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), 112 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 the appendix.

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.

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

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

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

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

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

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

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

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

in annotations) can only be generalized at the level of a let binding by explicitly specifying them in 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.

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

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

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:* $\tau_1 \preccurlyeq \tau_2 \iff \exists T \in \star(\tau_1), \theta : \mathcal{V}^X \to \mathcal{T}_\tau . T\theta = \tau_2$.

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 $\text{dom}(\theta)$ to denote the set of variables for which θ is not the identity (i.e., $\text{dom}(\theta) = \{X \mid X\theta \neq X\}$).

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 $\tau' \preccurlyeq \tau$.*

[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

In other words, the relationship between our new declarative approach (i.e., with the [Materialize] 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 $\mathsf{let}\ \alpha\ z = \lambda x\colon \alpha.\,\lambda y.\,x\ \mathsf{in}\ z\,42$, 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 $\lambda x\colon ?.\,x + 1$ 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

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

type τ (rules [Cast$^{\oplus}$] and [Cast$^{\ominus}$], respectively). Blame safety usually involves two subtyping 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 $(t_1 \mathrel{\dot\leq} t_2)$ 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 $(t_1 \mathrel{\dot\leq} t_2) \in D$, 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 $(t_1 \mathrel{\dot\leq} t_2)$ forces the substitution to unify t_1 and t_2. We use $\mathrel{\dot\leq}$ 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 $\mathrm{var}(e_1) = \mathrm{var}(\langle\!\langle e_1\colon \alpha\rangle\!\rangle)\setminus\{\alpha\}$.

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

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

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

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

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

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 $(x \mathrel{\dot{\preccurlyeq}} \alpha)$. 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 $(x \mathrel{\dot{\preccurlyeq}} \alpha)$ 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 $(\alpha \mathrel{\dot{\leq}} \alpha\theta_1)$ and $(\alpha\theta_1 \mathrel{\dot{\leq}} \alpha)$. 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 $\alpha \dot{\leq} t$ 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: $\{(\alpha \dot{\leq} \mathsf{Bool}\times\mathbb{1}), (\alpha \dot{\leq} \mathbb{1}\times\mathsf{Int}), (\mathsf{Int} \dot{\leq} \beta), (\alpha \dot{\leq} \beta)\}$, 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 $\alpha \dot{\leq} t_1$ and $\alpha \dot{\leq} t_2$, 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 $s_1 \dot{\leq} \beta$ and $s_2 \dot{\leq} \beta$ 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 $(\alpha \times \beta) \mathrel{\dot{\leq}} (\mathsf{Int} \times \mathsf{Int}) \vee (\mathsf{Bool} \times \mathsf{Bool})$ has two solutions, $\{\alpha := \mathsf{Int}, \beta := \mathsf{Int}\}$ and $\{\alpha := \mathsf{Bool}, \beta := \mathsf{Bool}\}$, 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:

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

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

(1) Let $\overline{T \mathrel{\dot{=}} \alpha}$ be $\{ (T_j \mathrel{\dot{=}} \alpha_j) \mid j \in J, \tau_j \neq \alpha_j \}$ where, for each $j \in J$, $T_j^\dagger = \tau_j$;
(2) Compute $\Theta = \mathsf{tally}^{\dot{=}}_{\Delta}\big(\{ (t_i^1 \mathrel{\dot{\leq}} t_i^2) \mid i \in I \} \cup \overline{T \mathrel{\dot{=}} \alpha}\big)$;
(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) $\overline{A} = \mathsf{var}_{\dot{\preccurlyeq}}(D)\theta_0 \cup \bigcup_{i \in I} \big(\mathsf{var}^{\pm}(t_i^1\theta_0) \cup \mathsf{var}^{\pm}(t_i^2\theta_0)\big)$
  (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 $(T_1 \mathrel{\dot{=}} \alpha)$ and $(T_2 \mathrel{\dot{=}} \alpha)$ 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

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

and that the new `mymap2` function has instead type

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

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.

A GRADUAL TYPING FOR HINDLEY-MILNER SYSTEMS: RESULTS

A.1 Type System of the Source Language

We use the symbol $\sharp$ throughout to denote disjointness of sets (usually, sets of type or frame variables). We write $A \sharp B$ for $A \cap B = \varnothing$, when A and B are sets. When multiple sets appear on the left- or on the right-hand side of $\sharp$ it is intended that their union should be disjoint from what is on the other side. When a type, a type scheme, a type environment, or an expression appears, we take the set of its type variables ($\mathsf{var}(\cdot)$). For type frames, we take both type and frame variables. For vectors, we take their elements (or the variables in their elements). When a substitution θ appears, we take $\mathsf{dom}(\theta) \cup \mathsf{var}(\theta)$. We write $\sharp\,\{A_1, \ldots, A_n\}$ to mean that $A_1, \ldots, A_n$ are pairwise disjoint.

We write $\mathcal{V}$ for $\mathcal{V}^\alpha \cup \mathcal{V}^X$. We use the metavariable A to range over it.

We say that a type substitution θ is *static* if it maps type variables to static types. When $\overline{\alpha}$ is a set of variables, we say that θ is *static on* $\overline{\alpha}$, and we write $\mathsf{static}(\theta, \overline{\alpha})$, to mean that $\alpha\theta$ is static for every $\alpha \in \overline{\alpha}$.

We show a weakening property which will be needed in the proof of completeness of type inference. First, we give some preliminary definitions and results.

Given two type schemes $S_1 = \forall\vec{\alpha}_1.\,\tau_1$ and $S_2 = \forall\vec{\alpha}_2.\,\tau_2$, we write $S_1 \sqsubseteq S_2$ when, for every instance $\tau_2\{\vec{\alpha}_2 := \vec{t}_2\}$ of S_2, there exists an instance $\tau_1\{\vec{\alpha}_1 := \vec{t}_1\}$ such that $\tau_1\{\vec{\alpha}_1 := \vec{t}_1\} \preccurlyeq \tau_2\{\vec{\alpha}_2 := \vec{t}_2\}$. We extend this definition to type environments: when Γ_1 and Γ_2 are two environments with the same domain, we write $\Gamma_1 \sqsubseteq \Gamma_2$ when, for every $x \in \mathsf{dom}(\Gamma_1)$, $\Gamma_1(x) \sqsubseteq \Gamma_2(x)$.

We have the following results.

Lemma A.1. *Let $S = \forall\vec{\alpha}.\,\tau$. The following hold:*

- *for every instance $\tau\{\vec{\alpha} := \vec{t}\}$ of S, $\mathsf{var}(S) \subseteq \mathsf{var}(\tau\{\vec{\alpha} := \vec{t}\})$;*
- *there exists an instance $\tau\{\vec{\alpha} := \vec{t}\}$ of S such that $\mathsf{var}(S) = \mathsf{var}(\tau\{\vec{\alpha} := \vec{t}\})$;*

Proof. For the first point, just observe that $\mathsf{var}(\tau\{\vec{\alpha} := \vec{t}\}) \subseteq \mathsf{var}(\tau) \setminus \vec{\alpha} = \mathsf{var}(S)$.
For the second point, we take any instance in which $\vec{t}$ is a vector of ground types. □

Lemma A.2. *If $\tau_1 \preccurlyeq \tau_2$, then $\mathsf{var}(\tau_1) \subseteq \mathsf{var}(\tau_2)$.*

Proof. Since $\tau_1 \preccurlyeq \tau_2$, we have $T_1\theta = \tau_2$ with T_1 such that $T_1^\dagger = \tau_1$ and with $\theta : \mathcal{V}^X \to \mathcal{T}_\tau$. Since θ only maps frame variables, every type variable $\alpha \in \mathsf{var}(\tau_1)$, which occurs in T_1, must also occur in $T_1\theta$. □

Lemma A.3. *If $S_1 \sqsubseteq S_2$, then $\mathsf{var}(S_1) \subseteq \mathsf{var}(S_2)$. If $\Gamma_1 \sqsubseteq \Gamma_2$, then $\mathsf{var}(\Gamma_1) \subseteq \mathsf{var}(\Gamma_2)$.*

Proof. Let $S_1 = \forall\vec{\alpha}_1.\,\tau_1$ and $S_2 = \forall\vec{\alpha}_2.\,\tau_2$ be such that $S_1 \sqsubseteq S_2$. By Lemma A.1, we can find an instance $\tau_2\{\vec{\alpha}_2 := \vec{t}_2\}$ of S_2 such that $\mathsf{var}(\tau_2\{\vec{\alpha}_2 := \vec{t}_2\}) = \mathsf{var}(S_2)$. By definition of $S_1 \sqsubseteq S_2$, there exists an instance $\tau_1\{\vec{\alpha}_1 := \vec{t}_1\}$ of S_1 such that $\tau_1\{\vec{\alpha}_1 := \vec{t}_1\} \preccurlyeq \tau_2\{\vec{\alpha}_2 := \vec{t}_2\}$. By Lemma A.1, we have $\mathsf{var}(S_1) \subseteq \mathsf{var}(\tau_1\{\vec{\alpha}_1 := \vec{t}_1\})$. By Lemma A.2, we have $\mathsf{var}(\tau_1\{\vec{\alpha}_1 := \vec{t}_1\}) \subseteq \mathsf{var}(\tau_2\{\vec{\alpha}_2 := \vec{t}_2\})$. Hence, $\mathsf{var}(S_1) \subseteq \mathsf{var}(S_2)$.
The result on type environments is a straightforward corollary. □

The following weakening lemma holds.

Lemma A.4.

$$\left.\begin{array}{l}\Gamma_2 \vdash e : \tau\\ \Gamma_1 \sqsubseteq \Gamma_2\end{array}\right\} \implies \Gamma_1 \vdash e : \tau$$

Proof. By induction on the derivation of $\Gamma_2 \vdash e : \tau$ and by case on the last rule applied.

Case: [Var]
We have $e = x$. By inversion of [Var], we have:

$$\Gamma_2(x) = \forall \vec{\alpha}_2.\, \tau_2$$
$$\tau = \tau_2\{\vec{\alpha}_2 := \vec{t}_2\}$$

By definition of $\Gamma_1 \sqsubseteq \Gamma_2$, we have $\Gamma_1(x) \sqsubseteq \Gamma_2(x)$. Let $\forall \vec{\alpha}_1.\, \tau_1$ be $\Gamma_1(x)$. Then we can find an instance $\tau_1\{\vec{\alpha}_1 := \vec{t}_1\}$ of $\Gamma_1(x)$ such that $\tau_1\{\vec{\alpha}_1 := \vec{t}_1\} \preccurlyeq \tau$. We have:

$\Gamma_1 \vdash x : \tau_1\{\vec{\alpha}_1 := \vec{t}_1\}$	by [Var]
$\Gamma_1 \vdash x : \tau$	by [Materialize]

Case: [Const] Straightforward.
Case: [Abstr], [AAbstr], [App], [Pair], [Proj], [Materialize]
By direct application of the induction hypothesis.
For [Abstr] and [AAbstr], note that, for every τ, $\tau \sqsubseteq \tau$ and therefore $(\Gamma_1, x : \tau) \sqsubseteq (\Gamma_2, x : \tau)$.
Case: [Let]
We have derived $\Gamma_2 \vdash (\text{let } \vec{\alpha}\ x = e_1 \text{ in } e_2) : \tau$ from the premises:

$$\Gamma_2 \vdash e_1 : \tau_1$$
$$\Gamma_2, x : \forall \vec{\alpha}, \vec{\beta}.\, \tau_1 \vdash e_2 : \tau$$
$$\vec{\alpha}, \vec{\beta} \sharp \Gamma_2 \text{ and } \vec{\beta} \sharp \Gamma_2$$

We have

(1)	$\Gamma_1 \vdash e_1 : \tau_1$	by IH
	$\Gamma_1, x : \forall \vec{\alpha}, \vec{\beta}.\, \tau_1 \sqsubseteq \Gamma_2, x : \forall \vec{\alpha}, \vec{\beta}.\, \tau_1$	because $S \sqsubseteq S$ for every type scheme S
(2)	$\Gamma_1, x : \forall \vec{\alpha}, \vec{\beta}.\, \tau_1 \vdash e_2 : \tau$	by IH
(3)	$\text{var}(\Gamma_1) \subseteq \text{var}(\Gamma_2)$	by Lemma A.3
(4)	$\vec{\alpha}, \vec{\beta} \sharp \Gamma_1$	from (3)
	$\Gamma_1 \vdash (\text{let } \vec{\alpha}\ x = e_1 \text{ in } e_2) : \tau$	from (1), (2), (4), and $\vec{\beta} \sharp e_1$

□

Proposition A.5. *For every types τ_1, τ_2,*

$$\tau_1 \sim \tau_2 \iff \exists \tau_0 \text{ such that } \begin{cases} \tau_1 \preccurlyeq \tau_0 \\ \tau_2 \preccurlyeq \tau_0 \end{cases}$$

Proof. • We first prove the implication from left to right. Let us first remark that if $\tau_1 = ?$ then it is sufficient to take $\tau_0 = \tau_2$ since $\tau_1 = ? \preccurlyeq \tau_2$ and $\tau_2 \preccurlyeq \tau_2$. Similarly, if $\tau_2 = ?$ then we can take $\tau_0 = \tau_1$ for the same reason. We then prove the result by induction on τ_1, for the cases where both τ_1 and τ_2 are not ?.
– $\tau_1 = \alpha$. Then we necessarily have $\tau_2 = \alpha$. Thus we can take $\tau_0 = \tau_1 = \tau_2$.
– $\tau_1 = b$. Then we necessarily have $\tau_2 = b$. Thus we can take $\tau_0 = \tau_1 = \tau_2$.
– $\tau_1 = \sigma_1 \times \sigma_1'$. By consistency, we have $\tau_2 = \sigma_2 \times \sigma_2'$ where $\sigma_1 \sim \sigma_2$ and $\sigma_1' \sim \sigma_2'$. Thus, by induction, there exists two types σ_0 and σ_0' such that $\sigma_i \preccurlyeq \sigma_0$ and $\sigma_i' \preccurlyeq \sigma_0'$ for every $i \in \{1, 2\}$. We then deduce that $\sigma_i \times \sigma_i' \preccurlyeq \sigma_0 \times \sigma_0'$ for every $i \in \{1, 2\}$, hence the result.
– $\tau_1 = \sigma_1 \to \sigma_1'$. This case is proved in the same way as the previous one.

$$\begin{array}{rrcl} \text{Gradual types} & \mathcal{T}_\tau \ni \tau & ::= & ? \mid b \mid \tau \to \tau \\ \text{Expressions} & e & ::= & x \mid c \mid \lambda x\colon \tau.\, e \mid e\ e \end{array}$$

$$[\textsc{Var}]\ \frac{}{\Gamma \vdash x\colon \Gamma(x)} \qquad [\textsc{Const}]\ \frac{}{\Gamma \vdash c\colon b_c} \qquad [\textsc{App}]\ \frac{\Gamma \vdash e_1\colon \tau' \to \tau \qquad \Gamma \vdash e_2\colon \tau'}{\Gamma \vdash e_1\ e_2\colon \tau}$$

$$[\textsc{AAbstr}]\ \frac{\Gamma, x\colon \tau' \vdash e\colon \tau}{\Gamma \vdash (\lambda x\colon \tau'.\, e)\colon \tau' \to \tau} \qquad [\textsc{Materialize}]\ \frac{\Gamma \vdash e\colon \tau'}{\Gamma \vdash e\colon \tau}\ \tau' \preccurlyeq \tau$$

Fig. 7. Monomorphic restriction of the implicative fragment

- We now prove the other direction. We first remark that, as before, if $\tau_1 = ?$ or $\tau_2 = ?$ then the result is immediate. Thus, we reason by induction over τ_0 for the cases where both τ_1 and τ_2 are not ?.
  - $\tau_0 = ?$. Necessarily, $\tau_1 = \tau_2 = ?$ in this case, which is forbidden.
  - $\tau_0 = \alpha$. Then necessarily $\tau_1 = \tau_2 = \alpha$, and the result is immediate.
  - $\tau_0 = b$. Same as before.
  - $\tau_0 = \sigma_0 \to \sigma_0'$. By materialization, we have $\tau_i = \sigma_i \to \sigma_i'$ where $\sigma_i \preccurlyeq \sigma_0$ and $\sigma_i' \preccurlyeq \sigma_0'$ for every $i \in \{1, 2\}$. By induction, we then have $\sigma_1 \sim \sigma_2$ and $\sigma_1' \sim \sigma_2'$ and the result follows by definition of consistency.
  - $\tau_0 = \sigma_0 \times \sigma_0'$. This case is proved in the same way as the previous one.

□

Denoting by $\vdash_{ST}$ the typing judgments obtained in the system of Siek and Taha [2006], and by $\vdash_1$ the typing judgments obtained in the monomorphic implicative restriction of our system shown in Figure 7, we have the following result:

Proposition A.6. *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 $\tau' \preccurlyeq \tau$.*

Proof. We prove the two results by induction over e and the last rule used in the typing derivation.

- $\Gamma \vdash_{ST} e : \tau$.
  - [GVar] $\Gamma \vdash_{ST} x : \tau$. By hypothesis, $\Gamma(x) = \tau$. We immediately conclude by rule [Var] that $\Gamma \vdash_1 x : \tau$.
  - [GConst]. $\Gamma \vdash_{ST} c : \tau$. By hypothesis, $\Delta c : \tau$, which is equivalent to $b_c = \tau$ in our system. Thus, we conclude by rule [Const] that $\Gamma \vdash_1 c : \tau$.
  - [GLam]. This rule is identical to [AAbstr].
  - [GApp1] $\Gamma \vdash_{ST} e_1\ e_2 : ?$, when $\Gamma \vdash_{ST} e_1 : ?$ and $\Gamma \vdash_{ST} e_2 : \tau_2$. By induction, we have $\Gamma \vdash_1 e_1 : ?$ and $\Gamma \vdash_1 e_2 : \tau_2$. Then, by [Materialize] we obtain $\Gamma \vdash_1 e_1 : \tau_2 \to ?$ since $? \preccurlyeq \tau_2 \to ?$. We can then apply rule [App] to deduce that $\Gamma \vdash_1 e_1\ e_2 : ?$.
  - [GApp2] $\Gamma \vdash_{ST} e_1\ e_2 : \tau'$, when $\Gamma \vdash_{ST} e_1 : \tau \to \tau'$ and $\Gamma \vdash_{ST} e_2 : \tau_2$ and $\tau \sim \tau_2$. By induction, we have $\Gamma \vdash_1 e_1 : \tau \to \tau'$ and $\Gamma \vdash_1 e_2 : \tau_2$. Moreover, by Proposition A.5, we know that there exists a type τ_0 such that $\tau \preccurlyeq \tau_0$ and $\tau_2 \preccurlyeq \tau_0$. Therefore, by applying [Materialize] we deduce that $\Gamma \vdash_1 e_1 : \tau_0 \to \tau'$ and $\Gamma \vdash_1 e_2 : \tau_0$. We then conclude by applying [App] to deduce that $\Gamma \vdash_{ST} e_1\ e_2 : \tau'$.
- $\Gamma \vdash_1 e : \tau$.
  - [Var] $\Gamma \vdash_1 x : \tau$. By hypothesis, $\Gamma(x) = \tau$. Thus we can immediately conclude by rule [GVar].

$$\begin{array}{rrcl} \text{Gradual types} & \mathcal{T}_\tau \ni \tau & ::= & ? \mid \alpha \mid b \mid \tau \to \tau \\ \text{Expressions} & e & ::= & x \mid c \mid \lambda x.\, e \mid \lambda x\colon \tau.\, e \mid e\ e \end{array}$$

$$[\textsc{Var}]\ \frac{}{\Gamma \vdash x\colon \Gamma(x)} \qquad [\textsc{Const}]\ \frac{}{\Gamma \vdash c\colon b_c} \qquad [\textsc{App}]\ \frac{\Gamma \vdash e_1\colon \tau' \to \tau \qquad \Gamma \vdash e_2\colon \tau'}{\Gamma \vdash e_1\ e_2\colon \tau}$$

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

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

Fig. 8. Polymorphic restriction of the implicative fragment

- [Const] $\Gamma \vdash_1 c : b_c$. $b_c = \Delta c$ in the system of Siek and Taha [2006]. Thus we can conclude by rule [GConst].
- [App] $\Gamma \vdash_1 e_1\ e_2 : \tau$ when $\Gamma \vdash_1 e_1 : \tau' \to \tau$ and $\Gamma \vdash_1 e_2 : \tau'$. By induction, we have $\Gamma \vdash_{ST} e_1 : \tau_1$ and $\Gamma \vdash_{ST} e_2 : \tau_2$ where $\tau_1 \preccurlyeq \tau' \to \tau$ and $\tau_2 \preccurlyeq \tau'$. Then, if $\tau_1 = ?$ then we deduce by rule [GApp1] that $\Gamma \vdash_{ST} e_1\ e_2 : ?$ and $? \preccurlyeq \tau$, hence the result. Otherwise, we have $\tau_1 = \sigma' \to \sigma$ where $\sigma' \preccurlyeq \tau'$ and $\sigma \preccurlyeq \tau$. Since $\tau_2 \preccurlyeq \tau'$, we deduce by Proposition A.5 that $\sigma' \sim \tau_2$. Therefore, we deduce by rule [GApp2] that $\Gamma \vdash_{ST} e_1\ e_2 : \sigma$ and the result follows from the fact that $\sigma \preccurlyeq \tau$.
- [AAbstr] $\Gamma \vdash_1 \lambda x : \tau'.e : \tau' \to \tau$ when $\Gamma, x : \tau' \vdash_1 e : \tau$. By induction, $\Gamma, x : \tau' \vdash_{ST} e : \sigma$ where $\sigma \preccurlyeq \tau$. Thus, by rule [GLam], we obtain $\Gamma \vdash_{ST} \lambda x : \tau'.e : \tau' \to \sigma$, and the result follows from the fact that $\tau' \to \sigma \preccurlyeq \tau' \to \tau$.
- [Materialize] $\Gamma \vdash_1 e : \tau$ when $\Gamma \vdash_1 e : \tau'$ and $\tau' \preccurlyeq \tau$. By induction, we have $\Gamma \vdash_{ST} e : \tau''$ where $\tau'' \preccurlyeq \tau'$. By transitivity of the materialization, $\tau'' \preccurlyeq \tau$ and the result follows.

□

Now, denoting by $\vdash_{GC}$ the typing judgments obtained in the system of Garcia and Cimini [2015], and by $\vdash_{\to}$ the typing judgments obtained in the polymorphic implicative restriction of our system shown in Figure 8, we have the following result:

Proposition A.7. *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* $\tau' \preccurlyeq \tau$.

Proof. The proof is mostly identical to the proof of Proposition A.6. The main difference is the presence of the rule for untyped lambda-abstractions [Abstr], which is however identical to the rule [Uλ] of Garcia and Cimini [2015]. □

A.2 Compilation

Figures 9 and 10 give, respectively, the typing rules of the cast language and the compilation rules.

Proposition A.8. *If* $\Gamma \vdash e\colon \tau$, *then there exists an* E *such that* $\Gamma \vdash e \rightsquigarrow E\colon \tau$.

Proof. By induction on the derivation of $\Gamma \vdash e\colon \tau$. □

Proposition A.9. *If* $\Gamma \vdash e \rightsquigarrow E\colon \tau$, *then* $\Gamma \vdash e\colon \tau$ *and* $\Gamma \vdash E\colon \tau$.

Proof. By induction on the derivation of $\Gamma \vdash e \rightsquigarrow E\colon \tau$ and by case on last rule applied. Showing $\Gamma \vdash e\colon \tau$ is trivial.

$$[\textsc{Var}]\ \frac{}{\Gamma \vdash x : \forall\vec{\alpha}.\,\tau}\ \Gamma(x) = \forall\vec{\alpha}.\,\tau \qquad [\textsc{Const}]\ \frac{}{\Gamma \vdash c : b_c}$$

$$[\textsc{Abstr}]\ \frac{\Gamma, x : \tau' \vdash E : \tau}{\Gamma \vdash (\lambda^{\tau'\to\tau} x.\,E) : \tau' \to \tau} \qquad [\textsc{App}]\ \frac{\Gamma \vdash E_1 : \tau' \to \tau \qquad \Gamma \vdash E_2 : \tau'}{\Gamma \vdash E_1\ E_2 : \tau}$$

$$[\textsc{Pair}]\ \frac{\Gamma \vdash E_1 : \tau_1 \qquad \Gamma \vdash E_2 : \tau_2}{\Gamma \vdash (E_1, E_2) : \tau_1 \times \tau_2} \qquad [\textsc{Proj}]\ \frac{\Gamma \vdash E : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\ E : \tau_i}$$

$$[\textsc{Let}]\ \frac{\Gamma \vdash E_1 : \forall\vec{\alpha}.\,\tau_1 \qquad \Gamma, x : \forall\vec{\alpha}.\,\tau_1 \vdash E_2 : \tau}{\Gamma \vdash (\text{let } x = E_1 \text{ in } E_2) : \tau}$$

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

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

Fig. 9. Typing rules of the cast language.

$$[\textsc{Var}]\ \frac{}{\Gamma \vdash x \rightsquigarrow x\ [\vec{t}] : \tau\{\vec{\alpha} := \vec{t}\}}\ \Gamma(x) = \forall\vec{\alpha}.\,\tau \qquad [\textsc{Const}]\ \frac{}{\Gamma \vdash c \rightsquigarrow c : b_c}$$

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

$$[\textsc{App}]\ \frac{\Gamma \vdash e_1 \rightsquigarrow E_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 \rightsquigarrow E_2 : \tau'}{\Gamma \vdash e_1\ e_2 \rightsquigarrow E_1\ E_2 : \tau}$$

$$[\textsc{Pair}]\ \frac{\Gamma \vdash e_1 \rightsquigarrow E_1 : \tau_1 \qquad \Gamma \vdash e_2 \rightsquigarrow E_2 : \tau_2}{\Gamma \vdash (e_1, e_2) \rightsquigarrow (E_1, E_2) : \tau_1 \times \tau_2} \qquad [\textsc{Proj}]\ \frac{\Gamma \vdash e \rightsquigarrow E : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\ e \rightsquigarrow \pi_i\ E : \tau_i}$$

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

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

Fig. 10. Compilation from the source language to the cast language.

Showing $\Gamma \vdash E : \tau$ is also straightforward. If the last rule is [Var], we use [Var] and [TApp]. If the last rule is [Const], [App], [Pair], or [Proj], we use the same rule. If is is [Abstr] or [AAbstr], we use [Abstr]. If it is [Materialize], we use [Cast$^{\oplus}$].

Finally, if the last rule is [Let], from the premise $\Gamma \vdash e_1 \rightsquigarrow E_1 : \tau_1$ we get, by IH, $\Gamma \vdash E_1 : \tau_1$. Then (since $\vec{\alpha},\vec{\beta} \sharp \Gamma$) we get $\Gamma \vdash \Lambda\vec{\alpha},\vec{\beta}.\, E_1 : \forall\vec{\alpha},\vec{\beta}.\, \tau_1$ by [TAbstr]. From the premise $\Gamma, x : \forall\vec{\alpha},\vec{\beta}.\, \tau_1 \vdash e_2 \rightsquigarrow E_2 : \tau$ we get, by IH, $\Gamma, x : \forall\vec{\alpha},\vec{\beta}.\, \tau_1 \vdash E_2 : \tau$. We apply [Let] to conclude. □

Corollary A.10. *If $\Gamma \vdash e : \tau$, then there exists an E such that $\Gamma \vdash e \rightsquigarrow E : \tau$ and $\Gamma \vdash E : \tau$.*

Proof. Corollary of Propositions A.8 and A.9. □

A.3 Type Inference

A.3.1 Type Constraint Solving. We assume that $\mathsf{unify}_{(\cdot)}(\cdot)$ satisfies the following properties:

- if $\mathsf{unify}_\Delta(\overline{T^1 \doteq T^2}) = \theta$, then $\mathsf{dom}(\theta) \subseteq \mathsf{var}(\overline{T^1 \doteq T^2}) \setminus \Delta$ and $\mathsf{var}(\theta) \subseteq \mathsf{var}(\overline{T^1 \doteq T^2}) \setminus \mathsf{dom}(\theta)$ and, for every $(T^1 \doteq T^2) \in \overline{T^1 \doteq T^2}$, we have $T^1\theta = T^2\theta$;
- if θ' is a unifier for $\overline{T^1 \doteq T^2}$ and $\mathsf{dom}(\theta') \cap \Delta = \varnothing$, then $\mathsf{unify}_\Delta(\overline{T^1 \doteq T^2}) = \theta$ and $\theta' = \theta\theta'$.

(We use $\mathsf{var}(\theta)$ to denote $\bigcup_{A \in \mathsf{dom}(\theta)} \mathsf{var}(A\theta)$, where A ranges over both type and frame variables.)

Proposition A.11. *If $\theta \in \mathsf{solve}_\Delta(D)$, then all of the following hold:*

- $\theta \Vdash_\Delta D$;
- $\mathsf{dom}(\theta) \subseteq \mathsf{var}(D)$;
- $\mathsf{var}(D)\theta \subseteq \mathsf{var}_{\dot{\preccurlyeq}}(D)\theta \cup \Delta$.

Proof. Let θ be in $\mathsf{solve}_\Delta(D)$, where $D = \{ (t_i^1 \mathrel{\dot{\leq}} t_i^2) \mid i \in I \} \cup \{ (\tau_j \mathrel{\dot{\leq}} \alpha_j) \mid j \in J \}$. Then, we have:

$$\theta = (\theta_0\theta_0')^\dagger|_{\mathcal{V}^\alpha} \qquad \theta_0 = \mathsf{unify}_\Delta(\overline{T^1 \doteq T^2}) \qquad \theta_0' = \{\vec{X} := \vec{\alpha}'\} \cup \{\vec{\alpha} := \vec{X}'\}$$

$$\overline{T^1 \doteq T^2} = \{ (t_i^1 \doteq t_i^2) \mid i \in I \} \cup \{ (T_j \doteq \alpha_j) \mid j \in J \}$$

$$\vec{X} = \mathcal{V}^X \cap \mathsf{var}_{\dot{\preccurlyeq}}(D)\theta_0 \qquad \vec{\alpha} = \mathsf{var}(D) \setminus (\Delta \cup \mathsf{dom}(\theta_0) \cup \mathsf{var}_{\dot{\preccurlyeq}}(D)\theta_0) \qquad \vec{\alpha}', \vec{X}' \text{ fresh}$$

We first prove $\theta \Vdash_\Delta D$. First, we show that, for every $i \in I$, we have $t_i^1\theta = t_i^2\theta$. Note that, since $\mathsf{var}(t_i^1) \cup \mathsf{var}(t_i^2) \subseteq \mathcal{V}^\alpha$, we have $t_i^1\theta = (t_i^1\theta_0\theta_0')^\dagger$ and $t_i^2\theta = (t_i^2\theta_0\theta_0')^\dagger$. By the properties of unification, we have $t_i^1\theta_0 = t_i^2\theta_0$. Then, we also have $t_i^1\theta_0\theta_0' = t_i^2\theta_0\theta_0'$ and finally $t_i^1\theta = t_i^2\theta$.

Now, we show that, for every $j \in J$, we have $\tau_j\theta \preccurlyeq \alpha_j\theta$. We have $\tau_j\theta = (\tau_j\theta_0\theta_0')^\dagger$ and $\alpha_j\theta = (\alpha_j\theta_0\theta_0')^\dagger$. By the properties of unification, we have $T_j\theta_0 = \alpha_j\theta_0$ and therefore $(T_j\theta_0\theta_0')^\dagger = (\alpha_j\theta_0\theta_0')^\dagger$. Therefore, we must show $(\tau_j\theta_0\theta_0')^\dagger \preccurlyeq (T_j\theta_0\theta_0')^\dagger$, which holds trivially since $\tau_j = T_j^\dagger$.

Now, we show that, for every $j \in J$ and every $\beta \in \mathsf{var}(\tau_j)$, $\beta\theta$ is a static type. Note that $\beta \in \mathsf{var}_{\dot{\preccurlyeq}}(D)$. We have $\beta\theta = (\beta\theta_0\theta_0')^\dagger$. If $\beta\theta$ were not static, there would be an $X \in \mathsf{var}(\beta\theta_0\theta_0')$: we show that this cannot happen. If there were an $X \in \mathsf{var}(\beta\theta_0\theta_0')$, then there would be an $A \in \mathcal{V}^\alpha \cup \mathcal{V}^X$ such that $A \in \mathsf{var}(\beta\theta_0)$ and $X \in \mathsf{var}(A\theta_0')$. We would have $A \in \mathsf{var}_{\dot{\preccurlyeq}}(D)\theta_0$. Therefore, if $A \in \mathcal{V}^X$, then $A \in \vec{X}$ and it would be mapped to a static type variable; if $A \in \mathcal{V}^\alpha$, then it could not be in $\mathsf{dom}(\theta_0')$, so it could not be mapped to a type containing frame variables.

Finally, we show that $\mathsf{dom}(\theta) \cap \Delta = \varnothing$. Let $\alpha \in \Delta$. We show $\alpha \notin \mathsf{dom}(\theta)$, that is, $\alpha\theta = \alpha$. We have $\alpha\theta = (\alpha\theta_0\theta_0')^\dagger$. By the properties of unification, since $\alpha \in \Delta$, we have $\alpha\theta_0 = \alpha$. We also have $\alpha\theta_0' = \alpha$ because $\alpha \notin \vec{\alpha}$.

To prove $\mathsf{dom}(\theta) \subseteq \mathsf{var}(D)$, consider $\alpha \notin \mathsf{var}(D)$. We prove $\alpha \notin \mathsf{dom}(\theta)$, that is, $\alpha\theta = \alpha$. We have $\alpha\theta = (\alpha\theta_0\theta_0')^\dagger$. By the properties of unification, since $\alpha \notin \mathsf{var}(D)$, $\alpha\theta_0 = \alpha$. Then, since $\alpha \notin \mathsf{var}(D)$, we have $\alpha \notin \vec{\alpha}$; hence, $\alpha\theta_0' = \alpha$.

