Gradual Typing: A New Perspective

GIUSEPPE CASTAGNA, CNRS - Université Paris Diderot, France
VICTOR LANVIN, Université Paris Diderot, France
TOMMASO PETRUCCIANI, Università degli Studi di Genova, Italy and Université Paris Diderot, France
JEREMY G. SIEK, University of Indiana, USA

We define a new, more semantic interpretation of gradual types and use it to "gradualize" two forms of polymorphism: subtyping polymorphism and implicit parametric polymorphism. In particular, we use the new interpretation to define three gradual type systems —Hindley-Milner, with subtyping, and with union and intersection types— in terms of two preorders, subtyping and materialization. These systems are defined both declaratively and algorithmically. The declarative presentation consists in adding two subsumption-like rules, one for each preorder, to the standard rules of each type system. This yields more intelligible and streamlined definitions and shows a direct correlation between cast insertion and materialization. For the algorithmic presentation, we show how it can be defined by reusing existing techniques such as unification and tallying.

CCS Concepts: • **Software and its engineering → Polymorphism**;

Additional Key Words and Phrases: Subtyping, Gradual Typing, Union Types, Intersection Types, Semantic Subtyping, Let-Polymorphism, Hindley-Milner.

ACM Reference Format:
Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. 2019. Gradual Typing: A New Perspective. *Proc. ACM Program. Lang.* 3, POPL, Article 16 (January 2019), 32 pages. https://doi.org/10.1145/3290329

1 INTRODUCTION

The goal of this work was to endow gradual typing, as defined by Siek and Taha [2006], with the semantic interpretation of types and subtyping introduced by Frisch et al. [2008]. To that end we explore a new idea, that is, to interpret gradual types using polymorphic type variables. This leads to revisiting the existing definitions of gradually-typed languages (with implicit parametric and/or subtyping polymorphism) and yields a streamlined and declarative way to define existing systems, as well as a way to extend them with subtyping and set-theoretic types. But let us proceed in order.

Semantic subtyping is a technique by Frisch et al. [2008] to define a type theory for union, intersection, and negation types, in the presence of higher-order functions. It gives a semantic interpretation to types as sets (e.g., sets of values of some language) and then defines the subtyping relation as set-containment of the interpretations, whence the name *set-theoretic types*. The advantage of such a technique is that, by definition, types satisfy natural distribution laws (e.g., $(t \times s_1) \vee (t \times s_2)$ and $t \times (s_1 \vee s_2)$ are equivalent, and so are $(s \to t_1) \wedge (s \to t_2)$ and $s \to (t_1 \wedge t_2)$).

Gradual typing is an approach that combines the safety guarantees of static typing with the flexibility of dynamic typing [Siek and Taha 2006]. The idea is to introduce an *unknown* type,

Authors' addresses: Giuseppe Castagna, CNRS - Université Paris Diderot, France; Victor Lanvin, Université Paris Diderot, France; Tommaso Petrucciani, Università degli Studi di Genova, Italy , Université Paris Diderot, France; Jeremy G. Siek, University of Indiana, USA.

2475-1421/2019/1-ART16
https://doi.org/10.1145/3290329

denoted by "?", used to inform the compiler that additional type checks may be needed at run time. Programmers can add type annotations to a program *gradually* and control precisely how much checking is done statically versus dynamically. The type-checker ensures that the parts of the program that are typed with *static types*—i.e., types that do not contain ?—enjoy the type safety guarantees of static typing (well-typed expressions never get stuck), while the parts annotated with *gradual types*—i.e., types in which the dynamic type ? occurs—enjoy the same property modulo the possibility to fail on some dynamic type check inserted by the type-driven compilation.

Some practical benefits of combining gradual typing with union and intersection types were presented by Castagna and Lanvin [2017] in a monomorphic setting. In this work we extend such benefits to a polymorphic setting. For an aperçu of what can be done in this setting, consider the following ML-like code snippet adapted from Siek and Vachharajani [2008a]:

```
let mymap (condition) (f) (x : ?) =
  if condition then Array.map f x else List.map f x
```

According to the value of the argument `condition`, the function `mymap` applies either the array version or the list version of `map` to the other two arguments. This example cannot be typed using only simple types: the type of `x` and the return type of `mymap` change depending on the value of `condition`. By annotating `x` with the gradual type `?`, the type reconstruction system for gradual types of Siek and Vachharajani [2008a] can type this piece of code with $\mathsf{Bool} \to (\alpha \to \beta) \to ? \to ?$. That is, type reconstruction recognizes that the parameter `condition` must be bound to a Boolean value, and the compilation process adds dynamic checks to ensure that the value bound to `x` will be, according to the case, either an array or a list whose elements are of a type compatible with the actual input type of `f`. This type however is still imprecise. For example, if we pass a value that is neither an array nor a list (e.g., an integer) as the last argument to `mymap`, then this application is well-typed, even though the execution will always fail, independently of the value of `condition`. Likewise, the type gives no useful information about the result of `mymap`, even though it will clearly be either a β-list or a β-array. These problems can be remedied by using set-theoretic types:

```
let mymap (condition) (f) (x : (α array | α list) & ?) =
  if condition then Array.map f x else List.map f x
```

where "|" denotes union and "&" denotes intersection. The union indicates that a value of this type is *either* an array *or* a list, both of α-elements. The intersection indicates that `x` has *both* type $(\alpha\ \mathtt{array} \mid \alpha\ \mathtt{list})$ *and* type `?`. Intuitively, this type annotation means that the function `mymap` accepts for `x` a value of any type (which is indicated by `?`), as long as this value is also either an array *or* a list of α elements, with α being the domain of the `f` argument. The use of the intersection of a union type with "?" to type a parameter corresponds to a programming style in which the programmer asks the system to *statically* enforce that the function will be applied only to arguments in the union type and delegates to the system any *dynamic* check regarding the use of the parameter in the body of the function. The system presented in Section 4 deduces for this definition the type:

$$\mathsf{Bool} \to (\alpha \to \beta) \to (\ (\alpha\ \mathtt{array} \mid \alpha\ \mathtt{list})\ \&\ ?\ ) \to (\beta\ \mathtt{array} \mid \beta\ \mathtt{list})$$

This type forces the last argument of `mymap` to be either an array or a list of elements whose type is the input type of the argument bound to `f`. Note that the return type of `mymap` is no longer gradual (as it was with the previous definition), since the union type allows us to define it without any loss of precision, as well as to capture the correlation with the return type of the argument bound to `f`. The derivation of this type is used by the compiler to insert dynamic type-checks that ensure type soundness. In particular, the compilation process described in Section 2.2.4 inserts in the body of `mymap` the casts that dynamically check that the first occurrence of `x` is bound to an array of

elements of the appropriate type, and that the second occurrence of x is bound to a list of such elements, producing a code like the following:

```
let mymap (condition) (f) (x : (α array | α list) & ?) =
  if condition then Array.map f (x<α array>) else List.map f (x<α list>)
```

where e<t> is a type-cast expression that dynamically checks whether the result of e has type t.

This kind of type discipline is out of reach of current systems. To obtain it we explore a new idea to interpret gradual types, namely, that the unknown type ? acts like a type variable, but a peculiar one since each occurrence of ? in a typing constraint can be considered as a placeholder for a possibly distinct type variable. This idea is the essence of our approach to gradual typing and we formalize it by defining an operation of *discrimination* which replaces each occurrence of ? in a gradual type by a type variable.

Discrimination is the cornerstone of our semantics for gradual types: by applying discrimination we map a polymorphic gradual type into a set of polymorphic static types (one for each possible replacement of occurrences of the dynamic type by a type variable); then we use the semantic subtyping interpretation of static types, to interpret, indirectly, our initial gradual type. We use this semantic interpretation to revisit some notions from the gradual typing literature: we restate some of them switching from syntactic to semantic definitions, make new connections, and introduce new concepts. In particular, we use discrimination to define two preorders on gradual types: the *subtyping relation* (by which $\tau_1 \leq \tau_2$ implies that an expression of type τ_1 can be safely used wherever one of type τ_2 is expected) and the *materialization relation* ($\tau_1 \preccurlyeq \tau_2$ if and only if τ_2 is more precise than τ_1—i.e., it was obtained from τ_1 by replacing some occurrences of ? by types).[1] These two preorders are at the core of our approach and, we claim, of gradual typing as well, since they can be combined to replace *consistency* and *consistent subtyping*, two notions that current systems rely on. This is particularly important because the materialization relation is a preorder (transitivity is what matters here); therefore it can be used in a subsumption-like rule that we call *materialize*. As we show in Section 2, adding gradual typing to ML then essentially amounts to adding this materialize rule to the standard set of rules for Hindley-Milner systems. Further, adding set-theoretic types is then just a matter of adding the regular subsumption rule for subtyping. The simplicity of this extension contrasts with current literature where gradual typing is obtained by embedding checks for consistency or for consistent subtyping (two non-transitive relations) in elimination rules.

Finally, our approach sheds some light on the logical meaning of gradual typing. It is well-known that there is a strong correspondence between systems with subtyping and systems without subtyping but with explicit coercions: every usage of the subsumption rule in the former corresponds to the insertion of an explicit coercion in the latter. Our definition of materialization yields an analogous correspondence between a gradually-typed language and the cast calculus in which the language is compiled: every usage of the materialize rule in the former corresponds to the insertion of an explicit cast in the latter. As such, the cast language looks like an important ingredient for a Curry-Howard isomorphism for gradual typing disciplines. An intriguing direction for future work is to study the logic associated with these expressions.

Overview and Contributions. We present our system gradually (pun intended) in three steps of increasing complexity.

The first step is to add gradual typing to ML-like languages: we do it in Section 2. We start by giving the definition of materialization and use it to give a declarative static semantics for a

[1] The fourth author prefers to call materialization the precision relation, using the terminology of Garcia [2013] but with ? at the bottom, as in the work of Siek and Vachharajani [2008b].

gradually-typed version of ML (§2.1). As customary for gradually-typed languages, we give the dynamic semantics by compiling well-typed terms into a cast calculus and we prove its soundness (§2.2). We conclude the section by studying the algorithmic aspects of typing, that is, we define a constraint-based type inference algorithm that we prove to be sound and complete (§2.3).

The second step, in Section 3, shows how to extend the system with subtyping. We define a subtyping relation and add a subsumption rule both to the type system of the gradual language and to the one of the cast calculus (§3.1). The presence of subsumption makes type inference more difficult since, in particular, constraint resolution involves computing intersections and unions of types (§3.2). Therefore, we postpone the development of the algorithmic aspects of this part to the following section, in which we add first-class union and intersection types to the systems.

The third and last step, presented in Section 4, is the addition of unions and intersections, which we achieve by applying the approach of semantic subtyping. This involves the addition of union and negation types (intersections are encoded by De Morgan's laws) as well as of recursive types (which are needed to solve type reconstruction with polymorphic types). We use a set-theoretic interpretation of types to define subtyping for static types. Using our interpretation of ? as type variables, we extend this subtyping relation and the previous materialization relation to gradual types (§4.1). We extend the cast calculus with set-theoretic types: this notably requires a non-trivial modification of the semantics to reduce composition of casts involving unions and intersections; we prove the extension to be conservative besides enjoying the usual safety properties (§4.2). Finally, we define a type inference algorithm and prove its soundness (§4.3).

A discussion on related (§5) and future (§6) work and a conclusion (§7) end this presentation.

For space reasons, some auxiliary definitions and results and all proofs are moved into an appendix available online in the ACM digital library under the "Source Materials" tab.

The main contributions of this work are:

(1) A new interpretation of gradual types in which every occurrence of the unknown type "?" is considered a placeholder for some type variable.
(2) The definition of gradual type systems in terms of two preorders, subtyping and materialization. The definition of these preorders is based on the new interpretation of types; it yields semantic-oriented definitions that are syntax independent and, as such, more resilient to language extensions or modifications.
(3) The first declarative definition of gradual type systems obtained by adding two subsumption-like rules, yielding a clearer and more streamlined definition of the type system.
(4) A better correspondence between type derivations and compilation, obtained by showing a one-to-one correspondence between cast insertions and the use of the *materialize* rule.
(5) A direct correlation between the safety of a cast and the polarity of its blame label —a consequence of the correspondence in (4)—, allowing for a simpler statement of *blame safety* for the cast calculus. We show in particular that our system never blames the context.
(6) The reformulation of the type inference problem for gradual type systems in terms of static type systems via the new interpretation. In particular, our two inference algorithms reuse existing algorithms such as unification and tallying.
(7) The extension of gradual typing to polymorphic type systems with set-theoretic types. In particular, the definition of the operational semantics for casts in the presence of unions and intersections is an important and far-from-obvious result.

2 GRADUAL TYPING FOR HINDLEY-MILNER SYSTEMS

In this section, we use our new approach to add gradual typing to a language with ML-style polymorphism. We describe the syntax of types and of the source language, the declarative type system, the cast language and how to compile expressions, and the type inference system.

2.1 Source Language

2.1.1 Types and Expressions. Let α, β, and γ range over a countable set $\mathcal{V}^\alpha$ of *type variables*. Let b range over a set $\mathcal{B}$ of *basic types* (e.g., $\mathcal{B} = \{\mathsf{Int}, \mathsf{Bool}\}$). Let c range over a set C of *constants*. Types and expressions are then defined as follows and explained in the following paragraphs.

$$\begin{aligned} \text{static types} \quad & \mathcal{T}_t \ni t ::= \alpha \mid b \mid t \times t \mid t \to t \\ \text{gradual types} \quad & \mathcal{T}_\tau \ni \tau ::= ? \mid \alpha \mid b \mid \tau \times \tau \mid \tau \to \tau \\ \text{source language expressions} \quad & e ::= x \mid c \mid \lambda x.\, e \mid \lambda x\colon \tau.\, e \mid e\ e \mid (e, e) \mid \pi_i\ e \mid \mathsf{let}\ \vec{\alpha}\ x = e\ \mathsf{in}\ e \end{aligned}$$

Static types $\mathcal{T}_t$ (ranged over by t) are the types of an ML-like language: type variables, basic types, products, and arrows. Gradual types $\mathcal{T}_\tau$ (ranged over by τ) add the unknown type ? to them.

The source language is a fairly standard λ-calculus with constants, pairs (e, e), projections for the elements of a pair $\pi_i\ e$ (where $i \in \{1, 2\}$), plus a let construct. There are two aspects to point out.

One is that there are two forms of λ-abstraction: $\lambda x.\, e$ and $\lambda x\colon \tau.\, e$. In the latter, the annotation τ fixes the type of the argument, whereas in the former the type can be chosen during typing (and will in practice be computed by inference). Furthermore, the type τ in the annotation is gradual, while in $\lambda x.\, e$ we require that the inferred type of the parameter be a static type t (*cf.* Figure 1, rule [Abstr]). This is the same restriction imposed by Garcia and Cimini [2015] to properly reject some ill-typed programs. For example, without this restriction we can type $\lambda x.(x + 1, \neg x)$ since it would be possible to infer the type ? for x so as to deduce for $\lambda x.(x + 1, \neg x)$ the type $? \to \mathsf{Int} \times \mathsf{Bool}$. But $\lambda x.(x + 1, \neg x)$ is not a well-typed term in ML, therefore by the principles of gradual typing (see Theorem 1 of Siek et al. [2015b]) it must be rejected unless its parameter is explicitly annotated by a type in which ? occurs (here, annotated by ? itself).

The second non-standard element of this syntax is that the let binding is decorated with a vector $\vec{\alpha}$ of type variables, as in $\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2$. This *decoration* (we reserve the word *annotation* for types annotating parameters in λ-abstractions) serves as a binder for the type variables that appear in annotations *occurring in* e_1. For instance, $\mathsf{let}\ \alpha\ z = \lambda x\colon \alpha.\, x\ \mathsf{in}\ e$ and $\mathsf{let}\ z = \lambda x.\, x\ \mathsf{in}\ e$ are equivalent, while $\mathsf{let}\ z = \lambda x\colon \alpha.\, x\ \mathsf{in}\ e$ implies that α was introduced in an outer expressions such as $\lambda y\colon \alpha.\, \mathsf{let}\ z = \lambda x\colon \alpha.\, x\ \mathsf{in}\ e$. The normal let from ML can be recovered as the case where $\vec{\alpha}$ is empty (which would always be the case if, as in ML, function parameters never had type annotations).

As customary, we consider expressions modulo α-renaming of bound variables. In $\lambda x.\, e$ and $\lambda x\colon \tau.\, e$, x is bound in e; in $\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2$, x is bound in e_2 and the $\vec{\alpha}$ variables are bound in e_1. It is also customary that we may refer to the source language as the *gradually-typed language*.

2.1.2 Type System. We describe the declarative type system of the source language. We use the standard notion for type schemes and type environments. A type scheme has the form $\forall\vec{\alpha}.\, \tau$, where $\vec{\alpha}$ is a vector of distinct variables. We identify type schemes with an empty $\vec{\alpha}$ with gradual types. A type environment Γ is a finite function from variables to type schemes. The type system is defined by the rules in Figure 1.

The first eight rules are almost those of a standard Hindley-Milner type system. In [Const], we use b_c to denote the basic type for a constant c (e.g., $b_3 = \mathsf{Int}$). One important aspect to note is that the types used to instantiate the type scheme in [Var] and the type used for the domain in [Abstr] must all be static types, as forced by the use of the metavariable t.

