon Language Specification, Version 1.0.

Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. 1995. Efficient Recursive Subtyping. *Mathematical Structures in Computer Science* 5, 1 (1995), 113–125.

Luigi Liquori and Claude Stolze. 2017. A Decidable Subtyping Logic for Intersection and Union Types. In *Topics in Theoretical Computer Science*. Springer International Publishing, Cham, CHE, 74–90.

Microsoft. 2012. TypeScript.

Microsoft. 2018. The Typescript Handbook.

John C. Mitchell. 1988. Polymorphic Type Inference and Containment. *Information and Computation* 76, 2/3 (1988), 211–249.

Fabian Muehlboeck and Ross Tate. 2018. Artifact for Article "Empowering Union and Intersection Types with Integrated Subtyping". In *OOPSLA*. ACM, New York, NY, USA.

Martin Odersky and Konstantin Läufer. 1996. Putting Type Annotations to Work. In *POPL*. ACM, New York, NY, USA, 54–67.

Jens Palsberg and Christina Pavlopoulou. 1998. From Polyvariant Flow Information to Intersection and Union Types. In *POPL*. ACM, New York, NY, USA, 197–208.

Benjamin C. Pierce. 1989. *A Decision Procedure for the Subtype Relation on Intersection Types with Bounded Variables*. Technical Report CMU-CS-89-169. Carnegie Mellon University.

Benjamin C. Pierce. 1991. *Programming with Intersection Types, Union Types, and Polymorphism*. Technical Report CMU-CS-91-106. Carnegie Mellon University.

Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Boston, MA, USA.

Garrell Pottinger. 1980. A Type Assignment for the Strongly Normalizable λ-Terms. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, London, GBR, New York, NY, USA.

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*. Birkhäuser Boston, Boston, MA, USA, 173–233.

Tiark Rompf and Nada Amin. 2016. Type Soundness for Dependent Object Types (DOT). In *OOPSLA*. ACM, New York, NY, USA, 624–641.

Richard Routley and Robert K. Meyer. 1972. The Semantics of Entailment: III. *Journal of Philosophical Logic* 1, 2 (1972), 192–208.

Ross Tate, Alan Leung, and Sorin Lerner. 2011. Taming Wildcards in Java's Type System. In *PLDI*. ACM, New York, NY, USA, 614–627.

The Coq Development Team. 1984. The Coq Proof Assistant.

The Dotty Development Team. 2015. Dotty.

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *POPL*. ACM, New York, NY, USA, 395–406.

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. Australian National University.

Adriaan van Wijngaarden, Barry J. Mailloux, John E. L. Peck, Cornelis H. A. Koster, Michel Sintzoff, Charles H. Lindsey, Lambert G. L. T. Meertens, and Richard G. Fisker (Eds.). 1975. Revised Report on the Algorithmic Language ALGOL 68. *Acta Informatica* 5 (1975), 1–236.

Luca Viganò. 2000. An O(n log n)-Space Decision Procedure for the Relevance Logic B+. *Studia Logica* 66, 3 (2000), 385–407.

Mirko Viroli. 2000. *On the Recursive Generation of Parametric Types*. Technical Report DEIS-LIA-00-002. University of Bologna.

Joe B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. 2002. A Calculus with Polymorphic and Polyvariant Flow Types. *Journal of Functional Programming* 12, 3 (2002), 183–227.

Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. 2018. Julia Subtyping: A Rational Reconstruction. In *OOPSLA*. ACM, New York, NY, USA, Article 113, 28 pages.