To prove $\mathsf{var}(D)\theta \subseteq \mathsf{var}_{\dot{\preccurlyeq}}(D)\theta \cup \Delta$, consider an arbitrary $\alpha \in \mathsf{var}(D)\theta$. We show $\alpha \in \mathsf{var}_{\dot{\preccurlyeq}}(D)\theta \cup \Delta$. By definition of $\mathsf{var}(D)\theta$, there must exist a $\beta \in \mathsf{var}(D)$ such that $\alpha \in \mathsf{var}(\beta\theta)$. We have $\beta\theta = (\beta\theta_0\theta_0')^\dagger$. Either $\alpha \in \mathsf{var}(\beta\theta_0) \setminus \mathsf{dom}(\theta_0')$ or $\alpha \in \mathsf{var}(\theta_0')$.

- If $\alpha \in \mathsf{var}(\beta\theta_0) \setminus \mathsf{dom}(\theta_0')$, then $\alpha \in \mathsf{var}(D)$ (because $\beta \in \mathsf{var}(D)$ and because solutions of unification do not introduce new variables). Then, $\alpha \in \Delta \cup \mathsf{dom}(\theta_0) \cup \mathsf{var}_{\dot{\preccurlyeq}}(D)\theta_0$. The case $\alpha \in \mathsf{dom}(\theta_0)$ is impossible because θ_0 is idempotent. Therefore, $\alpha \in \Delta \cup \mathsf{var}_{\dot{\preccurlyeq}}(D)\theta_0$ and (since $\alpha \notin \mathsf{dom}(\theta_0')$) $\alpha \in \Delta \cup \mathsf{var}_{\dot{\preccurlyeq}}(D)\theta$.
- If $\alpha \in \mathsf{var}(\theta_0')$, then $\alpha \in \vec{X}\theta_0'$. Therefore, there exists an $X \in \mathsf{var}_{\dot{\preccurlyeq}}(D)\theta_0$ such that $\alpha \in \mathsf{var}(X\theta_0')$. Hence, $\alpha \in \mathsf{var}_{\dot{\preccurlyeq}}(D)\theta$. □

Lemma A.12. *Let $\theta : \mathcal{V}^\alpha \to \mathcal{T}_\tau$ and $\theta' : \mathcal{V} \to \mathcal{T}_T$ be such that $\forall \alpha \in \mathcal{V}^\alpha.\ (\alpha\theta')^\dagger = \alpha\theta$. Then, for every T, we have $T^\dagger\theta \preccurlyeq (T\theta')^\dagger$.*

Proof. We choose $\hat{\theta} : \mathcal{V}^\alpha \to \mathcal{T}_T$ such that:

$$\forall \alpha \in \mathcal{V}^\alpha.\ (\alpha\hat{\theta})^\dagger = \alpha\theta \qquad \mathsf{var}_X(\hat{\theta}) \sharp \mathsf{dom}(\theta'), \mathsf{var}_X(T)\ .$$

We define $\check{\theta} : \mathcal{V}^X \to \mathcal{T}_\tau$ as $\check{\theta} = \{X \coloneqq (X\theta')^\dagger\}_{X \in \mathsf{dom}(\theta')} \cup \{X \coloneqq ?\}_{X \in \mathsf{var}_X(T\hat{\theta}) \setminus \mathsf{dom}(\theta')}$.

We have $(T\hat{\theta})^\dagger = T^\dagger\theta$ because:

- for every $\alpha \in \mathsf{var}(T)$, we have $(\alpha\hat{\theta})^\dagger = \alpha\theta = \alpha^\dagger\theta$;
- for every $X \in \mathsf{var}(T)$, we have $(X\hat{\theta})^\dagger = X^\dagger = ? = ?\theta = X^\dagger\theta$.

We have $T\hat{\theta}\check{\theta} = (T\theta')^\dagger$ because:

- for every $\alpha \in \mathsf{var}(T) \cap \mathsf{dom}(\hat{\theta})$, since $\mathsf{var}_X(\hat{\theta}) \sharp \mathsf{dom}(\check{\theta})$, we have $\alpha\hat{\theta}\check{\theta} = \alpha\hat{\theta}$ and $\alpha(\hat{\theta} \cup \check{\theta}) = \alpha\hat{\theta}$;
- for every $\alpha \in \mathsf{var}(T) \setminus \mathsf{dom}(\theta)$, since $\alpha\theta = \alpha$, also $\alpha\hat{\theta} = \alpha$ and $\alpha\theta' = \alpha$: then we have $\alpha\hat{\theta}\check{\theta} = \alpha = (\alpha\theta')^\dagger$;
- for every $X \in \mathsf{var}(T) \cap \mathsf{dom}(\theta')$, we have $X\hat{\theta}\check{\theta} = X\check{\theta} = (X\theta')^\dagger$;
- for every $X \in \mathsf{var}(T) \setminus \mathsf{dom}(\theta')$, we have $X \in \mathsf{var}(T\hat{\theta}) \setminus \mathsf{dom}(\theta')$: then, $X\hat{\theta}\check{\theta} = X\check{\theta} = ? = X^\dagger = (X\theta')^\dagger$.

Therefore, we have $T\hat{\theta} \in \star(T^\dagger\theta)$ and $T\hat{\theta}\check{\theta} = (T\theta')^\dagger$ with $\check{\theta} : \mathcal{V}^X \to \mathcal{T}_\tau$: hence, $T^\dagger\theta \preccurlyeq (T\theta')^\dagger$. □

Proposition A.13. *If $\theta \Vdash_\Delta D$, then there exist two substitutions θ' and θ'' such that:*

- *$\theta' \in \mathsf{solve}_\Delta(D)$;*
- *$\mathsf{dom}(\theta'') \subseteq \mathsf{var}(\theta') \setminus \mathsf{var}(D)$;*
- *for every α, $\alpha\theta'(\theta \cup \theta'') \preccurlyeq \alpha(\theta \cup \theta'')$;*
- *for every α such that $\alpha\theta'$ is static, $\alpha\theta'(\theta \cup \theta'') \preccurlyeq \alpha(\theta \cup \theta'')$;*

Proof. Let $D = \{(t_i^1 \mathrel{\dot{\preccurlyeq}} t_i^2) \mid i \in I\} \cup \{(\tau_j \mathrel{\dot{\preccurlyeq}} \alpha_j) \mid j \in J\}$ and let $\theta : \mathcal{V}^\alpha \to \mathcal{T}_\tau$ be such that $\theta \Vdash_\Delta D$. The first step of computing $\mathsf{solve}_\Delta(D)$ is to construct

$$\overline{T^1 \doteq T^2} = \{(t_i^1 \doteq t_i^2) \mid i \in I\} \cup \{(T_j \doteq \alpha_j) \mid j \in J\}$$

with each T_j such that $T_j^\dagger = \tau_j$ and with unique frame variables.

First, we show that from θ we can obtain a substitution $\check{\theta} : \mathcal{V} \to \mathcal{T}_T$ which is a unifier for $\overline{T^1 \doteq T^2}$. For every $j \in J$, we have $\tau_j\theta \preccurlyeq \alpha_j\theta$; furthermore, θ is static on all variables of τ_j. By definition of materialization, there exist a type frame $T_j' \in \star(\tau_j\theta)$ and a substitution $\theta_j : \mathcal{V}^X \to \mathcal{T}_\tau$ such that $T_j'\theta_j = \alpha_j\theta$. In particular, we can choose $T_j' = T_j\theta$ (because $T_j\theta \in \star(\tau_j\theta)$ and because it has unique frame variables) and we can assume $\mathsf{dom}(\theta_j) = \mathsf{var}_X(T_j)$. Let $\hat{\theta} = \theta \cup \bigcup_{j \in J} \theta_j$: $\hat{\theta}$ is well-defined since the frame variables in every T_j are distinct. We choose an arbitrary frame variable $\check{X}$. Let $\check{\theta} : \mathcal{V} \to \mathcal{T}_T$ be such that $\forall A \in \mathcal{V}.\ (A\check{\theta})^\dagger = A\hat{\theta}$ and that $\mathsf{var}_X(\check{\theta}) \subseteq \{\check{X}\}$. We have $\mathsf{dom}(\check{\theta}) \cap \Delta = \varnothing$, since $\mathsf{dom}(\theta) \cap \Delta = \varnothing$, $\mathsf{dom}(\check{\theta}) \setminus \mathsf{dom}(\theta) \subseteq \mathcal{V}^X$, and $\Delta \subseteq \mathcal{V}^\alpha$. Moreover, $\check{\theta}$ is a unifier for $\overline{T^1 \doteq T^2}$.

By the properties of unification, we have $\mathsf{unify}_\Delta(\overline{T^1 \doteq T^2}) = \theta_0$ and $\check{\theta} = \theta_0\check{\theta}$.

By definition of solve, we have:

$$\theta' \in \mathsf{solve}_\Delta(D) \qquad \theta' = (\theta_0\theta_0')^\dagger|_{\mathcal{V}^\alpha} \qquad \theta_0' = \{\vec{X} \coloneqq \vec{\alpha}'\} \cup \{\vec{\alpha} \coloneqq \vec{X}'\}$$
$$\vec{X} = \mathcal{V}^X \cap \mathsf{var}_{\dot{\preccurlyeq}}(D)\theta_0 \qquad \vec{\alpha} = \mathsf{var}(D) \setminus (\Delta \cup \mathsf{dom}(\theta_0) \cup \mathsf{var}_{\dot{\preccurlyeq}}(D)\theta_0) \qquad \vec{\alpha}', \vec{X}' \text{ fresh}$$

Since $\vec{\alpha}'$ and $\vec{X}'$ are fresh, we can assume they are outside $\mathsf{dom}(\check{\theta})$ and $\mathsf{var}(\check{\theta})$.

We choose $\theta'' = \{\vec{\alpha}' \coloneqq (\vec{X}\check{\theta})^\dagger\}$. Since $\vec{\alpha}'$ is chosen fresh by solve, it is outside of $\mathsf{var}(D)$: therefore, it is in $\mathsf{var}(\theta') \setminus \mathsf{var}(D)$.

We must show:

$$\forall\alpha.\ \alpha\theta'(\theta \cup \theta'') \preccurlyeq \alpha(\theta \cup \theta'') \qquad \forall\alpha.\ \alpha\theta' \implies \alpha\theta'(\theta \cup \theta'') = \alpha(\theta \cup \theta'')$$

If $\alpha \notin \mathsf{dom}(\theta')$, the results hold trivially.

We consider the case $\alpha \in \mathsf{dom}(\theta')$. Then, we have $\alpha \notin \vec{\alpha}'$.

We have:

$$\alpha(\theta \cup \theta'') = \alpha\theta = (\alpha\check{\theta})^\dagger = (\alpha\theta_0\check{\theta})^\dagger$$

We have:

$$\begin{aligned}
\alpha\theta'(\theta \cup \theta'') &= (\alpha\theta_0\theta_0')^\dagger(\theta \cup \theta'') \\
&\preccurlyeq (\alpha\theta_0\theta_0'(\check{\theta} \cup \{\vec{\alpha}' \coloneqq \vec{X}\check{\theta}\}))^\dagger && \text{by Lemma A.12} \\
&= (\alpha\theta_0(\{\vec{X} \coloneqq \vec{\alpha}'\} \cup \{\vec{\alpha} \coloneqq \vec{X}'\})(\check{\theta} \cup \{\vec{\alpha}' \coloneqq \vec{X}\check{\theta}\}))^\dagger \\
&= (\alpha\theta_0(\check{\theta}|_{\mathsf{dom}(\check{\theta})\setminus\vec{\alpha}} \cup \{\vec{\alpha} \coloneqq \vec{X}'\}))^\dagger \\
&\preccurlyeq (\alpha\theta_0\check{\theta})^\dagger
\end{aligned}$$

If $\alpha\theta'$ is static, then $\mathsf{var}_X(\alpha\theta_0\theta_0') = \varnothing$ and therefore $\mathsf{var}(\alpha\theta_0) \sharp \vec{\alpha}$ and $\mathsf{var}_X(\alpha\theta_0) \subseteq \vec{X}$. Then:

$$\begin{aligned}
\alpha\theta'(\theta \cup \theta'') &= (\alpha\theta_0\theta_0')^\dagger(\theta \cup \theta'') \\
&= \alpha\theta_0\theta_0'(\theta \cup \theta'') \\
&= \alpha\theta_0\{\vec{X} \coloneqq \vec{\alpha}'\}(\theta \cup \{\vec{\alpha}' \coloneqq (\vec{X}\check{\theta})^\dagger\} \\
&= \alpha\theta_0(\theta \cup \{\vec{X} \coloneqq (\vec{X}\check{\theta})^\dagger\} \\
&= (\alpha\theta_0\check{\theta})^\dagger
\end{aligned}$$

□

A.3.2 Constraint Simplification. Figure 11 presents the constraint simplification rules in a form in which we track explicitly the variables we introduce and state precise freshness conditions. In a derivation $\Gamma; \Delta \vdash C \rightsquigarrow D \mid \overline{\alpha}$, the set $\overline{\alpha}$ is the set of fresh variables introduced by simplification. We will still write $\Gamma; \Delta \vdash C \rightsquigarrow D$ when we are not interested in what variables are introduced (notably, in the soundness proof).

Figure 12 defines the compilation algorithm that, given an expression e, a derivation $\mathcal{D}$ of $\Gamma; \Delta \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D$, and a substitution θ such that $\theta \Vdash_{\mathsf{var}(e)} D$, produces a cast language expression $(\!|e|\!)^{\mathcal{D}}_{\theta}$. It is defined by induction on e. For each case, we deconstruct the derivation $\mathcal{D}$ to obtain the sub-derivations used to compile the sub-expressions of e; we write the derivation in a compressed form where we collapse applications of the rules for definition, existential, and conjunctive constraints. We write $\mathcal{D} :: \Gamma; \Delta \vdash C \rightsquigarrow D$ to denote a derivation of $\Gamma; \Delta \vdash C \rightsquigarrow D$ that we name $\mathcal{D}$.

Lemma A.14 (Stability of typing under type substitution). *If $\Gamma \vdash e \rightsquigarrow E : \tau$, then, for every static type substitution θ, we have $\Gamma\theta \vdash e\theta \rightsquigarrow E\theta : \tau\theta$.*

Proof. By induction on the derivation of $\Gamma \vdash e \rightsquigarrow E : \tau$ and by case on the last rule applied.

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

$$\frac{\Gamma(x) = \forall\vec{\alpha}.\,\tau \quad \vec{\beta} \sharp \Gamma}{\Gamma;\Delta \vdash (x \mathrel{\dot\preccurlyeq} \alpha) \rightsquigarrow \{\tau\{\vec{\alpha} \coloneqq \vec{\beta}\} \mathrel{\dot\preccurlyeq} \alpha\} \mid \vec{\beta}} \qquad \frac{(\Gamma, x\colon \tau);\Delta \vdash C \rightsquigarrow D \mid \overline{\alpha}}{\Gamma;\Delta \vdash \mathsf{def}\ x\colon \tau\ \mathsf{in}\ C \rightsquigarrow D \mid \overline{\alpha}}$$

$$\frac{\Gamma;\Delta \vdash C \rightsquigarrow D \mid \overline{\alpha}}{\Gamma;\Delta \vdash (\exists\vec{\alpha}.\,C) \rightsquigarrow D \mid \overline{\alpha} \cup \vec{\alpha}}\ \vec{\alpha} \sharp \Gamma, \overline{\alpha} \qquad \frac{\Gamma;\Delta \vdash C_1 \rightsquigarrow D_1 \mid \overline{\alpha}_1 \qquad \Gamma;\Delta \vdash C_2 \rightsquigarrow D_2 \mid \overline{\alpha}_2}{\Gamma;\Delta \vdash C_1 \wedge C_2 \rightsquigarrow D_1 \cup D_2 \mid \overline{\alpha}_1 \cup \overline{\alpha}_2}\ \overline{\alpha}_1 \sharp \overline{\alpha}_2$$

$$\frac{\begin{array}{c}\Gamma;\Delta \cup \vec{\alpha} \vdash C_1 \rightsquigarrow D_1 \mid \overline{\alpha}_1 \\ (\Gamma, x\colon \forall\vec{\alpha},\vec{\beta}.\,\alpha\theta_1);\Delta \vdash C_2 \rightsquigarrow D_2 \mid \overline{\alpha}_2\end{array}}{\Gamma;\Delta \vdash \mathsf{let}\ x\colon \forall\vec{\alpha};\alpha[C_1]^{\vec{\alpha}'}.\,\alpha\ \mathsf{in}\ C_2 \rightsquigarrow D_2 \cup \mathsf{equiv}(\theta_1, D_1) \mid \overline{\alpha}}$$

$$\begin{array}{l}\theta_1 \in \mathsf{solve}_{\Delta\cup\vec{\alpha}}(D_1) \\ \vec{\alpha} \sharp \Gamma\theta_1 \\ \vec{\beta} = \mathsf{var}(\alpha\theta_1) \setminus (\mathsf{var}(\Gamma\theta_1) \cup \vec{\alpha} \cup \vec{\alpha}') \\ \sharp\ \big\{\{\alpha\}, \vec{\alpha}, \overline{\alpha}_1, \overline{\alpha}_2, (\mathsf{var}(\theta_1) \setminus \mathsf{var}(D_1))\big\} \\ \alpha, \vec{\alpha} \sharp \Gamma, \Delta \\ \overline{\alpha} = \{\alpha\} \cup \vec{\alpha} \cup \overline{\alpha}_1 \cup \overline{\alpha}_2 \cup (\mathsf{var}(\theta_1) \setminus \mathsf{var}(D_1))\end{array}$$

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

Fig. 11. Constraint simplification rules with explicit variable introduction.

Case: [Var]

	$\Gamma \vdash x \rightsquigarrow x\,[\vec{t}]\colon \tau\{\vec{\alpha} \coloneqq \vec{t}\}$	Given
	$\Gamma(x) = \forall\vec{\alpha}.\,\tau$	Given
	$(\Gamma\theta)(x) = \forall\vec{\alpha}.\,\tau\theta$	since, by α-renaming, $\vec{\alpha} \sharp \theta$
(1)	$\Gamma\theta \vdash x \rightsquigarrow x\,[\vec{t}\theta]\colon \tau\theta\{\vec{\alpha} \coloneqq \vec{t}\theta\}$	by [Var], since the $\vec{t}\theta$ are all static
(2)	$\tau\theta\{\vec{\alpha} \coloneqq \vec{t}\theta\} = \tau\{\vec{\alpha} \coloneqq \vec{t}\}\theta$	since $\vec{\alpha} \sharp \theta$, $\forall\alpha \in \mathsf{var}(\tau).\ \alpha\theta\{\vec{\alpha} \coloneqq \vec{t}\theta\} = \alpha\{\vec{\alpha} \coloneqq \vec{t}\}\theta$
	$\Gamma\theta \vdash x \rightsquigarrow x\,[\vec{t}]\theta\colon \tau\{\vec{\alpha} \coloneqq \vec{t}\}\theta$	by (1) and (2)

Case: [Const]
Straightforward, since $b_c\theta = b_c$.
Case: [Abstr], [AAbstr], [App], [Pair], [Proj]
Direct application of the induction hypothesis. For [Abstr], note that $t\theta$ is always static.
Case: [Materialize]
$\tau' \preccurlyeq \tau$ implies $\tau'\theta \preccurlyeq \tau\theta$ for any type substitution θ.
Case: [Let]

$\Gamma \vdash (\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2) \rightsquigarrow (\mathsf{let}\ x = \Lambda\vec{\alpha},\vec{\beta}.\,E_1\ \mathsf{in}\ E_2)\colon \tau$ Given

By inversion of [Let]:

(1) $\Gamma \vdash e_1 \rightsquigarrow E_1\colon \tau_1$

(2) $\Gamma, x\colon \forall\vec{\alpha},\vec{\beta}.\,\tau_1 \vdash e_2 \rightsquigarrow E_2\colon \tau$

(3) $\vec{\alpha},\vec{\beta} \sharp \Gamma$ and $\vec{\beta} \sharp e_1$

$$(\!|x|\!)^{\mathcal{D}}_{\theta} = x\,[\vec{\beta}\theta]\langle\tau\{\vec{\alpha} \coloneqq \vec{\beta}\}\theta \overset{\ell}{\Rightarrow} \alpha\theta\rangle$$
with ℓ fresh
where $\mathcal{D} = \dfrac{}{\Gamma;\Delta \vdash \langle\!\langle x\colon t\rangle\!\rangle \rightsquigarrow \{(\tau\{\vec{\alpha} \coloneqq \vec{\beta}\} \mathrel{\dot{\preccurlyeq}} \alpha), (\alpha \mathrel{\dot{\leq}} t)\}}$

$$(\!|c|\!)^{\mathcal{D}}_{\theta} = c$$

$$(\!|\lambda x.\, e|\!)^{\mathcal{D}}_{\theta} = \lambda^{(\alpha_1\to\alpha_2)\theta}x.\,(\!|e|\!)^{\mathcal{D}'}_{\theta}$$
where $\mathcal{D} = \dfrac{\mathcal{D}' :: (\Gamma, x\colon \alpha_1);\Delta \vdash \langle\!\langle e\colon \alpha_2\rangle\!\rangle \rightsquigarrow D'}{\Gamma;\Delta \vdash \langle\!\langle (\lambda x.\, e)\colon t\rangle\!\rangle \rightsquigarrow D' \cup \{(\alpha_1 \mathrel{\dot{\preccurlyeq}} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\}}$

$$(\!|\lambda x\colon \tau.\, e|\!)^{\mathcal{D}}_{\theta} = (\lambda^{(\tau\to\alpha_2)\theta}x.\,(\!|e|\!)^{\mathcal{D}'}_{\theta})\langle(\tau\to\alpha_2)\theta \overset{\ell}{\Rightarrow} (\alpha_1\to\alpha_2)\theta\rangle$$
with ℓ fresh
where $\mathcal{D} = \dfrac{\mathcal{D}' :: (\Gamma, x\colon \tau);\Delta \vdash \langle\!\langle e\colon \alpha_2\rangle\!\rangle \rightsquigarrow D'}{\Gamma;\Delta \vdash \langle\!\langle (\lambda x\colon \tau.\, e)\colon t\rangle\!\rangle \rightsquigarrow D' \cup \{(\tau \mathrel{\dot{\preccurlyeq}} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\}}$

$$(\!|e_1\ e_2|\!)^{\mathcal{D}}_{\theta} = (\!|e_1|\!)^{\mathcal{D}_1}_{\theta}\ (\!|e_2|\!)^{\mathcal{D}_2}_{\theta}$$
where $\mathcal{D} = \dfrac{\mathcal{D}_1 :: \Gamma;\Delta \vdash \langle\!\langle e_1\colon \alpha\to t\rangle\!\rangle \rightsquigarrow D_1 \qquad \mathcal{D}_2 :: \Gamma;\Delta \vdash \langle\!\langle e_2\colon \alpha\rangle\!\rangle \rightsquigarrow D_2}{\Gamma;\Delta \vdash \langle\!\langle e_1\ e_2\colon t\rangle\!\rangle \rightsquigarrow D_1 \cup D_2}$

$$(\!|(e_1, e_2)|\!)^{\mathcal{D}}_{\theta} = ((\!|e_1|\!)^{\mathcal{D}_1}_{\theta}, (\!|e_2|\!)^{\mathcal{D}_2}_{\theta})$$
where $\mathcal{D} = \dfrac{\mathcal{D}_1 :: \Gamma;\Delta \vdash \langle\!\langle e_1\colon \alpha_1\rangle\!\rangle \rightsquigarrow D_1 \qquad \mathcal{D}_2 :: \Gamma;\Delta \vdash \langle\!\langle e_2\colon \alpha_2\rangle\!\rangle \rightsquigarrow D_2}{\Gamma;\Delta \vdash \langle\!\langle (e_1, e_2)\colon t\rangle\!\rangle \rightsquigarrow D_1 \cup D_2 \cup \{\alpha_1\times\alpha_2 \mathrel{\dot{\leq}} t\}}$

$$(\!|\pi_i\ e|\!)^{\mathcal{D}}_{\theta} = \pi_i\ (\!|e|\!)^{\mathcal{D}'}_{\theta}$$
where $\mathcal{D} = \dfrac{\mathcal{D}' :: \Gamma;\Delta \vdash \langle\!\langle e\colon \alpha_1\times\alpha_2\rangle\!\rangle \rightsquigarrow D'}{\Gamma;\Delta \vdash \langle\!\langle \pi_i\ e\colon t\rangle\!\rangle \rightsquigarrow D' \cup \{\alpha_i \mathrel{\dot{\leq}} t\}}$

$$(\!|\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2|\!)^{\mathcal{D}}_{\theta} = \mathsf{let}\ x = \Lambda\vec{\alpha}_1,\vec{\beta}_1.\,(\!|e_1|\!)^{\mathcal{D}_1}_{\theta_1}\rho\theta\ \mathsf{in}\ (\!|e_2|\!)^{\mathcal{D}_2}_{\theta}$$
where $\mathcal{D} = \dfrac{\mathcal{D}_1 :: \Gamma;\Delta\cup\vec{\alpha} \vdash C_1 \rightsquigarrow D_1 \qquad \mathcal{D}_2 :: (\Gamma, x\colon \forall\vec{\alpha},\vec{\beta}.\,\alpha\theta_1);\Delta \vdash C_2 \rightsquigarrow D_2}{\Gamma;\Delta \vdash \langle\!\langle \mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2\colon t\rangle\!\rangle \rightsquigarrow D_2 \cup \mathrm{equiv}(\theta_1, D_1)}$
and $\theta_1 \in \mathrm{solve}_{\Delta\cup\vec{\alpha}}(D_1)$ $\vec{\alpha}_1,\vec{\beta}_1$ fresh $\rho = \{\vec{\alpha} \coloneqq \vec{\alpha}_1\} \cup \{\vec{\beta} \coloneqq \vec{\beta}_1\}$

Fig. 12. Algorithmic compilation.

Let $\vec{\alpha}_1$ and $\vec{\beta}_1$ be vectors of distinct variables chosen outside $\mathrm{var}(\Gamma)$, $\mathrm{var}(e_1)$, $\mathrm{dom}(\theta)$, and $\mathrm{var}(\theta)$. Let $\rho = \{\vec{\alpha} \coloneqq \vec{\alpha}_1\} \cup \{\vec{\beta} \coloneqq \vec{\beta}_1\}$.

	$\Gamma\rho \vdash e_1\rho \rightsquigarrow E_1\rho\colon \tau_1\rho$	by IH from (1), since ρ is static
(4)	$\Gamma \vdash e_1\{\vec{\alpha} \coloneqq \vec{\alpha}_1\} \rightsquigarrow E_1\rho\colon \tau_1\rho$	by (3)
(5)	$\Gamma\theta \vdash e_1\{\vec{\alpha} \coloneqq \vec{\alpha}_1\}\theta \rightsquigarrow E_1\rho\theta\colon \tau_1\rho\theta$	by IH from (4)
(6)	$\Gamma\theta, x\colon (\forall\vec{\alpha},\vec{\beta}.\,\tau_1)\theta \vdash e_2\theta \rightsquigarrow E_2\theta\colon \tau\theta$	by IH from (2)
(7)	$\Gamma\theta, x\colon (\forall\vec{\alpha}_1,\vec{\beta}_1.\,\tau_1\rho)\theta \vdash e_2\theta \rightsquigarrow E_2\theta\colon \tau\theta$	by α-renaming from (6)
(8)	$\Gamma\theta, x\colon (\forall\vec{\alpha}_1,\vec{\beta}_1.\,\tau_1\rho\theta) \vdash e_2\theta \rightsquigarrow E_2\theta\colon \tau\theta$	from (7) since $\vec{\alpha}_1,\vec{\beta}_1 \sharp \theta$
	$\Gamma\theta \vdash (\mathsf{let}\ \vec{\alpha}_1\ x = e_1\{\vec{\alpha} \coloneqq \vec{\alpha}_1\}\theta\ \mathsf{in}\ e_2\theta) \rightsquigarrow$ $(\mathsf{let}\ x = \Lambda\vec{\alpha}_1,\vec{\beta}_1.\,E_1\rho\theta\ \mathsf{in}\ E_2\theta)\colon \tau\theta$	by [LET] from (5) and (8)

This concludes the proof because let $\vec{\alpha}_1\ x = e_1\{\vec{\alpha} := \vec{\alpha}_1\}\theta$ in $e_2\theta$ and (let $\vec{\alpha}\ x = e_1$ in $e_2)\theta$ are equivalent by α-renaming, as are let $x = \Lambda\vec{\alpha}_1,\vec{\beta}_1.\, E_1\rho\theta$ in $E_2\theta$ and (let $x = \Lambda\vec{\alpha},\vec{\beta}.\, E_1$ in $E_2)\theta$. □

Lemma A.15. *Let $\mathcal{D}$ be a derivation of $\Gamma;\Delta \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D$. Then:*

- *if $e = x$, then $\Gamma(x) = \forall\vec{\alpha}.\,\tau$ and $D = \{(\tau\{\vec{\alpha} := \vec{\beta}\} \mathrel{\dot{\preccurlyeq}} \alpha), (\alpha \mathrel{\dot{\leq}} t)\}$ (for some τ, α, $\vec{\alpha}$, $\vec{\beta}$);*
- *if $e = c$, then $D = \{b_c \mathrel{\dot{\leq}} t\}$;*
- *if $e = \lambda x.\, e'$, then $\mathcal{D}$ contains a sub-derivation of $(\Gamma, x : \alpha_1);\Delta \vdash \langle\!\langle e' : \alpha_2 \rangle\!\rangle \rightsquigarrow D'$, and $D = D' \cup \{(\alpha_1 \mathrel{\dot{\preccurlyeq}} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\}$;*
- *if $e = \lambda x : \tau.\, e'$, then $\mathcal{D}$ contains a sub-derivation of $(\Gamma, x : \tau);\Delta \vdash \langle\!\langle e' : \alpha_2 \rangle\!\rangle \rightsquigarrow D'$, and $D = D' \cup \{(\tau \mathrel{\dot{\preccurlyeq}} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\}$;*
- *if $e = e_1\ e_2$, then $\mathcal{D}$ contains two sub-derivations of $\Gamma;\Delta \vdash \langle\!\langle e_1 : \alpha \to t \rangle\!\rangle \rightsquigarrow D_1$ and $\Gamma;\Delta \vdash \langle\!\langle e_2 : \alpha \rangle\!\rangle \rightsquigarrow D_2$ (for some α, D_1, and D_2), and $D = D_1 \cup D_2$;*
- *if $e = (e_1, e_2)$, then $\mathcal{D}$ contains two sub-derivations of $\Gamma;\Delta \vdash \langle\!\langle e_1 : \alpha_1 \rangle\!\rangle \rightsquigarrow D_1$ and $\Gamma;\Delta \vdash \langle\!\langle e_2 : \alpha_2 \rangle\!\rangle \rightsquigarrow D_2$ (for some α_1, α_2, D_1, and D_2), and $D = D_1 \cup D_2 \cup \{\alpha_1 \times \alpha_2 \mathrel{\dot{\leq}} t\}$;*
- *if $e = \pi_i\, e'$, then $\mathcal{D}$ contains a sub-derivation of $\Gamma;\Delta \vdash \langle\!\langle e' : \alpha_1 \times \alpha_2 \rangle\!\rangle \rightsquigarrow D'$, and $D = D' \cup \{\alpha_i \mathrel{\dot{\leq}} t\}$;*
- *if $e = (\text{let } \vec{\alpha}\ x = e_1 \text{ in } e_2)$, then $\mathcal{D}$ contains two sub-derivations of $\Gamma;\Delta \cup \vec{\alpha} \vdash \langle\!\langle e_1 : \alpha \rangle\!\rangle \rightsquigarrow D_1$ and $(\Gamma, x : \forall\vec{\alpha},\vec{\beta}.\, \alpha\theta_1);\Delta \vdash \langle\!\langle e_2 : t \rangle\!\rangle \rightsquigarrow D_2$, and the following hold:*

$$D = D_2 \cup \mathsf{equiv}(\theta_1, D_1) \qquad \theta_1 \in \mathsf{solve}_{\Delta\cup\vec{\alpha}}(D_1)$$

$$\vec{\alpha} \sharp \mathsf{var}(\Gamma\theta_1) \qquad \vec{\beta} = \mathsf{var}(\alpha\theta_1) \setminus (\mathsf{var}(\Gamma\theta_1) \cup \vec{\alpha} \cup \mathsf{var}(e_1))$$

Proof. Straightforward, since the constraint simplification rules are syntax-directed. □

Lemma A.16. *If $\Gamma;\Delta \vdash C \rightsquigarrow D$, then $\mathsf{var}(\Gamma) \cap \mathsf{var}(D) \subseteq \mathsf{var}(C) \cup \mathsf{var}_{\dot{\preccurlyeq}}(D)$.*

Proof. By induction on C (the form of C determines the derivation).

Case: $C = (t_1 \mathrel{\dot{\leq}} t_2)$ or $C = (\tau \mathrel{\dot{\preccurlyeq}} \alpha)$ We have $\mathsf{var}(D) \subseteq \mathsf{var}(C)$.

Case: $C = (\tau \mathrel{\dot{\preccurlyeq}} \alpha)$ We have $\mathsf{var}(D) \subseteq \mathsf{var}_{\dot{\preccurlyeq}}(D) \cup \{\alpha\}$ and $\alpha \in \mathsf{var}(C)$.

Case: $C = (\text{def } x : \tau \text{ in } C')$ By IH, $\mathsf{var}(\Gamma, x : \tau) \cap \mathsf{var}(D) \subseteq \mathsf{var}(C') \cup \mathsf{var}_{\dot{\preccurlyeq}}(D)$. This directly yields the result since $\mathsf{var}(C') \subseteq \mathsf{var}(C)$.