The other non-standard aspect is the rule for [Let]. To type $\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2$, we type e_1 with some type τ_1; then, we type e_2 in the expanded environment in which x has type $\forall\vec{\alpha},\vec{\beta}.\, \tau_1$. The first side condition ($\vec{\alpha},\vec{\beta} \sharp \Gamma$) asks that all the variables we generalize do not occur free in Γ; this is standard. The second condition ($\vec{\beta} \sharp e_1$) states that the type variables $\vec{\beta}$ must not occur free in e_1. This means that the type variables that are explicitly introduced by the programmer (by using them in annotations) can only be generalized at the level of a let binding by explicitly specifying them in

$$[\textsc{Var}]\ \frac{}{\Gamma \vdash x\colon \tau\{\vec{\alpha} := \vec{t}\}}\ \Gamma(x) = \forall\vec{\alpha}.\,\tau \qquad [\textsc{Const}]\ \frac{}{\Gamma \vdash c\colon b_c} \qquad [\textsc{Proj}]\ \frac{\Gamma \vdash e\colon \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\, e\colon \tau_i}$$

$$[\textsc{Pair}]\ \frac{\Gamma \vdash e_1\colon \tau_1 \qquad \Gamma \vdash e_2\colon \tau_2}{\Gamma \vdash (e_1, e_2)\colon \tau_1 \times \tau_2} \qquad [\textsc{App}]\ \frac{\Gamma \vdash e_1\colon \tau' \to \tau \qquad \Gamma \vdash e_2\colon \tau'}{\Gamma \vdash e_1\, e_2\colon \tau}$$

$$[\textsc{Abstr}]\ \frac{\Gamma, x\colon t \vdash e\colon \tau}{\Gamma \vdash (\lambda x.\, e)\colon t \to \tau} \qquad [\textsc{AAbstr}]\ \frac{\Gamma, x\colon \tau' \vdash e\colon \tau}{\Gamma \vdash (\lambda x\colon \tau'.\, e)\colon \tau' \to \tau}$$

$$[\textsc{Let}]\ \frac{\Gamma \vdash e_1\colon \tau_1 \qquad \Gamma, x\colon \forall\vec{\alpha},\vec{\beta}.\,\tau_1 \vdash e_2\colon \tau}{\Gamma \vdash (\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2)\colon \tau}\ \vec{\alpha},\vec{\beta} \sharp \Gamma \text{ and } \vec{\beta} \sharp e_1$$

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash e\colon \tau'}{\Gamma \vdash e\colon \tau}\ \tau' \preccurlyeq \tau$$

Fig. 1. Declarative type system of the source language.

the decoration. In contrast, type variables introduced by the type system (i.e., the fresh variables in the t type in the [Abstr] rule) can be generalized at any let (implicitly, that is, by the type system), provided they do not occur in the environment. Note that we recover the standard Hindley-Milner rule for let bindings when expressions do not contain annotations and decorations are empty.

As anticipated, the type system does not need to deal with gradual types explicitly except in one rule. Indeed, the first eight rules do not check anything regarding gradual types (they only impose restrictions that some types must be static). The last rule, [Materialize], is a subsumption-like rule that allows us to make any gradual type more precise by replacing occurrences of ? with arbitrary gradual types. This is accomplished by the *materialization* relation $\preccurlyeq$ defined below.

Materialization. Intuitively, $\tau_1 \preccurlyeq \tau_2$ holds when τ_2 can be obtained from τ_1 by replacing some occurrences of ? with arbitrary gradual types, possibly different for every occurrence. This relation can be easily defined by the following inductive rules, which add the reflexive case for type variables to the rules of Siek and Vachharajani [2008b]:[2]

$$\frac{}{? \preccurlyeq \tau} \qquad \frac{}{\alpha \preccurlyeq \alpha} \qquad \frac{}{b \preccurlyeq b} \qquad \frac{\tau_1 \preccurlyeq \tau_1' \qquad \tau_2 \preccurlyeq \tau_2'}{\tau_1 \times \tau_2 \preccurlyeq \tau_1' \times \tau_2'} \qquad \frac{\tau_1 \preccurlyeq \tau_1' \qquad \tau_2 \preccurlyeq \tau_2'}{\tau_1 \to \tau_2 \preccurlyeq \tau_1' \to \tau_2'}$$

However, this definition is intrinsically tied to the syntax of types. Instead, we want the definition of materialization to remain valid also when we extend the language of types we use. Therefore, we give a definition based on our view, anticipated earlier, of occurrences of ? as type variables.

First, let us define a new sort of types, *type frames*, as follows:

$$\mathcal{T}_T \ni T ::= X \mid \alpha \mid b \mid T \times T \mid T \to T$$

where X ranges over a set $\mathcal{V}^X$ of *frame variables* disjoint from $\mathcal{V}^\alpha$. Type frames are like gradual types except that, instead of ?, they have frame variables. We write $\mathcal{T}_T$ for the set of all type frames.

Given a type frame T, we write $T^\dagger$ for the gradual type obtained by replacing all frame variables in T with ?. The reverse operation, which we call *discrimination*, is defined as follows.

Definition 2.1 (Discrimination of a gradual type). *Given a gradual type τ, the set $\star(\tau)$ of its* discriminations *is defined as:* $\star(\tau) \stackrel{\text{def}}{=} \{\, T \in \mathcal{T}_T \mid T^\dagger = \tau \,\}$.

[2] Henglein [1994] defines an equivalent relation for monomorphic types (called "subtyping") but with different rules.

The definition of materialization, stated formally below, says that τ_2 materializes τ_1 if it can be obtained from τ_1 by first replacing all occurrences of ? with arbitrary variables in $\mathcal{V}^X$, and then applying a substitution which replaces those variables with gradual types.

Definition 2.2 (Materialization). *We define the* materialization *relation on gradual types* $\tau_1 \preccurlyeq \tau_2$ *("*τ_2 *materializes* τ_1*") as follows:* $\tau_1 \preccurlyeq \tau_2 \overset{\text{def}}{\iff} \exists T \in \star(\tau_1), \theta : \mathcal{V}^X \to \mathcal{T}_\tau.\ T\theta = \tau_2$.

In the above, $\theta : \mathcal{V}^X \to \mathcal{T}_\tau$ is a type substitution (i.e., a mapping that is the identity on a cofinite set of variables) from frame variables to gradual types. We use $\mathrm{dom}(\theta)$ to denote the set of variables for which θ is not the identity (i.e., $\mathrm{dom}(\theta) = \{X \mid X\theta \neq X\}$).

It is not difficult to prove that the materialization relation of Definition 2.2 and the one deduced by the inductive rules that we have given in the previous page are equivalent, and that they are inverses of the precision relation [Garcia 2013] and of naive subtyping [Wadler and Findler 2009].

The presence of [Materialize] yields the static gradual guarantee property of Siek et al. [2015b] for free. We lift the materialization relation to terms as usual by relating type annotations via materialization. On the right is the rule for annotated λ-abstractions. The remaining rules are straightforward.

$$\frac{\tau \preccurlyeq \tau' \qquad e \preccurlyeq e'}{\lambda x\colon \tau.\, e \preccurlyeq \lambda x\colon \tau'.\, e'}$$

Proposition 2.3 (Static gradual guarantee). *If* $\varnothing \vdash e : \tau$ *and* $e' \preccurlyeq e$*, then* $\varnothing \vdash e' : \tau$.

We said that our type system is *declarative*. This is because all auxiliary relations (here materialization) are handled by structural rules (here [Materialize]) added to an existing set of logical and identity rules.[3] In a declarative system, every term may have different types and derivations; removing the structural rules corresponds to finding an algorithmic system that for every well-typed term chooses one particular derivation and, thus, one type of the declarative system. This is usually obtained by moving the checks of the auxiliary relations into the elimination rules: this yields a system that is easier to implement but less understandable. And this is exactly what current gradual type systems do. It is possible to show that the set of typable terms of our declarative system is the same as the set of typable terms of the existing gradual type systems that use consistency.

In particular, the relation between our system and the gradual type system of Siek and Taha [2006] can be stated formally. Let $\vdash_{ST}$ denote the typing judgments of Siek and Taha [2006] and let $\vdash_1$ denote the monomorphic restriction of the implicative fragment of our system (i.e., our gradual types without type variables and the typing rules of the simply-typed λ-calculus plus materialization: see Figure 7 in the appendix). Then we have the following result:

Proposition 2.4. *If* $\Gamma \vdash_{ST} e : \tau$ *then* $\Gamma \vdash_1 e : \tau$. *Conversely, if* $\Gamma \vdash_1 e : \tau$*, then there exists a type* τ' *such that* $\Gamma \vdash_{ST} e : \tau'$ *and* $\tau' \preccurlyeq \tau$.

The most enlightening case in the proof of the forward direction is for the rule [GApp2] of Siek and Taha [2006] here on the right. This rule is derivable in our system because $\tau_2 \sim \tau'$ implies that there is some τ_3 such that $\tau_2 \preccurlyeq \tau_3$ and $\tau' \preccurlyeq \tau_3$ [Siek and Vachharajani 2008a], then we have $\Gamma \vdash e_1 : \tau_3 \to \tau$ and $\Gamma \vdash e_2 : \tau_3$ by two uses of [Materialize]. Conversely, materialization can always be pushed to applications.

$$\text{[GApp2]}\quad \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_2 \sim \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

The (polymorphic) implicative fragment of our system (i.e., our system without products), denoted by $\vdash_{\to}$, is yet another well-known gradual type system, because it coincides with the ITGL type system of Garcia and Cimini [2015], denoted by $\vdash_{GC}$, as stated by the following result:

Proposition 2.5. *If* $\Gamma \vdash_{GC} e : \tau$ *then* $\Gamma \vdash_{\to} e : \tau$. *Conversely, if* $\Gamma \vdash_{\to} e : \tau$*, then there exists a type* τ' *such that* $\Gamma \vdash_{GC} e : \tau'$ *and* $\tau' \preccurlyeq \tau$.

In other words, the relationship between our new declarative approach (i.e., with the [Materialize]

[3] In logic, logical rules refer to a particular connective (here, a type constructor, that is, either $\to$, or $\times$, or b), while identity rules (e.g., axioms and cuts) and structural rules (e.g., weakening and contraction) do not.

$$[\textsc{Var}]\ \frac{}{\Gamma \vdash x : \forall\vec{\alpha}.\,\tau}\ \Gamma(x) = \forall\vec{\alpha}.\,\tau \qquad [\textsc{TAbstr}]\ \frac{\Gamma \vdash E : \tau}{\Gamma \vdash \Lambda\vec{\alpha}.\,E : \forall\vec{\alpha}.\,\tau}\ \vec{\alpha} \sharp \Gamma \qquad [\textsc{TApp}]\ \frac{\Gamma \vdash E : \forall\vec{\alpha}.\,\tau}{\Gamma \vdash E\,[\vec{t}] : \tau\{\vec{\alpha} := \vec{t}\}}$$

$$[\textsc{Cast}^{\oplus}]\ \frac{\Gamma \vdash E : \tau'}{\Gamma \vdash E\langle \tau' \overset{\ell}{\Rightarrow} \tau\rangle : \tau}\ \tau' \preccurlyeq \tau \qquad [\textsc{Cast}^{\ominus}]\ \frac{\Gamma \vdash E : \tau'}{\Gamma \vdash E\langle \tau' \overset{\bar{\ell}}{\Rightarrow} \tau\rangle : \tau}\ \tau \preccurlyeq \tau'$$

Fig. 2. Main Typing Rules for the Cast Language

rule) and the standard ones that use consistency (e.g., Siek and Taha [2006] and Garcia and Cimini [2015]) is analogous to the usual relationship between a declarative type system with subtyping (i.e., with a subsumption rule) and an algorithmic type system.

We conclude this section by stressing that our new interpretation of gradual types only concerns the relations on types, but it does not apply directly to the terms of the language. In particular, it does not apply to the occurrences of ? in the type annotations of a program: indeed, it can be that an occurrence of ? in a program cannot be replaced by a static type while maintaining typability.

2.2 Cast Language

As customary with gradual typing, the semantics of the gradually-typed language is given by translating its well-typed expressions into a cast language, which we define next.

2.2.1 Syntax. The syntax of the cast language is defined as follows:

$$E ::= x \mid c \mid \lambda^{\tau\to\tau} x.\,E \mid E\ E \mid (E, E) \mid \pi_i\ E \mid \mathsf{let}\ x = E\ \mathsf{in}\ E \mid \Lambda\vec{\alpha}.\,E \mid E\,[\vec{t}] \mid E\langle \tau \overset{p}{\Rightarrow} \tau\rangle$$

This is an explicitly-typed λ-calculus similar to the source language with a few differences and the addition of explicit casts.

There is now just one kind of λ-abstraction, which is annotated with its arrow type. Let-expressions no longer bind type variables; instead, there are explicit type abstractions $\Lambda\vec{\alpha}.\,E$ and applications $E\,[\vec{t}]$. For example, the source language expression $\mathsf{let}\ \alpha\, z = \lambda x : \alpha.\,\lambda y.\,x\ \mathsf{in}\ z\,42$, of type $\beta \to \mathsf{Int}$, is translated into the cast calculus as $\mathsf{let}\ z = \Lambda\alpha\beta.\,\lambda^{\alpha\to\beta\to\alpha} x.\,\lambda^{\beta\to\alpha} y.\,x\ \mathsf{in}\ z\,[\mathsf{Int}, \beta]\,42$. Despite the presence of type abstractions, the cast calculus does not support first-class polymorphism; the syntax of types remains unchanged from Section 2.1.1 and does not include universally quantified types. Finally, the important additions to the calculus are explicit casts of the form $E\langle \tau \overset{p}{\Rightarrow} \tau'\rangle$ where, as usual, p ranges over a set of blame labels. Such an expression dynamically checks whether E, of static type τ, produces a value of type τ'; if the cast fails, then the label p is used to blame the cast. These casts are inserted during compilation to perform runtime checks in dynamically-typed code: for instance, the function $\lambda x : ?.\,x + 1$ will be compiled into $\lambda^{?\to\mathsf{Int}} x.\,x\langle ? \overset{p}{\Rightarrow} \mathsf{Int}\rangle + 1$, which checks at runtime whether the function parameter is bound to an integer value (and if not blames the label p). As customary blame labels have a polarity and we follow the standard convention of using ℓ to range over positive labels and $\bar{\ell}$ for negative ones.

2.2.2 Type System. The main typing rules for the cast language are presented in Figure 2. Type environments associate variables to type schemes of the form $\forall\vec{\alpha}.\tau$ (rule [Var]) and we use the standard rules for the introduction [TAbstr] and elimination [TApp] of type abstractions. Our typing rules for casts are more precise than the current literature, since they capture invariants that are typically captured by a separate safe-for relation that is used to establish the Blame Theorem [Wadler and Findler 2009]. Our casts are well-typed if they go from the type of the casted expression τ' to either a more precise (positive label) or a less precise (negative label) gradual type τ (rules [Cast$^{\oplus}$] and [Cast$^{\ominus}$], respectively). Blame safety usually involves two subtyping

Cast Reductions.

$$
\begin{array}{rrcll}
\text{[ExpandL]} & V\langle \tau \overset{p}{\Rightarrow} ? \rangle & \hookrightarrow & V\langle \tau \overset{p}{\Rightarrow} \tau/? \rangle \langle \tau/? \overset{p}{\Rightarrow} ? \rangle & \text{if } \tau/? \neq \tau \text{ and } \tau \neq ? \\
\text{[ExpandR]} & V\langle ? \overset{p}{\Rightarrow} \tau \rangle & \hookrightarrow & V\langle ? \overset{p}{\Rightarrow} \tau/? \rangle \langle \tau/? \overset{p}{\Rightarrow} \tau \rangle & \text{if } \tau/? \neq \tau \text{ and } \tau \neq ? \\
\text{[CastId]} & V\langle \tau \overset{p}{\Rightarrow} \tau \rangle & \hookrightarrow & V & \\
\text{[Collapse]} & V\langle \rho \overset{p}{\Rightarrow} ? \rangle \langle ? \overset{q}{\Rightarrow} \rho \rangle & \hookrightarrow & V & \\
\text{[Blame]} & V\langle \rho \overset{p}{\Rightarrow} ? \rangle \langle ? \overset{q}{\Rightarrow} \rho' \rangle & \hookrightarrow & \texttt{blame}\ q & \text{if } \rho \neq \rho'
\end{array}
$$

Standard Reductions.

$$
\begin{array}{rrcll}
\text{[CastApp]} & V\langle \tau_1 \to \tau_2 \overset{p}{\Rightarrow} \tau_1' \to \tau_2' \rangle V' & \hookrightarrow & V(V'\langle \tau_1' \overset{\bar{p}}{\Rightarrow} \tau_1 \rangle) \langle \tau_2 \overset{p}{\Rightarrow} \tau_2' \rangle & \\
\text{[App]} & (\lambda^{\tau_1 \to \tau_2} x.\, E) V & \hookrightarrow & E\{x := V\} & \\
\text{[ProjCast]} & \pi_i\,(V\langle \tau_1 \times \tau_2 \overset{p}{\Rightarrow} \tau_1' \times \tau_2' \rangle) & \hookrightarrow & (\pi_i\, V) \langle \tau_i \overset{p}{\Rightarrow} \tau_i' \rangle & \\
\text{[Proj]} & \pi_i\,(V_1, V_2) & \hookrightarrow & V_i & \\
\text{[TypeApp]} & (\Lambda \vec{\alpha}.\, E)\,[\vec{t}] & \hookrightarrow & E\{\vec{\alpha} := \vec{t}\} & \\
\text{[Let]} & \texttt{let}\ x = V\ \texttt{in}\ E & \hookrightarrow & E\{x := V\} & \\
\text{[Context]} & \mathcal{E}[E] & \hookrightarrow & \mathcal{E}[E'] & \text{if } E \hookrightarrow E' \\
\text{[CtxBlame]} & \mathcal{E}[E] & \hookrightarrow & \texttt{blame}\ p & \text{if } E \hookrightarrow \texttt{blame}\ p
\end{array}
$$

Fig. 3. Semantics of the Cast Calculus

relations, called *positive subtyping* (written $\leq^+$) and *negative subtyping* (written $\leq^-$), characterizing respectively casts that cannot yield positive blame and casts that cannot yield negative blame. By the factoring theorem for naive subtyping [Wadler and Findler 2009], $\tau' \preccurlyeq \tau$ implies $\tau' <:^+ \tau$, so a cast that satisfies rule [Cast$^\oplus$] is safe for ℓ. Conversely, $\tau \preccurlyeq \tau'$ implies $\tau' <:^- \tau$, so a cast that satisfies rule [Cast$^\ominus$] is also safe for ℓ. The remaining rules are standard (Figure 9 of the appendix).

2.2.3 Semantics. The cast calculus has a strict reduction semantics defined by the reduction rules in Figure 3. The semantics is defined in terms of values (ranged over by V), evaluation contexts (ranged over by $\mathcal{E}$), and ground types (ranged over by ρ). The first two are defined as follows:

$$V ::= c \mid \lambda^{\tau \to \tau} x.\, E \mid (V, V) \mid V\langle \tau_1 \to \tau_2 \overset{p}{\Rightarrow} \tau_1' \to \tau_2' \rangle \mid V\langle \tau_1 \times \tau_2 \overset{p}{\Rightarrow} \tau_1' \times \tau_2' \rangle \mid V\langle \rho \overset{p}{\Rightarrow} ? \rangle$$

$$\mathcal{E} ::= \square \mid E\,\mathcal{E} \mid \mathcal{E}\,V \mid \mathcal{E}\,[\vec{t}] \mid (E, \mathcal{E}) \mid (\mathcal{E}, V) \mid \pi_i\,\mathcal{E} \mid \texttt{let}\ x = \mathcal{E}\ \texttt{in}\ E \mid \mathcal{E}\langle \tau \overset{p}{\Rightarrow} \tau \rangle$$

As usual there are three value forms with casts [Siek et al. 2015a].

The notion of *ground type* was introduced by Wadler and Findler [2009] to compare types in casts, with the idea that *incompatibility between ground types is the source of all blame.* We next give a definition of ground types equivalent to the one of Wadler and Findler [2009], but which uses a different notation that is more convenient when we extend the system to set-theoretic types (§4).

Definition 2.6 (Grounding and Ground Types). *For every type $\tau \in \mathcal{T}_\tau$, we define the* grounding *of τ with respect to ?, noted $\tau/?$, as follows:*

$$
\begin{array}{rclcrclcrcl}
b/? & = & b & & \alpha/? & = & \alpha & & ?/? & = & ? \\
\tau_1 \to \tau_2/? & = & ? \to ? & & \tau_1 \times \tau_2/? & = & ? \times ? & &
\end{array}
$$

Types τ such that $\tau \neq ?$ and that verify $\tau/? = \tau$ are called ground types *and are ranged over by ρ.*

The reduction rules of Figure 3 closely follow the presentation of Siek et al. [2015a]. They are divided into two groups, the reductions for the application of casts to a value and the reductions corresponding to the elimination of type constructors. For the former we use the technique by Wadler and Findler [2009] which consists in checking whether a cast is performed between two types

with the same toplevel constructor and failing when this is not the case. This amounts to checking whether *grounding* the two types (by the rules [Expand_]) yields the same ground type (rule [Collapse]) or not (rule [Blame]). In regards to an implementation, the [ExpandL] rule corresponds to tagging a value with its type constructor (as done in Lisp implementations) and the [Collapse] rule corresponds to untagging a value. Most of the rules of the standard reductions group are taken from Siek et al. [2015a] too: we added the rules for type abstractions and applications, for projections, and for let bindings (all absent in the cited work). As usual, the function $\bar{\cdot}$ is involutory, that is, $\bar{\bar{p}} = p$.

The soundness of the cast calculus is proved via progress and subject reduction. We do not give a direct proof of these properties. They follow from the corresponding properties of the cast calculus of Section 4 (Lemmas 4.9, 4.10) and the conservativity of the extension (Theorem 4.13). The same holds true for the property of *blame safety* (Corollary 4.12).

2.2.4 Compilation. The final ingredient of the declarative definition of the system is to show how to compile a well-typed expression of the source language into an expression of the cast calculus and prove that compilation preserves types. This result, combined with the soundness of the cast language, implies the soundness of the gradually-typed language: a well-typed expression is compiled into an expression that can only either return a value of the same type, or return a cast error, or diverge.

Compilation is driven by the derivation of the type for the source language expression. Conceptually, compilation is straightforward: every time the derivation uses the [Materialize] rule on some subexpression for a relation $\tau_1 \preccurlyeq \tau_2$, a cast $\langle \tau_1 \overset{\ell}{\Rightarrow} \tau_2 \rangle$ must be added to that subexpression. Technically, we achieve this by enriching the judgements of typing derivations with a compilation part: $\Gamma \vdash e \rightsquigarrow E : \tau$ means that the source language expression e of type τ compiles to the cast language expression E. These judgements are derived by the same rules as those given for the source language in Figure 1 to whose judgements we add the compilation part. The only rules that need non-trivial modifications are the following ones:

$$[\textsc{Var}]\ \frac{}{\Gamma \vdash x \rightsquigarrow x[\vec{t}] : \tau\{\vec{\alpha} := \vec{t}\}}\ \Gamma(x) = \forall \vec{\alpha}.\, \tau \qquad [\textsc{Abstr}]\ \frac{\Gamma, x : t \vdash e \rightsquigarrow E : \tau}{\Gamma \vdash (\lambda x.\, e) \rightsquigarrow (\lambda^{t \to \tau} x.\, E) : t \to \tau}$$

$$[\textsc{Let}]\ \frac{\Gamma \vdash e_1 \rightsquigarrow E_1 : \tau_1 \qquad \Gamma, x : \forall \vec{\alpha}, \vec{\beta}.\, \tau_1 \vdash e_2 \rightsquigarrow E_2 : \tau}{\Gamma \vdash \mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2 \rightsquigarrow \mathsf{let}\ x = \Lambda \vec{\alpha}, \vec{\beta}.\, E_1\ \mathsf{in}\ E_2 : \tau}\ \vec{\alpha}, \vec{\beta} \sharp \Gamma \text{ and } \vec{\beta} \sharp e_1$$

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash e \rightsquigarrow E : \tau'}{\Gamma \vdash e \rightsquigarrow E\langle \tau' \overset{\ell}{\Rightarrow} \tau \rangle : \tau}\ \tau' \preccurlyeq \tau$$

[Var] compiles occurrences of polymorphic variables by instantiating them with the needed types. [Abstr] explicitly annotates the function with the type deduced by inference. The compilation of a let-construct abstracts the type variables that are generalized. Finally, the core of compilation is given by the [Materialize] rule, which corresponds to the insertion of an explicit cast (with a positive fresh label ℓ). All the remaining rules are straightforward modifications of the rules in Figure 1 insofar as their conclusions simply compose the compiled expressions in the premisses.

Compilation is defined for all well-typed expressions and preserves well-typing:

Theorem 2.7. *If $\Gamma \vdash e : \tau$, then there exists an E such that $\Gamma \vdash e \rightsquigarrow E : \tau$ and $\Gamma \vdash E : \tau$.*

2.3 Type Inference

In this section we show how to decide whether a given term is well-typed or not: we define a type inference algorithm that is sound and complete with respect to the system of the previous section. The algorithm is mostly based on the work of Pottier and Rémy [2005] and of Castagna et al. [2016], adapted for gradual typing. Our algorithm differs from that of Garcia and Cimini [2015] in that ours literally reduces the inference problem to unification. To infer the type of an expression, we generate constraints that specify the conditions that must hold for the expression to be well-typed; then, we solve these constraints via unification to obtain a solution (a type substitution).

Our presentation proceeds as follows. We first introduce *type constraints* (§2.3.1) and show how to solve sets of type constraints using standard unification (§2.3.2). Then we show how to generate constraints for a given expression (§2.3.3). To keep constraint generation separated from solving, generation uses more complex *structured constraints* (this is essentially due to the presence of let-polymorphism) which are then solved by simplifying them into the simpler *type constraints* (§2.3.4). Finally, we present our soundness and completeness results for type inference.

2.3.1 Type Constraints and Solutions. A *type constraint* has either the form $(t_1 \mathrel{\dot\lesssim} t_2)$ or the form $(\tau \mathrel{\dot\preccurlyeq} \alpha)$, whose meaning we give below. *Type constraint sets* (ranged over by the metavariable D) are finite sets of type constraints. We write $\mathsf{var}(D)$ for the set of type variables appearing in the type constraints in D. We write $\mathsf{var}_{\dot\preccurlyeq}(D)$ for the set of type variables appearing in the gradual types in materialization constraints in D: that is, $\mathsf{var}_{\dot\preccurlyeq}(D) = \bigcup_{(\tau \mathrel{\dot\preccurlyeq} \alpha) \in D} \mathsf{var}(\tau)$. When $\overline{\alpha} \subseteq \mathcal{V}^\alpha$ is a set of type variables and θ is a type substitution, we define $\overline{\alpha}\theta = \bigcup_{\alpha \in \overline{\alpha}} \mathsf{var}(\alpha\theta)$.

We say that a type substitution $\theta : \mathcal{V}^\alpha \to \mathcal{T}_\tau$ is a *solution* of a type constraint set D (with respect to a finite set $\Delta \subseteq \mathcal{V}^\alpha$), and we write $\theta \Vdash_\Delta D$, if:

- for every $(t_1 \mathrel{\dot\lesssim} t_2) \in D$, we have $t_1\theta = t_2\theta$;
- for every $(\tau \mathrel{\dot\preccurlyeq} \alpha) \in D$, we have $\tau\theta \preccurlyeq \alpha\theta$ and, for all $\beta \in \mathsf{var}(\tau)$, $\beta\theta$ is a static type;
- $\mathsf{dom}(\theta) \cap \Delta = \varnothing$.

A subtyping type constraint $(t_1 \mathrel{\dot\lesssim} t_2)$ forces the substitution to unify t_1 and t_2. We use $\mathrel{\dot\lesssim}$ instead of, say $\doteq$, to have uniform syntax with the later section on subtyping.

A materialization type constraint $(\tau \mathrel{\dot\preccurlyeq} \alpha)$ imposes two distinct requirements: the solution must make α a materialization of τ and must map all variables in τ to static types. These two conditions might be separated but in practice they must always be imposed together, and their combination simplifies the description of constraint solving. Note that the constraint $(\alpha \mathrel{\dot\preccurlyeq} \alpha)$ forces $\alpha\theta$ to be static (since the other requirement, $\alpha\theta \preccurlyeq \alpha\theta$, is trivial).

Finally, the set Δ is used to force the solution *not* to instantiate certain type variables.

2.3.2 Type Constraint Solving. We solve a type constraint set in three steps: we convert the type constraints to unification constraints between type frames (notably, by changing every occurrence of ? into a different frame variable); then we compute a unifier; finally, we convert the unifier into a solution (by renaming some variables and then changing frame variables back to ?).

We define this process as an algorithm $\mathsf{solve}_{(\cdot)}(\cdot)$ which, given a type constraint set D and a finite set $\Delta \subseteq \mathcal{V}^\alpha$, computes a set of type substitutions $\mathsf{solve}_\Delta(D)$. This set is either empty, indicating failure, or a singleton set containing the solution (which is unique up to variable renaming).[4]

We do not describe a unification algorithm explicitly; rather, we rely on properties satisfied by standard implementations (e.g., that by Martelli and Montanari [1982]). We use unification on type frames: its input is a finite set $\overline{T^1 \doteq T^2}$ of equality constraints of the form $T^1 \doteq T^2$. We also include as input a finite set $\Delta \subseteq \mathcal{V}^\alpha$ that specifies the variables that unification must *not* instantiate

[4]We use a set because, in the extension with subtyping, constraint solving can produce multiple incomparable solutions.

(i.e., that should be treated as constants). We write $\mathsf{unify}_\Delta(\overline{T^1 \doteq T^2})$ for the result of the algorithm, which is either fail or a type substitution $\theta : \mathcal{V}^\alpha \cup \mathcal{V}^X \to \mathcal{T}_T$. We assume that unify satisfies the usual soundness and completeness properties and that it computes idempotent substitutions.

Unification is the main ingredient of our type constraint solving algorithm, but we need some extra steps to handle materialization constraints.

Let D be of the form $\{ (t_i^1 \dot{\leq} t_i^2) \mid i \in I \} \cup \{ (\tau_j \dot{\leqslant} \alpha_j) \mid j \in J \}$: then $\mathsf{solve}_\Delta(D)$ is defined as follows.

(1) Let $\overline{T^1 \doteq T^2}$ be $\{ (t_i^1 \doteq t_i^2) \mid i \in I \} \cup \{ (T_j \doteq \alpha_j) \mid j \in J \}$ where the T_j are chosen to ensure:
  (a) for each $j \in J$, $T_j^\dagger = \tau_j$;
  (b) every frame variable X occurs in at most one of the T_j, at most once.

(2) Compute $\mathsf{unify}_\Delta(\overline{T^1 \doteq T^2})$:
  (a) if $\mathsf{unify}_\Delta(\overline{T^1 \doteq T^2}) = \mathsf{fail}$, return $\varnothing$;
  (b) if $\mathsf{unify}_\Delta(\overline{T^1 \doteq T^2}) = \theta_0$, return $\{(\theta_0\theta_0')^\dagger|_{\mathcal{V}^\alpha}\}$ where:
    (i) $\theta_0' = \{\vec{X} \coloneqq \vec{\alpha}'\} \cup \{\vec{\alpha} \coloneqq \vec{X}'\}$
    (ii) $\vec{X} = \mathcal{V}^X \cap \mathsf{var}_{\dot{\leqslant}}(D)\theta_0$ and $\vec{\alpha} = \mathsf{var}(D) \setminus (\Delta \cup \mathsf{dom}(\theta_0) \cup \mathsf{var}_{\dot{\leqslant}}(D)\theta_0)$
    (iii) $\vec{\alpha}'$ and $\vec{X}'$ are vectors of fresh variables

In step 1, we convert D to a set of type frame equality constraints. To do so, we convert all gradual types in materialization constraints by replacing each occurrence of ? with a different frame variable. In step 2, we compute a unifier for these constraints. If a unifier θ_0 exists (step 2b), we use it to build our solution: however, we need a post-processing step to ensure that α and X variables are treated correctly. For example, a unifier could map $\alpha \coloneqq X$ when $(\alpha \dot{\leqslant} \alpha) \in D$: then, converting type frames back to gradual types would yield $\alpha \coloneqq ?$, which is not a solution because α is mapped to a gradual type, but a static type is required. Therefore, to obtain the result we first compose θ_0 with a renaming substitution θ_0'; then, we apply $\dagger$ to change type frames back to gradual types, and we restrict the domain to $\mathcal{V}^\alpha$. The renaming introduces fresh variables to replace some frame variables with type variables ($\{\vec{X} \coloneqq \vec{\alpha}'\}$) and some type variables with frame variables ($\{\vec{\alpha} \coloneqq \vec{X}'\}$). It has two purposes. One is to ensure that the variables in $\mathsf{var}_{\dot{\leqslant}}(D)$ are mapped to static types, which we need for $\theta \Vdash_\Delta D$ to hold. The other is to have the substitution introduce as few type variables as possible.

$\mathsf{solve}_{(\cdot)}(\cdot)$ satisfies the following properties.

- Soundness: if $\theta \in \mathsf{solve}_\Delta(D)$, then $\theta \Vdash_\Delta D$.
- Completeness: if $\theta \Vdash_\Delta D$, then there exist two substitutions θ' and θ'' such that
  - $\theta' \in \mathsf{solve}_\Delta(D)$;
  - for every α, $\alpha\theta'(\theta \cup \theta'') \preccurlyeq \alpha(\theta \cup \theta'')$ and, if $\alpha\theta'$ is static, $\alpha\theta'(\theta \cup \theta'') = \alpha(\theta \cup \theta'')$.
- If $\theta \in \mathsf{solve}_\Delta(D)$, then $\mathsf{var}(D)\theta \subseteq \Delta \cup \mathsf{var}_{\dot{\leqslant}}(D)\theta$.

The last property states that a solution θ returned by solve introduces as few variables as possible. In particular, the variables it introduces in D are only those in Δ and those that appear in the solutions of variables in $\mathsf{var}_{\dot{\leqslant}}(D)$ (whose solutions must be static). To ensure this, we perform the substitution $\{\vec{\alpha} \coloneqq \vec{X}'\}$. This property implies that we avoid useless materializations of ? to type variables (and thus the insertion of useless casts at compilation): for example, it ensures that, in $\mathsf{let}\ y = x\ \mathsf{in}\ e$, if x has type ?, then y is given type ? too. In the declarative system, it can be typed also as $\forall\alpha.\,\alpha$, but then the compiled expression has a cast: $\mathsf{let}\ y = \Lambda\alpha.\,x\langle ? \overset{\ell}{\Rightarrow} \alpha\rangle\ \mathsf{in}\ E$. We prefer the compilation without this cast, which is why we replace as many α variables as possible with ?.

2.3.3 Structured Constraints and Constraint Generation. In the absence of let-polymorphism, the type constraints we presented suffice to describe the conditions for a program to be well-typed

$$
\begin{aligned}
\langle\!\langle x \colon t \rangle\!\rangle &= \exists \alpha.\, (x \mathrel{\dot{\preccurlyeq}} \alpha) \wedge (\alpha \mathrel{\dot{\leq}} t) && \alpha \sharp t \\
\langle\!\langle c \colon t \rangle\!\rangle &= (b_c \mathrel{\dot{\leq}} t) \\
\langle\!\langle (\lambda x.\, e) \colon t \rangle\!\rangle &= \exists \alpha_1, \alpha_2.\, (\mathsf{def}\ x \colon \alpha_1\ \mathsf{in}\ \langle\!\langle e \colon \alpha_2 \rangle\!\rangle) \wedge (\alpha_1 \mathrel{\dot{\preccurlyeq}} \alpha_1) \wedge (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t) && \alpha_1, \alpha_2 \sharp t, e \\
\langle\!\langle (\lambda x \colon \tau.\, e) \colon t \rangle\!\rangle &= \exists \alpha_1, \alpha_2.\, (\mathsf{def}\ x \colon \tau\ \mathsf{in}\ \langle\!\langle e \colon \alpha_2 \rangle\!\rangle) \wedge (\tau \mathrel{\dot{\preccurlyeq}} \alpha_1) \wedge (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t) && \alpha_1, \alpha_2 \sharp t, \tau, e \\
\langle\!\langle e_1\ e_2 \colon t \rangle\!\rangle &= \exists \alpha.\, \langle\!\langle e_1 \colon \alpha \to t \rangle\!\rangle \wedge \langle\!\langle e_2 \colon \alpha \rangle\!\rangle && \alpha \sharp t, e_1, e_2 \\
\langle\!\langle (e_1, e_2) \colon t \rangle\!\rangle &= \exists \alpha_1, \alpha_2.\, \langle\!\langle e_1 \colon \alpha_1 \rangle\!\rangle \wedge \langle\!\langle e_2 \colon \alpha_2 \rangle\!\rangle \wedge (\alpha_1 \times \alpha_2 \mathrel{\dot{\leq}} t) && \alpha_1, \alpha_2 \sharp t, e_1, e_2 \\
\langle\!\langle \pi_i\, e \colon t \rangle\!\rangle &= \exists \alpha_1, \alpha_2.\, \langle\!\langle e \colon \alpha_1 \times \alpha_2 \rangle\!\rangle \wedge (\alpha_i \mathrel{\dot{\leq}} t) && \alpha_1, \alpha_2 \sharp t, e \\
\langle\!\langle \mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2 \colon t \rangle\!\rangle &= \mathsf{let}\ x \colon \forall \vec{\alpha}; \alpha[\langle\!\langle e_1 \colon \alpha \rangle\!\rangle]^{\mathrm{var}(e_1) \setminus \vec{\alpha}}.\, \alpha\ \mathsf{in}\ \langle\!\langle e_2 \colon t \rangle\!\rangle && \alpha \sharp \vec{\alpha}, e_1
\end{aligned}
$$

Fig. 4. Constraint generation.