Case: $C = (\exists\vec{\alpha}.\, C')$ By IH, $\mathsf{var}(\Gamma) \cap \mathsf{var}(D) \subseteq \mathsf{var}(C') \cup \mathsf{var}_{\dot{\preccurlyeq}}(D)$. The side condition on the rule imposes $\vec{\alpha} \sharp \Gamma$. Then, $\mathsf{var}(\Gamma) \cap \mathsf{var}(D) \subseteq \mathsf{var}(C) \cup \mathsf{var}_{\dot{\preccurlyeq}}(D)$ since $\mathsf{var}(C) = \mathsf{var}(C') \setminus \vec{\alpha}$.

Case: $C = (C_1 \wedge C_2)$ By IH, for both i, $\mathsf{var}(\Gamma) \cap \mathsf{var}(D_i) \subseteq \mathsf{var}(C_i) \cup \mathsf{var}_{\dot{\preccurlyeq}}(D_i)$. This directly implies $\mathsf{var}(\Gamma) \cap \mathsf{var}(D_1 \cup D_2) \subseteq \mathsf{var}(C_1 \wedge C_2) \cup \mathsf{var}_{\dot{\preccurlyeq}}(D_1 \cup D_2)$.

Case: $C = (\text{let } x : \forall\vec{\alpha}; \alpha[C_1]^{\vec{\alpha}_1}.\, \alpha \text{ in } C_2)$ By IH,

$$\mathsf{var}(\Gamma) \cap \mathsf{var}(D_1) \subseteq \mathsf{var}(C_1) \cup \mathsf{var}_{\dot{\preccurlyeq}}(D_1)$$

$$\mathsf{var}(\Gamma, x : \forall\vec{\alpha},\vec{\beta}.\, \alpha\theta_1) \cap \mathsf{var}(D_2) \subseteq \mathsf{var}(C_2) \cup \mathsf{var}_{\dot{\preccurlyeq}}(D_2)$$

We have

$$D = D_2 \cup \mathsf{equiv}(\theta_1, D_1)$$

$$\mathsf{var}(D) = \mathsf{var}(D_2) \cup \mathsf{var}(D_1)\theta_1 \cup \mathsf{var}_{\dot{\preccurlyeq}}(D_1) \cup S \cup S\theta_1$$

$$\mathsf{var}_{\dot{\preccurlyeq}}(D) = \mathsf{var}_{\dot{\preccurlyeq}}(D_2) \cup \mathsf{var}(D_1)\theta_1 \cup \mathsf{var}_{\dot{\preccurlyeq}}(D_1)$$

$$\mathsf{var}(C) = (\mathsf{var}(C_1) \setminus (\vec{\alpha} \cup \{\alpha\})) \cup \mathsf{var}(C_2)$$

where $S = \{\, \alpha \in \mathsf{dom}(\theta_1) \mid \alpha\theta_1 \text{ static} \,\}$.
Consider an arbitrary $\beta \in \mathsf{var}(\Gamma) \cap \mathsf{var}(D)$.

Case: $\beta \in \mathrm{var}(D_2)$ Then $\beta \in \mathrm{var}(C_2) \cup \mathrm{var}_{\dot{\preccurlyeq}}(D_2)$ and hence $\beta \in \mathrm{var}(C) \cup \mathrm{var}_{\dot{\preccurlyeq}}(D)$.
Case: $\beta \in \mathrm{var}(D_1)\theta_1 \cup \mathrm{var}_{\dot{\preccurlyeq}}(D_1)$ Then $\beta \in \mathrm{var}_{\dot{\preccurlyeq}}(D)$.
Case: $\beta \in S$
Then $\beta \in \mathrm{dom}(\theta_1)$. By Proposition A.11, $\beta \in \mathrm{var}(D_1)$.
Since $\beta \in \mathrm{var}(\Gamma) \cap \mathrm{var}(D_1)$, we have $\beta \in \mathrm{var}(C_1) \cup \mathrm{var}_{\dot{\preccurlyeq}}(D_1)$. Since $\beta \in \mathrm{var}(\Gamma)$, by the side conditions of the rule we know $\beta \neq \alpha$ and $\beta \notin \vec{\alpha}$. Therefore, $\beta \in \mathrm{var}(C) \cup \mathrm{var}_{\dot{\preccurlyeq}}(D)$.
Case: $\beta \in S\theta_1$
Then $\beta \in \mathrm{var}(\gamma\theta_1)$ for some $\gamma \in \mathrm{dom}(\theta_1)$ such that $\gamma\theta_1$ is static.
By Proposition A.11, $\gamma \in \mathrm{var}(D_1)$. Then $\beta \in \mathrm{var}(D_1)\theta_1 \subseteq \mathrm{var}_{\dot{\preccurlyeq}}(D)$. □

Lemma A.17. *Let θ and θ' be two type substitutions such that $\theta \Vdash_\Delta D$ and* $\mathrm{static}(\theta', \mathrm{var}(D)\theta)$. *If $(t_1 \mathrel{\dot{\preccurlyeq}} t_2) \in D$, then $t_1\theta\theta' = t_2\theta\theta'$.*

Proof. By definition of $\theta \Vdash_\Delta D$, we have $t_1\theta = t_2\theta$. Then, $t_1\theta\theta' = t_2\theta\theta'$. □

Lemma A.18. *Let θ and θ' be two type substitutions such that $\theta \Vdash_\Delta D$ and* $\mathrm{static}(\theta', \mathrm{var}(D)\theta)$. *If $(\tau \mathrel{\dot{\preccurlyeq}} \alpha) \in D$, then $\tau\theta\theta' \preccurlyeq \alpha\theta\theta'$.*

Proof. By definition of $\theta \Vdash_\Delta D$, we have $\tau\theta \preccurlyeq \alpha\theta$. Then, $\tau\theta\theta' \preccurlyeq \alpha\theta\theta'$. □

Lemma A.19.

$$\forall \Gamma, \Delta, e, \alpha, D, \theta. \left.\begin{array}{l} \Gamma; \Delta \vdash \langle\!\langle e : \alpha \rangle\!\rangle \rightsquigarrow D \\ \theta \in \mathrm{solve}_\Delta(D) \\ \mathrm{var}(e) \subseteq \Delta \\ \alpha \notin \mathrm{var}(\Gamma) \end{array}\right\} \implies \mathrm{static}(\theta, \mathrm{var}(\Gamma))$$

Proof. Consider an arbitrary $\beta \in \mathrm{var}(\Gamma)$. We show that $\beta\theta$ is static.
Case: $\beta \notin \mathrm{dom}(\theta)$ Then $\beta\theta = \beta$, which is static.
Case: $\beta \in \mathrm{dom}(\theta)$ Then $\beta \in \mathrm{var}(D)$ (by Proposition A.11), and therefore $\beta \in \mathrm{var}(\Gamma) \cap \mathrm{var}(D)$.
By Lemma A.16, $\beta \in \mathrm{var}(\langle\!\langle e : \alpha \rangle\!\rangle) \cup \mathrm{var}_{\dot{\preccurlyeq}}(D)$.
Case: $\beta \in \mathrm{var}(\langle\!\langle e : \alpha \rangle\!\rangle)$ This case is impossible because $\mathrm{var}(\langle\!\langle e : \alpha \rangle\!\rangle) = \mathrm{var}(e) \cup \{\alpha\}$, $\mathrm{dom}(\theta) \nparallel \mathrm{var}(e)$ (because $\mathrm{var}(e) \subseteq \Delta$), and $\alpha \notin \mathrm{var}(\Gamma)$.
Case: $\beta \in \mathrm{var}_{\dot{\preccurlyeq}}(D)$ Since $\theta \Vdash_\Delta D$, $\beta\theta$ must be static. □

Lemma A.20.

$$\forall \Gamma, \Delta, D_1, \theta_1, \rho, \theta, \theta'. \left.\begin{array}{l} \theta \Vdash_\Delta \mathrm{equiv}(\theta_1, D_1) \\ \mathrm{dom}(\rho) \nparallel \Gamma\theta_1 \\ \mathrm{static}(\theta', \mathrm{var}(\mathrm{equiv}(\theta_1, D_1))\theta) \\ \mathrm{static}(\theta_1, \mathrm{var}(\Gamma)) \end{array}\right\} \implies \Gamma\theta\theta' = \Gamma\theta_1\rho\theta\theta'$$

Proof. Consider an arbitrary $x \in \mathrm{dom}(\Gamma)$. We have $\Gamma(x) = \forall\vec{\alpha}.\, \tau$. We assume by α-renaming that $\vec{\alpha} \nparallel \theta_1, \rho, \theta, \theta'$; then, $(\Gamma\theta\theta')(x) = \forall\vec{\alpha}.\, \tau\theta\theta'$ and $(\Gamma\theta_1\rho\theta\theta')(x) = \forall\vec{\alpha}.\, \tau\theta_1\rho\theta\theta'$. We must show $\tau\theta\theta' = \tau\theta_1\rho\theta\theta'$. We show $\forall\alpha \in \mathrm{var}(\tau).\ \alpha\theta\theta' = \alpha\theta_1\rho\theta\theta'$. Consider an arbitrary $\alpha \in \mathrm{var}(\tau)$.
Case: $\alpha \in \vec{\alpha}$ Then (by our choice of naming) $\alpha\theta\theta' = \alpha$ and $\alpha\theta_1\rho\theta\theta' = \alpha$.
Case: $\alpha \notin \vec{\alpha}$ Then $\alpha \in \mathrm{var}(\Gamma)$ and hence: $\mathrm{var}(\alpha\theta_1) \subseteq \mathrm{var}(\Gamma\theta_1)$, and $\alpha\theta_1\rho = \alpha\theta_1$, and $\alpha\theta_1$ is static.
Case: $\alpha \notin \mathrm{dom}(\theta_1)$ Then $\alpha\theta_1 = \alpha$, $\alpha\theta_1\rho = \alpha$, and $\alpha\theta_1\rho\theta\theta' = \alpha\theta\theta'$.
Case: $\alpha \in \mathrm{dom}(\theta_1)$
Then $\{(\alpha \mathrel{\dot{\preccurlyeq}} \alpha\theta_1), (\alpha\theta_1 \mathrel{\dot{\preccurlyeq}} \alpha)\} \subseteq \mathrm{equiv}(\theta_1, D_1)$.
Therefore, we have $\alpha\theta_1\theta = \alpha\theta$ and $\alpha\theta_1\theta\theta' = \alpha\theta\theta'$. □

Theorem A.21. *Let $\mathcal{D}$ be a derivation of $\Gamma; \mathrm{var}(e) \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D$. Let θ be a type substitution such that $\theta \Vdash_{\mathrm{var}(e)} D$. Then, we have $\Gamma\theta \vdash e \rightsquigarrow (\!|e|\!)^{\mathcal{D}}_{\theta} : t\theta$.*

Proof. We show the following, stronger result (for all $\mathcal{D}$, Γ, Δ, e, t, D, θ, and θ'):

$$\left.\begin{array}{l} \mathcal{D} \text{ is a derivation of } \Gamma; \Delta \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D \\ \theta \Vdash_{\Delta} D \\ \mathrm{static}(\theta', \mathrm{var}(D)\theta) \\ \mathrm{var}(e) \subseteq \Delta \end{array}\right\} \implies \Gamma\theta\theta' \vdash e\theta' \rightsquigarrow (\!|e|\!)^{\mathcal{D}}_{\theta}\theta' : t\theta\theta'$$

This result implies the statement: we take $\Delta = \mathrm{var}(e)$ and $\theta' = \{\,\}$ (the identity substitution). The proof is by structural induction on e.

Case: $e = x$

(1)	$\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle x : t \rangle\!\rangle \rightsquigarrow D$	Given
(2)	$\theta \Vdash_{\Delta} D$	Given
(3)	$\mathrm{static}(\theta', \mathrm{var}(D)\theta)$	Given

By Lemma A.15 from (1):

$\Gamma(x) = \forall\vec{\alpha}.\, \tau$

$D = \{(\tau\{\vec{\alpha} := \vec{\beta}\} \mathrel{\dot{\preccurlyeq}} \alpha), (\alpha \mathrel{\dot{\leq}} t)\}$

Then:

$(\Gamma\theta\theta')(x) = \forall\vec{\alpha}.\, \tau\theta\theta'$	assuming $\vec{\alpha} \sharp \theta, \theta'$ by α-renaming
the types $\vec{\beta}\theta\theta'$ are static	by (2) and (3)
$\forall\alpha \in \mathrm{var}(\tau).\ \alpha\theta\theta'\{\vec{\alpha} := \vec{\beta}\theta\theta'\} = \alpha\{\vec{\alpha} := \vec{\beta}\}\theta\theta'$	since $\vec{\alpha} \sharp \theta, \theta'$
$\tau\theta\theta'\{\vec{\alpha} := \vec{\beta}\theta\theta'\} = \tau\{\vec{\alpha} := \vec{\beta}\}\theta\theta'$	
$\tau\{\vec{\alpha} := \vec{\beta}\}\theta\theta' \preccurlyeq \alpha\theta\theta'$	by Lemma A.18
$\alpha\theta\theta' = t\theta\theta'$	by Lemma A.17
$\Gamma\theta\theta' \vdash x \rightsquigarrow x\,[\vec{\beta}\theta\theta'] : \tau\theta\theta'\{\vec{\alpha} := \vec{\beta}\theta\theta'\}$	by [Var]
$\Gamma\theta\theta' \vdash x \rightsquigarrow x\,[\vec{\beta}\theta\theta']\langle\tau\{\vec{\alpha} := \vec{\beta}\}\theta\theta' \overset{\ell}{\Rightarrow} \alpha\theta\theta'\rangle : t\theta\theta'$	by [Materialize]

This concludes this case since $(\!|x|\!)^{\mathcal{D}}_{\theta}\theta' = x\,[\vec{\beta}\theta\theta']\langle\tau\{\vec{\alpha} := \vec{\beta}\}\theta\theta' \overset{\ell}{\Rightarrow} \alpha\theta\theta'\rangle$.

Case: $e = c$

$\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle c : t \rangle\!\rangle \rightsquigarrow D$	Given
$D = \{b_c \mathrel{\dot{\leq}} t\}$	by Lemma A.15
$b_c\theta\theta' = t\theta\theta'$	by Lemma A.17
$\Gamma\theta\theta' \vdash c\theta\theta' \rightsquigarrow c : t\theta\theta'$	by [Const]
$(\!|c|\!)^{\mathcal{D}}_{\theta}\theta' = c$	

Case: $e = \lambda x.\, e'$

$\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle \lambda x.\, e' : t \rangle\!\rangle \rightsquigarrow D$	Given

By Lemma A.15:

$\mathcal{D}' :: (\Gamma, x\colon \alpha_1); \Delta \vdash \langle\!\langle e'\colon \alpha_2 \rangle\!\rangle \rightsquigarrow D'$

$D = D' \cup \{(\alpha_1 \dot{\preccurlyeq} \alpha_1), (\alpha_1 \to \alpha_2 \dot{\leq} t)\}$

Then:

$\alpha_1\theta\theta'$ is static

$(\alpha_1 \to \alpha_2)\theta\theta' = t\theta\theta'$ by Lemma A.17

$\Gamma\theta\theta', x\colon \alpha_1\theta\theta' \vdash e'\theta\theta' \rightsquigarrow (\!|e'|\!)^{\mathcal{D}'}_{\theta}\theta'\colon \alpha_2\theta\theta'$ by IH

$\Gamma\theta\theta' \vdash (\lambda x.\, e'\theta\theta') \rightsquigarrow \lambda^{(\alpha_1\to\alpha_2)\theta\theta'} x.\, (\!|e'|\!)^{\mathcal{D}'}_{\theta}\theta'\colon (\alpha_1 \to \alpha_2)\theta\theta'$ by [Abstr]

$\Gamma\theta\theta' \vdash (\lambda x.\, e'\theta\theta') \rightsquigarrow \lambda^{(\alpha_1\to\alpha_2)\theta\theta'} x.\, (\!|e'|\!)^{\mathcal{D}'}_{\theta}\theta'\colon t\theta\theta'$

$(\!|\lambda x.\, e|\!)^{\mathcal{D}}_{\theta}\theta' = \lambda^{(\alpha_1\to\alpha_2)\theta\theta'} x.\, (\!|e'|\!)^{\mathcal{D}'}_{\theta}\theta'$

Case: $e = \lambda x\colon \tau.\, e'$

$\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle \lambda x\colon \tau.\, e'\colon t \rangle\!\rangle \rightsquigarrow D$ Given

By Lemma A.15:

$\mathcal{D}' :: (\Gamma, x\colon \tau); \Delta \vdash \langle\!\langle e'\colon \alpha_2 \rangle\!\rangle \rightsquigarrow D'$

$D = D' \cup \{(\tau \dot{\preccurlyeq} \alpha_1), (\alpha_1 \to \alpha_2 \dot{\leq} t)\}$

Then:

$\tau\theta\theta' \preccurlyeq \alpha_1\theta\theta'$ by Lemma A.18

$(\alpha_1 \to \alpha_2)\theta\theta' = t\theta\theta'$ by Lemma A.17

$\Gamma\theta\theta', x\colon \tau\theta\theta' \vdash e'\theta\theta' \rightsquigarrow (\!|e'|\!)^{\mathcal{D}'}_{\theta}\theta'\colon \alpha_2\theta\theta'$ by IH

$\Gamma\theta\theta' \vdash (\lambda x\colon \tau.\, e')\theta\theta' \rightsquigarrow \lambda^{(\tau\to\alpha_2)\theta\theta'} x.\, (\!|e'|\!)^{\mathcal{D}'}_{\theta}\theta'\colon (\tau \to \alpha_2)\theta\theta'$ by [AAbstr]

$\Gamma\theta\theta' \vdash (\lambda x\colon \tau.\, e')\theta\theta' \rightsquigarrow$
$\left(\lambda^{(\tau\to\alpha_2)\theta\theta'} x.\, (\!|e'|\!)^{\mathcal{D}'}_{\theta}\theta'\right)\langle(\tau \to \alpha_2)\theta\theta' \overset{\ell}{\Rightarrow} (\alpha_1 \to \alpha_2)\theta\theta'\rangle\colon (\alpha_1 \to \alpha_2)\theta\theta'$ by [Materialize]

$\Gamma\theta\theta' \vdash (\lambda x\colon \tau.\, e')\theta\theta' \rightsquigarrow$
$\left(\lambda^{(\tau\to\alpha_2)\theta\theta'} x.\, (\!|e'|\!)^{\mathcal{D}'}_{\theta}\theta'\right)\langle(\tau \to \alpha_2)\theta\theta' \overset{\ell}{\Rightarrow} (\alpha_1 \to \alpha_2)\theta\theta'\rangle\colon t\theta\theta'$

$(\!|\lambda x\colon \tau.\, e|\!)^{\mathcal{D}}_{\theta}\theta' =$
$\left(\lambda^{(\tau\to\alpha_2)\theta\theta'} x.\, (\!|e'|\!)^{\mathcal{D}'}_{\theta}\theta'\right)\langle(\tau \to \alpha_2)\theta\theta' \overset{\ell}{\Rightarrow} (\alpha_1 \to \alpha_2)\theta\theta'\rangle$

Case: $e = e_1\ e_2$

$\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle e_1\ e_2\colon t \rangle\!\rangle \rightsquigarrow D$ Given

By Lemma A.15:

$\mathcal{D}_1 :: \Gamma; \Delta \vdash \langle\!\langle e_1\colon \alpha \to t \rangle\!\rangle \rightsquigarrow D_1$

$\mathcal{D}_2 :: \Gamma; \Delta \vdash \langle\!\langle e_2\colon \alpha \rangle\!\rangle \rightsquigarrow D_2$

$D = D_1 \cup D_2$

Then:

$$\Gamma\theta\theta' \vdash e_1\theta\theta' \rightsquigarrow (\!|e_1|\!)^{\mathcal{D}_1}_{\theta}\theta' : (\alpha \to t)\theta\theta' \qquad \text{by IH}$$
$$\Gamma\theta\theta' \vdash e_2\theta\theta' \rightsquigarrow (\!|e_2|\!)^{\mathcal{D}_2}_{\theta}\theta' : \alpha\theta\theta' \qquad \text{by IH}$$
$$\Gamma\theta\theta' \vdash (e_1\ e_2)\theta\theta' \rightsquigarrow (\!|e_1|\!)^{\mathcal{D}_1}_{\theta}\theta'\ (\!|e_2|\!)^{\mathcal{D}_2}_{\theta}\theta' : t\theta\theta' \qquad \text{by [Appl]}$$
$$(\!|e_1\ e_2|\!)^{\mathcal{D}}_{\theta}\theta' = (\!|e_1|\!)^{\mathcal{D}_1}_{\theta}\theta'\ (\!|e_2|\!)^{\mathcal{D}_2}_{\theta}\theta'$$

Case: $e = (e_1, e_2)$

$$\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle (e_1, e_2) : t \rangle\!\rangle \rightsquigarrow D \qquad \text{Given}$$

By Lemma A.15:

$$\mathcal{D}_1 :: \Gamma; \Delta \vdash \langle\!\langle e_1 : \alpha_1 \rangle\!\rangle \rightsquigarrow D_1$$
$$\mathcal{D}_2 :: \Gamma; \Delta \vdash \langle\!\langle e_2 : \alpha_2 \rangle\!\rangle \rightsquigarrow D_2$$
$$D = D_1 \cup D_2 \cup \{\alpha_1 \times \alpha_2 \mathrel{\dot{\leq}} t\}$$

Then:

$$(\alpha_1 \times \alpha_2)\theta\theta' = t\theta\theta' \qquad \text{by Lemma A.17}$$
$$\Gamma\theta\theta' \vdash e_1\theta\theta' \rightsquigarrow (\!|e_1|\!)^{\mathcal{D}_1}_{\theta}\theta' : \alpha_1\theta\theta' \qquad \text{by IH}$$
$$\Gamma\theta\theta' \vdash e_2\theta\theta' \rightsquigarrow (\!|e_2|\!)^{\mathcal{D}_2}_{\theta}\theta' : \alpha_2\theta\theta' \qquad \text{by IH}$$
$$\Gamma\theta\theta' \vdash (e_1, e_2)\theta\theta' \rightsquigarrow \left((\!|e_1|\!)^{\mathcal{D}_1}_{\theta}\theta', (\!|e_2|\!)^{\mathcal{D}_2}_{\theta}\theta'\right) : t\theta\theta' \qquad \text{by [Pair]}$$
$$(\!|(e_1, e_2)|\!)^{\mathcal{D}}_{\theta}\theta' = \left((\!|e_1|\!)^{\mathcal{D}_1}_{\theta}\theta', (\!|e_2|\!)^{\mathcal{D}_2}_{\theta}\theta'\right)$$

Case: $e = \pi_i\ e'$

$$\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle \pi_i\ e' : t \rangle\!\rangle \rightsquigarrow D \qquad \text{Given}$$

By Lemma A.15:

$$\mathcal{D}' :: \Gamma; \Delta \vdash \langle\!\langle e' : \alpha_1 \times \alpha_2 \rangle\!\rangle \rightsquigarrow D'$$
$$D = D' \cup \{\alpha_i \mathrel{\dot{\leq}} t\}$$

Then:

$$\alpha_i\theta\theta' = t\theta\theta' \qquad \text{by Lemma A.17}$$
$$\Gamma\theta\theta' \vdash e'\theta\theta' \rightsquigarrow (\!|e'|\!)^{\mathcal{D}'}_{\theta}\theta' : (\alpha_1 \times \alpha_2)\theta\theta' \qquad \text{by IH}$$
$$\Gamma\theta\theta' \vdash (\pi_i\ e')\theta\theta' \rightsquigarrow \pi_i\ ((\!|e'|\!)^{\mathcal{D}'}_{\theta}\theta') : t\theta\theta' \qquad \text{by [Proj]}$$
$$(\!|\pi_i\ e'|\!)^{\mathcal{D}}_{\theta}\theta' = (\pi_i\ (\!|e'|\!)^{\mathcal{D}'}_{\theta})\theta'$$

Case: $e = (\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2)$

$$\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle \mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2 : t \rangle\!\rangle \rightsquigarrow D \qquad \text{Given}$$

By Lemma A.15:

$$\mathcal{D}_1 :: \Gamma; \Delta \cup \vec{\alpha} \vdash \langle\!\langle e_1 : \alpha \rangle\!\rangle \rightsquigarrow D_1$$
$$\mathcal{D}_2 :: (\Gamma, x : \forall \vec{\alpha}, \vec{\beta}.\, \alpha\theta_1); \Delta \vdash \langle\!\langle e_2 : t \rangle\!\rangle \rightsquigarrow D_2$$
$$D = D_2 \cup \mathsf{equiv}(\theta_1, D_1)$$
$$\theta_1 \in \mathsf{solve}_{\Delta \cup \vec{\alpha}}(D_1)$$
$$\vec{\alpha} \sharp \mathsf{var}(\Gamma\theta_1)$$
$$\vec{\beta} = \mathsf{var}(\alpha\theta_1) \setminus (\mathsf{var}(\Gamma\theta_1) \cup \vec{\alpha} \cup \mathsf{var}(e_1))$$

Let $\vec{\alpha}_1$ and $\vec{\beta}_1$ be vectors of distinct variables chosen outside $\mathsf{var}(e_1)$, $\mathsf{dom}(\theta)$, $\mathsf{var}(\theta)$, $\mathsf{dom}(\theta')$, and $\mathsf{var}(\theta')$. Let $\rho = \{\vec{\alpha} \coloneqq \vec{\alpha}_1\} \cup \{\vec{\beta} \coloneqq \vec{\beta}_1\}$. Then:

$e\theta' = (\mathsf{let}\ \vec{\alpha}_1\ x = e_1\rho\theta'\ \mathsf{in}\ e_2\theta')$ — since $\vec{\beta} \sharp e_1$ and $\vec{\alpha}_1 \sharp \theta'$

$(\!|e|\!)^{\mathcal{D}}_{\theta} = \big(\mathsf{let}\ x = (\Lambda\vec{\alpha}_1, \vec{\beta}_1.\, (\!|e_1|\!)^{\mathcal{D}_1}_{\theta_1}\rho\theta)\ \mathsf{in}\ (\!|e_2|\!)^{\mathcal{D}_2}_{\theta}\big)$

$(\!|e|\!)^{\mathcal{D}}_{\theta}\theta' = \big(\mathsf{let}\ x = (\Lambda\vec{\alpha}_1, \vec{\beta}_1.\, (\!|e_1|\!)^{\mathcal{D}_1}_{\theta_1}\rho\theta\theta')\ \mathsf{in}\ (\!|e_2|\!)^{\mathcal{D}_2}_{\theta}\theta'\big)$ — since $\vec{\alpha}_1, \vec{\beta}_1 \sharp \theta'$

For e_1:

$\theta_1 \Vdash_{\Delta \cup \vec{\alpha}} D_1$	
$\mathsf{static}(\rho\theta\theta', \mathsf{var}(D_1)\theta_1)$	proven below
$\mathsf{var}(e_1) \subseteq \Delta \cup \vec{\alpha}$	
$\Gamma\theta_1\rho\theta\theta' \vdash e_1\rho\theta\theta' \rightsquigarrow (\!|e_1|\!)^{\mathcal{D}_1}_{\theta_1}\rho\theta\theta' : \alpha\theta_1\rho\theta\theta'$	by IH
$e_1\rho\theta\theta' = e_1\rho\theta'$	since $\mathsf{dom}(\theta) \cap \mathsf{var}(e_1\rho) = \varnothing$
$\alpha \notin \mathsf{var}(\Gamma)$	by inversion
$\mathsf{static}(\theta_1, \mathsf{var}(\Gamma))$	by Lemma A.19
$\Gamma\theta\theta' = \Gamma\theta_1\rho\theta\theta'$	by Lemma A.20
$\Gamma\theta\theta' \vdash e_1\rho\theta' \rightsquigarrow (\!|e_1|\!)^{\mathcal{D}_1}_{\theta_1}\rho\theta\theta' : \alpha\theta_1\rho\theta\theta'$	

For e_2:

$\theta \Vdash_{\Delta} D_2$	
$\mathsf{static}(\theta', \mathsf{var}(D_2)\theta)$	
$\mathsf{var}(e_2) \subseteq \Delta$	
$\Gamma\theta\theta', x : (\forall\vec{\alpha}, \vec{\beta}.\, \alpha\theta_1)\theta\theta' \vdash e_2\theta' \rightsquigarrow (\!|e_2|\!)^{\mathcal{D}_2}_{\theta}\theta' : t\theta\theta'$	by IH
$(\forall\vec{\alpha}, \vec{\beta}.\, \alpha\theta_1)\theta\theta' = (\forall\vec{\alpha}_1, \vec{\beta}_1.\, \alpha\theta_1\rho\theta\theta')$	since $\vec{\alpha}_1, \vec{\beta}_1 \sharp \theta, \theta'$
$\Gamma\theta\theta', x : (\forall\vec{\alpha}_1, \vec{\beta}_1.\, \alpha\theta_1\rho\theta\theta') \vdash e_2\theta' \rightsquigarrow (\!|e_2|\!)^{\mathcal{D}_2}_{\theta}\theta' : t\theta\theta'$	
$\vec{\alpha}_1, \vec{\beta}_1 \sharp \Gamma\theta\theta'$ and $\vec{\beta}_1 \sharp e_1\rho\theta'$	

Finally:

$$\Gamma\theta\theta' \vdash e\theta' \rightsquigarrow (\!|e|\!)^{\mathcal{D}}_{\theta}\theta' : t\theta\theta' \qquad \text{by [Let]}$$

To check $\mathsf{static}(\rho\theta\theta', \mathsf{var}(D_1)\theta_1)$, take an arbitrary $\alpha \in \mathsf{var}(D_1)\theta_1$.

- If $\alpha \in \mathsf{dom}(\rho)$, then $\alpha\rho$ is a variable in $\vec{\alpha}_1, \vec{\beta}_1$ and $\alpha\rho = \alpha\rho\theta\theta'$ (because $\vec{\alpha}_1, \vec{\beta}_1 \sharp \theta, \theta'$): hence $\alpha\rho\theta\theta'$ is static.

- If $\alpha \notin \mathrm{dom}(\rho)$, then $\alpha\rho\theta\theta' = \alpha\theta\theta'$. We have $(\alpha \dot{\preccurlyeq} \alpha) \in \mathrm{equiv}(\theta_1, D_1)$. Since $\mathrm{equiv}(\theta_1, D_1) \subseteq D$, $\alpha\theta$ is static. Furthermore, $\mathrm{var}(\alpha\theta) \subseteq \mathrm{var}(D)\theta$; hence, $\alpha\theta\theta'$ is static too. □

Lemma A.22. *Let $\Gamma \vdash e : \tau$. Then:*

- *if $e = x$ then $\Gamma(x) = \forall\vec{\alpha}.\, \tau_x$ and $\tau_x\{\vec{\alpha} := \vec{t}\} \preccurlyeq \tau$;*
- *if $e = c$, then $\tau = b_c$;*
- *if $e = \lambda x.\, e_1$ then $\tau = t \to \tau_1$ and $\Gamma, x : t \vdash e_1 : \tau_1$;*
- *if $e = \lambda x : \tau'.\, e_1$ then $\tau = \tau_1' \to \tau_1$, $\tau' \preccurlyeq \tau_1'$, and $\Gamma, x : \tau' \vdash e_1 : \tau_1$;*
- *if $e = e_1\ e_2$, then $\Gamma \vdash e_1 : \tau' \to \tau$ and $\Gamma \vdash e_2 : \tau'$;*
- *if $e = (e_1, e_2)$, then $\tau = \tau_1 \times \tau_2$, $\Gamma \vdash e_1 : \tau_1$, and $\Gamma \vdash e_2 : \tau_2$;*
- *if $e = \pi_i\ e'$, then $\Gamma \vdash e' : \tau_1 \times \tau_2$ and $\tau = \tau_i$;*
- *if $e = (\mathrm{let}\ \vec{\alpha}\ x = e_1\ \mathrm{in}\ e_2)$, then $\Gamma \vdash e_1 : \tau_1$, $\Gamma, x : \forall\vec{\alpha},\vec{\beta}.\, \tau_1 \vdash e_2 : \tau$, $\vec{\alpha},\vec{\beta} \sharp \Gamma$, and $\vec{\beta} \sharp e_1$.*

Proof. The derivation of $\Gamma \vdash e : \tau$ must end with the rule corresponding to the shape of e, possibly followed by applications of [Materialize]. We proceed by case on the derivation, possibly applying [Materialize] to the derivations in the premises to obtain the needed results. □

We say that a set $\mathfrak{U} \subseteq \mathcal{V}$ is a *variable pool* if both $\mathfrak{U} \cap \mathcal{V}^\alpha$ and $\mathfrak{U} \cap \mathcal{V}^X$ are countably infinite. Throughout the proof of completeness, we will use variable pools to choose new fresh variables, and we will partition variable pools to obtain new pools. For example, we will write $\mathfrak{U} = \{\alpha\} \uplus \mathfrak{U}_1 \uplus \mathfrak{U}_2$ to mean that we partition $\mathfrak{U}$ into three sets: a singleton set α and two variable pools $\mathfrak{U}_1$ and $\mathfrak{U}_2$.

Lemma A.23. *Let θ and $\theta_1, \ldots, \theta_n$ be type substitutions, such that the θ_i are pairwise disjoint and every θ_i is disjoint from θ. Let $D_1, \ldots, D_n$ be type constraint sets such that, for every $i_1 \neq i_2$, $\theta_{i_1} \sharp \mathrm{var}(D_{i_2})$.*
If, for every $i \in \{1, \ldots, n\}$, we have $\theta \cup \theta_i \Vdash_\Delta D_i$, then $\theta \cup \bigcup_{i=1}^n \theta_i \Vdash_\Delta \bigcup_{i=1}^n D_i$.