(following the approach of Wand [1987], augmented with materialization constraints). With let-polymorphism, instead, we would need either to mix constraint generation and solving or to copy constraints for let-bound expressions multiple times. To avoid this, we use a kind of constraint that includes binding, following Pottier and Rémy [2005].

A *structured constraint* is a term generated by the following grammar:

$$C ::= (t \mathrel{\dot{\leq}} t) \mid (\tau \mathrel{\dot{\preccurlyeq}} \alpha) \mid (x \mathrel{\dot{\preccurlyeq}} \alpha) \mid \mathsf{def}\ x \colon \tau\ \mathsf{in}\ C \mid \exists \vec{\alpha}.\, C \mid C \wedge C \mid \mathsf{let}\ x \colon \forall \vec{\alpha}; \alpha[C]^{\vec{\alpha}}.\, \alpha\ \mathsf{in}\ C$$

Structured constraints are considered equal up to α-renaming of bound variables. In $\exists \vec{\alpha}.\, C$, the $\vec{\alpha}$ variables are bound in C. In $\mathsf{let}\ x \colon \forall \vec{\alpha}; \alpha[C_1]^{\vec{\alpha}'}.\, \alpha\ \mathsf{in}\ C_2$, α and the $\vec{\alpha}$ variables are bound in C_1.

Structured constraints include type constraints and five other forms. A constraint $(x \mathrel{\dot{\preccurlyeq}} \alpha)$ asks that the type scheme for x has an instance that materializes to the solution of α. Existential constraints $\exists \vec{\alpha}.\, C$ bind the type variables $\vec{\alpha}$ occurring in C; this simplifies freshness conditions, as in Pottier and Rémy [2005]. $C \wedge C$ is simply the conjunction of two constraints, while def and let constraints are generated to type λ-abstractions and let-expressions, as explained below.

Figure 4 defines a function $\langle\!\langle (\cdot) \colon (\cdot) \rangle\!\rangle$ such that, for every expression e and every static type t, $\langle\!\langle e \colon t \rangle\!\rangle$ expresses the conditions that must hold for e to have type $t\theta$ for some substitution θ.

We point out some peculiarities of the rules. For variables, we generate a constraint combining materialization and subtyping. This allows us to use the form $(x \mathrel{\dot{\preccurlyeq}} \alpha)$ instead of $(x \mathrel{\dot{\preccurlyeq}} t)$; more importantly, it means the same definition for constraint generation can be reused when we add subtyping. For a λ-abstraction, constraint generation wraps the constraint for the body in a def constraint to introduce the type of the parameter. In the absence of annotations, the constraint $(\alpha_1 \mathrel{\dot{\preccurlyeq}} \alpha_1)$ is used to ensure that the parameter will have a static type. For annotated functions, the constraint $(\tau \mathrel{\dot{\preccurlyeq}} \alpha_1)$ allows the domain of the function to be materialized. This is needed, for example, to obtain solvable constraints for the abstraction $(\lambda x \colon ?.\, x)$ in a context expecting $\mathsf{Int} \to \mathsf{Int}$. For let, we build a let constraint including the constraints of the two expressions and recording the variables that *must* be generalized ($\vec{\alpha}$) and those that must *not* ($\mathrm{var}(e_1) \setminus \vec{\alpha}$)[5]. In all rules, the side conditions force the choice of fresh variables.

2.3.4 Constraint Solving. While our definition of constraints is mostly based on the work of Pottier and Rémy [2005], we describe constraint solving differently, following Castagna et al. [2016]. We solve structured constraints in two steps: we convert a structured constraint to a type constraint set with the *constraint simplification* system of Figure 5; then, we compute a solution using the type constraint solving algorithm of §2.3.2. Because of let-polymorphism, constraint simplification also uses type constraint solving internally to compute partial solutions.

[5]We include the latter for convenience: actually, they can be recomputed from the rest since $\mathrm{var}(e_1) = \mathrm{var}(\langle\!\langle e_1 \colon \alpha \rangle\!\rangle) \setminus \{\alpha\}$.

$$\frac{}{\Gamma;\Delta \vdash (t_1 \dot{\leq} t_2) \rightsquigarrow \{t_1 \dot{\leq} t_2\}} \qquad \frac{}{\Gamma;\Delta \vdash (\tau \dot{\preccurlyeq} \alpha) \rightsquigarrow \{\tau \dot{\preccurlyeq} \alpha\}}$$

$$\frac{}{\Gamma;\Delta \vdash (x \dot{\preccurlyeq} \alpha) \rightsquigarrow \{\tau\{\vec{\alpha} := \vec{\beta}\} \dot{\preccurlyeq} \alpha\}} \begin{array}{l}\Gamma(x) = \forall \vec{\alpha}.\, \tau \\ \vec{\beta} \text{ fresh}\end{array} \qquad \frac{(\Gamma, x\colon \tau);\Delta \vdash C \rightsquigarrow D}{\Gamma;\Delta \vdash \mathsf{def}\ x\colon \tau\ \mathsf{in}\ C \rightsquigarrow D}$$

$$\frac{\Gamma;\Delta \vdash C \rightsquigarrow D}{\Gamma;\Delta \vdash (\exists \vec{\alpha}.\, C) \rightsquigarrow D}\ \vec{\alpha} \text{ fresh} \qquad \frac{\Gamma;\Delta \vdash C_1 \rightsquigarrow D_1 \qquad \Gamma;\Delta \vdash C_2 \rightsquigarrow D_2}{\Gamma;\Delta \vdash C_1 \wedge C_2 \rightsquigarrow D_1 \cup D_2}$$

$$\frac{\Gamma;\Delta \cup \vec{\alpha} \vdash C_1 \rightsquigarrow D_1 \qquad (\Gamma, x\colon \forall \vec{\alpha}, \vec{\beta}.\, \alpha\theta_1);\Delta \vdash C_2 \rightsquigarrow D_2}{\Gamma;\Delta \vdash \mathsf{let}\ x\colon \forall \vec{\alpha}; \alpha[C_1]^{\vec{\alpha}'}.\, \alpha\ \mathsf{in}\ C_2 \rightsquigarrow D_2 \cup \mathsf{equiv}(\theta_1, D_1)} \begin{array}{l}\theta_1 \in \mathsf{solve}_{\Delta \cup \vec{\alpha}}(D_1) \\ \vec{\alpha} \sharp \Gamma\theta_1 \\ \vec{\beta} = \mathsf{var}(\alpha\theta_1) \setminus (\mathsf{var}(\Gamma\theta_1) \cup \vec{\alpha} \cup \vec{\alpha}') \\ \vec{\alpha}, \alpha \text{ fresh}\end{array}$$

$$\text{where } \mathsf{equiv}(\theta, D) \stackrel{\text{def}}{=} \left\{ (\alpha \dot{\preccurlyeq} \alpha) \,\middle|\, \alpha \in \mathsf{var}_{\dot{\preccurlyeq}}(D) \cup \mathsf{var}(D)\theta \right\} \cup \bigcup_{\substack{\alpha \in \mathsf{dom}(\theta) \\ \alpha\theta \text{ static}}} \left\{ (\alpha \dot{\leq} \alpha\theta), (\alpha\theta \dot{\leq} \alpha) \right\}$$

Fig. 5. Constraint simplification.

Constraint simplification is a relation $\Gamma;\Delta \vdash C \rightsquigarrow D$. The Γ is a type environment used to assign types to the variables in constraints of the form $(x \dot{\preccurlyeq} \alpha)$. Δ is a finite subset of $\mathcal{V}^\alpha$ and is used to record variables that must not be instantiated. When simplifying constraints for a whole program, we take Γ to be empty and Δ to be the set of free type variables in the program (presumably empty as well). Finally, C is the constraint to be simplified and Δ the result of simplification.

The rules are syntax-directed and deterministic (modulo the choice of fresh variables). Subtyping and materialization constraints are left unchanged. Variable constraints $(x \dot{\preccurlyeq} \alpha)$ are converted to materialization constraints by replacing x with a fresh instance of its type scheme. To simplify a def constraint, we update the environment and simplify the inner constraint. For $\exists \vec{\alpha}.\, C$, we simplify C after performing α-renaming, if needed, to ensure that $\vec{\alpha}$ is fresh. To simplify $C_1 \wedge C_2$, we simplify C_1 and C_2 and take the union of the resulting sets.

Finally, the rule for let constraints is of course the most complicated. To simplify a constraint $\mathsf{let}\ x\colon \forall \vec{\alpha}; \alpha[C_1]^{\vec{\alpha}'}.\, \alpha\ \mathsf{in}\ C_2$, we perform five steps:

(1) we simplify the constraint C_1 to obtain a set D_1;
(2) we apply the solve algorithm to D_1 to obtain a solution θ_1, if one exists;
(3) we compute the type scheme for x by generalizing the type given by the solution;
(4) we simplify the constraint C_2 in the expanded environment to obtain a set D_2;
(5) finally, we add to D_2 the set $\mathsf{equiv}(\theta_1, D_1)$, whose purpose is to constrain the solution to be an instantiation of θ_1 and to yield static types where needed.

In steps (1) and (2), we add $\vec{\alpha}$ to Δ to ensure that the $\vec{\alpha}$ variables are not instantiated while solving C_1, otherwise we could not generalize them later. The type $\alpha\theta_1$ for x is generalized by quantifying over the $\vec{\alpha}$ variables (checking that they are not introduced in the environment by θ_1) as well as over $\vec{\beta}$, which contains all variables in $\alpha\theta_1$ that do not appear in any of $\Gamma\theta_1$, $\vec{\alpha}$, or $\vec{\alpha}'$. Recall that we record in $\vec{\alpha}'$ the variables that cannot be generalized (typically because they appeared in the expression but not in the decoration of the let construct).

We use the set $\mathsf{equiv}(\theta_1, D_1)$ to constrain a solution θ to adhere to θ_1 in two ways. First, θ must map to static types all variables in $\mathsf{var}_{\dot{\preccurlyeq}}(D_1)$ (which θ_1 had to map to static types) and all variables introduced by θ_1. Also, θ must satisfy $\alpha\theta_1\theta = \alpha\theta$ whenever $\alpha\theta_1$ is a static type. To ensure the latter, we add the two subtyping constraints $(\alpha \dot{\leq} \alpha\theta_1)$ and $(\alpha\theta_1 \dot{\leq} \alpha)$. Adding both is redundant here

(both require equality), but they are needed when we add subtyping. The freshness conditions are stated informally here. In the Appendix, we give a definition where we track explicitly the variables we introduce and state the conditions precisely (Figure 11).

The results of type inference can also be used to compile expressions. In particular, when e is an expression, $\mathcal{D}$ is a derivation of $\Gamma; \Delta \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D$, and $\theta \Vdash_\Delta D$, we can compute a cast language expression $(\!|e|\!)^{\mathcal{D}}_{\theta}$. For reasons of space, the (straightforward) definition of $(\!|e|\!)^{\mathcal{D}}_{\theta}$ is in the Appendix. The following results hold.

Theorem 2.8 (Soundness of type inference). *Let $\mathcal{D}$ be a derivation of $\Gamma; \mathsf{var}(e) \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D$. Let θ be a type substitution such that $\theta \Vdash_{\mathsf{var}(e)} D$. Then, we have $\Gamma\theta \vdash e \rightsquigarrow (\!|e|\!)^{\mathcal{D}}_{\theta} : t\theta$.*

Theorem 2.9 (Completeness of type inference). *If $\Gamma \vdash e : \tau$, then, for every fresh type variable α, there exist D and θ such that $\Gamma; \mathsf{var}(e) \vdash \langle\!\langle e : \alpha \rangle\!\rangle \rightsquigarrow D$ and $\{\alpha \coloneqq \tau\} \cup \theta \Vdash_{\mathsf{var}(e)} D$.*

The latter result, combined with completeness of solve, ensures that inference can compute most general types for all expressions. In particular, starting from a program (i.e., a closed expression) e, we pick a fresh variable α and generate $\langle\!\langle e : \alpha \rangle\!\rangle$. Theorem 2.9 ensures that, if the program is well-typed, we can find a derivation $\mathcal{D}$ for $\varnothing; \varnothing \vdash \langle\!\langle e : \alpha \rangle\!\rangle \rightsquigarrow D$ and D has a solution. Since solve is complete, we can compute the principal solution θ of D. Then, $\alpha\theta$ is the most general type for the program and $(\!|e|\!)^{\mathcal{D}}_{\theta}$ is its compilation driven by the derivation $\mathcal{D}$.

2.3.5 An Example of Type Inference. Let e be the term $\mathsf{let}\ \alpha\ x = (\lambda y : \alpha.\, y)\ \mathsf{in}\ 1 + \big(x\ ((\lambda z : ?.\, z)\ 3)\big)$ (we assume to have a + operator in the language). Since $x\ ((\lambda z : ?.\, z)\ 3)$ is used as a number, to be well-typed it should be given type Int. In the declarative system, $\lambda z : ?.\, z$ has type $? \to ?$, which can be materialized to $\mathsf{Int} \to \mathsf{Int}$; then its application to 3 has type Int; therefore applying the identity function x, we also get type Int. Inference can find this solution, as follows. We use a type variable β as the expected type, and we generate the constraints below. We have:

$$
\begin{aligned}
C = \langle\!\langle e : \beta \rangle\!\rangle = \langle\!\langle \mathsf{let}\ \alpha\ x = (\lambda y : \alpha.\, y)\ \mathsf{in}\ 1 + \big(x\ ((\lambda z : ?.\, z)\ 3)\big) : \beta \rangle\!\rangle &= \mathsf{let}\ x : \forall\alpha; \alpha_1[C_1]^{\epsilon}.\, \alpha_1\ \mathsf{in}\ C_2 \\
C_1 = \langle\!\langle (\lambda y : \alpha.\, y) : \alpha_1 \rangle\!\rangle & \\
= \exists \alpha_2, \alpha_3.\, \big(\mathsf{def}\ y : \alpha\ \mathsf{in}\ \langle\!\langle y : \alpha_3 \rangle\!\rangle\big) \wedge (\alpha \mathrel{\dot{\preccurlyeq}} \alpha_2) \wedge (\alpha_2 \to \alpha_3 \mathrel{\dot{\leq}} \alpha_1) & \\
C_2 = \langle\!\langle 1 + \big(x\ ((\lambda z : ?.\, z)\ 3)\big) : \beta \rangle\!\rangle = (\mathsf{Int} \mathrel{\dot{\leq}} \beta) \wedge \langle\!\langle x\ ((\lambda z : ?.\, z)\ 3) : \mathsf{Int} \rangle\!\rangle & \\
= (\mathsf{Int} \mathrel{\dot{\leq}} \beta) \wedge \big(\exists \alpha_4.\, \langle\!\langle x : \alpha_4 \to \mathsf{Int} \rangle\!\rangle & \\
\quad \wedge \big(\exists \alpha_5.\, \langle\!\langle (\lambda z : ?.\, z) : \alpha_5 \to \alpha_4 \rangle\!\rangle \wedge (b_3 \mathrel{\dot{\leq}} \alpha_5)\big)\big) & \\
\langle\!\langle y : \alpha_3 \rangle\!\rangle = \exists \alpha_6.\, (y \mathrel{\dot{\preccurlyeq}} \alpha_6) \wedge (\alpha_6 \mathrel{\dot{\leq}} \alpha_3) & \\
\langle\!\langle x : \alpha_4 \to \mathsf{Int} \rangle\!\rangle = \exists \alpha_7.\, (x \mathrel{\dot{\preccurlyeq}} \alpha_7) \wedge (\alpha_7 \mathrel{\dot{\leq}} \alpha_4 \to \mathsf{Int}) & \\
\langle\!\langle (\lambda z : ?.\, z) : \alpha_5 \to \alpha_4 \rangle\!\rangle = \exists \alpha_8, \alpha_9.\, \big(\mathsf{def}\ z : ?\ \mathsf{in}\ \exists \alpha_{10}.\, (z \mathrel{\dot{\preccurlyeq}} \alpha_{10}) \wedge (\alpha_{10} \mathrel{\dot{\leq}} \alpha_9)\big) & \\
\qquad \wedge (? \mathrel{\dot{\preccurlyeq}} \alpha_8) \wedge (\alpha_8 \to \alpha_9 \mathrel{\dot{\leq}} \alpha_5 \to \alpha_4) &
\end{aligned}
$$

We simplify C in the empty environment with $\Delta = \varnothing$. To do this, we first simplify C_1: we have $\varnothing; \{\alpha\} \vdash C_1 \rightsquigarrow \big\{(\alpha \mathrel{\dot{\preccurlyeq}} \alpha_6), (\alpha_6 \mathrel{\dot{\leq}} \alpha_3), (\alpha \mathrel{\dot{\preccurlyeq}} \alpha_2), (\alpha_2 \to \alpha_3 \mathrel{\dot{\leq}} \alpha_1)\big\}$. Then, through unification we can obtain the solution $\theta_1 = \{\alpha_1 \coloneqq (\alpha \to \alpha), \alpha_2 \coloneqq \alpha, \alpha_3 \coloneqq \alpha, \alpha_6 \coloneqq \alpha\}$. We obtain the expanded environment $x : \forall\alpha.\, \alpha \to \alpha$. Then, we simplify C_2. We have $(x : \forall\alpha.\, \alpha \to \alpha); \varnothing \vdash C_2 \rightsquigarrow D_2$ with $D_2 = \big\{(\gamma \to \gamma \mathrel{\dot{\preccurlyeq}} \alpha_7), (\alpha_7 \mathrel{\dot{\leq}} \alpha_4 \to \mathsf{Int}), (? \mathrel{\dot{\preccurlyeq}} \alpha_{10}), (\alpha_{10} \mathrel{\dot{\leq}} \alpha_9), (? \mathrel{\dot{\preccurlyeq}} \alpha_8), (\alpha_8 \to \alpha_9 \mathrel{\dot{\leq}} \alpha_5 \to \alpha_4), (b_3 \mathrel{\dot{\leq}} \alpha_5)\big\}$. The final constraint set is $D = D_2 \cup \mathsf{equiv}(\theta_1, D_1)$, with

$$
\begin{aligned}
\mathsf{equiv}(\theta_1, D_1) = \{&(\alpha \mathrel{\dot{\preccurlyeq}} \alpha), (\alpha_1 \mathrel{\dot{\leq}} \alpha \to \alpha), (\alpha \to \alpha \mathrel{\dot{\leq}} \alpha_1), \\
&(\alpha_2 \mathrel{\dot{\leq}} \alpha), (\alpha \mathrel{\dot{\leq}} \alpha_2), (\alpha_3 \mathrel{\dot{\leq}} \alpha), (\alpha \mathrel{\dot{\leq}} \alpha_3), (\alpha_6 \mathrel{\dot{\leq}} \alpha), (\alpha \mathrel{\dot{\leq}} \alpha_6)\}\,.
\end{aligned}
$$

A solution to D is

$$
\theta = \theta_1 \cup \{\alpha_4 \coloneqq \mathsf{Int}, \alpha_5 \coloneqq \mathsf{Int}, \alpha_7 \coloneqq (\mathsf{Int} \to \mathsf{Int}), \alpha_8 \coloneqq \mathsf{Int}, \alpha_9 \coloneqq \mathsf{Int}, \alpha_{10} \coloneqq \mathsf{Int}, \beta \coloneqq \mathsf{Int}, \gamma \coloneqq \mathsf{Int}\}\,.
$$

Let $\mathcal{D}$ be the derivation of constraint simplification that we have described. Then, the compiled expression $(\!|e|\!)^{\mathcal{D}}_{\theta}$ is (omitting identity casts)

$$\mathsf{let}\ x = (\Lambda\alpha.\, \lambda^{\alpha\to\alpha} y.\, y)\ \mathsf{in}\ \big(x\ [\mathsf{Int}]\big)\big((\lambda^{?\to\mathsf{Int}} z.\, z\langle ? \overset{\ell_1}{\Rightarrow} \mathsf{Int}\rangle\langle ? \to \mathsf{Int} \overset{\ell_2}{\Rightarrow} \mathsf{Int} \to \mathsf{Int}\rangle\ 3\big)\ .$$

3 GRADUAL TYPING WITH SUBTYPING

In this section, we add subtyping to the system of the previous section. We just outline the main differences and the necessary additions without giving the details. In particular, we present only the declarative systems since developing the algorithmic counterpart requires set-theoretic operations on types, a topic that we thoroughly deal with in Section 4. For this section we prioritize simplicity, which is why we give a simple syntactic definition for subtyping instead of the more complex but extension-robust semantic definition of it, that is postponed to Section 4.

3.1 Declarative System

3.1.1 Subtyping. We suppose to start from a predefined subtyping preorder relation $\leq$ on $\mathcal{B}$ (e.g., $\mathsf{Odd} \leq \mathsf{Int} \leq \mathsf{Real}$) and we extend it to the set $\mathcal{T}_\tau$ of gradual types by the inductive application of the following inference rules:

$$\frac{}{? \leq ?} \qquad \frac{}{\alpha \leq \alpha} \qquad \frac{\tau_1 \leq \tau_1' \qquad \tau_2 \leq \tau_2'}{\tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'} \qquad \frac{\tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$$

These rules are standard: covariance for products, co-contravariance for arrows. Just notice that, from the point of view of subtyping, the dynamic type ? is only related to itself, just like a type variable (*cf.* [Siek and Taha 2007]).

3.1.2 Type System. The extension of the source gradual language with subtyping could not be simpler: it suffices to add the standard subsumption rule to the declarative typing rules of Figure 1:

$$[\textsc{Subsume}]\ \frac{\Gamma \vdash e : \tau'}{\Gamma \vdash e : \tau}\ \tau' \leq \tau$$

The definition of the dynamic semantics does not require any essential change, either. The cast calculus is the same as in Section 2.2, except that the [Subsume] rule above must be added to its typing rules and the two cast reduction rules that use type equality must be generalized to subtyping (again, type soundness will be a consequence of the conservativity of the system in Section 4), namely:

$$\begin{array}{rlcll} [\textsc{Collapse}] & V\langle \rho \overset{p}{\Rightarrow} ?\rangle\langle ? \overset{q}{\Rightarrow} \rho'\rangle & \hookrightarrow & V & \text{if } \rho \leq \rho' \\ [\textsc{Blame}] & V\langle \rho \overset{p}{\Rightarrow} ?\rangle\langle ? \overset{q}{\Rightarrow} \rho'\rangle & \hookrightarrow & \mathtt{blame}\ q & \text{if } \rho \not\leq \rho' \end{array}$$

The definition of the compilation of the source language into the "new" cast calculus does not change either (subsumption is neutral for compilation). The proof that compilation preserves types stays essentially the same, since we have just added the subsumption rule to both systems.

3.2 Type Inference

The changes required to add subtyping to the declarative system are minimal: define the subtyping relation, add the subsumption rule, and recheck the proofs since they need slight modifications. On the contrary, defining algorithms to decide the relations we just defined is more complicated. As we saw in Section 2.3, this amounts to (1) generating a set of constraints and (2) solving it.

Constraint generation is not problematic. The form of the constraints and the generation algorithm given in Section 2.3 already account for the extension with subtyping: hence, they do not need to be changed, neither here nor in the next section. Constraint resolution, instead, is a different

matter. In the previous section, constraints of the form $\alpha \doteq t$ were actually equality constraints (i.e., $\alpha \doteq t$) that could be solved by unification. The same constraints now denote subtyping, and their resolution requires the computation of intersections and unions. To see why, consider the following OCaml code snippet (that does not involve any gradual typing):

```
fun x -> if (fst x) then (1 + snd x) else x
```

We want our system to deduce for this definition the following type:

$$\texttt{(Bool×Int)} \to \texttt{(Int | (Bool×Int))}$$

To that end, a constraint generation system like the one we present in the next section would assign to the function the type $\alpha \to \beta$ and generate the following set of four constraints: $\{(\alpha \doteq \mathsf{Bool}\times\mathbb{1}), (\alpha \doteq \mathbb{1}\times\mathsf{Int}), (\mathsf{Int} \doteq \beta), (\alpha \doteq \beta)\}$, where $\mathbb{1}$ denotes the top type (that is, the supertype of all types). The first constraint is generated because `fst x` is used in a position where a Boolean is expected; the second comes from the use of `snd x` in an integer position; the last two constraints are produced to type the result of an `if_then_else` expression (with a supertype of the types of both branches). To compute the solution of two constraints of the form $\alpha \doteq t_1$ and $\alpha \doteq t_2$, the resolution algorithm must compute the greatest lower bound of t_1 and t_2 (or an approximation thereof); likewise for two constraints of the form $s_1 \doteq \beta$ and $s_2 \doteq \beta$ the best solution is the least upper bound of s_1 and s_2. This yields $\mathsf{Bool} \times \mathsf{Int}$ for the domain —i.e., the intersection of the upper bounds for α— and `(Int | (Bool×Int))` for the codomain—i.e., the union of the lower bounds for β.

In summary, to perform type reconstruction in the presence of subtyping, one must be able to compute unions and intersections of types. In some cases, as for the domain in the example above, the solution of these operations is a type of ML (or of the language at issue): then the operations can be meta-operators computed by the type-checker but not exposed to the programmer. In other cases, as for the codomain in the example, the solution is a type which might not already exist in the language: therefore, the only solution to type the expression precisely is to add the corresponding set-theoretic operations to the types of the language.

The full range of these options can be found in the literature. For instance, Pottier [2001] defines intersection and union as meta-operations, and it is not possible to simplify the constraints to derive a type like the one above. Hosoya et al. [2000] implement a hybrid solution in which intersections are meta-operations while full-fledged unions —which are necessary to encode XML types— are included in types. Other systems include both intersections and unions in the types, starting from the earliest work by Aiken and Wimmers [1993] to more recent work by Dolan and Mycroft [2017]. Union and intersection types are the most expressive solution but also the one that is technically most challenging; this is why the cited works impose some restrictions on the use of unions and intersections (e.g., no unions in covariant position and no intersections in contravariant ones). In the next section, we embrace unrestricted union and intersection types, adding them to both static and gradual types. In particular, we follow the approach of *semantic subtyping* by Frisch et al. [2008], which also requires the addition of negation and recursive types.

4 GRADUAL TYPING WITH SET-THEORETIC TYPES

In this section we add set-theoretic types to our system. From a syntactic viewpoint, this means we add union and negation type connectives, plus the empty (or bottom) type, to all the previous categories of types; intersection and the top type are encoded. We also introduce recursive types: besides the interest of recursive types *per se*, we need them to solve subtyping constraints following a technique introduced by Courcelle [1983]. Instead of using explicit recursion, say, by a μ-binder, we define types coinductively as infinite trees satisfying regularity and contractivity conditions. Such a definition is equivalent to one using μ-binders, but it fits our framework better.

Definition 4.1 (Type syntax). *The sets $\mathcal{T}_t$, $\mathcal{T}_\tau$, and $\mathcal{T}_T$ are the sets of terms t, τ, and T produced coinductively by the following grammars*

$$\begin{array}{rl} \textit{static types} & t ::= \alpha \mid b \mid t \times t \mid t \to t \mid t \vee t \mid \neg t \mid \mathbb{0} \\ \textit{gradual types} & \tau ::= ? \mid \alpha \mid b \mid \tau \times \tau \mid \tau \to \tau \mid \tau \vee \tau \mid \neg\tau \mid \mathbb{0} \\ \textit{type frames} & T ::= X \mid \alpha \mid b \mid T \times T \mid T \to T \mid T \vee T \mid \neg T \mid \mathbb{0} \end{array}$$

and that satisfy the following conditions:

- (regularity) *the term has a finite number of different sub-terms;*
- (contractivity) *every infinite branch of a type contains an infinite number of occurrences of the product or arrow type constructors.*

We introduce the following abbreviations for types: $\tau_1 \wedge \tau_2 \stackrel{\text{def}}{=} \neg(\neg\tau_1 \vee \neg\tau_2)$, $\tau_1 \setminus \tau_2 \stackrel{\text{def}}{=} \tau_1 \wedge \neg\tau_2$, $\mathbb{1} \stackrel{\text{def}}{=} \neg\mathbb{0}$, and likewise for type frames. We refer to b, $\times$, and $\to$ as *type constructors* and to $\vee$, $\wedge$, $\neg$, and $\setminus$ as *type connectives*.

The contractivity condition is crucial because it removes ill-formed types such as $\tau = \tau \vee \tau$ (which does not carry any information about the set denoted by the type) or $\tau = \neg\tau$ (which cannot represent any set). It also ensures that the binary relation $\triangleright \subseteq \mathcal{T}_\tau^2$ defined by $\tau_1 \vee \tau_2 \triangleright \tau_i$, $\neg\tau \triangleright \tau$ is Noetherian (that is, strongly normalizing). This gives an induction principle on $\mathcal{T}_\tau$ that we will use without any further reference to the relation.[6] The same applies to type frames. Regularity is only necessary to ensure the decidability of the subtyping relation.

The semantics of the new types and connectives is given in terms of the subtyping relation: union and intersection are, respectively, the least upper bound and the greatest lower bound of the relation, while $\mathbb{0}$ and $\mathbb{1}$ are the extrema of the lattice. Therefore, to give meaning to this extension, we extend the subtyping relation of Section 3. We come here to the limits of the syntactic approach: not only is giving inference rules for set-theoretic types hard, but it also yields a system that is hardly intelligible. Therefore we follow the semantic subtyping approach of Frisch et al. [2008]: we give an interpretation of types as sets and then use this interpretation to define the subtyping relation in terms of set containment. We would like to view a type as the set of the values that have that type. However, values cannot be used directly to define the interpretation because of a problem of circularity. Indeed, in a higher-order language, values include well-typed λ-abstractions; hence to know which values inhabit a type —and thus define the interpretation— we need to have already defined the type system (to type λ-abstractions, in particular their bodies), which depends on the subtyping relation, which in turn depends on the interpretation of types. To break this circularity, types are instead interpreted as subsets of an *interpretation domain*, written $\mathcal{D}$ and defined below.

Definition 4.2 (Interpretation domain). *The* interpretation domain $\mathcal{D}$ *is produced inductively by the following grammar*

$$\mathcal{D} \ni d ::= c^L \mid (d, d)^L \mid \{(d, d_\Omega), \ldots, (d, d_\Omega)\}^L \qquad\qquad d_\Omega ::= d \mid \Omega$$

where L ranges over $\mathcal{P}_{\text{fin}}(\mathcal{V}^\alpha \cup \mathcal{V}^X)$ (i.e., on finite sets of variables), $c \in \mathcal{C}$, and Ω is a symbol not in $\mathcal{C}$.

The elements of $\mathcal{D}$ correspond, intuitively, to the results of evaluating expressions. Expressions can produce constants or pairs of results, so we include both in $\mathcal{D}$. In a higher-order language, the result of a computation can also be a function. Functions are represented by finite relations of the form $\{(d^1, d^1_\Omega), \ldots, (d^n, d^n_\Omega)\}$, where Ω (which is a constant not in $\mathcal{D}$) can appear in second components to signify that the function fails (i.e., evaluation is stuck) on the corresponding input.

[6] The induction principle derived from the relation "$\triangleright$" states that we can use induction on type connectives but not on type constructors. This is well-founded because contractivity ensures that there are finitely many type connectives between two type constructors. For instances of applications of this principle, see Definition B.3 and the proof of Proposition B.10 in the appendix. All formal details can be found in Appendix B.

The restriction to *finite* relations is standard in semantic subtyping [Frisch et al. 2008]: intuitively, one wants the domain to represent all function values; but the use of infinite relations is not possible for cardinality reasons (since $\mathcal{D}$ cannot contain its function space), therefore we include in $\mathcal{D}$ all finite approximations of the computable functions[7] in $\mathcal{D}$, which reproduces the construction of Scott's domains. Finally, the elements of $\mathcal{D}$ are tagged by finite sets of type variables. As explained later, these tags are used to define the set-theoretic interpretation of type variables. In particular, we write $\mathsf{tags}(d)$ for the outermost set of variables labeling d, that is, $\mathsf{tags}(c^L) = \mathsf{tags}((d_1, d_2)^L) = \mathsf{tags}(\{(d_1, d_1'), \ldots, (d_n, d_n')\}^L) = L$. The next step is to define the interpretation of types into subsets of $\mathcal{D}$. We do it for type frames and, thus, for static types as well.

Definition 4.3 (Set-theoretic interpretation). *We define the* set-theoretic interpretation *of type frames* $[\![\cdot]\!] : \mathcal{T}_T \to \mathcal{P}(\mathcal{D})$ *as follows:*

$$
\begin{aligned}
[\![\alpha]\!] &= \{\, d \mid \alpha \in \mathsf{tags}(d) \,\} & [\![T_1 \vee T_2]\!] &= [\![T_1]\!] \cup [\![T_2]\!] \\
[\![X]\!] &= \{\, d \mid X \in \mathsf{tags}(d) \,\} & [\![\neg T]\!] &= \mathcal{D} \setminus [\![T]\!] \\
[\![b]\!] &= \{\, c^L \mid c \in \mathbb{B}(b) \,\} & [\![\mathbb{0}]\!] &= \varnothing \\
[\![T_1 \times T_2]\!] &= \{\, (d_1, d_2)^L \mid d_1 \in [\![T_1]\!] \wedge d_2 \in [\![T_2]\!] \,\} \\
[\![T_1 \to T_2]\!] &= \{\, \{(d_1, d_1'), \ldots, (d_n, d_n')\}^L \mid \forall i.\, d_i \in [\![T_1]\!] \implies d_i' \in [\![T_2]\!] \,\}
\end{aligned}
$$

Strictly speaking the definition above is for inductive types. The reader will find in the appendix (Definition B.3) a formal definition that handles the coinductive definition of types and such that the equalities given in Definition 4.3 hold.

The interpretation of type connectives in semantic subtyping is mandatory: the interpretation of a union type is the union of the interpretations, negation is set-theoretic complementation, and $\mathbb{0}$ is the empty set. The interpretation of type constructors, instead, is not *a priori* fixed: it depends on the characteristics of the language we want to use the types for. This dependence is hardly visible in the interpretation of basic and product types: for basic types, we assume that a function $\mathbb{B}(\cdot)$ maps each basic type to a set of constants, while products are interpreted as Cartesian products.

The interpretation of arrow types instead is more open-ended and has a more important impact on the definition of the subtyping relation. In particular, in Definition 4.3 the arrow type $T_1 \to T_2$ is interpreted as the set of (finite) graphs that map elements in T_1 only to elements in T_2. For instance, $\mathsf{Int} \to \mathsf{Bool}$ contains all the functions that when applied to an integer either diverge or return a Boolean value; $\mathsf{Int} \to \mathbb{0}$ is the set of all functions that diverge on integer arguments (if they do not diverge, they must return a value in the empty set, which is impossible); $\mathbb{0} \to \mathbb{1}$ is the set of all functions. The type systems assigns a type to an expression only if the expression returns values only in that type; this implies that all expressions of the empty type $\mathbb{0}$ are diverging. This particular interpretation of function spaces fits languages that are: (1) *non-deterministic*: since the definition does not prevent the interpretation of a function space to contain a relation with two pairs (d, d_1) and (d, d_2) with $d_1 \neq d_2$; (2) *non-terminating* since the definition does not force a relation in $[\![T_1 \to T_2]\!]$ to have as first projection the whole $[\![T_1]\!]$; (3) with *overloaded functions*: since it does not make the two types $(T_1 \vee T_2) \to (T_1' \wedge T_2')$ and $(T_1 \to T_1') \wedge (T_2 \to T_2')$ equivalent (see Castagna [2005, §4.5] for details); and (4) *strict*: since the interpretation identifies divergence and type emptiness (see Petrucciani et al. [2018, §5] for a thorough discussion of this point). Languages with different characteristics may then require a different interpretation for arrows.