Proof. Straightforward since, because of the disjointness conditions, for every i_0 and every $\alpha \in \mathrm{var}(D_{i_0})$, we have $\alpha(\theta \cup \bigcup_{i=1}^n \theta_i) = \alpha(\theta \cup \theta_{i_0})$. □

Lemma A.24. *If $\Gamma; \Delta \vdash C \rightsquigarrow D \mid \overline{\alpha}$, then $\mathrm{var}(D) \subseteq \mathrm{var}(\Gamma) \cup \mathrm{var}(C) \cup \overline{\alpha}$.*

Proof. By induction on the derivation of $\Gamma; \Delta \vdash C \rightsquigarrow D \mid \overline{\alpha}$. All cases are straightforward except that of let constraints.

Let $C = \mathrm{let}\ x : \forall\vec{\alpha}; \alpha[C_1]^{\vec{\alpha}'}.\, \alpha\ \mathrm{in}\ C_2$. Assume $\Gamma; \Delta \vdash C \rightsquigarrow D_2 \cup \mathrm{equiv}(\theta_1, D_1) \mid \overline{\alpha}$. Consider an arbitrary $\beta \in \mathrm{var}(D_2) \cup \mathrm{equiv}(\theta_1, D_1)$. We must show $\beta \in \mathrm{var}(\Gamma) \cup \mathrm{var}(C) \cup \overline{\alpha}$.

Case: $\beta \in \mathrm{var}(D_2)$
By IH, we have $\beta \in \mathrm{var}(\Gamma) \cup \mathrm{var}(\forall\vec{\alpha},\vec{\beta}.\, \alpha\theta_1) \cup \mathrm{var}(C_2) \cup \overline{\alpha}_2$.
If $\beta \in \mathrm{var}(\Gamma) \cup \mathrm{var}(C_2) \cup \overline{\alpha}_2$, then $\beta \in \mathrm{var}(\Gamma) \cup \mathrm{var}(C) \cup \overline{\alpha}$.
If $\beta \in \mathrm{var}(\forall\vec{\alpha},\vec{\beta}.\, \alpha\theta_1)$, then either $\beta = \alpha$ or $\beta \in \mathrm{var}(\theta_1)$.
- If $\beta = \alpha$, then $\beta \in \overline{\alpha}$.
- If $\beta \in \mathrm{var}(\theta_1)$, either $\beta \in \mathrm{var}(D_1)$ or not.
  In the latter case, $\beta \in \overline{\alpha}$.
  In the former, by IH, we have $\beta \in \mathrm{var}(\Gamma) \cup \mathrm{var}(C_1) \cup \overline{\alpha}_1$. Note that $\mathrm{var}(C_1) \subseteq \mathrm{var}(C) \cup \{\alpha\} \cup \vec{\alpha}$. Then, $\beta \in \mathrm{var}(\Gamma) \cup \mathrm{var}(C) \cup \overline{\alpha}$.

Case: $\beta \in \mathrm{var}(\mathrm{equiv}(\theta_1, D_1))$ By Proposition A.11, $\mathrm{dom}(\theta_1) \subseteq \mathrm{var}(D_1)$. Then, $\beta \in \mathrm{var}(D_1) \cup \mathrm{var}(\theta_1)$. Both cases have already been treated above. □

Lemma A.25. *If $\Gamma; \Delta \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D \mid \overline{\alpha}$, then $\mathrm{var}(t) \subseteq \mathrm{var}(D)$.*

Proof. We define a function v mapping structured constraints to sets of type variables. We show these two results, which together imply the statement:

- for every t and e, $\mathsf{var}(t) \subseteq v(\langle\!\langle e : t \rangle\!\rangle)$;
- for every Γ, Δ, C, D, and $\overline{\alpha}$, if $\Gamma; \Delta \vdash C \rightsquigarrow D \mid \overline{\alpha}$, then $v(C) \subseteq \mathsf{var}(D)$.

The function v is defined by induction on the structured constraint as follows:

$$v(t_1 \mathrel{\dot{\leq}} t_2) = \mathsf{var}(t_2) \qquad v(\tau \mathrel{\dot{\preccurlyeq}} \alpha) = \varnothing \qquad v(x \mathrel{\dot{\preccurlyeq}} \alpha) = \varnothing$$
$$v(\mathsf{def}\ x : \tau\ \mathsf{in}\ C) = v(C) \qquad v(\exists \vec{\alpha}.\, C) = v(C) \setminus \vec{\alpha} \qquad v(C_1 \wedge C_2) = v(C_1) \cup v(C_2)$$
$$v(\mathsf{let}\ x : \forall \vec{\alpha}; \alpha[C_1]^{\vec{\alpha}'}.\, \alpha\ \mathsf{in}\ C_2) = v(C_2)$$

The two results are proven easily by induction, respectively on e and on the derivation of $\Gamma; \Delta \vdash C \rightsquigarrow D \mid \overline{\alpha}$. □

Theorem A.26 (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 $\{\alpha \coloneqq \tau\} \cup \theta \Vdash_{\mathsf{var}(e)} D$.*

Proof. We show the following, stronger result (for all Γ, θ, e, t, Δ, and $\mathfrak{U}$):

$$\left.\begin{array}{l} \Gamma\theta \vdash e : t\theta \\ \mathsf{static}(\theta, \Gamma) \\ \mathsf{dom}(\theta) \sharp \Delta \supseteq \mathsf{var}(e) \\ \mathfrak{U} \sharp \Delta, t, \Gamma, \mathsf{dom}(\theta) \end{array}\right\} \implies \exists D, \overline{\alpha}, \theta'.\ \begin{cases} \Gamma; \Delta \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D \mid \overline{\alpha} \\ \theta \cup \theta' \Vdash_\Delta D \\ \mathsf{dom}(\theta') \subseteq \overline{\alpha} \subseteq \mathfrak{U} \end{cases}$$

This result implies the statement: take $t = \alpha$ (with $\alpha \sharp \Gamma, \mathsf{var}(e)$), $\theta = \{\alpha \coloneqq \tau\}$, and $\Delta = \mathsf{var}(e)$. The proof is by structural induction on e.

Case: $e = x$

We have $\Gamma\theta \vdash x : t\theta$. Therefore, $x \in \mathsf{dom}(\Gamma)$.
Let $\Gamma(x)$ be $\forall \vec{\alpha}.\, \tau$ and assume, by α-renaming, $\vec{\alpha} \sharp \theta$. Then, $(\Gamma\theta)(x) = \forall \vec{\alpha}.\, \tau\theta$.
By inversion of the typing rules, there exists an instance $\tau\theta\{\vec{\alpha} \coloneqq \vec{t}\}$ of $(\Gamma\theta)(x)$ such that $\tau\theta\{\vec{\alpha} \coloneqq \vec{t}\} \preccurlyeq t\theta$.
We take $\alpha \in \mathfrak{U}$. Then, $\langle\!\langle x : t \rangle\!\rangle = \exists \alpha.\, (x \mathrel{\dot{\preccurlyeq}} \alpha) \wedge (\alpha \mathrel{\dot{\leq}} t)$ (since $\alpha \sharp t$).
We take $\vec{\beta} \in \mathfrak{U}$ (with $\vec{\beta} \sharp \alpha$). We have

$$\Gamma; \Delta \vdash (x \mathrel{\dot{\preccurlyeq}} \alpha) \rightsquigarrow \{\tau\{\vec{\alpha} \coloneqq \vec{\beta}\} \mathrel{\dot{\preccurlyeq}} \alpha\} \mid \vec{\beta} \qquad \Gamma; \Delta \vdash (\alpha \mathrel{\dot{\leq}} t) \rightsquigarrow \{\alpha \mathrel{\dot{\leq}} t\} \mid \varnothing$$

and therefore (since $\alpha \sharp \Gamma, \vec{\beta}$)

$$\Gamma; \Delta \vdash \langle\!\langle x : t \rangle\!\rangle \rightsquigarrow \{(\tau\{\vec{\alpha} \coloneqq \vec{\beta}\} \mathrel{\dot{\preccurlyeq}} \alpha), (\alpha \mathrel{\dot{\leq}} t)\} \mid \vec{\beta} \cup \{\alpha\}$$

We take $\theta' = \{\alpha \coloneqq t\theta\} \cup \{\beta \coloneqq \vec{t}\}$ and show $\theta \cup \theta' \Vdash_\Delta \{(\tau\{\vec{\alpha} \coloneqq \vec{\beta}\} \mathrel{\dot{\preccurlyeq}} \alpha), (\alpha \mathrel{\dot{\leq}} t)\}$:

- $\alpha(\theta \cup \theta') = t\theta$ and $t(\theta \cup \theta') = t\theta$;
- $\tau\{\vec{\alpha} \coloneqq \vec{\beta}\}(\theta \cup \theta') = \tau\theta\{\vec{\alpha} \coloneqq \vec{t}\}$ (because $\mathsf{var}(\tau) \setminus \vec{\alpha} \subseteq \mathsf{var}(\Gamma) \sharp \mathsf{dom}(\theta')$);
- $\theta \cup \theta'$ is static on $\mathsf{var}(\tau\{\vec{\alpha} \coloneqq \vec{\beta}\})$, because θ is static on $\mathsf{var}(\Gamma)$ and θ' is static on $\vec{\beta}$.

Case: $e = c$ We have:

$$t\theta = b_c \qquad \text{by Lemma A.22}$$
$$\langle\!\langle c : t \rangle\!\rangle = (b_c \mathrel{\dot{\leq}} t)$$
$$\Gamma; \Delta \vdash \langle\!\langle c : t \rangle\!\rangle \rightsquigarrow (b_c \mathrel{\dot{\leq}} t) \mid \varnothing$$

We take $\theta' = \{\,\}$. Then, $\theta \cup \theta' \Vdash_\Delta (b_c \mathrel{\dot{\leq}} t)$ holds since $b_c = t\theta$ and $\mathsf{dom}(\theta) \sharp \Delta$.

Case: $e = (\lambda x.\, e_1)$

By Lemma A.22:

$$t\theta = t_1 \to \tau_1 \qquad \Gamma\theta, x : t_1 \vdash e_1 : \tau_1$$

We partition the variable pool as $\mathfrak{U} = \{\alpha_1, \alpha_2\} \uplus \mathfrak{U}_1$. Let $\hat{\theta} = \theta \cup \{\alpha_1 \coloneqq t_1\} \cup \{\alpha_2 \coloneqq \tau_1\}$.
We have

$$\langle\!\langle (\lambda x.\, e_1) \colon t \rangle\!\rangle = \exists \alpha_1, \alpha_2.\, (\mathsf{def}\ x \colon \alpha_1\ \mathsf{in}\ \langle\!\langle e_1 \colon \alpha_2 \rangle\!\rangle) \wedge (\alpha_1 \mathrel{\dot{\preccurlyeq}} \alpha_1) \wedge (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)$$

since $\alpha_1, \alpha_2 \sharp t, e_1$.
We have:

$$\Gamma\theta = \Gamma\hat{\theta} \text{ and } t\theta = t\hat{\theta} \qquad \text{since } \alpha_1, \alpha_2 \sharp t, \Gamma$$
$$\mathsf{static}(\hat{\theta}, (\Gamma, x \colon \alpha_1))$$
$$(\Gamma, x \colon \alpha_1)\hat{\theta} \vdash e_1 \colon \alpha_2\hat{\theta}$$

By IH:

$$(\Gamma, x \colon \alpha_1); \Delta \vdash \langle\!\langle e_1 \colon \alpha_2 \rangle\!\rangle \rightsquigarrow D_1 \mid \overline{\alpha}_1 \qquad \hat{\theta} \cup \theta_1' \Vdash_\Delta D_1 \qquad \mathsf{dom}(\theta_1') \subseteq \overline{\alpha}_1 \subseteq \mathfrak{U}_1$$

Then we have

$$\Gamma; \Delta \vdash \langle\!\langle (\lambda x.\, e_1) \colon t \rangle\!\rangle \rightsquigarrow D_1 \cup \{(\alpha_1 \mathrel{\dot{\preccurlyeq}} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\} \mid \overline{\alpha}_1 \cup \{\alpha_1, \alpha_2\}$$

since $\alpha_1, \alpha_2 \sharp \Gamma, \overline{\alpha}_1$.
We take $\theta' = \{\alpha_1 \coloneqq t_1\} \cup \{\alpha_2 \coloneqq \tau_1\} \cup \theta_1'$. Note that $\theta \cup \theta' = \hat{\theta} \cup \theta_1'$.
We have $\theta \cup \theta' \Vdash_\Delta D_1 \cup \{(\alpha_1 \mathrel{\dot{\preccurlyeq}} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\}$ because $\alpha_1(\theta \cup \theta') = t_1$ is static and because $(\alpha_1 \to \alpha_2)(\theta \cup \theta') = t_1 \to \tau_1 = t\theta = t(\theta \cup \theta')$.

Case: $e = (\lambda x \colon \tau.\, e_1)$
By Lemma A.22:

$$t\theta = \tau' \to \tau_1 \qquad \tau \preccurlyeq \tau' \qquad \Gamma\theta, x \colon \tau \vdash e_1 \colon \tau_1$$

We partition the variable pool as $\mathfrak{U} = \{\alpha_1, \alpha_2\} \uplus \mathfrak{U}_1$. Let $\hat{\theta} = \theta \cup \{\alpha_1 \coloneqq \tau'\} \cup \{\alpha_2 \coloneqq \tau_1\}$.
We have

$$\langle\!\langle (\lambda x \colon \tau.\, e_1) \colon t \rangle\!\rangle = \exists \alpha_1, \alpha_2.\, (\mathsf{def}\ x \colon \tau\ \mathsf{in}\ \langle\!\langle e_1 \colon \alpha_2 \rangle\!\rangle) \wedge (\tau \mathrel{\dot{\preccurlyeq}} \alpha_1) \wedge (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)$$

since $\alpha_1, \alpha_2 \sharp t, \tau, e_1$.
We have:

$$\Gamma\theta = \Gamma\hat{\theta} \text{ and } t\theta = t\hat{\theta} \qquad \text{since } \alpha_1, \alpha_2 \sharp t, \Gamma$$
$$\tau\hat{\theta} = \tau\theta = \tau \qquad \text{since } \alpha_1, \alpha_2 \sharp \tau \text{ and } \mathsf{var}(\tau) \subseteq \Delta$$
$$\mathsf{static}(\hat{\theta}, (\Gamma, x \colon \tau))$$
$$(\Gamma, x \colon \tau)\hat{\theta} \vdash e_1 \colon \alpha_2\hat{\theta}$$

By IH:

$$(\Gamma, x \colon \tau); \Delta \vdash \langle\!\langle e_1 \colon \alpha_2 \rangle\!\rangle \rightsquigarrow D_1 \mid \overline{\alpha}_1 \qquad \hat{\theta} \cup \theta_1' \Vdash_\Delta D_1 \qquad \mathsf{dom}(\theta_1') \subseteq \overline{\alpha}_1 \subseteq \mathfrak{U}_1$$

Then we have

$$\Gamma; \Delta \vdash \langle\!\langle (\lambda x \colon \tau.\, e_1) \colon t \rangle\!\rangle \rightsquigarrow D_1 \cup \{(\tau \mathrel{\dot{\preccurlyeq}} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\} \mid \overline{\alpha}_1 \cup \{\alpha_1, \alpha_2\}$$

since $\alpha_1, \alpha_2 \sharp \Gamma, \overline{\alpha}_1$.
We take $\theta' = \{\alpha_1 \coloneqq \tau'\} \cup \{\alpha_2 \coloneqq \tau_1\} \cup \theta_1'$. Note that $\theta \cup \theta' = \hat{\theta} \cup \theta_1'$.
We have $\theta \cup \theta' \Vdash_\Delta D_1 \cup \{(\tau \mathrel{\dot{\preccurlyeq}} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\}$ because $\tau(\theta \cup \theta') = \tau \preccurlyeq \tau' = \alpha_1(\theta \cup \theta')$, because $\theta \cup \theta'$ is static on τ (since it is the identity), and because $(\alpha_1 \to \alpha_2)(\theta \cup \theta') = \tau' \to \tau_1 = t\theta = t(\theta \cup \theta')$.

Case: $e = e_1\ e_2$

By Lemma A.22:

$$\Gamma\theta \vdash e_1 : \tau \to t\theta$$
$$\Gamma\theta \vdash e_2 : \tau$$

We partition the variable pool as $\mathfrak{U} = \{\alpha\} \uplus \mathfrak{U}_1 \uplus \mathfrak{U}_2$. Let $\hat{\theta} = \theta \cup \{\alpha \coloneqq \tau\}$. We have:

$\langle\!\langle e_1\ e_2 : t \rangle\!\rangle = \exists\alpha.\ \langle\!\langle e_1 : \alpha \to t \rangle\!\rangle \wedge \langle\!\langle e_2 : \alpha \rangle\!\rangle$ since $\alpha \sharp t, e_1, e_2$

$\Gamma\theta = \Gamma\hat{\theta}$ and $t\theta = t\hat{\theta}$ since $\alpha \sharp t, \Gamma$

$\mathrm{static}(\hat{\theta}, \Gamma)$

$\Gamma\hat{\theta} \vdash e_1 : (\alpha \to t)\hat{\theta}$

$\Gamma\hat{\theta} \vdash e_2 : \alpha\hat{\theta}$

By IH:

$$\Gamma; \Delta \vdash \langle\!\langle e_1 : \alpha \to t \rangle\!\rangle \rightsquigarrow D_1 \mid \overline{\alpha}_1 \qquad \hat{\theta} \cup \theta_1' \Vdash_\Delta D_1 \qquad \mathrm{dom}(\theta_1') \subseteq \overline{\alpha}_1 \subseteq \mathfrak{U}_1$$
$$\Gamma; \Delta \vdash \langle\!\langle e_2 : \alpha \rangle\!\rangle \rightsquigarrow D_2 \mid \overline{\alpha}_2 \qquad \hat{\theta} \cup \theta_2' \Vdash_\Delta D_2 \qquad \mathrm{dom}(\theta_2') \subseteq \overline{\alpha}_2 \subseteq \mathfrak{U}_2$$

Then:

$$\Gamma; \Delta \vdash \langle\!\langle e_1\ e_2 : t \rangle\!\rangle \rightsquigarrow D_1 \cup D_2 \mid \overline{\alpha}_1 \cup \overline{\alpha}_2 \cup \{\alpha\} \qquad \text{since } \overline{\alpha}_1 \sharp \overline{\alpha}_2 \text{ and } \alpha \sharp \Gamma, (\overline{\alpha}_1 \cup \overline{\alpha}_2)$$

We take $\theta' = \{\alpha \coloneqq \tau\} \cup \theta_1' \cup \theta_2'$.
By Lemma A.24, we have that $\theta_1' \sharp \mathrm{var}(D_2)$ and $\theta_2' \sharp \mathrm{var}(D_1)$.
Then, by Lemma A.23, $\theta \cup \theta' \Vdash_\Delta D_1 \cup D_2$.

Case: $e = (\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2)$

By Lemma A.22:

$$\Gamma\theta \vdash e_1 : \tau_1 \qquad \Gamma\theta, x : \forall\vec{\alpha},\vec{\beta}.\ \tau_1 \vdash e_2 : t\theta \qquad \vec{\alpha},\vec{\beta} \sharp \Gamma\theta \qquad \vec{\beta} \sharp e_1$$

By α-renaming, we can assume $\vec{\alpha} \subseteq \mathfrak{U}$. We partition the variable pool as $\mathfrak{U} = \{\alpha\} \uplus \vec{\alpha} \uplus \mathfrak{U}_1 \uplus \mathfrak{U}_2 \uplus \mathfrak{U}_3$. Let $\hat{\theta} = \theta \cup \{\alpha \coloneqq \tau_1\}$. We have:

$$\langle\!\langle e : t \rangle\!\rangle = \mathsf{let}\ x : \forall\vec{\alpha}; \alpha[\langle\!\langle e_1 : \alpha \rangle\!\rangle]^{\mathrm{var}(e_1)\setminus\vec{\alpha}}.\ \alpha\ \mathsf{in}\ \langle\!\langle e_2 : t \rangle\!\rangle$$
$$\Gamma\theta = \Gamma\hat{\theta} \text{ and } t\theta = t\hat{\theta}$$
$$\mathrm{static}(\hat{\theta}, \Gamma)$$
$$\Gamma\hat{\theta} \vdash e_1 : \alpha\hat{\theta}$$

By IH (using $\Delta \cup \vec{\alpha}$ instead of Δ):

$$\Gamma; \Delta \cup \vec{\alpha} \vdash \langle\!\langle e_1 : \alpha \rangle\!\rangle \rightsquigarrow D_1 \mid \overline{\alpha}_1 \qquad \hat{\theta} \cup \theta_1' \Vdash_{\Delta\cup\vec{\alpha}} D_1 \qquad \mathrm{dom}(\theta_1') \subseteq \overline{\alpha}_1 \subseteq \mathfrak{U}_1$$

Since $\hat{\theta} \cup \theta_1' \Vdash_{\Delta\cup\vec{\alpha}} D_1$, by Proposition A.13, there exist two substitutions θ_1 and $\tilde{\theta}_1$ such that

$$\theta_1 \in \mathrm{solve}_{\Delta\cup\vec{\alpha}}(D_1) \qquad \mathrm{dom}(\tilde{\theta}_1) \subseteq \mathrm{var}(\theta_1) \setminus \mathrm{var}(D_1)$$
$$\forall\alpha.\ \alpha\theta_1(\hat{\theta} \cup \theta_1' \cup \tilde{\theta}_1) \preccurlyeq \alpha(\hat{\theta} \cup \theta_1' \cup \tilde{\theta}_1)$$
$$\forall\alpha.\ \alpha\theta_1 \text{ static} \implies \alpha\theta_1(\hat{\theta} \cup \theta_1' \cup \tilde{\theta}_1) = \alpha(\hat{\theta} \cup \theta_1' \cup \tilde{\theta}_1)$$

We can choose the variables in $\mathrm{var}(\theta_1) \setminus \mathrm{var}(D_1)$ freely from a set of fresh variables: we take them from $\mathfrak{U}_3$.
Let $\breve{\theta} = \theta \cup \{\alpha \coloneqq \tau_1\} \cup \theta_1' \cup \tilde{\theta}_1$.

We have $\Gamma\theta = \Gamma\check{\theta}$ and $t\theta = t\check{\theta}$.
Let $\vec{\gamma} = \mathrm{var}(\alpha\theta_1) \setminus (\mathrm{var}(\Gamma\theta_1) \cup \vec{\alpha} \cup (\mathrm{var}(e_1) \setminus \vec{\alpha})) = \mathrm{var}(\alpha\theta_1) \setminus (\mathrm{var}(\Gamma\theta_1) \cup \vec{\alpha} \cup \mathrm{var}(e_1))$. Let $\Gamma' = (\Gamma, x\colon \forall\vec{\alpha},\vec{\gamma}.\, \alpha\theta_1)$.
We show $\mathrm{static}(\theta_1, \Gamma)$. Take $\beta \in \mathrm{var}(\Gamma)$. If $\beta \notin \mathrm{var}(D_1)$, then $\beta\theta_1 = \beta$, which is static. Otherwise, by Lemma A.16, we have $\beta \in \mathrm{var}(\langle\!\langle e_1 \colon \alpha \rangle\!\rangle) \cap \mathrm{var}_{\preccurlyeq}(D_1)$. We have $\mathrm{var}(\langle\!\langle e_1 \colon \alpha \rangle\!\rangle) = \mathrm{var}(e_1) \cap \{\alpha\}$. The case $\beta = \alpha$ is impossible because $\alpha \notin \mathrm{var}(\Gamma)$. If $\beta \in \mathrm{var}(e_1)$, then $\beta\theta_1 = \beta$. If $\beta \in \mathrm{var}_{\preccurlyeq}(D_1)$, then $\beta\theta_1$ is static.
Note that $\check{\theta} = \hat{\theta} \cup \theta_1' \cup \tilde{\theta}_1$. Therefore we have:

$$\forall\alpha.\ \ \alpha\theta_1\check{\theta} \preccurlyeq \alpha\check{\theta} \qquad \forall\alpha.\ \ \alpha\theta_1 \text{ static} \implies \alpha\theta_1\check{\theta} = \alpha\check{\theta}$$

We have $\Gamma\check{\theta} = \Gamma\theta_1\check{\theta}$ because, for every $\alpha \in \mathrm{var}(\Gamma)$, $\alpha\theta_1$ is static.
We show $\mathfrak{U}_2 \sharp \Gamma'$. We already have $\mathfrak{U}_2 \sharp \Gamma$. It remains to show that the variables of $\forall\vec{\alpha},\vec{\gamma}.\, \alpha\theta_1$ are not in $\mathfrak{U}_2$, which is true because all these variables are either α or variables in $\mathrm{var}(\theta_1)$, and $\mathrm{var}(\theta_1) \subseteq \mathrm{var}(D_1) \cup \mathfrak{U}_3$.
We show $\mathrm{static}(\check{\theta}, \Gamma')$. We have $\mathrm{static}(\check{\theta}, \Gamma)$ since $\check{\theta}$ and θ are equal on $\mathrm{var}(\Gamma)$. We must show that, for every variable $\beta \in \mathrm{var}(\forall\vec{\alpha},\vec{\gamma}.\, \alpha\theta_1)$, $\beta\check{\theta}$ is a static type. We have $\beta \in \mathrm{var}(\alpha\theta_1) \setminus (\vec{\alpha} \cup \vec{\gamma})$. By definition of $\vec{\gamma}$, we have $\beta \in \mathrm{var}(\Gamma\theta_1) \cup \mathrm{var}(e_1)$. If $\beta \in \mathrm{var}(\Gamma\theta_1)$, then there exists a $\gamma \in \mathrm{var}(\Gamma)$ such that $\beta \in \mathrm{var}(\gamma\theta_1)$; since $\gamma\check{\theta}$ is static and $\gamma\check{\theta} = \gamma\theta_1\check{\theta}$, $\gamma\theta_1\check{\theta}$ is static; therefore, $\beta\check{\theta}$ is static as well. If $\beta \in \mathrm{var}(e_1)$, then $\beta \in \Delta$ and $\beta\check{\theta} = \beta$.
We show $\mathrm{static}(\check{\theta}, \mathrm{var}(D_1)\theta_1)$. Consider $\gamma \in \mathrm{var}(D_1)\theta_1$. By Proposition A.11, $\gamma \in \mathrm{var}_{\preccurlyeq}(D_1)\theta_1 \cup \Delta \cup \vec{\alpha}$. If $\gamma \in \Delta \cup \vec{\alpha}$, we have $\gamma\check{\theta} = \gamma$. If $\gamma \in \mathrm{var}_{\preccurlyeq}(D_1)\theta_1$, there exists $\gamma' \in \mathrm{var}_{\preccurlyeq}(D_1)$ such that $\gamma \in \mathrm{var}(\gamma'\theta_1)$. We know that $\gamma'\check{\theta}$ is static. Since $\gamma'\theta_1$ is static too, we have $\gamma'\theta_1\check{\theta} = \gamma'\check{\theta}$. This implies that $\gamma\check{\theta}$ must be static.
Now we show $\Gamma'\check{\theta} \vdash e_2\colon t\check{\theta}$. We apply Lemma A.4 by showing $\Gamma'\check{\theta} \trianglerighteq (\Gamma\theta, x\colon \forall\vec{\alpha},\vec{\beta}.\, \tau_1)$. Since $\Gamma\check{\theta} = \Gamma\theta$, we must only show $(\forall\vec{\alpha},\vec{\gamma}.\, \alpha\theta_1)\check{\theta} \trianglerighteq \forall\vec{\alpha},\vec{\beta}.\, \tau_1$. Note that $\alpha\check{\theta} = \tau_1$ and $\alpha\theta_1\check{\theta} \preccurlyeq \alpha\check{\theta}$. Therefore, we have $\forall\vec{\alpha},\vec{\beta}.\, \alpha\theta_1\check{\theta} \trianglerighteq \forall\vec{\alpha},\vec{\beta}.\, \tau_1$. Since $\trianglerighteq$ is transitive, we can conclude by showing $(\forall\vec{\alpha},\vec{\gamma}.\, \alpha\theta_1)\check{\theta} \trianglerighteq \forall\vec{\alpha},\vec{\beta}.\, \alpha\theta_1\check{\theta}$. We choose fresh variables $\vec{\alpha}_1,\vec{\gamma}_1$ (ensuring $\vec{\alpha}_1,\vec{\gamma}_1 \sharp \check{\theta}$) and let $\rho = \{\vec{\alpha} := \vec{\alpha}_1\} \cup \{\vec{\gamma} := \vec{\gamma}_1\}$; then $(\forall\vec{\alpha},\vec{\gamma}.\, \alpha\theta_1)\check{\theta} = \forall\vec{\alpha}_1,\vec{\gamma}_1.\, \alpha\theta_1\rho\check{\theta}$. To show $\forall\vec{\alpha}_1,\vec{\gamma}_1.\, \alpha\theta_1\rho\check{\theta} \trianglerighteq \forall\vec{\alpha},\vec{\beta}.\, \alpha\theta_1\check{\theta}$, we consider an arbitrary instance $\alpha\theta_1\check{\theta}\tilde{\theta}$ of $\forall\vec{\alpha},\vec{\beta}.\, \alpha\theta_1\check{\theta}$, with $\tilde{\theta} : \vec{\alpha},\vec{\beta} \to \mathcal{T}_t$. We choose the instance $\alpha\theta_1\rho\check{\theta}\tilde{\theta}'$ of $\forall\vec{\alpha}_1,\vec{\gamma}_1.\, \alpha\theta_1\rho\check{\theta}$, with $\tilde{\theta}' = \{\vec{\alpha}_1 := \vec{\alpha}\tilde{\theta}\} \cup \{\vec{\gamma}_1 := \vec{\gamma}\check{\theta}\tilde{\theta}\}$. We must show that $\tilde{\theta}'$ is a valid instantiation. It has the correct domain, but it remains to show that $\vec{\alpha}\tilde{\theta}$ and $\vec{\gamma}\check{\theta}\tilde{\theta}$ are static. For $\vec{\alpha}\tilde{\theta}$, the result is immediate. If $\gamma \in \vec{\gamma}$, instead, we must show that $\gamma\check{\theta}\tilde{\theta}$ is static. We have $\gamma \in \mathrm{var}(\alpha\theta_1)$. By Lemma A.25, we have $\alpha \in \mathrm{var}(D_1)$. Hence, $\gamma \in \mathrm{var}(D_1)\theta_1$. We have already shown $\mathrm{static}(\check{\theta}, \mathrm{var}(D_1)\theta_1)$. Hence, $\gamma\check{\theta}$ is static; since $\tilde{\theta}$ is static, $\gamma\check{\theta}\tilde{\theta}$ is static too. Now, we must show $\alpha\theta_1\rho\check{\theta}\tilde{\theta}' \preccurlyeq \alpha\theta_1\check{\theta}\tilde{\theta}$; actually, we show that the two types are equal. Consider $\beta \in \mathrm{var}(\alpha\theta_1)$: we must show $\beta\rho\check{\theta}\tilde{\theta}' = \beta\check{\theta}\tilde{\theta}$.

- If $\beta \in \mathrm{dom}(\rho)$, then $\beta\rho\check{\theta}\tilde{\theta}' = \beta\rho\tilde{\theta}'$. In particular, if $\beta \in \vec{\alpha}$, then $\beta\rho\check{\theta}\tilde{\theta}' = \beta\tilde{\theta} = \beta\check{\theta}\tilde{\theta}$ (because $\beta\check{\theta} = \beta$ since $\check{\theta}$ is not defined on $\vec{\alpha}$). If $\beta \in \vec{\gamma}$, then $\beta\rho\check{\theta}\tilde{\theta}' = \beta\check{\theta}\tilde{\theta}$.
- If $\beta \notin \mathrm{dom}(\rho)$, then $\beta\rho\check{\theta}\tilde{\theta}' = \beta\check{\theta}$. Since $\beta \in \mathrm{var}(\alpha\theta_1)$, necessarily $\beta \in \mathrm{var}(\Gamma\theta_1) \cup \mathrm{var}(e_1)$. If $\beta \in \mathrm{var}(\Gamma\theta_1)$, then $\mathrm{var}(\beta\check{\theta}) \subseteq \mathrm{var}(\beta\theta_1\check{\theta}) = \mathrm{var}(\Gamma\theta)$; but then, since $\mathrm{dom}(\tilde{\theta}) \sharp \Gamma\theta$, we have $\beta\check{\theta}\tilde{\theta} = \beta\check{\theta}$. If $\beta \in \mathrm{var}(e_1)$, since $\beta \notin \alpha$, we have $\beta \in \Delta$ and therefore $\beta\check{\theta}\theta = \beta = \beta\check{\theta}$.