Finally, notice that the elements of $\mathcal{D}$ are labeled by finite sets of variables and that the interpretation of a variable is the set of all the elements it tags. This is a technique proposed by Gesbert et al. [2015] to let type variables range over arbitrary subsets of $\mathcal{D}$, implementing the idea of convex model defined by Castagna and Xu [2011]. We refer the reader to the cited papers for more details.

[7] A computable function f can be approximated by the set of finite graph functions g such that $\forall x.\, g(x)\Downarrow \implies g(x) = f(x)$.

Definition 4.4 (Subtyping). *The* subtyping *relation* $\leq_T$ *between type frames is defined by*

$$T_1 \leq_T T_2 \stackrel{\text{def}}{\iff} [\![T_1]\!] \subseteq [\![T_2]\!]$$

We write $\simeq_T$ *for the* subtype equivalence *relation defined as* $T_1 \simeq_T T_2 \stackrel{\text{def}}{\iff} (T_1 \leq_T T_2) \wedge (T_2 \leq_T T_1)$.

The subtyping relation is decidable. We invite the reader to peruse Castagna and Frisch [2005] for a simple introduction to semantic subtyping which shows how to derive a subtyping algorithm from the set-theoretic interpretation. A detailed description of the implementation of the subtyping algorithm can be found in Castagna [2018]. For the extension of subtyping to type variables the reader can refer to Castagna and Xu [2011] and Gesbert et al. [2015].

4.1 Materialization and Subtyping for Set-Theoretic Types

In the previous section we have defined subtyping on type frames (and static types, which are a subset of type frames), but not on gradual types. This section shows how to define the two relations we need for the type system: materialization and subtyping on gradual types.

For materialization, nothing needs to change. Definition 2.2, based on discrimination and type substitutions, is equally valid here though we have changed the syntax of types. Conversely, an inductive definition would no longer work because types are defined coinductively.

As for subtyping, in Section 3 we treated ? exactly like a type variable. We might be tempted to do the same here: $\tau_1 \leq \tau_2$ would hold if and only if $T_1 \leq_T T_2$ holds, where T_i is τ_i in which every occurrence of ? is replaced by a distinguished frame variable X°. This relation is not satisfactory. Indeed, note that it would validate $? \setminus ? \leq \mathbb{0}$ (because $X^\circ \setminus X^\circ \leq_T \mathbb{0}$). As a consequence, combined with materialization, it would imply that the declarative type system would type *every* program, even fully static and nonsensical ones (it would insert casts that always fail).[8] Therefore to define subtyping, the idea of replacing ? with type variables requires some care: we must distinguish occurrences that appear below negation from those that do not.

We say that an occurrence of a frame variable X in a type frame T is *positive* if it is below an even number of negations and *negative* otherwise. A type frame is *polarized* if no frame variable has both positive and negative occurrences in it.[9] We write $\mathcal{T}_T^{\text{pol}}$ for the set of polarized type frames. The *polarized discriminations* of a gradual type are defined as $\star^{\text{pol}}(\tau) \stackrel{\text{def}}{=} \star(\tau) \cap \mathcal{T}_T^{\text{pol}}$.

Using polarized discrimination, we can define subtyping as follows.

Definition 4.5 (Subtyping on gradual types). *The* subtyping *relation* $\leq$ *between gradual types is defined by*

$$\tau_1 \leq \tau_2 \stackrel{\text{def}}{\iff} \exists T_1 \in \star^{\text{pol}}(\tau_1), T_2 \in \star^{\text{pol}}(\tau_2).\ T_1 \leq_T T_2$$

We write $\simeq$ *for the* subtype equivalence *relation defined as* $\tau_1 \simeq \tau_2 \stackrel{\text{def}}{\iff} (\tau_1 \leq \tau_2) \wedge (\tau_2 \leq \tau_1)$.

It is easy to check that this is a conservative extension of the definition in Section 3: if τ_1 and τ_2 are non-recursive and do not contain union, negation, or $\mathbb{0}$, then $\tau_1 \leq \tau_2$ holds if and only if it can be derived by those inductive rules.

This definition of subtyping could be computationally problematic because of the existential quantification. However, it turns out that we do not need to check every discrimination. It is enough to use the discrimination in which just two frame variables appear (thus eliminating the existential quantification): one (say, X^1) to replace all positive occurrences of ? and another (say, X^0) for all negative ones. Given τ, we denote this discrimination as $\tau^\oplus$. The following result holds.

Proposition 4.6. *Let* τ_1 *and* τ_2 *be gradual types. Then,* $\tau_1 \leq \tau_2$ *holds if and only if* $\tau_1^\oplus \leq_T \tau_2^\oplus$.

[8] This is because any type could then be converted to any other type: for example, $\mathsf{Int} \leq \mathsf{Int} \setminus (? \setminus ?) \preccurlyeq \mathsf{Int} \setminus (\mathsf{Int} \setminus ?) \leq \mathbb{0} \leq \mathsf{Bool}$.
[9] The notion of polarized type frame is not directly related to the polarity of blame labels.

This only holds for subtyping: for materialization, we must consider discriminations using more variables to replace positive occurrences of ? (to allow, for instance, $? \to ? \preccurlyeq \mathsf{Int} \to \mathsf{Bool}$).

This result proves not only that subtyping on gradual types is decidable, but also that it reduces in linear time to subtyping on static types (clearly, $\tau^{\oplus}$ can be computed from τ in linear time).

Note that we have defined positive and negative occurrences based solely on negation. They do not coincide with covariant and contravariant occurrences: in $X \to Y$, X is contravariant but positive; in $(\neg X) \to Y$, it is covariant but negative. However, using variance instead of polarity to define $\star^{\mathsf{pol}}$ and $\tau^{\oplus}$ gives exactly the same relation (we elaborate on this in Appendix B.4 and B.5.)

The following result shows that we can commute subtyping and materialization so that materialization always occurs first, which is useful for type inference.

Proposition 4.7. *If $\tau_1 \leq \tau_2 \preccurlyeq \tau_3$, then there exists a τ_2' such that $\tau_1 \preccurlyeq \tau_2' \leq \tau_3$.*

4.2 Cast Calculus

We extend the cast language of Section 2 to support set-theoretic types. Expressions and typing rules remain as in Section 2.2, except that we use the new definition of gradual types for casts, annotations and type applications, and that we add the typing rule [Subsume] as in §3.1.1.

The operational semantics must be redefined insofar as it depends on the syntax of types. The first definition we extend is that of *grounding*. The idea is the same as in §2.2.3: to compute an intermediate type between two types that are in the materialization relation. However, in §2.2.3 one of these two types was always ? for non-trivial materializations (so that [Collapse] and [Blame] could then eliminate it); but now, because of type connectives, both endpoints may be different from ?. For example, the cast $\langle (\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool}) \xRightarrow{p} (\mathsf{Int} \to \mathsf{Int}) \wedge ? \rangle$ makes a transition between $\mathsf{Bool} \to \mathsf{Bool}$ and ?, which can be decomposed by first transitioning to the intermediate type $? \to ?$, as done in Section 2. The intermediate type for this cast would therefore be $(\mathsf{Int} \to \mathsf{Int}) \wedge (? \to ?)$ and the endpoint $(\mathsf{Int} \to \mathsf{Int}) \wedge ?$. The intuition to generalize this idea is to apply the grounding operation of Section 2 recursively under type connectives, as formalized in the following definition.

Definition 4.8 (Grounding and Relative Ground Types). *For all types $\tau, \tau' \in \mathcal{T}_\tau$ such that $\tau' \preccurlyeq \tau$, we define the* grounding *of τ with respect to τ', noted τ/τ', as follows:*

$$
\begin{array}{rcl@{\qquad}rcl}
(\tau_1 \vee \tau_2)/(\tau_1' \vee \tau_2') & = & (\tau_1/\tau_1') \vee (\tau_2/\tau_2') & \neg\tau/\neg\tau' & = & \neg(\tau/\tau') \\
(\tau_1 \vee \tau_2)/? & = & (\tau_1/?) \vee (\tau_2/?) & \neg\tau/? & = & \neg(\tau/?) \\
(\tau_1 \to \tau_2)/? & = & ? \to ? & (\tau_1 \times \tau_2)/? & = & ? \times ? \\
b/? & = & b & \mathbb{0}/? & = & \mathbb{0} \\
\alpha/? & = & \alpha & \tau/\tau' & = & \tau' \qquad \textit{otherwise}
\end{array}
$$

A type τ is ground with respect to τ' *if and only if* $\tau/\tau' = \tau$.

Note that $\tau' \preccurlyeq \tau$ is a precondition to computing τ/τ'. Therefore to ease the presentation any further reference to τ/τ' will implicitly imply that $\tau' \preccurlyeq \tau$.

In Section 2, *ground types* are types ρ such that $\rho/? = \rho$. They are "skeletons" of types whose only information is the top-level constructor. The values of the form $V\langle \rho \xRightarrow{p} ? \rangle$ record the essence of the loss of information induced by materialization. We extend this definition to match the new definition of grounding by saying that a type τ is *ground* with respect to τ' if $\tau/\tau' = \tau$. Then, the expressions of the form $V\langle \tau \xRightarrow{p} \tau' \rangle$ are values whenever τ is ground with respect to τ'. Intuitively, casts of this form *lose information* about the top-level constructors of a type: an example is the cast $\langle (\mathsf{Int} \to \mathsf{Int}) \wedge (? \to ?) \xRightarrow{p} (\mathsf{Int} \to \mathsf{Int}) \wedge ? \rangle$, where we lose information about the $? \to ?$ part, which becomes ?. Once again, this kind of cast records the essence of this loss.

Cast Reductions.

$$
\begin{array}{rrcll}
[\textsc{ExpandL}] & V\langle \tau_1 \xRightarrow{p} \tau_2\rangle & \hookrightarrow & V\langle \tau_1 \xRightarrow{p} \tau_1/\tau_2\rangle\langle \tau_1/\tau_2 \xRightarrow{p} \tau_2\rangle & \text{if } \tau_1/\tau_2 \neq \tau_1,\ \tau_1/\tau_2 \neq \tau_2 \\
[\textsc{ExpandR}] & V\langle \tau_1 \xRightarrow{p} \tau_2\rangle & \hookrightarrow & V\langle \tau_1 \xRightarrow{p} \tau_2/\tau_1\rangle\langle \tau_2/\tau_1 \xRightarrow{p} \tau_2\rangle & \text{if } \tau_2/\tau_1 \neq \tau_1,\ \tau_2/\tau_1 \neq \tau_2 \\
[\textsc{CastId}] & V\langle \tau \xRightarrow{p} \tau\rangle & \hookrightarrow & V & \\
[\textsc{Collapse}] & V\langle \tau_1 \xRightarrow{p} \tau_2\rangle\langle \tau_1' \xRightarrow{q} \tau_2'\rangle & \hookrightarrow & V & \text{if } \tau_1 \leq \tau_2' \\
 & & & & \text{with } \tau_2'/\tau_1' = \tau_2' \text{ and } (\tau_1/\tau_2 = \tau_1 \text{ or } \tau_2/\tau_1 = \tau_1) \\
[\textsc{Blame}] & V\langle \tau_1 \xRightarrow{p} \tau_2\rangle\langle \tau_1' \xRightarrow{q} \tau_2'\rangle & \hookrightarrow & \texttt{blame}\ q & \text{if } \tau_1 \not\leq \tau_2' \\
 & & & & \text{with } \tau_2'/\tau_1' = \tau_2' \text{ and } (\tau_1/\tau_2 = \tau_1 \text{ or } \tau_2/\tau_1 = \tau_1) \\
[\textsc{UpSimpl}] & V\langle \tau_1 \xRightarrow{p} \tau_2\rangle\langle \tau_1' \xRightarrow{q} \tau_2'\rangle & \hookrightarrow & V\langle \tau_1 \xRightarrow{p} \tau_2\rangle & \text{if } \tau_2 \leq \tau_2',\quad \tau_1/\tau_2 = \tau_2,\ \tau_2'/\tau_1' = \tau_2' \\
[\textsc{UpBlame}] & V\langle \tau_1 \xRightarrow{p} \tau_2\rangle\langle \tau_1' \xRightarrow{q} \tau_2'\rangle & \hookrightarrow & \texttt{blame}\ q & \text{if } \tau_2 \not\leq \tau_2',\quad \tau_1/\tau_2 = \tau_2,\ \tau_2'/\tau_1' = \tau_2' \\
[\textsc{UnboxSimpl}] & V\langle \tau_1 \xRightarrow{p} \tau_2\rangle & \hookrightarrow & V & \text{if } \mathrm{type}(V) \leq \tau_2,\quad \tau_2/\tau_1 = \tau_2,\ V \text{ is unboxed} \\
[\textsc{UnboxBlame}] & V\langle \tau_1 \xRightarrow{p} \tau_2\rangle & \hookrightarrow & \texttt{blame}\ p & \text{if } \mathrm{type}(V) \not\leq \tau_2,\quad \tau_2/\tau_1 = \tau_2,\ V \text{ is unboxed}
\end{array}
$$

Standard Reductions.

$$
\begin{array}{rrcll}
[\textsc{CastApp}] & V\langle \tau \xRightarrow{p} \tau'\rangle V' & \hookrightarrow & (V\ V'\langle \tau_1' \xRightarrow{\bar{p}} \tau_1\rangle)\langle \tau_2 \xRightarrow{p} \tau_2'\rangle & \text{if } \tau'/\tau = \tau \text{ or } \tau/\tau' = \tau' \\
 & & & \text{where } \langle \tau \xRightarrow{p} \tau'\rangle \circ \mathrm{type}(V') = \langle \tau_1 \to \tau_2 \xRightarrow{p} \tau_1' \to \tau_2'\rangle & \\
[\textsc{CastProj}] & \pi_i\,(V\langle \tau \xRightarrow{p} \tau'\rangle) & \hookrightarrow & (\pi_i\ V)\langle \tau_i \xRightarrow{p} \tau_i'\rangle & \text{if } \tau'/\tau = \tau \text{ or } \tau/\tau' = \tau' \\
 & & & \text{where } \pi_i\,(\langle \tau \xRightarrow{p} \tau'\rangle) = \langle \tau_i \xRightarrow{p} \tau_i'\rangle & \\
[\textsc{SimplApp}] & V\langle \tau \xRightarrow{p} \tau'\rangle V' & \hookrightarrow & V\ V' & \text{if } \tau/\tau' = \tau \\
[\textsc{SimplProj}] & \pi_i\,(V\langle \tau \xRightarrow{p} \tau'\rangle) & \hookrightarrow & \pi_i\ V & \text{if } \tau/\tau' = \tau \\
[\textsc{FailApp}] & V\langle \tau \xRightarrow{p} \tau'\rangle V' & \hookrightarrow & \texttt{blame p} & \text{if } \langle \tau \xRightarrow{p} \tau'\rangle \circ \mathrm{type}(V') \text{ undef.} \\
[\textsc{FailProj}] & \pi_i\,(V\langle \tau \xRightarrow{p} \tau'\rangle) & \hookrightarrow & \texttt{blame p} & \text{if } \pi_i\,(\langle \tau \xRightarrow{p} \tau'\rangle) \text{ undef.}
\end{array}
$$

Fig. 6. Cast Reductions for the Cast Calculus

We have accounted for one kind of cast value, but we also need to update the definition of cast values of the form $V\langle \tau_1 \to \tau_2 \xRightarrow{p} \tau_1' \to \tau_2'\rangle$ (and similarly for pairs), because function types are not necessarily syntactic arrows anymore (they can be unions and/or intersections thereof). This can be done by considering the opposite case of the previous definition, that is, types such that $\tau/\tau' = \tau'$. Intuitively, a cast $\langle \tau \xRightarrow{p} \tau'\rangle$ where $\tau/\tau' = \tau'$ *does not lose or gain* information about the top-level constructors of a type: it only acts *below* the top constructors. That is, both the origin and target of such a cast have the same syntactic structure "above" constructors, the same "skeleton". For example, $\langle(\mathsf{Int} \to \mathsf{Int}) \wedge (? \to ?) \xRightarrow{p} (\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})\rangle$ is such a cast.

Putting everything together, we obtain the following new definition of values:

$$
\begin{array}{rcl}
V ::= & c \mid \lambda^{\tau \to \tau} x.\, E \mid (V, V) \mid \Lambda \vec{\alpha}.\, E & \\
 & \mid V\langle \tau_1 \xRightarrow{p} \tau_2\rangle & \text{where } \tau_1 \neq \tau_2 \text{ and where } \tau_1/\tau_2 = \tau_1 \text{ or } \tau_1/\tau_2 = \tau_2 \text{ or } \tau_2/\tau_1 = \tau_1
\end{array}
$$

We say that a value is *unboxed* if it is not of the form $V\langle \tau_1 \xRightarrow{p} \tau_2\rangle$. We next need to define a new operator "type" on values (except type abstractions) to resolve particular casts:

$$
\begin{array}{rclcrcl}
\mathrm{type}(c) & = & b_c & \qquad & \mathrm{type}(\lambda^{\tau_1 \to \tau_2} x.\, E) & = & \tau_1 \to \tau_2 \\
\mathrm{type}((V_1, V_2)) & = & \mathrm{type}(V_1) \times \mathrm{type}(V_2) & & \mathrm{type}(V\langle \tau_1 \xRightarrow{p} \tau_2\rangle) & = & \tau_2
\end{array}
$$

The semantics of the cast calculus for set-theoretic types is given in Figure 6. We only include the rules that are different from Section 2; the other rules (for let, non-cast applications, type applications, etc.) are unchanged.

The rules [ExpandL] and [ExpandR] are the immediate counterparts of the rules of the same name presented in Section 2, adapted for the new grounding operator. The other rules of this group use the information provided by the grounding operator to reduce to types that can be easily compared. For example, consider $V\langle\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle\langle\tau_1' \overset{q}{\Rightarrow} \tau_2'\rangle$. If $\tau_1 / \tau_2 = \tau_1$, then τ_1 contains all the information about type constructors which the cast lost by going into τ_2. Likewise, if $\tau_2' / \tau_1' = \tau_2'$, then all the information about type constructors is in τ_2', so the second cast *adds* constructor information. Therefore, to simplify the expressions, it suffices to compare τ_1 and τ_2', which is what is done in the rules [Collapse] and [Blame] (the set-theoretic counterparts of their namesakes in Section 2.2.3). The remaining rules for cast reductions follow the same idea, but handle cases that only arise because of set-theoretic types. For example, we can give a constant a dynamic type by subtyping (e.g., $\mathsf{Int} \leq \mathsf{Int} \vee ?$ implies $3 : \mathsf{Int} \vee ?$), and thus we can immediately cast the type of a constant to a more precise type, as in the expression $3\langle \mathsf{Int} \vee ? \overset{p}{\Rightarrow} \mathsf{Int} \vee (? \to ?)\rangle$. The rules [UnboxSimpl] and [UnboxBlame] handle such cases by checking if the cast can be removed. The intuition is that the dynamic part of such casts is useless since it has been introduced by subtyping.

The rules for applications and projections also need to be updated because function and product types can now be unions and intersections of arrows or products. For applications, we define a new operator, written $\circ$, which, given a function cast and the type of the argument, computes an approximation of the cast such that both its origin and target types are arrows, so that the usual rule for cast applications defined in Section 2 can be applied. More formally, the operation $\langle\tau \overset{p}{\Rightarrow} \tau'\rangle \circ \tau_v$ computes a cast $\langle\tau_1 \to \tau_2 \overset{p}{\Rightarrow} \tau_1' \to \tau_2'\rangle$ such that $\tau_v \leq \tau_1'$, $\tau_2' = \min\{\tau \mid \tau' \leq \tau_v \to \tau\}$, $\tau \leq \tau_1 \to \tau_2$, and such that the materialization relation between the two parts of the cast is preserved. This ensures that the resulting approximation is still well-typed. The definition of this operator is quite involved, so we relegate it to the appendix (see Definition B.68). The most important point of this definition is that it requires both types of the cast to be syntactically identical above their constructors, which explains the presence of the grounding condition in [CastApp]. Moreover, this operator can also be undefined in some cases, such as if the origin type of the cast is not an arrow type or if the second type is empty (e.g. $\langle(? \to ?) \wedge \neg(\mathsf{Int} \to \mathsf{Int}) \overset{p}{\Rightarrow} (\mathsf{Int} \to \mathsf{Int}) \wedge \neg(\mathsf{Int} \to \mathsf{Int})\rangle$). Such ill-formed casts are handled by [FailApp]. We apply the same idea to projections and define an operator, written π_i, that computes an approximation of the first or second component of a cast between two product types. This yields the rules [CastProj] and [FailProj]. The two remaining rules, [SimplApp] and [SimplProj], handle cases that only appear due to the presence of set-theoretic types. For instance, it is now possible to apply (or project) a value that has a dynamic type: $V\langle(\mathsf{Int} \to \mathsf{Int}) \wedge (? \to ?) \overset{p}{\Rightarrow} (\mathsf{Int} \to \mathsf{Int}) \wedge ?\rangle V'$. Here, by subtyping, the function has both type $\mathsf{Int} \to \mathsf{Int}$ and $?$, so it can be applied but it is also dynamic. We show that such casts are unnecessary and can be harmlessly removed; the rules [SimplApp] and [SimplProj] do just that.

We next state the usual type soundness lemmas and theorems for this cast calculus.

Lemma 4.9 (Progress). *For every term E such that $\varnothing \vdash E : \forall\vec{\alpha}.\tau$, either there exists a value V such that $E = V$, or there exists a term E' such that $E \hookrightarrow E'$, or there exists a label p such that $E \hookrightarrow \texttt{blame}\ p$.*

Lemma 4.10 (Subject Reduction). *For all terms E, E' and every context Γ, if $\Gamma \vdash E : \forall\vec{\alpha}.\tau$ and $E \hookrightarrow E'$, then $\Gamma \vdash E' : \forall\vec{\alpha}.\tau$.*

Theorem 4.11 (Soundness). *For every term E such that $\varnothing \vdash E : \forall\vec{\alpha}.\tau$, either there exists a value V such that $E \hookrightarrow^* V$, or there exists a label p such that $E \hookrightarrow^* \texttt{blame}\ p$, or E diverges.*

Another important result for our calculus is *Blame Safety*, introduced by Wadler and Findler [2009], which guarantees that the statically typed part of a program cannot be blamed. In our

system, recall that the typing rules that we presented in Section 2.2 enforce the correspondence between the polarity of the label of a cast and the direction of materialization. That is, we only have casts of the form $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle$ where $\tau' \preccurlyeq \tau$ (i.e., $\tau \leq_n \tau'$) for a negative p and $\tau \preccurlyeq \tau'$ (i.e., $\tau' \leq_n \tau$) for a positive p. Since all this information is encoded in the typing rules, blame safety is a corollary of Lemma 4.10, and can be stated without resorting to positive and negative subtyping:

Corollary 4.12 (Blame Safety). *For every term E such that $\varnothing \vdash E : \forall \vec{\alpha}.\tau$, and every blame label ℓ, $E \not\rightarrow^* \texttt{blame}\ \bar{\ell}$.*

Lastly, an important aspect of the cast language defined in this section is that it is a conservative extension of the cast calculus defined in Section 3; this justifies the choice of the reduction rules. Denoting by Sub the system defined in Section 3 and by Set the system defined in this section, there is a strong bisimulation relation between Set and Sub, as stated by the following result.

Theorem 4.13 (Conservativity). *For every term E such that $\varnothing \vdash_{SUB} E : \tau$, $E \hookrightarrow_{\text{Sub}} E' \iff E \hookrightarrow_{\text{Set}} E'$ and $E \hookrightarrow_{\text{Sub}} \texttt{blame}\ p \iff E \hookrightarrow_{\text{Set}} \texttt{blame}\ p$.*

4.3 Type Inference

To add set-theoretic types to the source language, we do not need to change the syntax, except, of course, by allowing set-theoretic types in annotations. The typing rules remain as in Section 2.1, plus the rule [Subsume] from Section 3.1 which now uses the subtyping relation of Definition 4.4; likewise for compilation, which is the same as in §2.2.4 plus a rule for subsumption that acts as the identity on the compiled expressions. Type inference requires adaptation, though. In Section 2.3, we have described it for the system without subtyping. That description was intended to be extended here; this motivated some design choices, such as the use of subtyping constraints. Now we describe what must be changed to adapt the system to set-theoretic types.

4.3.1 Type Constraints and Solutions. We keep the same definition for type constraints except, of course, for the different definition of types. However, the conditions for a type substitution θ to be a solution of a constraint D in Δ must be changed: subtyping constraints now require subtyping instead of equality. So we write $\theta \Vdash_\Delta D$ when:

- for every $(t_1 \mathrel{\dot{\leq}} t_2) \in D$, we have $t_1\theta \leq t_2\theta$;
- for every $(\tau \mathrel{\dot{\preccurlyeq}} \alpha) \in D$, we have $\tau\theta \preccurlyeq \alpha\theta$ and, for all $\beta \in \text{var}(\tau)$, $\beta\theta$ is a static type;
- $\text{dom}(\theta) \cap \Delta = \varnothing$.

4.3.2 Type Constraint Solving. To solve type constraint sets, we replace unification with an algorithm designed for set-theoretic types and semantic subtyping.

In particular, we use the *tallying* algorithm of Castagna et al. [2015]. Given a set $\overline{t^1 \mathrel{\dot{\leq}} t^2}$ of subtyping constraints, tallying computes a finite set Θ of type substitutions such that, for all $\theta \in \Theta$ and $(t^1 \mathrel{\dot{\leq}} t^2) \in \overline{t^1 \mathrel{\dot{\leq}} t^2}$, we have $t^1\theta \leq_T t^2\theta$. The set computed by tallying can contain multiple incomparable substitutions (unlike unification, where the principal solution to the problem is a unique substitution). For example, the constraint $(\alpha \times \beta) \mathrel{\dot{\leq}} (\text{Int} \times \text{Int}) \vee (\text{Bool} \times \text{Bool})$ has two solutions, $\{\alpha \coloneqq \text{Int}, \beta \coloneqq \text{Int}\}$ and $\{\alpha \coloneqq \text{Bool}, \beta \coloneqq \text{Bool}\}$, which are not comparable. Nevertheless, the tallying algorithm of Castagna et al. [2015] is sound and complete with respect to the tallying problem (i.e., checking whether there exists a substitution solving a set $\overline{t^1 \mathrel{\dot{\leq}} t^2}$ of subtyping constraints) insofar as *the set* of substitutions computed by the algorithm is principal: any other solution is an instance of one in the set.

We want to use tallying to define an algorithm to solve type constraints. Previously, we converted materialization constraints $(\tau \mathrel{\dot{\preccurlyeq}} \alpha)$ to equality constraints $(T \doteq \alpha)$ and used unification. To do the same here, we first need to extend tallying to handle such equality constraints. This is easy to do in

our case by adding simple pre- and post-processing steps.[10] The resulting algorithm $\mathsf{tally}^{\dot{=}}_{(\cdot)}(\cdot)$ is defined in the appendix. It satisfies the following property:

$$\forall \theta \in \mathsf{tally}^{\dot{=}}_{\Delta}\Big(\overline{t^1 \mathrel{\dot{\leq}} t^2} \cup \overline{T \doteq \alpha}\Big).\ \begin{cases} \forall (t^1 \mathrel{\dot{\leq}} t^2) \in \overline{t^1 \mathrel{\dot{\leq}} t^2}.\ t^1\theta \leq_T t^2\theta \\ \forall (T \doteq \alpha) \in \overline{T \doteq \alpha}.\ T\theta = \alpha\theta \\ \mathsf{dom}(\theta) \subseteq \mathsf{var}\big(\overline{t^1 \mathrel{\dot{\leq}} t^2} \cup \overline{T \doteq \alpha}\big) \setminus \Delta \end{cases}$$

Using $\mathsf{tally}^{\dot{=}}$, we can define the version of solve for set-theoretic types following the same approach as before. However, there are two difficulties.

The main difficulty is the presence of recursive types and their behaviour with respect to materialization. Consider the recursive type defined by the equation $\tau = (? \times \tau) \vee b$, where b is some basic type. It corresponds to the type of lists of elements of type ?, terminated by a constant in b. Since recursive types in our definition are infinite regular trees (and not finite trees with explicit binders), $\tau = (? \times \tau) \vee b$ and $\tau' = (? \times ((? \times \tau') \vee b)) \vee b$ denote exactly the same type. What types can τ materialize to? Clearly, both $\tau_1 = (\mathsf{Int} \times \tau_2) \vee b$ and $\tau_2 = (\mathsf{Int} \times ((\mathsf{Bool} \times \tau_2) \vee b)) \vee b$ are possible. Indeed, ? occurs infinitely many times in τ. Materialization could in principle allow us to change each occurrence to a different type. However, since types must be regular trees, only a finite number of occurrences can be replaced with different types (otherwise, the resulting tree would not be a gradual type). While finite, this number is unbounded.

Recall that step 1 of solve picked a discrimination T_j of each τ_j such that no frame variable appeared more than once in T_j. If we consider the recursive type τ above, there is no T such that $T^\dagger = \tau$ and that T has no repeated frame variables: it would need to have infinitely many frame variables and thus be non-regular. While we will never need infinitely many variables, we do not know in advance (in this pre-processing step) how many we will need.

A solution to this would be to change the tallying algorithm so that discrimination is performed during tallying. Then, it could be done lazily, introducing as many frame variables as needed. However, this sacrifices some of the modularity of our current approach.

Currently, we give a definition where no constraint is placed on how many frame variables are used to replace ?. Of course, a sensible choice is to use different variables as much as possible except for the infinitely many occurrences of ? in a recursive loop.

There is a second difficulty. For a subtyping constraint $(t_1 \mathrel{\dot{\leq}} t_2)$, a substitution θ computed by tallying ensures $t_1\theta \leq_T t_2\theta$. However, what we want is rather $(t_1\theta)^\dagger \leq (t_2\theta)^\dagger$. This does not necessarily hold unless the type frames $t_1\theta$ and $t_2\theta$ are polarized. For example, if the constraint is $(\alpha \mathrel{\dot{\leq}} \mathbb{0})$ and the substitution is $\{\alpha \coloneqq X \setminus X\}$, we have $X \setminus X \leq_T \mathbb{0}$ but $? \setminus ? \not\leq \mathbb{0}$. We define solve so that it ensures polarization in these cases by tweaking the variable renaming step we already had.

Having described these differences, we can give the definition of the algorithm. Let D be of the form $\{ (t_i^1 \mathrel{\dot{\leq}} t_i^2) \mid i \in I \} \cup \{ (\tau_j \mathrel{\dot{\not\leq}} \alpha_j) \mid j \in J \}$: then $\mathsf{solve}_\Delta(D)$ is defined as follows.

(1) Let $\overline{T \doteq \alpha}$ be $\{ (T_j \doteq \alpha_j) \mid j \in J, \tau_j \neq \alpha_j \}$ where, for each $j \in J$, $T_j^\dagger = \tau_j$;
(2) Compute $\Theta = \mathsf{tally}^{\dot{=}}_{\Delta}\big(\{ (t_i^1 \mathrel{\dot{\leq}} t_i^2) \mid i \in I \} \cup \overline{T \doteq \alpha}\big)$;
(3) Return $\{ (\theta_0\theta_0')^\dagger|_{\mathcal{V}^\alpha} \mid \theta_0 \in \Theta \}$, where, for every $\theta_0 \in \Theta$, θ_0' is computed as follows:
   (a) $\theta_0' = \{\vec{X} \coloneqq \vec{\alpha}'\} \cup \{\vec{\alpha} \coloneqq \vec{X}'\}$
   (b) $\overline{A} = \mathsf{var}_{\dot{\not\leq}}(D)\theta_0 \cup \bigcup_{i \in I} \big(\mathsf{var}^{\pm}(t_i^1\theta_0) \cup \mathsf{var}^{\pm}(t_i^2\theta_0)\big)$
   (c) $\vec{X} = \mathcal{V}^X \cap \overline{A}$ and $\vec{\alpha} = \mathsf{var}(D) \setminus (\Delta \cup \mathsf{dom}(\theta_0) \cup \overline{A})$
   (d) $\vec{\alpha}'$ and $\vec{X}'$ are vectors of fresh variables

In step 3b, we write $\mathsf{var}^{\pm}(T)$ to denote the set of all variables (both α and X) that have both positive and negative occurrences in T. A type frame T is polarized when $\mathsf{var}^{\pm}(T) \cap \mathcal{V}^X = \varnothing$: the

[10] We rely on some properties of the constraints we generate: e.g., we never have both $(T_1 \doteq \alpha)$ and $(T_2 \doteq \alpha)$ with $T_1 \neq T_2$.

renaming substitution θ'_0 is constructed to ensure this for all type frames $t_i^1\theta_0\theta'_0$ and $t_i^2\theta_0\theta'_0$. This algorithm is sound, though not complete: if $\theta \in \text{solve}_\Delta(D)$, then $\theta \Vdash_\Delta D$.

4.3.3 Structured Constraints, Generation, and Simplification. The syntax of structured constraints can be kept unchanged except for the change in the syntax of types. Constraint generation is also unchanged. Constraint simplification still uses the same rules, but it relies on the new solve algorithm. Soundness still holds, with the same statement as Theorem 2.8.

Theorem 4.14 (Soundness of type inference). *Let $\mathcal{D}$ be a derivation of $\Gamma; \text{var}(e) \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D$. Let θ be a type substitution such that $\theta \Vdash_{\text{var}(e)} D$. Then, we have $\Gamma\theta \vdash e \rightsquigarrow (\!|e|\!)^{\mathcal{D}}_{\theta} : t\theta$.*

However, completeness no longer holds, mainly as a consequence of the possible materializations of ? in recursive types. Therefore, the first step to attempt to recover completeness for inference would be to study how to change the solve algorithm to make it complete.

Note also that type constraint solving can now produce more than one incomparable solution. So constraint simplification is non-deterministic: in the rule for let constraints, there can be multiple solutions to try. Soundness ensures that any solution will give a type and a compiled expression that are sound with respect to the declarative system.

We conclude the technical presentation of this work with a word about decidability. Althought we did not always explicitly state it, all the algorithms we presented in this paper terminate, either because we reduce them to existing typing and subtyping problems that are known to be decidable (e.g., subtyping and materialization for gradual types) or because of some obvious decreasing measure (e.g., constraint simplification). This, combined with the soundness and completeness results implies the decidability of all properties (or just the semi-decidability when, like in the case of type inference for set-theoretic polymorphic gradual types, only soundness holds).

5 RELATED WORK

The contributions of this paper include the replacement of consistency with the materialization rule and the integration of gradual typing with set-theoretic types (intersection, union, negation, recursive) and Hindley-Milner polymorphism (with inference). The integration of all of these features is novel, but prior work has studied the combination of subsets of these features.

Castagna and Lanvin [2017] study the combination of gradual typing with set-theoretic types, but without polymorphism. They employ the approach of Garcia et al. [2016] that uses abstract interpretation to guide the design of the operations on types. Compared to the work of Castagna and Lanvin [2017], the present paper adds Hindley-Milner polymorphism with type inference and gives a new operational semantics that includes blame tracking and better lines up with the prior work on gradual typing. Ortin and García [2011] also investigate the combination of intersection and union types with gradual typing, but without higher-order functions and polymorphism. Toro and Tanter [2017] introduce a new kind of union type inspired by gradual typing, that provides implicit downcasts from a union to any of its constituent types. There is some overlap in the intended use-cases of these gradual union types and our design, though there are considerable differences as well, given that our work handles polymorphism and the full range of set-theoretic types. A similar overlap exists with the work by Jafery and Dunfield [2017] who introduce gradual sum types, yet, with the same kind of limitations as Toro and Tanter [2017]. Ângelo and Florido [2018] study the combination of gradual typing and intersection types, but in a somewhat limited form, as the design does not support subtyping or the other set-theoretic types.

As discussed in the introduction, Siek and Vachharajani [2008a] showed how to do unification-based inference in a gradually typed language. Garcia and Cimini [2015] took this a step further and provide inference for Hindley-Milner polymorphism and prove that their algorithm yields

principal types. The present paper builds on this prior work and contributes the additional insight that a special-purpose constraint solver is not needed to handle gradual typing, but an off-the-shelf unification algorithm can be used in combination of some pre and post-processing of the solution. In another line of work, Rastogi et al. [2012] develop a flow-based type inference algorithm for ActionScript to facilitate type specialization and the removal of runtime checks as part of their optimizing compiler. Campora et al. [2017] improve the support for migrating from dynamic to static typing by integrating gradual typing with variational types. They define a constraint-based type inference algorithm that accounts for the combination of these two features.