We apply the IH using the premises:

$$\Gamma'\check{\theta} \vdash e_2\colon t\check{\theta} \qquad \mathrm{static}(\check{\theta}, \Gamma') \qquad \mathrm{dom}(\check{\theta}) \sharp \Delta \supseteq \mathrm{var}(e_2) \qquad \mathfrak{U}_2 \sharp \Delta, t, \Gamma', \mathrm{dom}(\check{\theta})$$

We derive:

$$\Gamma'; \Delta \vdash \langle\!\langle e_2 \colon t \rangle\!\rangle \rightsquigarrow D_2 \mid \overline{\alpha}_2 \qquad \breve{\theta} \cup \theta'_2 \Vdash_\Delta D_2 \qquad \mathsf{dom}(\theta'_2) \subseteq \overline{\alpha}_2 \subseteq \mathfrak{U}_2$$

We show $\vec{\alpha} \sharp \Gamma\theta_1$ by contradiction. Assume that there exists an $\alpha \in \vec{\alpha}$ such that $\alpha \in \mathsf{var}(\Gamma\theta_1)$. Then, since $\breve{\theta}$ is not defined on $\vec{\alpha}$, we would have $\alpha \in \mathsf{var}(\Gamma\theta_1\breve{\theta})$. But $\Gamma\theta_1\breve{\theta} = \Gamma\breve{\theta} = \Gamma\theta$. Then, we would have $\alpha \in \mathsf{var}(\Gamma\theta)$, which is impossible.
From the premises

$$\Gamma; \Delta \cup \vec{\alpha} \vdash \langle\!\langle e_1 \colon \alpha \rangle\!\rangle \rightsquigarrow D_1 \mid \overline{\alpha}_1 \qquad (\Gamma, x \colon \forall \vec{\alpha}, \vec{\gamma}.\, \alpha\theta_1); \Delta \vdash \langle\!\langle e_2 \colon t \rangle\!\rangle \rightsquigarrow D_2 \mid \overline{\alpha}_2$$
$$\theta_1 \in \mathsf{solve}_{\Delta \cup \vec{\alpha}}(D_1) \qquad \vec{\alpha} \sharp \Gamma\theta_1 \qquad \vec{\gamma} = \mathsf{var}(\alpha\theta_1) \setminus (\mathsf{var}(\Gamma\theta_1) \cup \vec{\alpha} \cup (\mathsf{var}(e_1) \setminus \vec{\alpha}))$$

we derive

$$\Gamma; \Delta \vdash \langle\!\langle e \colon t \rangle\!\rangle \rightsquigarrow D_2 \cup \mathsf{equiv}(\theta_1, D_1) \mid \overline{\alpha}$$

where $\overline{\alpha} = \{\alpha\} \cup \vec{\alpha} \cup \overline{\alpha}_1 \cup \overline{\alpha}_2 \cup (\mathsf{var}(\theta_1) \setminus \mathsf{var}(D_1)) \subseteq \mathfrak{U}$.
Let $\theta' = \{\alpha \coloneqq \tau_1\} \cup \theta'_1 \cup \tilde{\theta}_1 \cup \theta'_2$. We have $\mathsf{dom}(\theta') \subseteq \overline{\alpha} \subseteq \mathfrak{U}$.
It remains to prove that $\theta \cup \theta' \Vdash_\Delta D_2 \cup \mathsf{equiv}(\theta_1, D_1)$. Note that $\theta \cup \theta' = \breve{\theta} \cup \theta'_2$. Therefore, we have $\theta \cup \theta' \Vdash_\Delta D_2$. We show that $\theta \cup \theta'$ solves $\mathsf{equiv}(\theta_1, D_1)$.

- When $\beta \in \mathsf{var}_{\gtrless}(D_1)$, we must show that $\beta(\theta \cup \theta')$ is a static type. Note that, since $\hat{\theta} \cup \theta'_1 \Vdash_{\Delta \cup \vec{\alpha}} D_1$, we know $\beta(\hat{\theta} \cup \theta'_1)$ is static. This gives the result we need since $\theta \cup \theta' = (\hat{\theta} \cup \theta'_1) \cup (\tilde{\theta}_1 \cup \theta'_2)$ and $\mathsf{dom}(\tilde{\theta}_1 \cup \theta'_2) \sharp \mathsf{var}(D_1)$.
- When $\beta \in \mathsf{var}(D_1)\theta_1$, we must show that $\beta(\theta \cup \theta')$ is a static type. We have shown $\mathsf{static}(\breve{\theta}, \mathsf{var}(D_1)\theta_1)$. This is sufficient because $\theta \cup \theta' = \breve{\theta} \cup \theta'_2$ and $\mathsf{dom}(\theta'_2) \sharp \mathsf{var}(D_1) \cup \mathsf{var}(\theta_1)$.
- When $\beta \in \mathsf{dom}(\theta_1)$ and $\beta\theta_1$ is static, we must show $\beta(\theta \cup \theta') = \beta\theta_1(\theta \cup \theta')$. We have $\beta\breve{\theta} = \beta\theta_1\breve{\theta}$, which gives the result we need since $\breve{\theta}$ and $\theta \cup \theta'$ differ only on variables outside $\mathsf{dom}(\theta_1)$ and $\mathsf{var}(\theta_1)$. □

B FULL DEFINITIONS AND RESULTS

B.1 Type Frames

Let $\mathcal{V}$ be a countable set of *type variables*, which we partition into two sets $\mathcal{V}^\alpha$ (ranged over by α) and $\mathcal{V}^X$ (ranged over by X). In this section, α and X variables are treated identically; we introduce two separate metavariables because we will need to distinguish them later.

Let $\mathcal{C}$ be a set of *language constants* (ranged over by c) and $\mathcal{B}$ a set of *basic types* (ranged over by b). We assume that there exists a function $\mathbb{B} : \mathcal{B} \to \mathcal{P}(\mathcal{C})$. For two basic types $b_1 \neq b_2$, $\mathbb{B}(b_1)$ and $\mathbb{B}(b_2)$ might have a non-empty intersection. We assume that there exists a basic type $\mathbb{1}_\mathcal{B} \in \mathcal{B}$ such that $\mathbb{B}(\mathbb{1}_\mathcal{B}) = \mathcal{C}$. We also assume that there exists a function $b_{(\cdot)} : \mathcal{C} \to \mathcal{B}$ which assigns a basic type b_c to every constant c, such that $c \in \mathbb{B}(b_c)$.

Definition B.1 (Type frames). *The set $\mathcal{T}_T$ of* type frames *is the set of terms T produced coinductively by the following grammar*

$$T ::= \alpha \mid X \mid b \mid T \times T \mid T \to T \mid T \vee T \mid \neg T \mid \mathbb{0}$$

and that satisfy the following conditions:

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

We introduce the following abbreviations:

$$T_1 \wedge T_2 \stackrel{\text{def}}{=} \neg(\neg T_1 \vee \neg T_2) \qquad T_1 \setminus T_2 \stackrel{\text{def}}{=} T_1 \wedge (\neg T_2) \qquad \mathbb{1} \stackrel{\text{def}}{=} \neg\mathbb{0}\,.$$

We refer to type frames also as *types* when no ambiguity is possible. We refer to b, $\times$, and $\to$ as *type constructors* and to $\vee$, $\wedge$, $\neg$, and $\setminus$ as *type connectives*.

The condition on infinite branches bars out ill-formed types such as $T = T \vee T$ (which does not carry any information about the set denoted by the type) or $T = \neg T$ (which cannot represent any set). It also ensures that the binary relation $\triangleright \subseteq \mathcal{T}_T^2$ defined by $T_1 \vee T_2 \triangleright T_i$, $\neg T \triangleright T$ is Noetherian (that is, strongly normalizing). This gives an induction principle on $\mathcal{T}_T$ that we will use without any further reference to the relation.

Let T be a type frame. We use $\mathrm{var}(T)$ to denote the set of type variables (both α and X) occurring in T. We write $\mathrm{var}_\alpha(T)$ for $\mathrm{var}(T) \cap \mathcal{V}^\alpha$ and $\mathrm{var}_X(T)$ for $\mathrm{var}(T) \cap \mathcal{V}^X$. We say that T is *ground* or *closed* if and only if $\mathrm{var}(T)$ is empty.

B.2 Semantic Subtyping for Type Frames

Let Ω be a symbol that is not in C.

Definition B.2 (Interpretation domain). *The* interpretation domain $\mathcal{D}$ *is the set of finite terms* d *produced inductively by the following grammar*

$$\begin{aligned} d &::= c^L \mid (d, d)^L \mid \{(d, d_\Omega), \ldots, (d, d_\Omega)\}^L \\ d_\Omega &::= d \mid \Omega \end{aligned}$$

where L *ranges over* $\mathcal{P}_{\mathrm{fin}}(\mathcal{V})$ *(that is, on finite sets of type variables).*

We write $\mathrm{tags}(d)$ for the outermost set of labels in d, that is,

$$\mathrm{tags}(c^L) = \mathrm{tags}((d_1, d_2)^L) = \mathrm{tags}(\{(d_1, d_1'), \ldots, (d_n, d_n')\}^L) = L\,.$$

Definition B.3 (Set-theoretic interpretation). *We define a binary predicate* $(d : T)$*, where* $d \in \mathcal{D}$ *and* $T \in \mathcal{T}_T$*, by induction on the pair* (d, T) *ordered lexicographically (we use the induction principle on types mentioned earlier). The predicate is defined as follows:*

$$\begin{aligned} (d : \alpha) &= \alpha \in \mathrm{tags}(d) \\ (d : X) &= X \in \mathrm{tags}(d) \\ (c^L : b) &= c \in \mathbb{B}(b) \\ ((d_1, d_2)^L : T_1 \times T_2) &= (d_1 : T_1) \wedge (d_2 : T_2) \\ (\{(d_1, d_1'), \ldots, (d_n, d_n')\}^L : T_1 \to T_2) &= \forall i \in \{1, \ldots, n\}.\, (d_i : T_1) \implies (d_i' : T_2) \\ (d : T_1 \vee T_2) &= (d : T_1) \vee (d : T_2) \\ (d : \neg T) &= \neg (d : T) \\ (d : T) &= \mathsf{false} \qquad \textit{otherwise} \end{aligned}$$

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

$$[\![T]\!] \stackrel{\mathrm{def}}{=} \{\, d \in \mathcal{D} \mid (d : T) \,\}\,.$$

We have the following equalities:

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

Definition B.4 (Subtyping). *The* subtyping *relation $\leq_T$ between types is defined by*

$$T_1 \leq_T T_2 \iff^{\text{def}} [\![T_1]\!] \subseteq [\![T_2]\!] \,.$$

We write $\simeq_T$ for the subtype equivalence *relation defined as*

$$T_1 \simeq_T T_2 \iff^{\text{def}} (T_1 \leq_T T_2) \wedge (T_2 \leq_T T_1) \,.$$

Definition B.5 (Type substitutions). *A type substitution θ is a mapping from type variables to type frames. We write $\{\alpha := T\}$ or $\{X := T\}$ for the type substitution mapping α or X, respectively, to T. We write* $\mathsf{dom}(\theta)$ *for the domain of the substitution θ.*

We write $T\theta$ for the application *of the substitution θ to the type frame T, which is defined coinductively by the following equations.*

$$\begin{aligned}
\alpha\theta &= \begin{cases} \theta(\alpha) & \text{if } \alpha \in \mathsf{dom}(\theta) \\ \alpha & \text{otherwise} \end{cases} \\
X\theta &= \begin{cases} \theta(X) & \text{if } X \in \mathsf{dom}(\theta) \\ X & \text{otherwise} \end{cases} \\
b\theta &= b \\
(T_1 \times T_2)\theta &= (T_1\theta) \times (T_2\theta) \\
(T_1 \to T_2)\theta &= (T_1\theta) \to (T_2\theta) \\
(T_1 \vee T_2)\theta &= (T_1\theta) \vee (T_2\theta) \\
(\neg T)\theta &= \neg(T\theta) \\
\mathbb{0}\theta &= \mathbb{0}
\end{aligned}$$

Proposition B.6. *If $T_1 \leq_T T_2$, then $T_1\theta \leq_T T_2\theta$ for every type substitution θ.*

We refer to a type frame of the form b, $T_1 \times T_2$, or $T_1 \to T_2$ as an *atom*. We write $\mathcal{A}_{\mathsf{basic}}$, $\mathcal{A}_{\mathsf{prod}}$, and $\mathcal{A}_{\mathsf{fun}}$ for the set of type frames of the forms b, $T_1 \times T_2$, and $T_1 \to T_2$, respectively.

Lemma B.7. *Let P, N be two finite subsets of $\mathcal{A}_{\mathsf{prod}}$. Then:*

$$\bigwedge_{T_1 \times T_2 \in P} T_1 \times T_2 \leq_T \bigvee_{T_1 \times T_2 \in N} T_1 \times T_2 \iff$$
$$\forall N' \subseteq N.\, \Big(\bigwedge_{T_1 \times T_2 \in P} T_1 \leq_T \bigvee_{T_1 \times T_2 \in N'} T_1 \Big) \vee \Big(\bigwedge_{T_1 \times T_2 \in P} T_2 \leq_T \bigvee_{T_1 \times T_2 \in N \setminus N'} T_2 \Big)$$

(with the convention $\bigwedge_{T \in \varnothing} T = \mathbb{1} \times \mathbb{1}$).

Lemma B.8. *Let P, N be two finite subsets of $\mathcal{A}_{\mathsf{fun}}$. Then:*

$$\bigwedge_{T_1\to T_2\in P} T_1\to T_2 \le_T \bigvee_{T_1\to T_2\in N} T_1\to T_2 \iff \exists(\overline{T}_1\to\overline{T}_2)\in N.$$
$$\left(\overline{T}_1 \le_T \bigvee_{T_1\to T_2\in P} T_1\right) \wedge \left(\forall P'\subsetneq P.\left(\overline{T}_1 \le_T \bigvee_{T_1\to T_2\in P'} T_1\right)\vee\left(\bigwedge_{T_1\to T_2\in P\setminus P'} T_2 \le_T \overline{T}_2\right)\right)$$

(with the convention $\bigwedge_{T\in\varnothing} T = \mathbb{0}\to\mathbb{1}$).

Lemma B.9. *Let P, N be two finite subsets of $\mathcal{A}_{\mathsf{basic}}\cup\mathcal{A}_{\mathsf{prod}}\cup\mathcal{A}_{\mathsf{fun}}$ and P', N' be two finite subsets of $\mathcal{V}$. If $P'\cap N' = \varnothing$, then*

$$\bigwedge_{a\in P} a\wedge\bigwedge_{a\in N}\neg a\wedge\bigwedge_{a\in P'} a\wedge\bigwedge_{a\in N'}\neg a \le_T \mathbb{0} \iff \bigwedge_{a\in P} a\wedge\bigwedge_{a\in N}\neg a\le_T\mathbb{0}$$

Proof. The implication $\Leftarrow$ is trivial. We prove the other direction by contrapositive. Assume that the subtyping relation on the right does not hold. Then, we have

$$d\in[\![\textstyle\bigwedge_{a\in P} a\wedge\bigwedge_{a\in N}\neg a]\!]\,.$$

Therefore $d\in[\![a]\!]$ holds for all $a\in P$ and $d\in\mathcal{D}\setminus[\![N]\!]$ holds for all $a\in N$.

Note that every $a\in P\cup N$ is of the forms b, $T_1\times T_2$, or $T_1\to T_2$. For such types, if $d\in[\![a]\!]$, then every d' that differs from d only for its outermost set of tags satisfies $d'\in[\![a]\!]$.

We consider the domain element $\overline{d}$ which is d changed to have $\mathsf{tags}(\overline{d}) = P'$. By construction, it is in the interpretation of all variables in P' and in none of the interpretations of the variables in N'. Hence, we have

$$\overline{d}\in[\![\textstyle\bigwedge_{a\in P} a\wedge\bigwedge_{a\in N}\neg a\wedge\bigwedge_{a\in P'} a\wedge\bigwedge_{a\in N'}\neg a]\!]\,.$$ □

Given two type substitutions θ_1 and θ_2, we write $\theta_1\le_T\theta_2$ when, for every A, $A\theta_1\le_T A\theta_2$.

When $\overline{A}\subseteq\mathcal{V}$, we define $\theta|_{\overline{A}}$ as the type substitution such that $A\theta|_{\overline{A}} = A\theta$ if $A\in\overline{A}$ and $A\theta|_{\overline{A}} = A$ otherwise.

Proposition B.10.

$$\forall T,\theta_1,\theta_2.\quad \left.\begin{array}{l}\theta_1|_{\mathsf{var}^{\mathsf{cov}}(T)}\le_T\theta_2|_{\mathsf{var}^{\mathsf{cov}}(T)}\\ \theta_2|_{\mathsf{var}^{\mathsf{cnt}}(T)}\le_T\theta_1|_{\mathsf{var}^{\mathsf{cnt}}(T)}\end{array}\right\}\implies T\theta_1\le_T T\theta_2$$

Proof. We define

$$P(T,\theta_1,\theta_2)\overset{\text{def}}{\iff}\left(\theta_1|_{\mathsf{var}^{\mathsf{cov}}(T)}\le_T\theta_2|_{\mathsf{var}^{\mathsf{cov}}(T)}\right)\text{ and }\left(\theta_2|_{\mathsf{var}^{\mathsf{cnt}}(T)}\le_T\theta_1|_{\mathsf{var}^{\mathsf{cnt}}(T)}\right)$$

and note that the following hold

$$\begin{aligned}P(A,\theta_1,\theta_2)&\implies A\theta_1\le_T A\theta_2\\ P(T_1\times T_2,\theta_1,\theta_2)&\implies P(T_1,\theta_1,\theta_2)\text{ and }P(T_2,\theta_1,\theta_2)\\ P(T_1\to T_2,\theta_1,\theta_2)&\implies P(T_1,\theta_2,\theta_1)\text{ and }P(T_2,\theta_1,\theta_2)\\ P(T_1\vee T_2,\theta_1,\theta_2)&\implies P(T_1,\theta_1,\theta_2)\text{ and }P(T_2,\theta_1,\theta_2)\\ P(\neg T',\theta_1,\theta_2)&\implies P(T',\theta_2,\theta_1)\end{aligned}$$

We show the following result (which implies the statement)

$$\forall\theta_1,\theta_2,d,T.\quad\left.\begin{array}{l}P(T,\theta_1,\theta_2)\\ (d:T\theta_1)\end{array}\right\}\implies(d:T\theta_2)$$

by induction on (d,T).

Case: $T = b$ or $T = \mathbb{0}$ Trivial, since $T\theta_1 = T = T\theta_2$.
Case: $T = A$ We have $A\theta_1 \leq_T A\theta_2$ and $(d : A\theta_1)$, which implies $(d : A\theta_2)$.
Case: $T = T_1 \times T_2$
We have $T\theta_1 = (T_1\theta_1) \times (T_2\theta_1)$ and $T\theta_2 = (T_1\theta_2) \times (T_2\theta_2)$.
Since $(d : T\theta_1)$, we have $d = (d_1, d_2)$ and $(d_i : T_i\theta_1)$.
Since $P(T_i, \theta_1, \theta_2)$ holds for both i, by IH we have $(d_i : T_i\theta_2)$. Then, $(d : T\theta_2)$.
Case: $T = T_1 \to T_2$
We have $T\theta_1 = (T_1\theta_1) \to (T_2\theta_1)$ and $T\theta_2 = (T_1\theta_2) \to (T_2\theta_2)$.
Since $(d : T\theta_1)$, we have $d = \{ (d_i, d_i') \mid i \in I \}$ and $\forall i \in I.\ (d_i : T_1\theta_1) \implies (d_i' : T_2\theta_1)$.
We have $P(T_1, \theta_2, \theta_1)$ and $P(T_2, \theta_1, \theta_2)$.
For every d_i such that $(d_i : T_1\theta_2)$, by IH we have $(d_i : T_1\theta_1)$, therefore $(d_i' : T_2\theta_1)$, and, by IH, $(d_i' : T_2\theta_2)$. Therefore, $\forall i \in I.\ (d_i : T_1\theta_2) \implies (d_i' : T_2\theta_2)$, and hence $(d : T\theta_2)$.
Case: $T = T_1 \vee T_2$
We have either $(d : T_1\theta_1)$ or $(d : T_2\theta_1)$. Therefore, since $P(T_i, \theta_1, \theta_2)$ holds for both i, by IH we have either $(d : T_1\theta_2)$ or $(d : T_2\theta_2)$, and hence $(d : T\theta_2)$.
Case: $T = \neg T'$
We have $\neg(d : T'\theta_1)$. Since $P(T', \theta_2, \theta_1)$, by IH $(d : T'\theta_2) \implies (d : T'\theta_1)$. Therefore, by contrapositive, we have $\neg(d : T'\theta_2)$, hence $(d : \neg T'\theta_2)$. □

B.3 Static and Gradual Types

We define the set of *static types* $\mathcal{T}_t$, ranged over by t, as

$$\mathcal{T}_t = \{ T \in \mathcal{T}_T \mid \mathsf{var}_X(T) = \varnothing \}$$

that is, the set of T containing no X variable. The types in $\mathcal{T}_t$ are therefore generated by the grammar

$$t ::= \alpha \mid b \mid t \times t \mid t \to t \mid t \vee t \mid \neg t \mid \mathbb{0}$$

interpreted coinductively, with the same regularity and contractivity conditions as in Definition B.1.

Definition B.11 (Gradual types). *The set $\mathcal{T}_\tau$ of* gradual types *is the set of terms τ produced coinductively by the following grammar*

$$\tau ::= ? \mid \alpha \mid b \mid \tau \times \tau \mid \tau \to \tau \mid \tau \vee \tau \mid \neg\tau \mid \mathbb{0}$$

and that satisfy the following conditions:

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

For both static and gradual types we use the same abbreviations and the same notation introduced for type frames in general. On static types, we have the relations $\leq_T$ and $\simeq_T$ since static types are just a subset of type frames.

We extend the definition of application of a type substitution to gradual types by defining $?\theta = ?$.

When V is a set of type variables and T a set of types, we write $\theta : V \to T$ to mean $\mathsf{dom}(\theta) = V$ and to restrict which types can be in the range of θ. For instance, we write $\theta_1 : \mathcal{V}^X \to \mathcal{T}_t$ if θ maps X variables to static types and $\theta_2 : \mathcal{V}^X \to \mathcal{T}_\tau$ if θ_2 maps X variables to gradual types.

B.4 Discriminations of Gradual Types

Polarity. Given a type frame and an occurrence of a variable in it, we speak of that occurrence as being either in *positive* or in *negative* position, and we refer to this property of the occurrence as its *polarity*. Polarity is defined as follows. The root of the type frame is in positive position. In a

type frame T of the form $T_1 \times T_2$, $T_1 \vee T_2$ or $T_1 \to T_2$, both T_1 and T_2 have the same polarity as T. In a type frame $T = \neg T'$, T' has opposite polarity with respect to T.

We write $\mathrm{var}^+_X(T)$ for the set of variables in $\mathcal{V}^X$ that have an occurrence in positive position in T and $\mathrm{var}^-_X(T)$ for the set of variables in $\mathcal{V}^X$ that have an occurrence in negative position in T.

We say that T is *polarized* if each variable $X \in \mathcal{V}^X$ occurs in T always with the same polarity; that is, if $\mathrm{var}^+_X(T) \cap \mathrm{var}^-_X(T) = \varnothing$. We write $\mathcal{T}^{\mathsf{pol}}_T$ for the set of polarized type frames.

We also use these notions for type variables as well as for frame variables. We write $\mathrm{var}^+(T)$ and $\mathrm{var}^-(T)$ for the set of type and frame variables that have, respectively, positive or negative occurrences in T. We write $\mathrm{var}^\pm(T)$ for the set of type and frame variables that have both positive and negative occurrences in T.

Variance. Similarly, given a type frame and an occurrence of a variable in it, we also speak of that occurrence as being either in *covariant* or in *contravariant* position. The definition is identical to the definition of *polarity*, except that in a type frame $T = T_1 \to T_2$, T_1 has opposite variance with respect to T, and T_2 has the same variance as T.

We write $\mathrm{var}^{\mathsf{cov}}_X(T)$ for the set of variables in $\mathcal{V}^X$ that have an occurrence in covariant position in T and $\mathrm{var}^{\mathsf{cnt}}_X(T)$ for the set of variables in $\mathcal{V}^X$ that have an occurrence in contravariant position in T.

We say that T is *variance-polarized* if each variable $X \in \mathcal{V}^X$ occurs in T always with the same variance; that is, if $\mathrm{var}^{\mathsf{cov}}_X(T) \cap \mathrm{var}^{\mathsf{cnt}}_X(T) = \varnothing$. We write $\mathcal{T}^{\mathsf{var}}_T$ for the set of variance-polarized type frames.

Parity. We also speak of the *parity* of an occurrence of a variable in a type frame. An occurrence is *even* if it appears to the left of an even number of arrows; it is odd otherwise. That is, the root of a type is even; in $T_1 \to T_2$, parity is flipped for T_1. We write $\mathrm{var}^{\mathsf{even}}_X(\cdot)$ and $\mathrm{var}^{\mathsf{odd}}_X(\cdot)$ for frame variables with even and off occurrences.

Note that any two of polarity, variance, and parity of an occurrence determine the third. Notably, covariant occurrences are either positive and even or negative and odd. Contravariant occurrences are either negative and even or positive and odd.

Given a type frame T, we write $T^\dagger$ for the gradual type τ obtained from T by replacing all variables in $\mathcal{V}^X$ with ?.

Definition B.12 (Discrimination of a gradual type). *Given a gradual type τ, the set $\star(\tau)$ of the* discriminations *of τ is defined by*

$$\star(\tau) = \{\, T \in \mathcal{T}^{pol}_T \mid T^\dagger = \tau \,\}\,.$$

In the following, we assume that X^1 and X^0 are two variables in $\mathcal{V}^X$. We will define subtyping on gradual types by replacing these variables for positive and negative occurrences (respectively) of ? in a gradual type in order to obtain a type frame.

We say that a type frame T is *strongly polarized* if $\mathrm{var}_X(T) \subseteq \{X^1, X^0\}$, $\mathrm{var}^+_X(T) \subseteq \{X^1\}$, and $\mathrm{var}^-_X(T) \subseteq \{X^0\}$. We say that it is *strongly negatively polarized* if $\mathrm{var}_X(T) \subseteq \{X^1, X^0\}$, $\mathrm{var}^+_X(T) \subseteq \{X^0\}$, and $\mathrm{var}^-_X(T) \subseteq \{X^1\}$. We write $\mathcal{T}^{\mathsf{pol1}}_T$ and $\mathcal{T}^{\mathsf{pol0}}_T$ for the sets of strongly polarized type frames and of strongly negatively polarized type frames, respectively.

Similarly, we define and write $\mathcal{T}^{\mathsf{var1}}_T$ and $\mathcal{T}^{\mathsf{var0}}_T$ for variance-polarized type frames verifying the same conditions.

Given a gradual type τ, we define its *positive discrimination* $\tau^\oplus$ as the unique element of $\star(\tau) \cap \mathcal{T}^{\mathsf{pol1}}_T$ and its *negative discrimination* $\tau^\ominus$ as the unique element of $\star(\tau) \cap \mathcal{T}^{\mathsf{pol0}}_T$. We have the following

equalities:

$$\begin{array}{rcl@{\qquad}rcl} ?^{\oplus} &=& X^1 & \alpha^{\ominus} &=& X^0 \\ \alpha^{\oplus} &=& \alpha & \alpha^{\ominus} &=& \alpha \\ b^{\oplus} &=& b & b^{\ominus} &=& b \\ (\tau_1 \times \tau_2)^{\oplus} &=& \tau_1^{\oplus} \times \tau_2^{\oplus} & (\tau_1 \times \tau_2)^{\ominus} &=& \tau_1^{\ominus} \times \tau_2^{\ominus} \\ (\tau_1 \to \tau_2)^{\oplus} &=& \tau_1^{\ominus} \to \tau_2^{\oplus} & (\tau_1 \to \tau_2)^{\ominus} &=& \tau_1^{\oplus} \to \tau_2^{\ominus} \\ (\tau_1 \vee \tau_2)^{\oplus} &=& \tau_1^{\oplus} \vee \tau_2^{\oplus} & (\tau_1 \vee \tau_2)^{\ominus} &=& \tau_1^{\ominus} \vee \tau_2^{\ominus} \\ \mathbb{0}^{\oplus} &=& \mathbb{0} & \mathbb{0}^{\ominus} &=& \mathbb{0} \\ (\neg\tau)^{\oplus} &=& \tau^{\ominus} & (\neg\tau)^{\ominus} &=& \tau^{\oplus} \end{array}$$

When $T \in \mathcal{T}_T^{\mathsf{pol1}}$, we have $(T^\dagger)^{\oplus} = T$. When $T \in \mathcal{T}_T^{\mathsf{pol0}}$, we have $(T^\dagger)^{\ominus} = T$.

Moreover, given a gradual type τ, we define its *covariant polarization* $\tau^{\circledwedge}$ as the unique element $T \in \mathcal{T}_T^{\mathsf{var1}}$ such that $T^\dagger = \tau$, and its *contravariant polarization* $\tau^{\circledvee}$ as the unique element $T \in \mathcal{T}_T^{\mathsf{var0}}$ verifying $T^\dagger = \tau$. We have the following equalities:

$$\begin{array}{rcl@{\qquad}rcl} ?^{\circledwedge} &=& X^1 & \alpha^{\circledvee} &=& X^0 \\ \alpha^{\circledwedge} &=& \alpha & \alpha^{\circledvee} &=& \alpha \\ b^{\circledwedge} &=& b & b^{\circledvee} &=& b \\ (\tau_1 \times \tau_2)^{\circledwedge} &=& \tau_1^{\circledwedge} \times \tau_2^{\circledwedge} & (\tau_1 \times \tau_2)^{\circledvee} &=& \tau_1^{\circledvee} \times \tau_2^{\circledvee} \\ (\tau_1 \to \tau_2)^{\circledwedge} &=& \tau_1^{\circledvee} \to \tau_2^{\circledwedge} & (\tau_1 \to \tau_2)^{\circledvee} &=& \tau_1^{\circledwedge} \to \tau_2^{\circledvee} \\ (\tau_1 \vee \tau_2)^{\circledwedge} &=& \tau_1^{\circledwedge} \vee \tau_2^{\circledwedge} & (\tau_1 \vee \tau_2)^{\circledvee} &=& \tau_1^{\circledvee} \vee \tau_2^{\circledvee} \\ \mathbb{0}^{\circledwedge} &=& \mathbb{0} & \mathbb{0}^{\circledvee} &=& \mathbb{0} \\ (\neg\tau)^{\circledwedge} &=& \tau^{\circledvee} & (\neg\tau)^{\circledvee} &=& \tau^{\circledwedge} \end{array}$$

We write $\mathsf{var}_X^{+\mathsf{cov}}(T)$ to denote the set of frame variables that have an occurrence in T that is both positive and covariant. Similarly, we use the notations $\mathsf{var}_X^{+\mathsf{cnt}}(T) \subseteq \{X^{+\vee}\}$, $\mathsf{var}_X^{-\mathsf{cov}}(T) \subseteq \{X^{-\wedge}\}$, and $\mathsf{var}_X^{-\mathsf{cnt}}(T) \subseteq \{X^{-\vee}\}$.

In the following, we assume that $X^{+\wedge}, X^{+\vee}, X^{-\wedge}$, and $X^{-\vee}$ are four distinguished variables in $\mathcal{V}^X$. Given a gradual type τ, we define $\tau^\bullet$ as the unique type frame T such that $T^\dagger = \tau$, that $\mathsf{var}_X^{+\mathsf{cov}}(T) \subseteq \{X^{+\wedge}\}$, that $\mathsf{var}_X^{+\mathsf{cnt}}(T) \subseteq \{X^{+\vee}\}$, that $\mathsf{var}_X^{-\mathsf{cov}}(T) \subseteq \{X^{-\wedge}\}$, and that $\mathsf{var}_X^{-\mathsf{cnt}}(T) \subseteq \{X^{-\vee}\}$.

B.5 Relations on Gradual Types

The *polarized discriminations* of a gradual type are defined as $\star^{\mathsf{pol}}(\tau) \stackrel{\text{def}}{=} \star(\tau) \cap \mathcal{T}_T^{\mathsf{pol}}$.

Definition B.13 (Subtyping on gradual types). *The* subtyping *relation* $\leq$ *between gradual types is defined by*

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

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

We write $\mathsf{var}(T_1, \ldots, T_n)$ for $\mathsf{var}(T_1) \cup \cdots \cup \mathsf{var}(T_n)$ (and similarly for var_X, var_X^+, etc.).

Lemma B.14. *Let* T *be a type frame with* $\mathsf{var}(T) = \{A_i \mid i \in I\}$. *There exists a type frame* T' *such that the four sets*

$$\begin{array}{ll} \mathsf{var}^{+\mathsf{cov}}(T') \subseteq \{A_i^{+\wedge} \mid i \in I\} & \mathsf{var}^{+\mathsf{cnt}}(T') \subseteq \{A_i^{+\vee} \mid i \in I\} \\ \mathsf{var}^{-\mathsf{cov}}(T') \subseteq \{A_i^{-\wedge} \mid i \in I\} & \mathsf{var}^{-\mathsf{cnt}}(T') \subseteq \{A_i^{-\vee} \mid i \in I\} \end{array}$$

are pairwise disjoint and that

$$T = T'\big(\{A_i^{+\wedge} \coloneqq A_i\}_{i\in I} \cup \{A_i^{+\vee} \coloneqq A_i\}_{i\in I} \cup \{A_i^{-\wedge} \coloneqq A_i\}_{i\in I} \cup \{A_i^{-\vee} \coloneqq A_i\}_{i\in I}\big)\ .$$

Proof. Clearly, T' is definable as a tree: it is the tree that coincides with T except on variables, and that, where T has a variable A_i, has one of $A_i^{+\wedge}$, $A_i^{+\vee}$, $A_i^{-\wedge}$, or $A_i^{-\vee}$ depending on the position of that occurrence of A_i. The tree T' is also clearly contractive and the sets of variables in different positions are disjoint.

For T' to be a type frame, it must also be regular. Since T is regular, it can be described by a finite system of equations

$$\begin{cases} x_1 = \bar{T}_1 \\ \quad \vdots \\ x_n = \bar{T}_n \end{cases}$$

such that every $\bar{T}_i$ is an inductively generated term of the grammar

$$\bar{T} ::= x \mid X \mid \alpha \mid b \mid \bar{T} \times \bar{T} \mid \bar{T} \to \bar{T} \mid \bar{T} \vee \bar{T} \mid \neg \bar{T} \mid \mathbb{0}$$

(x serves as a recursion variable) and that (reading the equations as a tree) $T = x_1$.