The combination of gradual typing with subtyping has been studied by many authors in the context of object-oriented languages. Siek and Taha [2007] showed how to augment an object calculus with gradual typing. Their declarative type system uses consistency in the elimination rules and has a subsumption rule to support subtyping. Their algorithmic type system combines consistency and subtyping into a single relation, consistent-subtyping. Many subsequent works adapted consistent-subtyping to different settings [Bierman et al. 2014; Garcia et al. 2016; Ina and Igarashi 2011; Lehmann and Tanter 2017; Maidl et al. 2014; Swamy et al. 2014; Xie et al. 2018].

There is a long history of type inference with intersection types [Kfoury and Wells 2004; Ronchi Della Rocca 1988]. The style of type inference known as soft typing employed union types [Aiken et al. 1994; Cartwright and Fagan 1991]. The set-constraints of Aiken and Wimmers [1993] employed both intersection and union types. Our work builds on recent results by Castagna et al. [2016] regarding type inference for languages with set-theoretic types and Hindley-Milner inference. Our work extends their approach to handle gradual typing. The addition of subtyping to a language presents a significant challenge for type inference, and there is a long line of work on this problem [Aiken and Wimmers 1993; Dolan and Mycroft 2017; Fuh and Mishra 1988; Mitchell 1991; Pottier 2001]. This challenge is intertwined with that of inference with intersection and union types, as we discussed in Section 3.2.

Ours is not the first line of work that tries to attack the syntactic hegemony currently ruling the gradual types community. The first and, alas hitherto unique, other example of this is the already cited work of Garcia et al. [2016] on "Abstracting Gradual Typing" (AGT) (and its several follow-ups) which was a source of inspiration both for our work and for Castagna and Lanvin [2017]. AGT uses abstract interpretation to relate gradual types to sets of static types. This is done via two functions: a *concretization* function that maps a gradual type τ into the set of static types obtained by replacing static types for all occurrences of ? in τ; an *abstraction* function that maps a set of static types to the gradual type whose concretization best approximates the set. Like AGT, we map gradual types into sets of static types, although they are different from those obtained by concretization, since we use type variables rather than generic static types. As long as only concretization is involved, we can follow and reproduce the AGT approach in ours: (1) AGT concretizations of a type τ can be defined in our system as the set of static types to which τ can materialize; (2) this definition can then be used to give a different characterization of the AGT's consistency relation; and (3) by using that characterization we can show consistency to be decidable, define consistent subtyping, and show that the problem of deciding consistent subtyping in AGT reduces in linear time to deciding semantic subtyping.

But then it is not possible to follow the approach further since the AGT definition of the abstraction function is inherently syntactic and, thus, is unfit to handle type connectives whose definition is fundamentally of semantic nature. In other terms, we have no idea about whether —let alone how— AGT could handle set-theoretic types and this is why we had to find a new semantic characterizations of constructions that in AGT are smoothly obtained by a simple application of the abstraction function.

On the topic of gradual typing and polymorphism, there has been considerable work on explicit parametric polymorphism, in the context of System F [Ahmed et al. 2011, 2017; Igarashi et al. 2017] and Java Generics [Ina and Igarashi 2011]. The presence of first-class polymorphism, as in System F, requires considerable care in the operational semantics of a cast calculus. In contrast, the second-class polymorphism (in the sense of Harper [2006]) in this paper does not significantly impact the operational semantics because casts do not need to handle the universal type.

The operational semantics for cast calculi are informed by research on runtime contract enforcement, especially regarding blame tracking [Findler and Felleisen 2002]. There is a large body of research on contracts; the most closely related to this paper are the intersection and union contracts of Keil and Thiemann [2015] and the polymorphic contracts of Sekiyama et al. [2017].

6 FUTURE WORK

This work lays a foundation for integrating gradual typing and full set-theoretic types and, as such, it opens many new questions and issues. There are in particular two practical issues that we want to address in the near future.

The first is to address a restriction we imposed to our system: namely, that it is not possible to assign intersection types to a function. Forbidding that (other than by subsumption) was an early design choice of this work, motivated by several reasons: its absence would complicate the dynamic semantics of the cast calculus (see Castagna and Lanvin [2017], where this restriction is not present); it would make type reconstruction and constraint solving much more difficult, and it would have probably hindered completeness even for simple systems; a system without this restriction would have been interesting only if the language had a type-case construct, which we wanted to avoid for simplicity and for sticking as close as possible to ML. The drawback is that we have function types that are less expressive than they could be. For instance, as noted by one of the referees of POPL, the type deduced for `mymap` in Section 1 is not completely satisfactory insofar as it does not capture the precise correlation between input and output. As a matter of fact, the following program (which transforms lists into arrays and viceversa) would get the same type:

```
let mymap2 (condition) (f) (x : (α array | α list) & ?) =
  if condition then Array.to_list(Array.map f x) else Array.of_list(List.map f x)
```

We plan to remove this restriction in future, so as to allow the system to check that (the unannotated version of) `mymap` has the type

$$\texttt{Bool} \to (\alpha \to \beta) \to \texttt{(}\ \texttt{(}\alpha\ \texttt{array \& ?)} \to \beta\ \texttt{array)} \ \&\ \texttt{(}\alpha\ \texttt{list \& ?)} \to \beta\ \texttt{list)}\ \texttt{)}$$

and that the new `mymap2` function has instead type

$$\texttt{Bool} \to (\alpha \to \beta) \to \texttt{(}\ \texttt{(}\alpha\ \texttt{array \& ?)} \to \beta\ \texttt{list)} \ \&\ \texttt{(}\alpha\ \texttt{list \& ?)} \to \beta\ \texttt{array)}\ \texttt{)}$$

two types where the correlation between the input and the output is more precisely described. In the long term not only we would like to check the types above, but also we plan to develop flow analyses that are able to infer such types for code without any type annotation.

The second practical issue we want to address is the implementation of the cast calculus. While it is still subject of a lively debate whether the insertion of casts significantly penalizes performances or not (see Takikawa et al. [2016] vs. Bauman et al. [2017]), it is clear that a naive implementation of the semantics of Figure 6 would be impractical. Therefore, we plan to study how to improve the performance of the compiled code. For that we will follow a two-pronged approach: on the one hand, we will try to define abstract machines and suitable restrictions of the cast calculus with set-theoretic types to target performance; on the other hand, we plan to use the fact that the declarative semantics of our gradual language provides a choice of different compilation strategies (corresponding to different ways of using the [Materialize] rule) that can be selected according

to some code analysis. We hope that by coupling the two we can achieve important performance gains in the compiled code.

7 CONCLUSIONS

The original goal of this work was to combine polymorphic gradual typing and set-theoretic types. We soon realized that the task was hard, because the systems were intrinsically different: gradual typing is of syntactic nature ("?" is a syntactic placeholder), while set-theoretic types rely on a semantic-based definition of subtyping. To overcome this discrepancy, the only feasible option seemed to be to give a semantic-oriented interpretation of gradual types: dealing syntactically with set-theoretic types is unfeasible. This had to be done from scratch, since all existing formalizations of gradual typing were essentially syntax-based, even the remarkable AGT approach of Garcia et al. [2016]: although it gives an interpretation of gradual types via a "concretization" function, it relies on an "abstraction" function whose definition is syntax-based.

The solution we found to this impasse was to give a semantic interpretation of gradual types indirectly, by mapping them into sets of types that already had a semantic interpretation, namely those of Castagna and Xu [2011]. Switching to a more semantic-oriented formalization makes all the chickens come home to roost. We realized that gradual typing, which was hitherto blurred in the typing rules, could be neatly perceived and captured by a subsumption-like rule using the preorder on types that we refer to as materialization. We also realized that the materialization preorder was orthogonal to the much more common preorder on types that is subtyping and that, therefore, the two preorders could be coupled without much interference (but a lot of interplay).

More than that: when, for pedagogical purposes, we studied a restricted version of our system (no set-theoretic types and no subtyping, that is, the system of Section 2) we realized that the restriction of materialization to non set-theoretic types yielded a well-known relation with many names (precision, less-or-equally-informative, and, ouch, naive subtyping). While the relation was well known, it had never been singled out in a dedicated, structural rule of the type system. We did so, and thereby we demonstrated how adding the [Materialize] rule alone is enough to endow a declarative type system with graduality. We believe that this declarative formulation is a valuable contribution to the understanding of gradual typing and complements the algorithmic systems on which previous work has focused. As an example, materialization gives a new meaning to the cast calculus: its expressions encode the proofs of the declarative systems, and casts, in particular, spot the places where [Materialize] was used. Casts thus satisfy much stronger invariants than by using consistency, allowing for a simpler statement of blame safety.

That said, it is not all a bed of roses. While materialization may enlighten the cast calculus by a previously unseen logical meaning, to define its reduction rules we had to go back to the down-and-dirty syntax of types, which is not so easy (as witnessed by the 80-page appendix). Nevertheless, we believe that our declarative formalization makes graduality more intelligible and that our work raises new questions and opens fresh, unforeseen perspectives such as: what is the logical meaning of gradual types, what is a complete inference system for gradual set-theoretic types, what is a denotational semantics of the cast calculus and could it be used to simplify, revise, and, above all, understand the operational one, how can all of this be transposed to real-world programming languages. We plan to explore all these issues in future work.

ACKNOWLEDGMENTS

We wish to thank the anonymous POPL reviewers for their detailed comments. This work was partially supported by a Google PhD Fellowship Program for Victor Lanvin and is partially based upon work supported by the National Science Foundation under Grant No. 1518844 and 1763922.

REFERENCES

Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 2011. Blame for all. *ACM SIGPLAN Notices* 46, 1 (2011), 201–214.

Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for Free for Free: Parametricity, With and Without Types. In *International Conference on Functional Programming (ICFP)*.

Alexander Aiken and Edward L. Wimmers. 1993. Type Inclusion Constraints and Type Inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*. ACM, New York, NY, USA, 31–41. https://doi.org/10.1145/165180.165188

Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. 1994. Soft typing with conditional types. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 163–173.

Pedro Ângelo and Mário Florido. 2018. Gradual Intersection Types. In *Workshop on Intersection Types and Related Systems*.

Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. 2017. Sound Gradual Typing: Only Mostly Dead. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 54 (Oct. 2017), 24 pages. https://doi.org/10.1145/3133878

Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Lecture Notes in Computer Science, Vol. 8586. Springer Berlin Heidelberg, 257–281.

John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2017. Migrating Gradual Types. *Proc. ACM Program. Lang.* 2, POPL (Dec. 2017), 15:1–15:29.

Robert Cartwright and Mike Fagan. 1991. Soft typing. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 278–292.

Giuseppe Castagna. 2005. Semantic subtyping: challenges, perspectives, and open problems. In *ICTCS 2005, Italian Conference on Theoretical Computer Science (Lecture Notes in Computer Science)*. Springer, 1–20.

Giuseppe Castagna. 2018. Covariance and Contravariance: a fresh look at an old issue (a primer in advanced type systems for learning functional programmers). (2018). First version: 02/2013, last revision: 09/2018. Unpublished manuscript.

Giuseppe Castagna and Alain Frisch. 2005. A gentle introduction to semantic subtyping. In Proceedings of *PPDP '05, the 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, pages 198-208, ACM Press (full version) and *ICALP '05, 32nd International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science n. 3580, pages 30-34, Springer (summary). Lisboa, Portugal. Joint ICALP-PPDP keynote talk.

Giuseppe Castagna and Victor Lanvin. 2017. Gradual Typing with Union and Intersection Types. *Proc. ACM Program. Lang.* 1, ICFP '17, Article 41 (Sept. 2017).

Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, and Pietro Abate. 2015. Polymorphic Functions with Set-Theoretic Types. Part 2: Local Type Inference and Type Reconstruction. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL '15)*. ACM, 289–302.

Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyen. 2016. Set-theoretic Types for Polymorphic Variants. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, New York, NY, USA, 378–391. https://doi.org/10.1145/2951913.2951928

Giuseppe Castagna and Zhiwu Xu. 2011. Set-theoretic Foundation of Parametric Polymorphism and Subtyping. In *ICFP '11: 16th ACM-SIGPLAN International Conference on Functional Programming*. 94–106.

Bruno Courcelle. 1983. Fundamental properties of infinite trees. *Theoretical Computer Science* 25 (1983), 95–169.

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, Subtyping, and Type Inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 60–72. https://doi.org/10.1145/3009837.3009882

Robert Bruce Findler and Matthias Felleisen. 2002. *Contracts for Higher-Order Functions*. Technical Report NU-CCS-02-05. Northeastern University.

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic Subtyping: dealing set-theoretically with function, union, intersection, and negation types. *J. ACM* 55, 4 (2008), 1–64.

You-Chin Fuh and Prateek Mishra. 1988. Type inference with subtypes. In *ESOP '88*, H. Ganzinger (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 94–114.

Ronald Garcia. 2013. Calculating Threesomes, with Blame. In *ICFP '13: Proceedings of the International Conference on Functional Programming*.

Ronald Garcia and Matteo Cimini. 2015. Principal Type Schemes for Gradual Programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, 303–315.

Ronald Garcia, Alison M Clark, and Éric Tanter. 2016. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, 429–442.

Nils Gesbert, Pierre Genevès, and Nabil Layaïda. 2015. A Logical Approach to Deciding Semantic Subtyping. *ACM Trans. Program. Lang. Syst.* 38, 1 (2015), 3. https://doi.org/10.1145/2812805

Robert Harper. 2006. *Programming Languages: Theory and Practice.* Carnegie Mellon University. Available on the web: http://fpl.cs.depaul.edu/jriely/547/extras/online.pdf.

Fritz Henglein. 1994. Dynamic typing: syntax and proof theory. *Science of Computer Programming* 22, 3 (June 1994), 197–230.

Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. 2000. Regular Expression Types for XML. In *ICFP '00 (SIGPLAN Notices)*, Vol. 35(9). http://www.cis.upenn.edu/~hahosoya/papers/regsub.ps

Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017. On Polymorphic Gradual Typing. In *International Conference on Functional Programming (ICFP)*. ACM.

Lintaro Ina and Atsushi Igarashi. 2011. Gradual typing for generics. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications (OOPSLA '11)*.

Khurram A. Jafery and Joshua Dunfield. 2017. Sums of Uncertainty: Refinements go gradual. In *Symposium on Principles of Programming Languages (POPL)*.

Matthias Keil and Peter Thiemann. 2015. Blame Assignment for Higher-order Contracts with Intersection and Union. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 375–386.

Assaf J. Kfoury and Joe B. Wells. 2004. Principality and type inference for intersection types using expansion variables. *Theoretical Computer Science* 311, 1 (2004), 1 – 70.

Nicolás Lehmann and Éric Tanter. 2017. Gradual Refinement Types. In *Symposium on Principles of Programming Languages (POPL)*.

André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. 2014. Typed Lua: An Optional Type System for Lua. In *Proceedings of the Workshop on Dynamic Languages and Applications (Dyla'14)*. ACM, New York, NY, USA, Article 3, 10 pages.

Alberto Martelli and Ugo Montanari. 1982. An Efficient Unification Algorithm. *ACM Trans. Program. Lang. Syst.* 4, 2 (1982), 258–282.

John C. Mitchell. 1991. Type inference with simple subtypes. *Journal of Functional Programming* 1, 3 (1991), 245–285. https://doi.org/10.1017/S0956796800000113

Francisco Ortin and Miguel García. 2011. Union and intersection types to support both dynamic and static typing. *Inform. Process. Lett.* 111, 6 (2011), 278 – 286. https://doi.org/10.1016/j.ipl.2010.12.006

Tommaso Petrucciani, Giuseppe Castagna, Davide Ancona, and Elena Zucca. 2018. *Semantic subtyping for non-strict languages.* Technical Report. https://arxiv.org/abs/1810.05555.

François Pottier. 2001. Simplifying subtyping constraints: a theory. *Inf. Comput.* 170, 2 (2001), 153–183.

François Pottier and Didier Rémy. 2005. The essence of ML type inference. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press, Chapter 10, 389–489.

Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. 2012. The ins and outs of gradual type inference. In *Symposium on Principles of Programming Languages (POPL)*. 481–494.

Simona Ronchi Della Rocca. 1988. Principal type scheme and unification for intersection type discipline. *Theor. Comput. Sci.* 59, 1-2 (1988), 181–209.

Taro Sekiyama, Atsushi Igarashi, and Michael Greenberg. 2017. Polymorphic Manifest Contracts, Revised and Resolved. *ACM Trans. Program. Lang. Syst.* 39, 1 (Feb. 2017), 3:1–3:36.

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Proceedings of Scheme and Functional Programming Workshop*. ACM, 81–92.

Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *European Conference on Object-Oriented Programming (LCNS)*, Vol. 4609. 2–27.

Jeremy G. Siek, Peter Thiemann, and Philip Wadler. 2015a. Blame and coercion: together again for the first time. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 425–435.

Jeremy G. Siek and Manish Vachharajani. 2008a. Gradual typing with unification-based inference. In *Proceedings of the 2008 Symposium on Dynamic languages*. ACM, 7.

Jeremy G. Siek and Manish Vachharajani. 2008b. *Gradual Typing with Unification-based Inference.* Technical Report CU-CS-1039-08. University of Colorado at Boulder.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015b. Refined criteria for gradual typing. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Nikhil Swamy, Cédric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. 2014. Gradual Typing Embedded Securely in JavaScript. In *ACM Conference on Principles of Programming Languages (POPL)*.

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual Typing Dead?. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, 456–468. https://doi.org/10.1145/2837614.2837630

Matías Toro and Éric Tanter. 2017. A Gradual Interpretation of Union Types. In *Proceedings of the 24th Static Analysis Symposium (SAS 2017) (Lecture Notes in Computer Science)*, Vol. 10422. Springer-Verlag, New York City, NY, USA, 382–404.

Philip Wadler and Robert Bruce Findler. 2009. Well-typed programs can't be blamed. In *European Symposium on Programming*. Springer, 1–16.

Mitchell Wand. 1987. A simple algorithm and proof for type inference. *Fundamenta Informaticae* 10 (1987), 115–122.

Ningning Xie, Xuan Bi, and Bruno C. d. S. Oliveira. 2018. Consistent Subtyping for All. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 3–30.