Then, T' can be defined as $x_1^{+\wedge}$ where

$$\begin{cases} x_1^{+\wedge} = f^{+\wedge}(\bar{T}_1) \\ x_1^{+\vee} = f^{+\vee}(\bar{T}_1) \\ x_1^{-\wedge} = f^{-\wedge}(\bar{T}_1) \\ x_1^{-\vee} = f^{-\vee}(\bar{T}_1) \\ \quad \vdots \\ x_n^{-\vee} = f^{-\vee}(\bar{T}_n) \end{cases}$$

and where (defining $\overline{+} = -$, $\overline{=} = +$, $\overline{\wedge} = \vee$, and $\overline{\vee} = \wedge$) $f^{pv}(\bar{T})$ is defined inductively as:

$$f^{pv}(x) = x^{pv} \qquad f^{pv}(X) = X^{pv} \qquad f^{pv}(\alpha) = \alpha^{pv}$$

$$f^{pv}(b) = b \qquad f^{pv}(\bar{T}_1 \times \bar{T}_2) = f^{pv}(\bar{T}_1) \times f^{pv}(\bar{T}_2) \qquad f^{pv}(\bar{T}_1 \to \bar{T}_2) = f^{p\overline{v}}(\bar{T}_1) \to f^{pv}(\bar{T}_2)$$

$$f^{pv}(\bar{T}_1 \vee \bar{T}_2) = f^{pv}(\bar{T}_1) \vee f^{pv}(\bar{T}_2) \qquad f^{pv}(\neg \bar{T}') = \neg f^{\overline{pv}}(T') \qquad f^{pv}(\mathbb{0}) = \mathbb{0}$$

At most $4n$ equations are needed to define T' (they could be less, since some x_i^{pv} could be unreachable from $x_1^{+\wedge}$). Therefore, T' is regular. □

Corollary B.15. *Let T be a type frame with $\mathrm{var}_X(T) = \{X_1, \ldots, X_n\}$. There exists a type frame T', with $\mathrm{var}_X^{\mathrm{cov}}(T') \subseteq \{X_1, \ldots, X_n\}$ disjoint from $\mathrm{var}_X^{\mathrm{cnt}}(T') \subseteq \{X_1', \ldots, X_n'\}$, such that $T = T'\{X_i' := X_i\}_{i=1}^n$.*

Proof. Consequence of Lemma B.14. We apply the lemma to find a type where type and frame variables are renamed according to their position (polarity and variance); then, we apply a substitution to unify the positions we do not want to distinguish. □

Corollary B.16. *Let T be a type frame with $\mathrm{var}_X(T) = \{X_1, \ldots, X_n\}$. There exists a type frame T', with $\mathrm{var}_X^{\mathrm{even}}(T') \subseteq \{X_1, \ldots, X_n\}$ disjoint from $\mathrm{var}_X^{\mathrm{odd}}(T') \subseteq \{X_1', \ldots, X_n'\}$, such that $T = T'\{X_i' := X_i\}_{i=1}^n$.*

Proof. Consequence of Lemma B.14, similarly to Corollary B.15. □

Corollary B.17. *Let τ be a gradual type with $\mathrm{var}(\tau) = \{\alpha_1, \ldots, \alpha_n\}$. There exists a gradual type τ', with $\mathrm{var}^+(\tau') \subseteq \{\alpha_1, \ldots, \alpha_n\}$ disjoint from $\mathrm{var}^-(\tau') \subseteq \{\alpha_1', \ldots, \alpha_n'\}$, such that $\tau = \tau'\{\alpha_i' := \alpha_i\}_{i=1}^n$.*

Proof. Consequence of Lemma B.14, similarly to Corollary B.15. We first choose a T such that $T^\dagger = \tau$; then, we apply the lemma and a substitution to unify the positions that we do not need to distinguish; finally, we apply $\dagger$ to obtain a gradual type. □

Lemma B.18.

$$\left.\begin{array}{l} T \not\leq_T \mathbb{0} \\ \text{either } \{X,Y\} \sharp \mathrm{var}^-_X(T) \text{ or } \{X,Y\} \sharp \mathrm{var}^+_X(T) \end{array}\right\} \implies T\{Y \coloneqq X\} \not\leq_T \mathbb{0}$$

Proof. We first give some auxiliary definitions.
Let σ range over the two symbols $\boxplus$ and $\boxminus$. We define $\overline{\sigma}$ as follows: $\overline{\boxplus} \stackrel{\text{def}}{=} \boxminus$ and $\overline{\boxminus} \stackrel{\text{def}}{=} \boxplus$.
Given a type frame T', we write $T' \vDash \boxplus$ if $\{X,Y\} \sharp \mathrm{var}^-_X(T)$ and $T' \vDash \boxminus$ if $\{X,Y\} \sharp \mathrm{var}^+_X(T)$.
Note that, for all T', T_1, and T_2, we have:

$$\begin{aligned} (\neg T' \vDash \sigma) &\implies (T' \vDash \overline{\sigma}) \\ (T_1 \vee T_2 \vDash \sigma) &\implies (T_1 \vDash \sigma) \wedge (T_2 \vDash \sigma) \\ (T_1 \times T_2 \vDash \sigma) &\implies (T_1 \vDash \sigma) \wedge (T_2 \vDash \sigma) \\ (T_1 \to T_2 \vDash \sigma) &\implies (T_1 \vDash \sigma) \wedge (T_2 \vDash \sigma) \end{aligned}$$

We define a function F^σ on domain element tags (finite sets of variables) as:

$$F^{\boxplus}(L) = \begin{cases} L \cup \{X,Y\} & \text{if } X \in L \text{ or } Y \in L \\ L & \text{otherwise} \end{cases} \qquad F^{\boxminus}(L) = \begin{cases} L \setminus \{X,Y\} & \text{if } X \notin L \text{ or } Y \notin L \\ L & \text{otherwise} \end{cases}$$

We also define F on domain elements as follows:

$$\begin{aligned} F^\sigma(c^L) &= c^{F^\sigma(L)} \\ F^\sigma((d_1,d_2)^L) &= (F^\sigma(d_1), F^\sigma(d_2))^{F^\sigma(L)} \\ F^\sigma(\{(d_1,d_1'),\ldots,(d_n,d_n')\}^L) &= \{(F^{\overline{\sigma}}(d_1), F^\sigma(d_1')),\ldots,(F^{\overline{\sigma}}(d_n), F^\sigma(d_n'))\}^{F^\sigma(L)} \\ F^\sigma(\Omega) &= \Omega \end{aligned}$$

We must show:

$$\left.\begin{array}{l} T \not\leq_T \mathbb{0} \\ \text{either } \{X,Y\} \sharp \mathrm{var}^-_X(T) \text{ or } \{X,Y\} \sharp \mathrm{var}^+_X(T) \end{array}\right\} \implies T\{Y \coloneqq X\} \not\leq_T \mathbb{0}$$

This can be restated as:

$$\left.\begin{array}{l} \exists d \in \mathcal{D}.\, (d : T) \\ \exists \sigma.\, T \vDash \sigma \end{array}\right\} \implies \exists d' \in \mathcal{D}.\, (d' : T\{Y \coloneqq X\})$$

We prove the following, stronger claim:

$$\forall d, T, \sigma. \quad T \vDash \sigma \implies \begin{cases} (d : T) \implies (F^\sigma(d) : T\{Y \coloneqq X\}) \\ \neg(d : T) \implies \neg(F^{\overline{\sigma}}(d) : T\{Y \coloneqq X\}) \end{cases}$$

by induction on the pair (d, T), ordered lexicographically. For a given d, T, and σ, we assume $T \vDash \sigma$ and proceed by case analysis on T and d.
Let $\theta = \{Y \coloneqq X\}$.

Case: $T = \alpha$

Since $\alpha\theta = \alpha$, we must show

$$(d : \alpha) \implies (F^\sigma(d) : \alpha) \qquad \neg(d : \alpha) \implies \neg(F^{\overline{\sigma}}(d) : \alpha)\,.$$

If $(d : \alpha)$, then $\alpha \in \mathrm{tags}(d)$ and also $\alpha \in \mathrm{tags}(F^{\sigma}(d))$. Likewise, if $d \notin [\![\alpha]\!]$, then $\alpha \notin \mathrm{tags}(d)$ and also $\alpha \notin \mathrm{tags}(F^{\overline{\sigma}}(d))$.

Case: $T = Z$, with $Z \neq X$ and $Z \neq Y$

Like the previous case.

Case: $T = X$

Since $X \in \mathrm{var}^{+}_{X}(X)$, we have $\sigma = \boxplus$.

We must show

$$(d : X) \implies (F^{\boxplus}(d) : X) \qquad \neg(d : X) \implies \neg(F^{\boxminus}(d) : X)\,.$$

If $(d : X)$, then $X \in \mathrm{tags}(d)$ and $X \in \mathrm{tags}(F^{\boxplus}(d))$. If $\neg(d : X)$, then $X \notin \mathrm{tags}(d)$ and $X \notin \mathrm{tags}(F^{\boxminus}(d))$.

Case: $T = Y$

Since $Y \in \mathrm{var}^{+}_{X}(Y)$, we have $\sigma = \boxplus$.

We must show

$$(d : Y) \implies (F^{\boxplus}(d) : X) \qquad \neg(d : Y) \implies \neg(F^{\boxminus}(d) : X)\,.$$

If $(d : Y)$, then $Y \in \mathrm{tags}(d)$ and $X \in \mathrm{tags}(F^{\boxplus}(d))$. If $\neg(d : Y)$, then $Y \notin \mathrm{tags}(d)$ and then $X \notin \mathrm{tags}(F^{\boxminus}(d))$.

Case: $T = b$

Since $b\theta = b$, we must show

$$(d : b) \implies (F^{\sigma}(d) : b) \qquad \neg(d : b) \implies (F^{\overline{\sigma}}(d) : b)\,.$$

If $(d : b)$, then $d = c^{L}$ with $c \in \mathbb{B}(b)$. Then, $F^{\sigma}(d) = c^{F^{\sigma}(L)}$ and $(F^{\sigma}(d) : b)$.

If $\neg(d : b)$ and d is of the form c^{L}, then $c \notin \mathbb{B}(b)$: then, $F^{\overline{\sigma}}(d) \notin [\![b]\!]$. If d is not of the form c^{L}, then $F^{\overline{\sigma}}(d)$ is not either and we have $F^{\overline{\sigma}}(d) \notin [\![b]\!]$.

Case: $T = T_1 \times T_2$

Since $T \vDash \sigma$, we have $T_1 \vDash \sigma$ and $T_2 \vDash \sigma$.

We must show

$$(d : T_1 \times T_2) \implies (F^{\sigma}(d) : T_1\theta \times T_2\theta)$$
$$\neg(d : T_1 \times T_2) \implies \neg(F^{\overline{\sigma}}(d) : T_1\theta \times T_2\theta)\,.$$

If $(d : T_1 \times T_2)$, then d is of the form $(d_1, d_2)^{L}$ and, for both i, $(d_i : T_1)$. We have $F^{\sigma}(d) = (F^{\sigma}(d_1), F^{\sigma}(d_2))^{F^{\sigma}(L)}$. By IH, $(d_1 : T_1)$ implies $(F^{\sigma}(d_1) : T_1\theta)$; likewise for d_2. Therefore, $(F^{\sigma}(d) : T_1\theta \times T_2\theta)$.

If $\neg(d : T_1 \times T_2)$ and $d = (d_1, d_2)^{L}$, then either $\neg(d_1 : T_1)$ or $\neg(d_2 : T_2)$. Then, by IH, either $\neg(F^{\overline{\sigma}}(d_1) : T_1\theta)$ or $\neg(F^{\overline{\sigma}}(d_2) : T_2\theta)$. Therefore, $\neg(F^{\overline{\sigma}}(d) : T_1\theta \times T_2\theta)$. If d is of another form, then the result is immediate.

Case: $T = T_1 \to T_2$

Since $T \vDash \sigma$, we have $T_1 \vDash \sigma$ and $T_2 \vDash \sigma$.

We must show

$$(d : T_1 \to T_2) \implies (F^{\sigma}(d) : T_1\theta \to T_2\theta)$$
$$\neg(d : T_1 \to T_2) \implies \neg(F^{\overline{\sigma}}(d) : T_1\theta \to T_2\theta)\,.$$

If $(d : T_1 \to T_2)$, then d is of the form $\{\, (d_j, d'_j) \mid j \in J \,\}^{L}$ and, for all $j \in J$, we have:

$$(d_j : T_1) \implies (d'_j : T_2)\,.$$

We have $F^{\sigma}(d) = \{\, (F^{\overline{\sigma}}(d_j), F^{\sigma}(d'_j)) \mid j \in J \,\}^{F^{\sigma}(L)}$.

For every j, by the induction hypothesis applied to T_1 and d_j, and to T_2 and d'_j, we get

$$(d_j : T_1) \implies (F^\sigma(d_j) : T_1\theta) \qquad \neg(d_j : T_1) \implies \neg(F^{\overline{\sigma}}(d_j) : T_1\theta)$$
$$(d'_j : T_2) \implies (F^\sigma(d'_j) : T_2\theta) \qquad \neg(d'_j : T_2) \implies \neg(F^{\overline{\sigma}}(d'_j) : T_2\theta)\,.$$

We must show, for all $j \in J$:

$$(F^{\overline{\sigma}}(d_j) : T_1\theta) \implies (F^\sigma(d'_j) : T_2\theta)$$

which we prove using the induction hypothesis (in particular, using the contrapositive of the second implication derived by induction).
If $\neg(d : T_1 \to T_2)$ and d is of the form $\{\,(d_j, d'_j) \mid j \in J\,\}^L$, then there exists a $j_0 \in J$ such that

$$(d_{j_0} : T_1) \qquad \neg(d'_{j_0} \in T_2)\,.$$

We have $F^{\overline{\sigma}}(d) = \{\,(F^\sigma(d_j), F^{\overline{\sigma}}(d'_j)) \mid j \in J\,\}^{F^{\overline{\sigma}}(L)}$. By IH, we show

$$(F^\sigma(d_{j_0}) : T_1\theta) \qquad \neg(F^{\overline{\sigma}}(d_{j_0}) : T_2\theta)\,.$$

If d is of another form, we have the result directly. then we get the result directly.

Case: $T = T_1 \vee T_2$

Since $T \vDash \sigma$, we have $T_1 \vDash \sigma$ and $T_2 \vDash \sigma$.
By the induction hypothesis applied to d and T_i, we get

$$(d : T_i) \implies (F^\sigma(d) : T_i\theta) \qquad \neg(d : T_i) \implies \neg(F^{\overline{\sigma}}(d) : T_i\theta)\,.$$

We must show

$$(d : T_1 \vee T_2) \implies (F^\sigma(d) : T_1\theta \vee T_2\theta) \qquad \neg(d : T_1 \vee T_2) \implies (F^{\overline{\sigma}}(d) : T_1\theta \vee T_2\theta)\,.$$

To show the first implication, assume $(d : T_1 \vee T_2)$: then either $(d : T_1)$ or $(d : T_2)$; then either $(F^\sigma(d) : T_1\theta)$ or $(F^\sigma(d) : T_2\theta)$; then $(F^\sigma(d) : T_1\theta \vee T_2\theta)$. To show the second, assume $\neg(d : T_1 \vee T_2)$: then $\neg(d : T_1)$ and $\neg(d : T_2)$; then $\neg(F^{\overline{\sigma}}(d) : T_1)$ and $\neg(F^{\overline{\sigma}}(d) : T_2)$; then $\neg(F^{\overline{\sigma}}(d) : T_1 \vee T_2)$.

Case: $T = \neg T'$

Since $T \vDash \sigma$, $T' \vDash \overline{\sigma}$.
By applying the induction hypothesis to d and T', we get

$$(d : T') \implies (F^{\overline{\sigma}}(d) : T'\theta) \qquad \neg(d : T') \implies \neg(F^\sigma(d) : T'\theta)\,.$$

We must show

$$(d : \neg T') \implies (F^\sigma(d) : \neg(T'\theta)) \qquad \neg(d : \neg T') \implies \neg(F^{\overline{\sigma}}(d) : \neg(T'\theta))\,.$$

For the first implication, assume $(d : \neg T')$: then $\neg(d : T')$, $\neg(F^\sigma(d) : T'\theta)$, and $(F^\sigma(d) : \neg(T'\theta))$.
For the second, assume $\neg(d : \neg T')$: then $\neg\neg(d : T')$, that is, $(d : T')$; hence $(F^{\overline{\sigma}}(d) : T'\theta)$, and $\neg(F^{\overline{\sigma}}(d) : \neg(T'\theta))$.

Case: $T = \mathbb{0}$

Both implications are trivial. □

Lemma B.19.

$$\left.\begin{array}{l} T_1 \leq_T T_2 \\ X \in \mathsf{var}^+_X(T_1) \implies X \notin \mathsf{var}^+_X(T_2) \\ X \in \mathsf{var}^-_X(T_1) \implies X \notin \mathsf{var}^-_X(T_2) \\ Y \sharp T_1, T_2, X \end{array}\right\} \implies T_1\{X := Y\} \leq_T T_2$$

Proof. If $X \notin \mathrm{var}_X(T_1)$, the result is immediate because $T_1\{X \coloneqq Y\} = T_1$. If $X \notin \mathrm{var}_X(T_2)$, then we have $T_2 = T_2\{X \coloneqq Y\}$ and the result can be derived by Lemma B.6. We consider the case $X \in \mathrm{var}_X(T_1) \cap \mathrm{var}_X(T_2)$. In this case, we have $X \notin \mathrm{var}^+_X(T_1) \cap \mathrm{var}^-_X(T_1)$: otherwise, X could not occur in T_2. Therefore, X occurs only positively or only negatively in T_1.

Given T_1, T_2, X, and Y satisfying

$$X \in \mathrm{var}^+_X(T_1) \implies X \notin \mathrm{var}^+_X(T_2) \qquad X \in \mathrm{var}^-_X(T_1) \implies X \notin \mathrm{var}^-_X(T_2) \qquad Y \sharp T_1, T_2, X\,,$$

we must show $T_1 \leq_T T_2 \implies T_1\{X \coloneqq Y\} \leq_T T_2$.

We show the contrapositive: $T_1\{X \coloneqq Y\} \not\leq_T T_2 \implies T_1 \not\leq_T T_2$. Assume $T_1\{X \coloneqq Y\} \not\leq_T T_2$.

We have $T_1 = T_1\{X \coloneqq Y\}\{Y \coloneqq X\}$ and $T_2 = T_2\{Y \coloneqq X\}$. Let $T = T_1\{X \coloneqq Y\} \setminus T_2$. We have $T \not\leq_T \mathbb{0}$ by definition of subtyping.

We show that either $\{X, Y\} \sharp \mathrm{var}^-_X(T)$ or $\{X, Y\} \sharp \mathrm{var}^+_X(T)$ holds. Note that

$$\mathrm{var}^+_X(T) = \mathrm{var}^+_X(T_1\{X \coloneqq Y\}) \cup \mathrm{var}^-_X(T_2) \qquad \mathrm{var}^-_X(T) = \mathrm{var}^-_X(T_1\{X \coloneqq Y\}) \cup \mathrm{var}^+_X(T_2)\,.$$

If $X \in \mathrm{var}^+_X(T_1)$, then $X \notin \mathrm{var}^-_X(T_1)$ and $X \notin \mathrm{var}^+_X(T_2)$: therefore, $\{X, Y\} \sharp \mathrm{var}^-_X(T)$. If $X \in \mathrm{var}^-_X(T_1)$, then $X \notin \mathrm{var}^+_X(T_1)$ and $X \notin \mathrm{var}^-_X(T_2)$: therefore, $\{X, Y\} \sharp \mathrm{var}^+_X(T)$.

By Lemma B.18, we have $T\{Y \coloneqq X\} \not\leq_T \mathbb{0}$: that is, $(T_1\{X \coloneqq Y\} \setminus T_2)\{Y \coloneqq X\} \not\leq_T \mathbb{0}$; that is, $T_1\{X \coloneqq Y\}\{Y \coloneqq X\} \not\leq_T T_2\{Y \coloneqq X\}$, which is $T_1 \not\leq_T T_2$. □

Lemma B.20.

$$\left.\begin{array}{l} T_1 \leq_T T_2 \\ \forall X \in \vec{X}.\ \begin{cases} X \in \mathrm{var}^+_X(T_1) \implies X \notin \mathrm{var}^+_X(T_2) \\ X \in \mathrm{var}^-_X(T_1) \implies X \notin \mathrm{var}^-_X(T_2) \end{cases} \\ \vec{Y} \sharp T_1, T_2, \vec{X} \end{array}\right\} \implies T_1\{\vec{X} \coloneqq \vec{Y}\} \leq_T T_2$$

Proof. By induction on $\vec{X}$. If $\vec{X}$ is empty, there is nothing to prove.

Otherwise, we have $\vec{X} = X_0\vec{X}'$ and $\vec{Y} = Y_0\vec{Y}'$. By Lemma B.19, we have $T_1\{X_0 \coloneqq Y_0\} \leq_T T_2$. Then, by IH, we have $T_1\{X_0 \coloneqq Y_0\}\{\vec{X}' \coloneqq \vec{Y}'\} \leq_T T_2$ and we conclude since $T_1\{X_0 \coloneqq Y_0\}\{\vec{X}' \coloneqq \vec{Y}'\} = T_1\{\vec{X} \coloneqq \vec{Y}\}$. □

Lemma B.21.

$$\left.\begin{array}{l} T \not\leq_T \mathbb{0} \\ X \notin \mathrm{var}^{\mathrm{even}}_X(T) \\ Y \notin \mathrm{var}^{\mathrm{odd}}_X(T) \end{array}\right\} \implies T\{Y \coloneqq X\} \not\leq_T \mathbb{0}$$

Proof. We first give some auxiliary definitions.

Let σ range over the two symbols $\triangle$ and $\triangledown$. We define $\overline{\sigma}$ as follows: $\overline{\triangle} \stackrel{\text{def}}{=} \triangledown$ and $\overline{\triangledown} \stackrel{\text{def}}{=} \triangle$.

Given a type frame T', we write $T' \vDash \triangle$ if $X \notin \mathrm{var}^{\mathrm{odd}}_X(T')$ and $Y \notin \mathrm{var}^{\mathrm{even}}_X(T')$; we write $T' \vDash \triangledown$ if $X \notin \mathrm{var}^{\mathrm{even}}_X(T')$ and $Y \notin \mathrm{var}^{\mathrm{odd}}_X(T')$.

Note that, for all T', T_1, and T_2, we have:

$$\begin{aligned} (\neg T' \vDash \sigma) &\implies (T' \vDash \sigma) \\ (T_1 \vee T_2 \vDash \sigma) &\implies (T_1 \vDash \sigma) \wedge (T_2 \vDash \sigma) \\ (T_1 \times T_2 \vDash \sigma) &\implies (T_1 \vDash \sigma) \wedge (T_2 \vDash \sigma) \\ (T_1 \to T_2 \vDash \sigma) &\implies (T_1 \vDash \overline{\sigma}) \wedge (T_2 \vDash \sigma) \end{aligned}$$

We define a function F^σ on domain element tags (finite sets of variables) as:

$$F^\triangle(L) = L \qquad F^\triangledown(L) = \begin{cases} L \cup \{X\} & \text{if } Y \in L \\ L \setminus \{X\} & \text{if } Y \notin L \end{cases}$$

We also define F on domain elements as follows:

$$\begin{aligned} F^\sigma(c^L) &= c^{F^\sigma(L)} \\ F^\sigma((d_1, d_2)^L) &= (F^\sigma(d_1), F^\sigma(d_2))^{F^\sigma(L)} \\ F^\sigma(\{(d_1, d_1'), \ldots, (d_n, d_n')\}^L) &= \{(F^{\overline{\sigma}}(d_1), F^\sigma(d_1')), \ldots, (F^{\overline{\sigma}}(d_n), F^\sigma(d_n'))\}^{F^\sigma(L)} \\ F^\sigma(\Omega) &= \Omega \end{aligned}$$

We must show:

$$\left.\begin{array}{l} T \not\leq_T \mathbb{0} \\ X \notin \mathrm{var}_X^{\mathsf{even}}(T) \\ Y \notin \mathrm{var}_X^{\mathsf{odd}}(T) \end{array}\right\} \implies T\{Y \coloneqq X\} \not\leq_T \mathbb{0}$$

This can be restated as:

$$\left.\begin{array}{l} \exists d \in \mathcal{D}.\, (d : T) \\ T \vDash \triangledown \end{array}\right\} \implies \exists d' \in \mathcal{D}.\, (d' : T\{Y \coloneqq X\})$$

We prove the following, stronger claim:

$$\forall d, T, \sigma. \quad T \vDash \sigma \implies \big((d : T) \iff (F^\sigma(d) : T\{Y \coloneqq X\})\big)$$

by induction on the pair (d, T), ordered lexicographically. For a given d, T, and σ, we assume $T \vDash \sigma$ and proceed by case analysis on T and d.

Let $\theta = \{Y \coloneqq X\}$.

Case: $T = \alpha$

Note that $\alpha\theta = \alpha$.

$$\begin{aligned} (d : \alpha) &\iff \alpha \in \mathrm{tags}(d) \\ &\iff \alpha \in \mathrm{tags}(F^\sigma(d)) && \text{neither } F^\triangle \text{ nor } F^\triangledown \text{ affect variables other than } X \\ &\iff (F^\sigma(d) : \alpha) \end{aligned}$$

Case: $T = Z$, with $Z \neq X$ and $Z \neq Y$

Like the previous case.

Case: $T = X$

Note that we must have $T \vDash \triangle$ because $X \in \mathrm{var}_X^{\mathsf{even}}(X)$ and $X \notin \mathrm{var}_X^{\mathsf{odd}}(X)$.

Note that $X\theta = X$.

$$\begin{aligned} (d : X) &\iff X \in \mathrm{tags}(d) \\ &\iff X \in \mathrm{tags}(F^\triangle(d)) \\ &\iff (F^\triangle(d) : X) \end{aligned}$$

Case: $T = Y$

Note that we must have $T \vDash \triangledown$ because $Y \in \mathrm{var}_X^{\mathsf{even}}(Y)$ and $Y \notin \mathrm{var}_X^{\mathsf{odd}}(Y)$.

Note that $Y\theta = X$.

$$\begin{aligned}(d : Y) &\iff Y \in \mathsf{tags}(d)\\ &\iff X \in \mathsf{tags}(F^{\triangledown}(d))\\ &\iff (F^{\triangledown}(d) : X)\end{aligned}$$

Case: $T = b$

Note that $b\theta = b$.

If $(d : b)$, then d must be of the form c^L with $c \in \mathbb{B}(b)$. Then, $F^\sigma(d) = c^{F^\sigma(L)}$ and $(F^\sigma(d) : b)$.

If $(F^\sigma(d) : b)$, then $F^\sigma(d)$ must be of the form c^L with $c \in \mathbb{B}(b)$. Then, $d = c^{L'}$ and $(d : b)$.

Case: $T = T_1 \times T_2$

If $(d : T_1 \times T_2)$, then $d = (d_1, d_2)^L$, $(d_1 : T_1)$, and $(d_2 : T_2)$. We have $F^\sigma(d) = (F^\sigma(d_1), F^\sigma(d_2))^{F^\sigma(L)}$. By IH we have, for $i \in \{1, 2\}$, $(d_i : T_i) \iff (F^\sigma(d_i) : T_i\theta)$; hence, $(F^\sigma(d) : T_1\theta \times T_2\theta)$.

If $(F^\sigma(d) : T_1\theta \times T_2\theta)$, then $F^\sigma(d) = (d_1, d_2)^L$, $(d_1 : T_1\theta)$, and $(d_2 : T_2\theta)$. Then, we have $d = (d_1', d_2')^{L'}$, with $d_1 = F^\sigma(d_1')$ and $d_2 = F^\sigma(d_2')$. By IH we have, for $i \in \{1, 2\}$, $(d_i' : T_i) \iff (d_i : T_i\theta)$; hence, $(d : T_1 \times T_2)$.

Case: $T = T_1 \to T_2$

Note that, since $T \vDash \sigma$, we have $T_1 \vDash \overline{\sigma}$ and $T_2 \vDash \sigma$.

If $(d : T_1 \to T_2)$, then $d = \{\, (d_j, d_j') \mid j \in J \,\}^L$ and

$$\forall j \in J.\ (d_j : T_1) \implies (d_j' : T_2)\ .$$

Then, $F^\sigma(d) = \{\, (F^{\overline{\sigma}}(d_j), F^\sigma(d_j')) \mid j \in J \,\}^{F^\sigma(L)}$. By IH, for every $j \in J$,

$$(d_j : T_1) \iff (F^{\overline{\sigma}}(d_j) : T_1\theta) \qquad (d_j' : T_2) \iff (F^\sigma(d_j') : T_2\theta)\ .$$

Therefore, we have

$$\forall j \in J.\ (F^{\overline{\sigma}}(d_j) : T_1\theta) \implies (F^\sigma(d_j') : T_2\theta)$$

and hence $(F^\sigma(d) : T_1\theta \to T_2\theta)$.

If $(F^\sigma(d) : T_1\theta \to T_2\theta)$, then $F^\sigma(d) = \{\, (d_j, d_j') \mid j \in J \,\}^L$ and

$$\forall j \in J.\ (d_j : T_1\theta) \implies (d_j' : T_2\theta)\ .$$

Then, $d = \{\, (\bar{d}_j, \bar{d}_j') \mid j \in J \,\}^{L'}$, with, for every $j \in J$, $F^{\overline{\sigma}}(\bar{d}_j) = d_j$ and $F^\sigma(\bar{d}_j') = d_j'$. By IH, for every $j \in J$,

$$(\bar{d}_j : T_1) \iff (d_j : T_1\theta) \qquad (\bar{d}_j' : T_2) \iff (d_j' : T_2\theta)\ .$$

Therefore, we have

$$\forall j \in J.\ (\bar{d}_j : T_1) \implies (\bar{d}_j' : T_2)$$

and hence $(d : T_1 \to T_2)$.

Case: $T = T_1 \vee T_2$

$$\begin{aligned}(d : T_1 \vee T_2) &\iff (d : T_1) \vee (d : T_2)\\ &\iff (F^\sigma(d) : T_1\theta) \vee (F^\sigma(d) : T_2\theta) && \text{by IH}\\ &\iff (F^\sigma(d) : T_1\theta \vee T_2\theta)\end{aligned}$$

Case: $T = \neg T'$

$$\begin{aligned}(d : \neg T') &\iff \neg(d : T')\\ &\iff \neg(F^\sigma(d) : T'\theta) && \text{by IH}\\ &\iff (F^\sigma(d) : \neg(T'\theta))\end{aligned}$$

Case: $T = \mathbb{0}$

Trivial, since $(d : \mathbb{0})$ never holds for any d and since $\mathbb{0}\theta = \mathbb{0}$. □

Lemma B.22.

$$\left.\begin{array}{l} T \not\leq_T \mathbb{0} \\ \vec{X} \sharp \mathrm{var}_X^{\mathsf{even}}(T) \\ \vec{Y} \sharp \mathrm{var}_X^{\mathsf{odd}}(T), \vec{X} \end{array}\right\} \implies T\{\vec{Y} \coloneqq \vec{X}\} \not\leq_T \mathbb{0}$$

Proof. By induction on $\vec{X}$. If $\vec{X}$ is empty, there is nothing to prove.

Otherwise, we have $\vec{X} = X_0\vec{X}'$ and $\vec{Y} = Y_0\vec{Y}'$. By Lemma B.21, we have $T\{Y_0 \coloneqq X_0\} \not\leq_T \mathbb{0}$. Then, by IH, we have $T\{Y_0 \coloneqq X_0\}\{\vec{Y}' \coloneqq \vec{X}'\} \not\leq_T \mathbb{0}$ and we conclude since $T\{Y_0 \coloneqq X_0\}\{\vec{Y}' \coloneqq \vec{X}'\} = T\{\vec{Y} \coloneqq \vec{X}\}$. □

Lemma B.23.

$$T \leq_T \mathbb{0} \implies \exists T', \vec{X}, \vec{Y}. \begin{cases} T' \leq_T \mathbb{0} \\ T = T'\{\vec{Y} \coloneqq \vec{X}\} \\ \mathrm{var}_X^{\mathsf{even}}(T') \sharp \mathrm{var}_X^{\mathsf{odd}}(T') \end{cases}$$

Proof. Assume that $\mathrm{var}_X(T) = \{X_1, \ldots, X_n\}$.

By Corollary B.16, we can find T' such that $\mathrm{var}_X^{\mathsf{even}}(T') \subseteq \{X_1, \ldots, X_n\}$ is disjoint from $\mathrm{var}_X^{\mathsf{odd}}(T') \subseteq \{X'_1, \ldots, X'_n\}$ and that $T = T'\{X'_i \coloneqq X_i\}_{i=1}^n$.

We must prove $T' \leq_T \mathbb{0}$. We have $T \leq_T \mathbb{0}$, which is $T'\{X'_i \coloneqq X_i\}_{i=1}^n \leq_T \mathbb{0}$. Therefore, we also have $T'\{X'_i \coloneqq X_i\}_{i=1}^n\{X_i \coloneqq X'_i\}_{i=1}^n \leq_T \mathbb{0}$ (by Proposition B.6), which is $T'\{X_i \coloneqq X'_i\}_{i=1}^n \leq_T \mathbb{0}$.

Let $\vec{X}$ be the vector $X_1 \ldots X_n$ and $\vec{X}'$ be the vector $X'_1 \ldots X'_n$. We have $\vec{X} \sharp \mathrm{var}_X^{\mathsf{odd}}(T')$ and $\vec{X}' \sharp \mathrm{var}_X^{\mathsf{even}}(T')$. We also have $\vec{X} \sharp \vec{Y}$.

By Lemma B.22, we have

$$T' \not\leq_T \mathbb{0} \implies T'\{\vec{X} \coloneqq \vec{X}'\} \not\leq_T \mathbb{0}$$

and, by contrapositive,

$$T'\{\vec{X} \coloneqq \vec{X}'\} \leq_T \mathbb{0} \implies T' \leq_T \mathbb{0}$$

which yields $T' \leq_T \mathbb{0}$. □

Lemma B.24.

$$T_1 \leq_T T_2 \implies \exists T'_1, T'_2, \vec{X}, \vec{Y}. \begin{cases} T'_1 \leq_T T'_2 \\ T_1 = T'_1\{\vec{Y} \coloneqq \vec{X}\} \\ T_2 = T'_2\{\vec{Y} \coloneqq \vec{X}\} \\ \mathrm{var}_X^{\mathsf{even}}(T'_1, T'_2) \sharp \mathrm{var}_X^{\mathsf{odd}}(T'_1, T'_2) \end{cases}$$

Proof. Let $T = T_1 \setminus T_2$. We have $T \leq_T \mathbb{0}$ by definition of subtyping.

By Lemma B.23, we find T', $\vec{X}$, and $\vec{Y}$ such that

$$T' \leq_T \mathbb{0} \qquad T = T'\{\vec{Y} \coloneqq \vec{X}\} \qquad \mathrm{var}_X^{\mathsf{even}}(T') \sharp \mathrm{var}_X^{\mathsf{odd}}(T') \,.$$

Since T' is empty, it cannot be a type variable or a frame variable. Then, we must have $T' = T'_1 \setminus T'_2$ for two types such that $T_1 = T'_1\{\vec{Y} \coloneqq \vec{X}\}$ and $T_2 = T'_2\{\vec{Y} \coloneqq \vec{X}\}$.

We have $T'_1 \leq_T T'_2$ by definition of subtyping.

We have $\mathrm{var}_X^{\mathsf{even}}(T'_1, T'_2) = \mathrm{var}_X^{\mathsf{even}}(T')$ and $\mathrm{var}_X^{\mathsf{odd}}(T'_1, T'_2) = \mathrm{var}_X^{\mathsf{odd}}(T')$, therefore the two sets are disjoint. □

Lemma B.25.

$$\left.\begin{array}{l} T_1 \leq_T T_2 \\ T_1^\dagger = \tau_1 \text{ and } T_2^\dagger = \tau_2 \\ \mathsf{var}_X^{+\mathsf{cov}}(T_1, T_2),\ \mathsf{var}_X^{+\mathsf{cnt}}(T_1, T_2),\ \mathsf{var}_X^{-\mathsf{cov}}(T_1, T_2), \\ \qquad \text{and } \mathsf{var}_X^{-\mathsf{cnt}}(T_1, T_2) \text{ are pairwise disjoint} \end{array}\right\} \implies \tau_1^\bullet \leq_T \tau_2^\bullet$$

Proof. We define

$$\theta = \{X := X^{+\wedge}\}_{X \in \mathsf{var}_X^{+\mathsf{cov}}(T_1,T_2)} \cup \{X := X^{+\vee}\}_{X \in \mathsf{var}_X^{+\mathsf{cnt}}(T_1,T_2)} \\ \cup \{X := X^{-\vee}\}_{X \in \mathsf{var}_X^{-\mathsf{cov}}(T_1,T_2)} \cup \{X := X^{-\wedge}\}_{X \in \mathsf{var}_X^{-\mathsf{cnt}}(T_1,T_2)} .$$

θ is well-defined because the four sets are disjoint. We have $T_1\theta = \tau_1^\bullet$ and $T_2\theta = \tau_2^\bullet$. We have $T_1\theta \leq_T T_2\theta$ by Proposition B.6. □

Lemma B.26. *If $\tau_1 \leq \tau_2$, then $\tau_1^\bullet \leq_T \tau_2^\bullet$.*

Proof. By definition of $\tau_1 \leq \tau_2$, there exist T_1 and T_2 such that:

$$T_1^\dagger = \tau_1 \text{ and } T_2^\dagger = \tau_2 \qquad \mathsf{var}_X^+(T_1) \sharp \mathsf{var}_X^-(T_1) \text{ and } \mathsf{var}_X^+(T_2) \sharp \mathsf{var}_X^-(T_2) \qquad T_1 \leq_T T_2 .$$

Let $\vec{X} = \left(\mathsf{var}_X^+(T_1) \cap \mathsf{var}_X^-(T_2)\right) \cup \left(\mathsf{var}_X^-(T_1) \cap \mathsf{var}_X^+(T_2)\right)$ and let $\vec{Y}$ be a vector of variables outside T_1 and T_2. Since T_1 and T_2 are polarized, we have

$$\forall X \in \vec{X}.\ \begin{cases} X \in \mathsf{var}_X^+(T_1) \implies X \notin \mathsf{var}_X^+(T_2) \\ X \in \mathsf{var}_X^-(T_1) \implies X \notin \mathsf{var}_X^-(T_2) \end{cases}$$

and we can apply Lemma B.20 to derive $T_1\{\vec{X} := \vec{Y}\} \leq_T T_2$.

We have

$$\mathsf{var}_X^+(T_1\{\vec{X} := \vec{Y}\}, T_2) \sharp \mathsf{var}_X^-(T_1\{\vec{X} := \vec{Y}\}, T_2) .$$

We apply Lemma B.24 to $T_1\{\vec{X} := \vec{Y}\}$ and T_2 to find T_1', T_2', $\vec{X}'$, and $\vec{Y}'$ such that:

$$T_1' \leq_T T_2' \qquad T_1\{\vec{X} := \vec{Y}\} = T_1'\{\vec{Y}' := \vec{X}'\} \text{ and } T_2 = T_2'\{\vec{Y}' := \vec{X}'\} \qquad \mathsf{var}_X^{\mathsf{even}}(T_1', T_2') \sharp \mathsf{var}_X^{\mathsf{odd}}(T_1', T_2').$$

We have

$$\tau_1 = T_1^\dagger = (T_1\{\vec{X} := \vec{Y}\})^\dagger = (T_1'\{\vec{Y}' := \vec{X}'\})^\dagger = (T_1')^\dagger$$
$$\tau_2 = T_2^\dagger = (T_2'\{\vec{Y}' := \vec{X}'\})^\dagger = (T_2')^\dagger .$$

We also have

$$\mathsf{var}_X^+(T_1', T_2') \sharp \mathsf{var}_X^-(T_1', T_2') \qquad \mathsf{var}_X^{\mathsf{even}}(T_1', T_2') \sharp \mathsf{var}_X^{\mathsf{odd}}(T_1', T_2')$$

and therefore the following four sets are disjoint

$$\mathsf{var}_X^{+\mathsf{cov}}(T_1', T_2') \qquad \mathsf{var}_X^{+\mathsf{cnt}}(T_1', T_2') \qquad \mathsf{var}_X^{-\mathsf{cov}}(T_1', T_2') \qquad \mathsf{var}_X^{-\mathsf{cnt}}(T_1', T_2') .$$

Then, by Lemma B.25, we have $\tau_1^\bullet \leq_T \tau_2^\bullet$. □

Lemma B.27. *Let τ_1 and τ_2 be two gradual types. Assume that there exist $T_1 \in \star(\tau_1)$ and $T_2 \in \star(\tau_2)$ such that T_1 and T_2 are variance-polarized and that $T_1 \leq_T T_2$. Then, $\tau_1^\bullet \leq_T \tau_2^\bullet$.*

Proof. We have

$$T_1^\dagger = \tau_1 \text{ and } T_2^\dagger = \tau_2 \qquad \mathsf{var}_X^{\mathsf{cov}}(T_1) \sharp \mathsf{var}_X^{\mathsf{cnt}}(T_1) \text{ and } \mathsf{var}_X^{\mathsf{cov}}(T_2) \sharp \mathsf{var}_X^{\mathsf{cnt}}(T_2) \qquad T_1 \leq_T T_2 \,.$$

We apply Lemma B.24 to T_1 and T_2 to find T_1', T_2', $\vec{X}$, and $\vec{Y}$ such that:

$$T_1' \leq_T T_2' \qquad T_1 = T_1'\{\vec{Y} := \vec{X}\} \text{ and } T_2 = T_2'\{\vec{Y} := \vec{X}\} \qquad \mathsf{var}_X^{\mathsf{even}}(T_1', T_2') \sharp \mathsf{var}_X^{\mathsf{odd}}(T_1', T_2') \,.$$

Since we have

$$\mathsf{var}_X^{\mathsf{cov}}(T_1') \sharp \mathsf{var}_X^{\mathsf{cnt}}(T_1') \text{ and } \mathsf{var}_X^{\mathsf{cov}}(T_2') \sharp \mathsf{var}_X^{\mathsf{cnt}}(T_2') \qquad \mathsf{var}_X^{\mathsf{even}}(T_1', T_2') \sharp \mathsf{var}_X^{\mathsf{odd}}(T_1', T_2') \,,$$

we also have

$$\mathsf{var}_X^{+}(T_1') \sharp \mathsf{var}_X^{-}(T_1') \text{ and } \mathsf{var}_X^{+}(T_2') \sharp \mathsf{var}_X^{-}(T_2') \,.$$

Let $\vec{X}' = \left(\mathsf{var}_X^{+}(T_1') \cap \mathsf{var}_X^{-}(T_2')\right) \cup \left(\mathsf{var}_X^{-}(T_1') \cap \mathsf{var}_X^{+}(T_2')\right)$ and let $\vec{Y}'$ be a vector of variables outside T_1' and T_2'. We have

$$\forall X \in \vec{X}'. \quad \begin{cases} X \in \mathsf{var}_X^{+}(T_1') \implies X \notin \mathsf{var}_X^{+}(T_2') \\ X \in \mathsf{var}_X^{-}(T_1') \implies X \notin \mathsf{var}_X^{-}(T_2') \end{cases}$$

and we can apply Lemma B.20 to derive $T_1'\{\vec{X}' := \vec{Y}'\} \leq_T T_2'$.

We have

$$\tau_1 = T_1^\dagger = (T_1'\{\vec{Y} := \vec{X}\})^\dagger = (T_1')^\dagger = (T_1'\{\vec{X}' := \vec{Y}'\})^\dagger$$
$$\tau_2 = T_2^\dagger = (T_2'\{\vec{Y} := \vec{X}\})^\dagger = (T_2')^\dagger \,.$$

Let $T_1'' = T_1'\{\vec{X}' := \vec{Y}'\}$.

We also have

$$\mathsf{var}_X^{+}(T_1'', T_2') \sharp \mathsf{var}_X^{-}(T_1'', T_2') \qquad \mathsf{var}_X^{\mathsf{even}}(T_1'', T_2') \sharp \mathsf{var}_X^{\mathsf{odd}}(T_1'', T_2')$$

and therefore the following four sets are disjoint

$$\mathsf{var}_X^{+\mathsf{cov}}(T_1'', T_2') \qquad \mathsf{var}_X^{+\mathsf{cnt}}(T_1'', T_2') \qquad \mathsf{var}_X^{-\mathsf{cov}}(T_1'', T_2') \qquad \mathsf{var}_X^{-\mathsf{cnt}}(T_1'', T_2') \,.$$

Then, by Lemma B.25, we have $\tau_1^\bullet \leq_T \tau_2^\bullet$. □

Proposition B.28. *Let τ_1 and τ_2 be two gradual types. The following statements are all equivalent:*

(1) $\tau_1 \leq \tau_2$;
(2) $\tau_1^\oplus \leq_T \tau_2^\oplus$;
(3) $\tau_1^\ominus \leq_T \tau_2^\ominus$;
(4) *there exist $T_1 \in \star(\tau_1)$ and $T_2 \in \star(\tau_2)$ such that T_1 and T_2 are variance-polarized and that $T_1 \leq_T T_2$;*
(5) $\tau_1^{\textcircled{\scriptsize\wedge}} \leq_T \tau_2^{\textcircled{\scriptsize\wedge}}$;
(6) $\tau_1^{\textcircled{\scriptsize\vee}} \leq_T \tau_2^{\textcircled{\scriptsize\vee}}$;
(7) $\tau_1^\bullet \leq_T \tau_2^\bullet$.

Proof. We have (1) $\implies$ (7) by Lemma B.26 and (4) $\implies$ (7) by Lemma B.27.

The equivalences (2) $\iff$ (3) and (5) $\iff$ (6) are shown trivially by Proposition B.6 since, for every τ, we have $\tau^\oplus = \tau^\ominus\{X^0 := X^1, X^1 := X^0\}$ and similarly for the others.

We can show (7) $\implies$ (2) $\wedge$ (5) by Proposition B.6. If $\tau_1^\bullet \leq_T \tau_2^\bullet$, then $\tau_1^\bullet\theta \leq_T \tau_2^\bullet\theta$ holds for every type substitution θ. To show (2), we choose $\theta = \{X^{+\wedge} := X^1, X^{+\vee} := X^1, X^{-\wedge} := X^0, X^{-\vee} := X^0\}$ and have $\tau_1^\bullet\theta = \tau_1^\oplus$ and $\tau_2^\bullet\theta = \tau_2^\oplus$. We prooced analogously to show (5).

The implication (2) $\implies$ (1) holds because, for any τ, $\tau^\oplus \in \star^{\mathsf{pol}}(\tau)$. Likewise for the implication (5) $\implies$ (4). □

Definition B.29 (Consistency). *We define the* consistency *relation* $\sim$ *on gradual types as follows:*

$$\tau_1 \sim \tau_2 \overset{\text{def}}{\iff} \exists T_1 \in \star(\tau_1), T_2 \in \star(\tau_2), \theta : \mathcal{V}^X \to \mathcal{T}_t.\ T_1\theta \simeq_T T_2\theta$$

Definition B.30 (Consistent subtyping). *The* consistent subtyping *relation* $\widetilde{\leq}$ *between types is defined by*

$$\tau_1 \mathrel{\widetilde{\leq}} \tau_2 \overset{\text{def}}{\iff} \exists T_1 \in \star(\tau_1), T_2 \in \star(\tau_2), \theta : \mathcal{V}^X \to \mathcal{T}_t.\ T_1\theta \leq_T T_2\theta$$

Definition B.31 (Materialization). *We define the* materialization *relation on gradual types* $\tau_1 \preccurlyeq \tau_2$ *("*τ_2 *materializes* τ_1*") as follows:*

$$\tau_1 \preccurlyeq \tau_2 \overset{\text{def}}{\iff} \exists T_1 \in \star(\tau_1), \exists \theta : \mathcal{V}^X \to \mathcal{T}_\tau.\ T_1\theta = \tau_2$$

Proposition B.32. *If* $\tau_1 \leq \tau_2$*, then, for any static type substitution* θ*, we have* $\tau_1\theta \leq \tau_2\theta$*.*

Proof. If $\tau_1 \leq \tau_2$, then by Proposition B.28 we have $\tau_1^{\oplus} \leq_T \tau_2^{\oplus}$. Then, $\tau_1^{\oplus}\theta \leq_T \tau_2^{\oplus}\theta$ by Proposition B.6. We have $\tau_1^{\oplus}\theta = (\tau_1\theta)^{\oplus}$ because $(\tau_1^{\oplus})^{\dagger} = \tau_1\theta$ and because $\tau_1^{\oplus}\theta$ is strongly polarized (since θ does not introduce frame variables). Similarly, we have $\tau_2^{\oplus}\theta = (\tau_2\theta)^{\oplus}$. Therefore, $\tau_1\theta \leq \tau_2\theta$. □

We write $\lessapprox$ for the reflexive and transitive relation on gradual types that combines subtyping and materialization, defined inductively by:

$$\frac{}{\tau \lessapprox \tau} \qquad \frac{\tau_1 \leq \tau_2 \qquad \tau_2 \lessapprox \tau_3}{\tau_1 \lessapprox \tau_3} \qquad \frac{\tau_1 \preccurlyeq \tau_2 \qquad \tau_2 \lessapprox \tau_3}{\tau_1 \lessapprox \tau_3}$$

Lemma B.33. *If* $\tau_1 \preccurlyeq \tau_2$*, then there exist a* T *and a* $\theta : \mathrm{var}_X(T) \to \mathcal{T}_\tau$ *such that* $T^{\dagger} = \tau_1$*, that* $T\theta = \tau_2$*, and that* $\mathrm{var}_X^{\mathrm{cov}}(T) \cap \mathrm{var}_X^{\mathrm{cnt}}(T) = \varnothing$*.*

Proof. By definition of $\tau_1 \preccurlyeq \tau_2$, there exist a T_1 and a $\theta_1 : \mathcal{V}^X \to \mathcal{T}_\tau$ such that $T_1^{\dagger} = \tau_1$ and that $T_1\theta_1 = \tau_2$. Let $\mathrm{var}_X(T_1) = \{X_1, \ldots, X_n\}$.

By Corollary B.15, we can find a T such that $\mathrm{var}_X^{\mathrm{cov}}(T) \subseteq \{X_1, \ldots, X_n\}$ is disjoint from $\mathrm{var}_X^{\mathrm{cnt}}(T) \subseteq \{X_1', \ldots, X_n'\}$ and such that $T_1 = T\{X_i' \coloneqq X_i\}_{i=1}^n$. Clearly, $T^{\dagger} = T_1^{\dagger} = \tau_1$.

We take θ to be $\{X_i \coloneqq X_i\theta_1\}_{i=1}^n \cup \{X_i' \coloneqq X_i\theta_1\}_{i=1}^n$ restricted to $\mathrm{var}_X(T)$. We have:

$$T\theta = T(\{X_i \coloneqq X_i\theta_1\}_{i=1}^n \cup \{X_i' \coloneqq X_i\theta_1\}_{i=1}^n) = T\{X_i' \coloneqq X_i\}_{i=1}^n\theta_1 = T_1\theta_1 = \tau_2\ .$$ □

Proposition B.34. *If* $\tau_1 \leq \tau_2 \preccurlyeq \tau_3$*, then there exists a* τ_2' *such that* $\tau_1 \preccurlyeq \tau_2' \leq \tau_3$*.*

Proof. By Lemma B.33, since $\tau_2 \preccurlyeq \tau_3$, there exist T_2 and $\theta : \mathrm{var}_X(T_2) \to \mathcal{T}_\tau$ such that $T_2^{\dagger} = \tau_2$, that $T_2\theta = \tau_3$, and that $\mathrm{var}_X^{\mathrm{cov}}(T_2) \cap \mathrm{var}_X^{\mathrm{cnt}}(T_2) = \varnothing$. Assume that $\mathrm{var}_X^{\mathrm{cov}} = \{X_1, \ldots, X_n\}$ and $\mathrm{var}_X^{\mathrm{cnt}} = \{Y_1, \ldots, Y_m\}$.

Let $\bar{\theta} = \{X_i \coloneqq (X_i\theta)^{\mathbb{O}}\}_{i=1}^n \cup \{Y_i \coloneqq (Y_i\theta)^{\mathbb{V}}\}_{i=1}^n$. We have $(T_2\bar{\theta})^{\dagger} = T_2\theta = \tau_3$.

Let $\hat{\theta} = \{X_i \coloneqq \bigwedge_{j=1}^n X_j\bar{\theta}\}_{i=1}^n \cup \{Y_i \coloneqq \bigvee_{j=1}^m Y_j\bar{\theta}\}_{i=1}^m$ and $\check{\theta} = \{X^1 \coloneqq \bigwedge_{j=1}^n X_j\bar{\theta}, X^0 \coloneqq \bigvee_{j=1}^m Y_j\bar{\theta}\}$.

We have:

$$\forall i = 1, \ldots, n.\ X_i\hat{\theta} \leq_T X_i\bar{\theta} \qquad \forall i = 1, \ldots, m.\ Y_i\bar{\theta} \leq_T Y_i\theta$$

We take $\tau_2' = (\tau_1^{\mathbb{O}}\check{\theta})^{\dagger}$. We must show:

$$\tau_1 \preccurlyeq (\tau_1^{\mathbb{O}}\check{\theta})^{\dagger} \qquad (\tau_1^{\mathbb{O}}\check{\theta})^{\dagger} \leq \tau_3$$

The former holds because $(\tau_1^{\mathbb{O}}\check{\theta})^{\dagger} = \tau_1^{\mathbb{O}}\{X^1 \coloneqq \bigwedge_{j=1}^n X_j\theta, X^0 \coloneqq \bigvee_{j=1}^m Y_j\theta\}$ and $\tau_1^{\mathbb{O}} \in \star(\tau_1)$.

To show the latter, we show:

$$(\tau_1^{\mathbb{O}}\check{\theta})^{\dagger} \leq (\tau_2^{\mathbb{O}}\check{\theta})^{\dagger} \qquad \tau_2^{\mathbb{O}}\check{\theta} = T_2\hat{\theta} \qquad (T_2\hat{\theta})^{\dagger} \leq (T_2\bar{\theta})^{\dagger}$$

We show $(\tau_1^{\textcircled{\scriptsize\Delta}}\check{\theta})^\dagger \leq (\tau_2^{\textcircled{\scriptsize\Delta}}\check{\theta})^\dagger$. By Proposition B.28, $\tau_1 \leq \tau_2$ implies $\tau_1^{\textcircled{\scriptsize\Delta}} \leq_T \tau_2^{\textcircled{\scriptsize\Delta}}$. By Proposition B.6, $\tau_1^{\textcircled{\scriptsize\Delta}}\check{\theta} \leq_T \tau_2^{\textcircled{\scriptsize\Delta}}\check{\theta}$. Both $\tau_1^{\textcircled{\scriptsize\Delta}}\check{\theta}$ and $\tau_2^{\textcircled{\scriptsize\Delta}}\check{\theta}$ are strongly polarized according to variance; therefore, $(\tau_1^{\textcircled{\scriptsize\Delta}}\check{\theta})^{\dagger\,\textcircled{\scriptsize\Delta}} = \tau_1^{\textcircled{\scriptsize\Delta}}\check{\theta}$ and $(\tau_2^{\textcircled{\scriptsize\Delta}}\check{\theta})^{\dagger\,\textcircled{\scriptsize\Delta}} = \tau_2^{\textcircled{\scriptsize\Delta}}\check{\theta}$. Hence, $(\tau_1^{\textcircled{\scriptsize\Delta}}\check{\theta})^\dagger \leq (\tau_2^{\textcircled{\scriptsize\Delta}}\check{\theta})^\dagger$.

To show $\tau_2^{\textcircled{\scriptsize\Delta}}\check{\theta} = T_2\hat{\theta}$, just note that $\tau_2^{\textcircled{\scriptsize\Delta}} = T_2(\{X_i \coloneqq X^1\}_{i=1}^n \cup \{Y_i \coloneqq X^0\}_{i=1}^m)$.

Now we show $(T_2\hat{\theta})^\dagger \leq (T_2\bar{\theta})^\dagger$. First, note that $\hat{\theta}|_{\mathsf{var}^{\mathsf{cov}}(T_2)} \leq_T \bar{\theta}|_{\mathsf{var}^{\mathsf{cov}}(T_2)}$ and $\bar{\theta}|_{\mathsf{var}^{\mathsf{cnt}}(T_2)} \leq_T \hat{\theta}|_{\mathsf{var}^{\mathsf{cnt}}(T_2)}$. Hence, by Proposition B.10, we have $T_2\hat{\theta} \leq_T T_2\bar{\theta}$. Since both $T_2\hat{\theta}$ and $T_2\bar{\theta}$ are strongly polarized according to variance, we have $T_2\hat{\theta} = ((T_2\hat{\theta})^\dagger)^{\textcircled{\scriptsize\Delta}}$ and $T_2\bar{\theta} = ((T_2\bar{\theta})^\dagger)^{\textcircled{\scriptsize\Delta}}$. This yields the result we need. □

Corollary B.35. *If $\tau_1 \lessdot\!\lessdot \tau_2$, then there exists a type τ such that $\tau_1 \lessdot \tau \leq \tau_2$.*

Proof. By induction on the derivation of $\tau_1 \lessdot\!\lessdot \tau_2$.

If $\tau_1 = \tau_2$, then $\tau_1 \lessdot \tau_1 \leq \tau_2$.

If $\tau_1 \leq \tau' \lessdot\!\lessdot \tau_2$, then by IH we find τ'' such that $\tau' \lessdot \tau'' \leq \tau_2$, by Proposition B.34 we find τ such that $\tau_1 \lessdot \tau \leq \tau''$, and finally (by transitivity of $\leq$) we have $\tau_1 \lessdot \tau \leq \tau_2$.

If $\tau_1 \lessdot \tau' \lessdot\!\lessdot \tau_2$, then by IH we find τ'' such that $\tau' \lessdot \tau'' \leq \tau_2$, and (by transitivity of $\lessdot$) we have $\tau_1 \lessdot \tau'' \leq \tau_2$. □

Proposition B.36. *If $\tau_1 \lessdot \tau_2$, then, for any type substitution θ, we have $\tau_1\theta \lessdot \tau_2\theta$.*

Proof. By definition of $\tau_1 \lessdot \tau_2$, we have $T_1\theta_1 = \tau_2$ for a T_1 such that $T_1^\dagger = \tau_1$ and a $\theta_1 : \mathcal{V}^X \to \mathcal{T}_\tau$.

Choose a $\theta' : \mathcal{V}^\alpha \to \mathcal{T}_T$ such that, for every α, $(\alpha\theta')^\dagger = \alpha\theta$ and that $\mathrm{var}_X(\theta') \cap \mathrm{dom}(\theta_1) = \varnothing$.

Then we have $(T_1\theta')^\dagger = \tau_1\theta$ and therefore $T_1\theta' \in \star(\tau_1\theta)$.

Consider $\theta_1' = \{X \coloneqq X\theta_1\theta\}_{X\in\mathrm{dom}(\theta_1)} \cup \{X \coloneqq ?\}_{X\in\mathrm{var}_X(\theta')}$.

We have $T_1\theta'\theta_1' = T_1\theta_1\theta$ because:

- for every $\alpha \in \mathrm{var}(T_1)$, if $\alpha \in \mathrm{dom}(\theta)$, then $\alpha\theta'\theta_1' = (\alpha\theta')^\dagger = \alpha\theta = \alpha\theta_1\theta$, and, if $\alpha \notin \mathrm{dom}(\theta)$, then $\alpha\theta'\theta_1' = \alpha = \alpha\theta_1\theta$;
- for every $X \in \mathrm{var}(T_1)$, we must have $X \in \mathrm{dom}(\theta_1)$ (otherwise, $T_1\theta_1$ would not be a gradual type): then $X\theta'\theta_1' = X\theta_1' = X\theta_1\theta$.

Since $T_1\theta_1\theta = \tau_2\theta$, we have $\tau_1\theta \lessdot \tau_2\theta$. □

Proposition B.37. *Let τ be a gradual type and θ_1 and θ_2 two substitutions such that $\forall\alpha \in \mathrm{var}(\tau)$. $\alpha\theta_1 \simeq \alpha\theta_2$. Then, $\tau\theta_1 \simeq \tau\theta_2$.*

Proof. Let $\mathrm{var}(\tau) = \{\alpha_1, \ldots, \alpha_n\}$. By Corollary B.17, we find τ' such that $\mathrm{var}^+(\tau') \subseteq \{\alpha_1, \ldots, \alpha_n\}$ is disjoint from $\mathrm{var}^-(\tau') \subseteq \{\alpha_1', \ldots, \alpha_n'\}$ and that $\tau = \tau'\{\alpha_i' \coloneqq \alpha_i\}_{i=1}^n$.

Now, we define

$$\hat{\theta}_1 = \{\alpha_i \coloneqq (\alpha_i\theta_1)^\oplus\}_{i=1}^n \cup \{\alpha_i' \coloneqq (\alpha_i\theta_1)^\ominus\}_{i=1}^n \qquad \hat{\theta}_2 = \{\alpha_i \coloneqq (\alpha_i\theta_2)^\oplus\}_{i=1}^n \cup \{\alpha_i' \coloneqq (\alpha_i\theta_2)^\ominus\}_{i=1}^n .$$

Let $T = \tau'^\oplus$.

We show that, for every A, $A\hat{\theta}_1 \simeq_T A\hat{\theta}_2$. Note that, for every $i \in I$, we have $\alpha_i\theta_1 \simeq \alpha_i\theta_2$ and therefore, by Proposition B.28, $(\alpha_i\theta_1)^\oplus \simeq_T (\alpha_i\theta_2)^\oplus$ and $(\alpha_i\theta_1)^\ominus \simeq_T (\alpha_i\theta_2)^\ominus$. If $A \notin \{\alpha_i \mid i \in I\} \cup \{\alpha_i \mid i \in I\}$, then $A\hat{\theta}_1 = A = A\hat{\theta}_2$. If $A = \alpha_i$ for some $i \in I$, then $A\hat{\theta}_1 = (\alpha_i\theta_1)^\oplus \simeq_T (\alpha_i\theta_2)^\oplus = A\hat{\theta}_2$. If $A = \alpha_i'$ for some $i \in I$, then $A\hat{\theta}_1 = (\alpha_i\theta_1)^\ominus \simeq_T (\alpha_i\theta_2)^\ominus = A\hat{\theta}_2$.

Since, for every A, $A\hat{\theta}_1 \simeq_T A\hat{\theta}_2$, we have $\hat{\theta}_1|_{\mathsf{var}^{\mathsf{cov}}(T)} \leq_T \hat{\theta}_2|_{\mathsf{var}^{\mathsf{cov}}(T)}$, $\hat{\theta}_2|_{\mathsf{var}^{\mathsf{cnt}}(T)} \leq_T \hat{\theta}_1|_{\mathsf{var}^{\mathsf{cnt}}(T)}$, $\hat{\theta}_2|_{\mathsf{var}^{\mathsf{cov}}(T)} \leq_T \hat{\theta}_1|_{\mathsf{var}^{\mathsf{cov}}(T)}$, and $\hat{\theta}_1|_{\mathsf{var}^{\mathsf{cnt}}(T)} \leq_T \hat{\theta}_2|_{\mathsf{var}^{\mathsf{cnt}}(T)}$. By Proposition B.10, we have $T\hat{\theta}_1 \simeq_T T\hat{\theta}_2$.

We have:

$$T\hat{\theta}_1 = \tau'^\oplus\hat{\theta}_1 = (\tau\theta_1)^\oplus \qquad T\hat{\theta}_2 = \tau'^\oplus\hat{\theta}_2 = (\tau\theta_2)^\oplus$$

Therefore, we have $(\tau\theta_1)^\oplus \simeq_T (\tau\theta_2)^\oplus$. Hence, $\tau\theta_1 \simeq \tau\theta_2$. □

B.6 Normal Forms and Decompositions for Type Frames

In the following, we use the metavariable a to range over the set $\mathcal{A}_{\mathsf{basic}} \cup \mathcal{A}_{\mathsf{prod}} \cup \mathcal{A}_{\mathsf{fun}} \cup \mathcal{V}$.

Definition B.38 (Uniform normal form). *A* uniform (disjunctive) normal form *(UDNF) is a type frame T of the form*

$$\bigvee_{i \in I} \left(\bigwedge_{a \in P_i} a \wedge \bigwedge_{a \in N_i} \neg a \right)$$

such that, for all $i \in I$, one of the following three condition holds:

- $P_i \cap \mathcal{A}_{\mathsf{basic}} \neq \varnothing$ *and* $(P_i \cup N_i) \cap (\mathcal{A}_{\mathsf{prod}} \cup \mathcal{A}_{\mathsf{fun}}) = \varnothing$;
- $P_i \cap \mathcal{A}_{\mathsf{prod}} \neq \varnothing$ *and* $(P_i \cup N_i) \cap (\mathcal{A}_{\mathsf{basic}} \cup \mathcal{A}_{\mathsf{fun}}) = \varnothing$;
- $P_i \cap \mathcal{A}_{\mathsf{fun}} \neq \varnothing$ *and* $(P_i \cup N_i) \cap (\mathcal{A}_{\mathsf{basic}} \cup \mathcal{A}_{\mathsf{prod}}) = \varnothing$;

We define here a function $\mathsf{UDNF}(T)$ which, given a type frame T, produces a uniform normal form that is equivalent to T.

We first define two mutually recursive functions $\mathcal{N}$ and $\mathcal{N}'$ on type frames. These are inductive definitions as no recursive uses of the functions occur below type constructors.

$$\begin{aligned}
\mathcal{N}(a) &= a \\
\mathcal{N}(T_1 \vee T_2) &= \mathcal{N}(T_1) \vee \mathcal{N}(T_2) \\
\mathcal{N}(\neg T) &= \mathcal{N}'(T) \\
\mathcal{N}(\mathbb{0}) &= \mathbb{0} \\
\mathcal{N}'(a) &= \neg a \\
\mathcal{N}'(T_1 \vee T_2) &= \bigvee_{i \in I, j \in J} \left(\bigwedge_{a \in P_i \cup P_j} a \wedge \bigwedge_{a \in N_i \cup N_j} \neg a \right) \\
&\qquad \text{where } \mathcal{N}'(T_1) = \bigvee_{i \in I} \left(\bigwedge_{a \in P_i} a \wedge \bigwedge_{a \in N_i} \neg a \right) \text{ and } \mathcal{N}'(T_2) = \bigvee_{j \in J} \left(\bigwedge_{a \in P_j} a \wedge \bigwedge_{a \in N_j} \neg a \right) \\
\mathcal{N}'(\neg T) &= \mathcal{N}(T) \\
\mathcal{N}'(\mathbb{0}) &= \mathbb{1}
\end{aligned}$$

In the definition above, we see $\mathbb{0}$ as the empty union $\bigvee_{i \in \varnothing} T_i$ and $\mathbb{1}$ as the singleton union of the empty intersection $\bigvee_{i \in \{i_0\}} \bigwedge_{a \in \varnothing} a$.

The first step in the computation of $\mathsf{UDNF}(T)$ is to compute $\mathcal{N}(T)$. Then, assuming

$$\mathcal{N}(T) = \bigvee_{i \in I} \underbrace{\left(\bigwedge_{a \in P_i} a \wedge \bigwedge_{a \in N_i} \neg a \right)}_{\mathcal{I}_i}$$

we define

$$\mathsf{UDNF}(T) \stackrel{\text{def}}{=} \bigvee_{i \in I} \mathcal{I}_i^{\mathsf{basic}} \vee \bigvee_{i \in I} \mathcal{I}_i^{\mathsf{prod}} \vee \bigvee_{i \in I} \mathcal{I}_i^{\mathsf{fun}}$$

where

$$\mathcal{I}_i^{\mathsf{basic}} \stackrel{\text{def}}{=} \mathbb{1}_{\mathcal{B}} \wedge \bigwedge_{a \in P_i \cap (\mathcal{A}_{\mathsf{basic}} \cup \mathcal{V})} a \wedge \bigwedge_{a \in N_i \cap (\mathcal{A}_{\mathsf{basic}} \cup \mathcal{V})} \neg a$$

$$\mathcal{I}_i^{\mathsf{prod}} \stackrel{\text{def}}{=} (\mathbb{1} \times \mathbb{1}) \wedge \bigwedge_{a \in P_i \cap (\mathcal{A}_{\mathsf{prod}} \cup \mathcal{V})} a \wedge \bigwedge_{a \in N_i \cap (\mathcal{A}_{\mathsf{prod}} \cup \mathcal{V})} \neg a$$

$$\mathcal{I}_i^{\mathsf{fun}} \stackrel{\text{def}}{=} (\mathbb{0} \to \mathbb{1}) \wedge \bigwedge_{a \in P_i \cap (\mathcal{A}_{\mathsf{fun}} \cup \mathcal{V})} a \wedge \bigwedge_{a \in N_i \cap (\mathcal{A}_{\mathsf{fun}} \cup \mathcal{V})} \neg a$$

Lemma B.39. *Given any type frame T, $\mathsf{UDNF}(T)$ is a uniform normal form and $\mathsf{UDNF}(T) \simeq_T T$. Moreover, if T is strongly polarized, then $\mathsf{UDNF}(T)$ is strongly polarized.*

Proof. Let T be a type frame. We can check on the definition of $\mathcal{N}$ and $\mathcal{N}'$ that $\mathcal{N}(T)$ is a union of intersection of atoms, assuming that we see $\mathbb{0}$ and $\mathbb{1}$ as described above and that atoms are interpreted as singleton unions of singleton intersections. We can check by induction on T that

$$[\![T]\!] = [\![\mathcal{N}(T)]\!] = [\![\neg\mathcal{N}'(T)]\!] .$$

Moreover, when T is strongly polarized, $\mathcal{N}(T)$ is strongly polarized too, because every atom of T appears in $\mathcal{N}(T)$ with the same polarity.

We now consider $\mathsf{UDNF}(T)$. It is trivial to check that it is always in disjunctive normal form. Preservation of strong polarization is also ensured by the fact that we are maintaining the polarity every atom had in $\mathcal{N}(T)$. The conditions that every intersection contains at least one positive atom and that the intersections are uniform are ensured by construction.

It remains to check $\mathsf{UDNF}(T) \simeq_T \mathcal{N}(T)$. Note that $\mathbb{1} \simeq_T \mathbb{1}_{\mathcal{B}} \vee (\mathbb{1} \times \mathbb{1}) \vee (\mathbb{0} \to \mathbb{1})$. We have the following equivalences.

$$\begin{aligned}
\mathcal{I}_i &\simeq_T \mathbb{1} \wedge \mathcal{I}_i \\
&\simeq_T \big(\mathbb{1}_{\mathcal{B}} \vee (\mathbb{1} \times \mathbb{1}) \vee (\mathbb{0} \to \mathbb{1})\big) \wedge \mathcal{I}_i \\
&\simeq_T \big(\mathbb{1}_{\mathcal{B}} \wedge \mathcal{I}_i\big) \vee \big((\mathbb{1} \times \mathbb{1}) \wedge \mathcal{I}_i\big) \vee \big((\mathbb{0} \to \mathbb{1}) \wedge \mathcal{I}_i\big)
\end{aligned}$$

We show the following three results.

$$\begin{aligned}
\mathbb{1}_{\mathcal{B}} \wedge \mathcal{I}_i &\simeq_T \mathcal{I}_i^{\mathsf{basic}} \\
(\mathbb{1} \times \mathbb{1}) \wedge \mathcal{I}_i &\simeq_T \mathcal{I}_i^{\mathsf{prod}} \\
(\mathbb{0} \to \mathbb{1}) \wedge \mathcal{I}_i &\simeq_T \mathcal{I}_i^{\mathsf{fun}}
\end{aligned}$$

For the first implication, we have

$$\begin{aligned}
\mathbb{1}_{\mathcal{B}} \wedge \mathcal{I}_i &= \mathbb{1}_{\mathcal{B}} \wedge \textstyle\bigwedge_{a \in P_i} a \wedge \bigwedge_{a \in N_i} \neg a \\
&\simeq_T \mathbb{1}_{\mathcal{B}} \wedge \textstyle\bigwedge_{a \in P_i \cap (\mathcal{A}_{\mathsf{basic}} \cup \mathcal{V})} a \wedge \bigwedge_{a \in N_i \cap (\mathcal{A}_{\mathsf{basic}} \cup \mathcal{V})} \neg a \wedge \\
&\qquad \textstyle\bigwedge_{a \in P_i \cap (\mathcal{A}_{\mathsf{prod}} \cup \mathcal{A}_{\mathsf{fun}})} a \wedge \bigwedge_{a \in N_i \cap (\mathcal{A}_{\mathsf{prod}} \cup \mathcal{A}_{\mathsf{fun}})} \neg a \\
&\simeq_T \mathbb{1}_{\mathcal{B}} \wedge \textstyle\bigwedge_{a \in P_i \cap (\mathcal{A}_{\mathsf{basic}} \cup \mathcal{V})} a \wedge \bigwedge_{a \in N_i \cap (\mathcal{A}_{\mathsf{basic}} \cup \mathcal{V})} \neg a \wedge \bigwedge_{a \in N_i \cap (\mathcal{A}_{\mathsf{prod}} \cup \mathcal{A}_{\mathsf{fun}})} \neg a
\end{aligned}$$

(because $P_i \cap (\mathcal{A}_{\mathsf{prod}} \cup \mathcal{A}_{\mathsf{fun}}) = \varnothing$: otherwise the intersection would be empty because, when $a \in \mathcal{A}_{\mathsf{prod}} \cup \mathcal{A}_{\mathsf{fun}}$, $[\![a]\!] \cap [\![\mathbb{1}_{\mathcal{B}}]\!] = \varnothing$)

$$\simeq_T \mathbb{1}_{\mathcal{B}} \wedge \textstyle\bigwedge_{a \in P_i \cap (\mathcal{A}_{\mathsf{basic}} \cup \mathcal{V})} a \wedge \bigwedge_{a \in N_i \cap (\mathcal{A}_{\mathsf{basic}} \cup \mathcal{V})} \neg a$$

(because, when $a \in \mathcal{A}_{\mathsf{prod}} \cup \mathcal{A}_{\mathsf{fun}}$, since $[\![a]\!] \cap [\![\mathbb{1}_{\mathcal{B}}]\!] = \varnothing$, we have $[\![\mathbb{1}_{\mathcal{B}}]\!] \subseteq [\![\neg a]\!]$). The other two implications are shown identically.

To conclude, we observe the following equivalence.

$$\begin{aligned}\bigvee_{i\in I}\mathcal{I}_i &\simeq_T \bigvee_{i\in I}\Big(\big(\mathbb{1}_{\mathcal{B}}\wedge\mathcal{I}_i\big)\vee\big((\mathbb{1}\times\mathbb{1})\wedge\mathcal{I}_i\big)\vee\big((\mathbb{0}\to\mathbb{1})\wedge\mathcal{I}_i\big)\Big)\\ &\simeq_T \bigvee_{i\in I}\big(\mathbb{1}_{\mathcal{B}}\wedge\mathcal{I}_i\big)\vee\bigvee_{i\in I}\big((\mathbb{1}\times\mathbb{1})\wedge\mathcal{I}_i\big)\vee\bigvee_{i\in I}\big((\mathbb{0}\to\mathbb{1})\wedge\mathcal{I}_i\big)\\ &\simeq_T \bigvee_{i\in I}\mathcal{I}_i^{\mathsf{basic}}\vee\bigvee_{i\in I}\mathcal{I}_i^{\mathsf{prod}}\vee\bigvee_{i\in I}\mathcal{I}_i^{\mathsf{fun}}\end{aligned}$$

□

Definition B.40 (Product decomposition and projections). *Given a type frame $T\leq_T \mathbb{1}\times\mathbb{1}$, we define its decomposition $\pi(T)$ as*

$$\pi(T)\stackrel{\text{def}}{=}\bigcup_{i\in I\,,\mathcal{I}_i\not\leq_T\mathbb{0}}\Bigg\{\Bigg(\underbrace{\bigwedge_{T_1\times T_2\in\overline{P}_i}T_1\wedge\bigwedge_{T_1\times T_2\in N'}\neg T_1}_{\overline{T}_1},\underbrace{\bigwedge_{T_1\times T_2\in\overline{P}_i}T_2\wedge\bigwedge_{T_1\times T_2\in\overline{N}_i\setminus N'}\neg T_2}_{\overline{T}_2}\Bigg)\;\Bigg|\;N'\subseteq\overline{N}_i,\overline{T}_1\not\leq_T\mathbb{0},\overline{T}_2\not\leq_T\mathbb{0}\Bigg\}$$

and its i-th projection $\pi_i(T)$ as

$$\pi_i(T)\stackrel{\text{def}}{=}\bigvee_{(T_1,T_2)\in\pi(T)}T_i$$

where

$$\mathsf{UDNF}(T)=\bigvee_{i\in I}\underbrace{\Bigg(\bigwedge_{a\in P_i}a\wedge\bigwedge_{a\in N_i}\neg a\Bigg)}_{\mathcal{I}_i}$$

and where $\overline{P}_i=P_i\cap\mathcal{A}_{\mathsf{prod}}$ and $\overline{N}_i=N_i\cap\mathcal{A}_{\mathsf{prod}}$.

Lemma B.41. *Let T be a type frame such that $T\leq_T\mathbb{1}\times\mathbb{1}$. Then, for all type frames T_1 and T_2,*

$$T\leq_T T_1\times T_2\iff\bigvee_{(T_1',T_2')\in\pi(T)}T_1'\times T_2'\leq_T T_1\times T_2\,.$$

Proof. Given T, we have

$$\mathsf{UDNF}(T)=\bigvee_{i\in I}\underbrace{\big(\bigwedge_{a\in P_i}a\wedge\bigwedge_{a\in N_i}\neg a\big)}_{\mathcal{I}_i}$$

and, by Lemma B.39, $T\simeq_T\mathsf{UDNF}(T)$. Then, since $T\leq_T\mathbb{1}\times\mathbb{1}$, we have

$$\forall i\in I.\,\mathcal{I}_i=\textstyle\bigwedge_{a\in P_i}a\wedge\bigwedge_{a\in N_i}\neg a\leq_T\mathbb{1}\times\mathbb{1}\,.$$

Consider a i such that $\mathcal{I}_i\not\leq_T\mathbb{0}$. Since each P_i must contain an atom, we have that P_i contains a type frame of the form $T_1\times T_2$. Hence, since intersections are uniform, $P_i\cup N_i\subseteq\mathcal{A}_{\mathsf{prod}}\cup\mathcal{V}$. Moreover, $\mathcal{V}\cap P_i\cap N_i=\varnothing$, otherwise $\mathcal{I}_i$ would be empty.

We have

$$\begin{aligned}T\leq_T T_1\times T_2&\iff\textstyle\bigvee_{i\in I}\big(\bigwedge_{a\in P_i}a\wedge\bigwedge_{a\in N_i}\neg a\big)\leq_T T_1\times T_2\\ &\iff\textstyle\bigvee_{i\in I,\mathcal{I}_i\not\leq_T\mathbb{0}}\big(\bigwedge_{a\in P_i}a\wedge\bigwedge_{a\in N_i}\neg a\big)\leq_T T_1\times T_2\\ &\iff\forall i\in I,\mathcal{I}_i\not\leq_T\mathbb{0}.\;\textstyle\bigwedge_{a\in P_i}a\wedge\bigwedge_{a\in N_i}\neg a\wedge\neg(T_1\times T_2)\leq_T\mathbb{0}\\ &\iff\forall i\in I,\mathcal{I}_i\not\leq_T\mathbb{0}.\;\textstyle\bigwedge_{a\in P_i\cap\mathcal{A}_{\mathsf{prod}}}a\wedge\bigwedge_{a\in N_i\cap\mathcal{A}_{\mathsf{prod}}}\neg a\wedge\neg(T_1\times T_2)\leq_T\mathbb{0}\end{aligned}$$

(by Lemma B.9, since $P_i \cap \mathcal{V}$ and $N_i \cap \mathcal{V}$ are disjoint; let $\overline{P}_i = P_i \cap \mathcal{A}_{\text{prod}}$ and $\overline{N}_i = N_i \cap \mathcal{A}_{\text{prod}}$)

$$\iff \forall i \in I, \mathcal{I}_i \not\leq_T \mathbb{0}.\ \bigwedge_{T_1' \times T_2' \in \overline{P}_i} T_1' \times T_2' \wedge \bigwedge_{T_1' \times T_2' \in \overline{N}_i} \neg(T_1' \times T_2') \wedge \neg(T_1 \times T_2) \leq_T \mathbb{0}$$

$$\iff \forall i \in I, \mathcal{I}_i \not\leq_T \mathbb{0}.\ \bigwedge_{T_1' \times T_2' \in \overline{P}_i} T_1' \times T_2' \leq_T \left(\bigvee_{T_1' \times T_2' \in \overline{N}_i} T_1' \times T_2'\right) \vee (T_1 \times T_2)$$

We now apply Lemma B.7 to the above to derive the following equivalence.

$$T \leq_T T_1 \times T_2 \iff \forall i \in I, \mathcal{I}_i \not\leq_T \mathbb{0}.$$

$$\left(\forall N' \subseteq \overline{N}_i. \left(\bigwedge_{T_1' \times T_2' \in \overline{P}_i} T_1' \leq_T \bigvee_{T_1' \times T_2' \in N'} T_1'\right) \vee \left(\bigwedge_{T_1' \times T_2' \in \overline{P}_i} T_2' \leq_T \left(\bigvee_{T_1' \times T_2' \in \overline{N}_i \setminus N'} T_2'\right) \vee T_2\right)\right) \wedge$$

$$\left(\forall N' \subseteq \overline{N}_i. \left(\bigwedge_{T_1' \times T_2' \in \overline{P}_i} T_1' \leq_T \left(\bigvee_{T_1' \times T_2' \in N'} T_1'\right) \vee T_1\right) \vee \left(\bigwedge_{T_1' \times T_2' \in \overline{P}_i} T_2' \leq_T \bigvee_{T_1' \times T_2' \in \overline{N}_i \setminus N'} T_2'\right)\right)$$

We have split the quantification over all N' into two: we consider the type $T_1 \times T_2$ in the second case and not in the first.

We also have

$$\bigvee_{(T_1'', T_2'') \in \pi(T)} T_1'' \times T_2'' \leq_T T_1 \times T_2 \iff \forall (T_1'', T_2'') \in \pi(T).\ T_1'' \times T_2'' \leq_T T_1 \times T_2$$

$$\iff \forall (T_1'', T_2'') \in \pi(T).\ (T_1'' \leq_T T_1) \wedge (T_2'' \leq_T T_2)$$

with

$$\pi(T) = \bigcup_{i \in I, \mathcal{I}_i \not\leq_T \mathbb{0}} \left\{ \Big(\underbrace{\bigwedge_{T_1' \times T_2' \in \overline{P}_i} T_1' \wedge \bigwedge_{T_1' \times T_2' \in N'} \neg T_1'}_{\overline{T}_1}, \underbrace{\bigwedge_{T_1' \times T_2' \in \overline{P}_i} T_2' \wedge \bigwedge_{T_1' \times T_2' \in \overline{N}_i \setminus N'} \neg T_2'}_{\overline{T}_2} \Big) \;\middle|\; N' \subseteq \overline{N}_i, \overline{T}_1 \not\leq_T \mathbb{0}, \overline{T}_2 \not\leq_T \mathbb{0} \right\}$$

To show

$$T \leq_T T_1 \times T_2 \iff \bigvee_{(T_1'', T_2'') \in \pi(T)} T_1'' \times T_2'' \leq_T T_1 \times T_2$$

we first show the implication from left to right. Let $(T_1'', T_2'') \in \pi(T)$. Since $\pi(T)$ is a union, (T_1'', T_2'') must be in at least one set in the union; we assume it is the set indexed by $i_0 \in I$. We must show $T_1'' \leq_T T_1$ and $T_2'' \leq_T T_2$.

By definition, (T_1'', T_2'') is a pair corresponding to some $N' \subseteq \overline{N}_{i_0}$. In that case, we must check

$$\bigwedge_{T_1' \times T_2' \in \overline{P}_{i_0}} T_1' \wedge \bigwedge_{T_1' \times T_2' \in N'} \neg T_1' \leq_T T_1 \qquad \bigwedge_{T_1' \times T_2' \in \overline{P}_{i_0}} T_2' \wedge \bigwedge_{T_1' \times T_2' \in \overline{N}_{i_0} \setminus N'} \neg T_2' \leq_T T_2\,,$$

which is

$$\bigwedge_{T_1' \times T_2' \in \overline{P}_{i_0}} T_1' \leq_T \left(\bigvee_{T_1' \times T_2' \in N'} T_1'\right) \vee T_1 \qquad \bigwedge_{T_1' \times T_2' \in \overline{P}_{i_0}} T_2' \leq_T \left(\bigvee_{T_1' \times T_2' \in \overline{N}_{i_0} \setminus N'} T_2'\right) \vee T_2\,.$$

We know from the condition on the set that

$$\bigwedge_{T_1' \times T_2' \in \overline{P}_{i_0}} T_1' \not\leq_T \bigvee_{T_1' \times T_2' \in N'} T_1' \qquad \bigwedge_{T_1' \times T_2' \in \overline{P}_{i_0}} T_2' \not\leq_T \bigvee_{T_1' \times T_2' \in \overline{N}_{i_0} \setminus N'} T_2'\,.$$

We can check the two relations in the decomposition above obtained by Lemma B.7 using the relations that do not hold to eliminate one case in the disjunction.

To check the other direction of the implication, we assume that, for all $(T_1'', T_2'') \in \pi(T)$, we have $(T_1'' \leq_T T_1) \wedge (T_2'' \leq_T T_2)$. We prove that the conditions obtained from the decomposition of subtyping hold. Consider an arbitrary $i \in I$. As $\pi(T)$ is a union, we will consider the set indexed by the same i. For the first condition, we take an arbitrary N'. If there is a pair corresponding to the same N' in $\pi(T)$, then we show the second disjunct. If there is no such pair, it is because $\overline{T}_1$ or $\overline{T}_2$

is empty, which also allows us to conclude. For the second condition, we consider an arbitrary N' and proceed analogously. □

Lemma B.42. *Let T be a type frame such that $T \leq_T \mathbb{1} \times \mathbb{1}$. Then $T \leq_T \pi_1(T) \times \pi_2(T)$. Moreover, if $T \leq_T T_1 \times T_2$, then $\pi_1(T) \leq_T T_1$ and $\pi_2(T) \leq_T T_2$.*

Proof. We have $\bigvee_{(T_1',T_2')\in\pi(T)} T_1' \times T_2' \leq_T (\bigvee_{(T_1',T_2')\in\pi(T)} T_1') \times (\bigvee_{(T_1',T_2')\in\pi(T)} T_2') = \pi_1(T) \times \pi_2(T)$. Hence, by Lemma B.42, we have $T \leq_T \pi_1(T) \times \pi_2(T)$.

If $T \leq_T T_1 \times T_2$, again by Lemma B.42, we have $\bigvee_{(T_1',T_2')\in\pi(T)} T_1' \times T_2' \leq_T T_1 \times T_2$. Hence, for every $(T_1', T_2') \in \pi(T)$, we have $T_1' \times T_2' \leq_T T_1 \times T_2$ and, by the definition of subtyping, $T_1' \leq_T T_1$ and $T_2' \leq_T T_2$, since T_1' and T_2' are not empty. As a result, we also have $\bigvee_{(T_1',T_2')\in\pi(T)} T_1' \leq_T T_1$ and $\bigvee_{(T_1',T_2')\in\pi(T)} T_2' \leq_T T_2$, that is, $\pi_1(T) \leq_T T_1$ and $\pi_2(T) \leq_T T_2$. □

Lemma B.43. *Let T be a type frame such that $T \leq_T \mathbb{1} \times \mathbb{1}$. If T is strongly polarized, then $\pi_1(T)$ and $\pi_2(T)$ are strongly polarized.*

Proof. If T is strongly polarized, then, by Lemma B.39, $\mathsf{UDNF}(T)$ is strongly polarized too. We can check on the definition of $\pi(T)$ that, in every $(T_1, T_2) \in \pi(T)$, subterms of $\mathsf{UDNF}(T)$ appear in T_i with the same polarity as in $\mathsf{UDNF}(T)$. Then, $\pi_i(T)$ also preserves polarity. □

Definition B.44 (Function domain and decomposition). *Given a type frame $T \leq_T \mathbb{0} \to \mathbb{1}$, we define its* domain $\mathsf{dom}(T)$ *as*

$$\mathsf{dom}(T) \stackrel{\text{def}}{=} \bigwedge_{i\in I, \mathcal{I}_i \not\leq_T \mathbb{0}} \bigvee_{T_1\to T_2 \in \overline{P}_i} T_1$$

and its decomposition $\phi(T)$ *as*

$$\phi(T) \stackrel{\text{def}}{=} \bigcup_{i\in I, \mathcal{I}_i \not\leq_T \mathbb{0}} \left\{ \left(\bigvee_{T_1\to T_2 \in P'} T_1, \bigwedge_{T_1\to T_2 \in \overline{P}_i \setminus P'} T_2 \right) \middle| P' \subsetneq \overline{P}_i \right\}$$

where

$$\mathsf{UDNF}(T) = \bigvee_{i\in I} \underbrace{\left(\bigwedge_{a\in P_i} a \wedge \bigwedge_{a\in N_i} \neg a \right)}_{\mathcal{I}_i}$$

and where $\overline{P}_i = P_i \cap \mathcal{A}_{\mathsf{fun}}$ and $\overline{N}_i = N_i \cap \mathcal{A}_{\mathsf{fun}}$.

Definition B.45 (Application result type). *Given two type frames T and T' such that $T \leq_T \mathbb{0} \to \mathbb{1}$ and $T' \leq_T \mathsf{dom}(T)$, we define the* application result type $T \circ T'$ *as*

$$T \circ T' \stackrel{\text{def}}{=} \bigvee_{\substack{(T_1,T_2)\in\phi(T) \\ T' \not\leq_T T_1}} T_2 \,.$$

Lemma B.46. *Let T be a type frame such that $T \leq_T \mathbb{0} \to \mathbb{1}$. Then, for all type frames T' and T'',*

$$T \leq_T T' \to T'' \iff \begin{cases} \forall (T_1', T_2') \in \phi(T).\ (T' \leq_T T_1') \vee (T_2' \leq_T T'') \\ \wedge \\ T' \leq_T \mathsf{dom}(T) \end{cases}$$

Proof. Given T, we have

$$\mathsf{UDNF}(T) = \bigvee_{i\in I}\left(\bigwedge_{a\in P_i} a \wedge \bigwedge_{a\in N_i} \neg a\right)$$

and, by Lemma B.39, $T \simeq_T \mathsf{UDNF}(T)$. Then, since $T \leq_T \mathbb{0} \to \mathbb{1}$, we have

$$\forall i \in I.\ \bigwedge_{a\in P_i} a \wedge \bigwedge_{a\in N_i} \neg a \leq_T \mathbb{0} \to \mathbb{1}\ .$$

For every non-empty intersection $\mathcal{I}_i$, since each P_i must contain an atom, we have that P_i contains a type frame of the form $T_1 \to T_2$. Hence, since intersections are uniform, $P_i \cup N_i \subseteq \mathcal{A}_{\mathsf{fun}} \cup \mathcal{V}$. Moreover, $\mathcal{V} \cap P_i \cap N_i = \varnothing$ otherwise $\mathcal{I}_i$ would be empty.

We have

$$\begin{aligned}
T \leq_T T' \to T'' &\iff \bigvee_{i\in I}\left(\bigwedge_{a\in P_i} a \wedge \bigwedge_{a\in N_i} \neg a\right) \leq_T T' \to T'' \\
&\iff \bigvee_{i\in I, \mathcal{I}_i \not\leq_T \mathbb{0}}\left(\bigwedge_{a\in P_i} a \wedge \bigwedge_{a\in N_i} \neg a\right) \leq_T T' \to T'' \\
&\iff \forall i \in I, \mathcal{I}_i \not\leq_T \mathbb{0}.\ \bigwedge_{a\in P_i} a \wedge \bigwedge_{a\in N_i} \neg a \wedge \neg(T' \to T'') \leq_T \mathbb{0} \\
&\iff \forall i \in I, \mathcal{I}_i \not\leq_T \mathbb{0}.\ \bigwedge_{a\in P_i\cap\mathcal{A}_{\mathsf{fun}}} a \wedge \bigwedge_{a\in N_i\cap\mathcal{A}_{\mathsf{fun}}} \neg a \wedge \neg(T' \to T'') \leq_T \mathbb{0}
\end{aligned}$$

(by Lemma B.9, since $P_i \cap \mathcal{V}$ and $N_i \cap \mathcal{V}$ are disjoint; let $\overline{P}_i = P_i \cap \mathcal{A}_{\mathsf{fun}}$ and $\overline{N}_i = N_i \cap \mathcal{A}_{\mathsf{fun}}$)

$$\begin{aligned}
&\iff \forall i \in I, \mathcal{I}_i \not\leq_T \mathbb{0}.\ \bigwedge_{T_1\to T_2\in\overline{P}_i} T_1 \to T_2 \wedge \bigwedge_{T_1\to T_2\in\overline{N}_i} \neg(T_1 \to T_2) \wedge \neg(T' \to T'') \leq_T \mathbb{0} \\
&\iff \forall i \in I, \mathcal{I}_i \not\leq_T \mathbb{0}.\ \bigwedge_{T_1\to T_2\in\overline{P}_i} T_1 \to T_2 \leq_T \left(\bigvee_{T_1\to T_2\in\overline{N}_i} T_1 \to T_2\right) \vee (T' \to T'')
\end{aligned}$$

Now consider the statement of Lemma B.8. Let $\mathbb{P}_i(\overline{T}_1, \overline{T}_2)$ be the proposition

$$\left(\overline{T}_1 \leq_T \bigvee_{T_1\to T_2\in\overline{P}_i} T_1\right) \wedge \left(\forall P' \subsetneq \overline{P}_i.\ \left(\overline{T}_1 \leq_T \bigvee_{T_1\to T_2\in P'} T_1\right) \vee \left(\bigwedge_{T_1\to T_2\in\overline{P}_i\setminus P'} T_2 \leq_T \overline{T}_2\right)\right)$$

By Lemma B.8, we have

$$\bigwedge_{T_1\to T_2\in\overline{P}_i} T_1 \to T_2 \leq_T \bigvee_{T_1\to T_2\in\overline{N}_i} T_1 \to T_2 \iff \exists(\overline{T}_1 \to \overline{T}_2) \in \overline{N}_i.\ \mathbb{P}_i(\overline{T}_1, \overline{T}_2)$$

and, since for all i such that $\mathcal{I}_i \not\leq_T \mathbb{0}$ the subtyping relation on the left does not hold, we know that $\mathbb{P}_i(\overline{T}_1, \overline{T}_2)$ is false for all such i and all $T_1 \to T_2 \in \overline{N}_i$.

We apply Lemma B.8 again to derive

$$\begin{aligned}
&\bigwedge_{T_1\to T_2\in\overline{P}_i} T_1 \to T_2 \leq_T \left(\bigvee_{T_1\to T_2\in\overline{N}_i} T_1 \to T_2\right) \vee (T' \to T'') \iff \\
&\qquad\qquad \left(\exists(\overline{T}_1 \to \overline{T}_2) \in \overline{N}_i.\ \mathbb{P}_i(\overline{T}_1, \overline{T}_2)\right) \vee \mathbb{P}_i(T', T'')
\end{aligned}$$

and hence $\mathbb{P}_i(T', T'')$ must be true for all i verifying $\mathcal{I}_i \not\leq_T \mathbb{0}$ in order for subtyping to hold.

We have therefore shown

$$\begin{aligned}
T \leq_T T' \to T'' &\iff \forall i \in I, \mathcal{I}_i \not\leq_T \mathbb{0}.\ \mathbb{P}_i(T', T'') \\
&\iff \forall i \in I, \mathcal{I}_i \not\leq_T \mathbb{0}.\ \left(T' \leq_T \bigvee_{T_1\to T_2\in\overline{P}_i} T_1\right) \wedge \\
&\qquad \left(\forall P' \subsetneq \overline{P}_i.\ \left(T' \leq_T \bigvee_{T_1\to T_2\in P'} T_1\right) \vee \left(\bigwedge_{T_1\to T_2\in\overline{P}_i\setminus P'} T_2 \leq_T T''\right)\right)
\end{aligned}$$

and we must now show

$$\forall i \in I, \mathcal{I}_i \not\leq_T \mathbb{0}.\ \left(T' \leq_T \bigvee_{T_1 \to T_2 \in \overline{P}_i} T_1\right) \wedge \left(\forall P' \subsetneq \overline{P}_i.\ \left(T' \leq_T \bigvee_{T_1 \to T_2 \in P'} T_1\right) \vee \left(\bigwedge_{T_1 \to T_2 \in \overline{P}_i \setminus P'} T_2 \leq_T T''\right)\right) \iff \begin{cases} \forall (T_1', T_2') \in \phi(T).\ (T' \leq_T T_1') \vee (T_2' \leq_T T'') \\ \wedge \\ T' \leq_T \mathsf{dom}(T) \end{cases}$$

where

$$\mathsf{dom}(T) = \bigwedge_{i \in I, \mathcal{I}_i \not\leq_T \mathbb{0}} \bigvee_{T_1 \to T_2 \in \overline{P}_i} T_1$$

$$\phi(T) = \bigcup_{i \in I, \mathcal{I}_i \not\leq_T \mathbb{0}} \left\{ \left(\bigvee_{T_1 \to T_2 \in P'} T_1, \bigwedge_{T_1 \to T_2 \in \overline{P}_i \setminus P'} T_2\right) \,\middle|\, P' \subsetneq \overline{P}_i \right\}$$

We first prove the implication from left to right. To prove $T' \leq_T \mathsf{dom}(T)$, note that

$$\left(\forall i \in I, \mathcal{I}_i \not\leq_T \mathbb{0}.\ T' \leq_T \bigvee_{T_1 \to T_2 \in \overline{P}_i} T_1\right) \implies T' \leq_T \bigwedge_{i \in I, \mathcal{I}_i \not\leq_T \mathbb{0}} \bigvee_{T_1 \to T_2 \in \overline{P}_i} T_1\ .$$

To prove the first condition, consider an arbitrary $(T_1', T_2') \in \phi(T)$. We have, for some $i \in I$ and $P' \subsetneq \overline{P}_i$ (verifying $\mathcal{I}_i \not\leq_T \mathbb{0}$),

$$(T_1', T_2') = \left(\bigvee_{T_1 \to T_2 \in P'} T_1, \bigwedge_{T_1 \to T_2 \in \overline{P}_i \setminus P'} T_2\right).$$

P' is necessarily also one of the P' considered in the premise of the implication, so the result follows.

To prove the reverse implication, consider an arbitrary $i \in I$. The first condition follows from $T' \leq_T \mathsf{dom}(T)$. Moreover, for every P', a pair exists in $\phi(T)$ such that the second condition holds. □

Lemma B.47. *Let T be a type frame such that $T \leq_T \mathbb{0} \to \mathbb{1}$. Then, $T \leq_T \mathsf{dom}(T) \to \mathbb{1}$. Moreover, if $T \leq_T T' \to \mathbb{1}$, then $T' \leq_T \mathsf{dom}(T)$.*

Proof. Consider the equivalence of Lemma B.46, with $T' = \mathsf{dom}(T)$ and $T'' = \mathbb{1}$. The three conditions on the right-hand side are all verified, the first two since $\mathbb{1}$ is the top element of subtyping and the third by reflexivity. Hence, $T \leq_T \mathsf{dom}(T) \to \mathbb{1}$.

When $T \leq_T T' \to \mathbb{1}$, again by Lemma B.46 we have $T' \leq_T \mathsf{dom}(T)$. □

Lemma B.48. *Let T be a type frame such that $T \leq_T \mathbb{0} \to \mathbb{1}$. If T is strongly polarized, then* $\mathsf{dom}(T)$ *is strongly negatively polarized.*

Proof. If T is strongly polarized, then, by Lemma B.39, $\mathsf{UDNF}(T)$ is strongly polarized too. We just check on the definition of $\mathsf{dom}(T)$ that every T_1 appears in positive position, whereas it appeared in negative position in $\mathsf{UDNF}(T)$ (because it appeared on the left on an arrow $T_1 \to T_2$ in positive position). □

Lemma B.49. *Let T and T' be type frames such that $T \leq_T \mathbb{0} \to \mathbb{1}$ and $T' \leq_T \mathsf{dom}(T)$. Then, $T \leq_T T' \to (T \circ T')$. Moreover, if $T \leq_T T' \to T''$, then $T \circ T' \leq_T T''$.*

Proof. We prove $T \leq_T T' \to (T \circ T')$ by Lemma B.46. We must show the two conditions

$$\forall (T_1', T_2') \in \phi(T).\ (T' \leq_T T_1') \vee (T_2' \leq_T (T \circ T')) \qquad T' \leq_T \mathsf{dom}(T)$$

the second of which holds by hypothesis. To show the first condition, we take an arbitrary $(T_1', T_2') \in \phi(T)$. Either $T' \leq_T T_1'$ holds or not. If it does not hold, then $T_2' \leq_T T \circ T'$ holds because T_2 is a summand in the union of $T \circ T'$.

Now, assuming $T \leq_T T' \to T''$, we must show $T \circ T' \leq_T T''$. By Lemma B.46, we have

$$\forall (T'_1, T'_2) \in \phi(T).\ (T' \leq_T T'_1) \vee (T'_2 \leq_T T'')$$

Hence, for every $(T'_1, T'_2) \in \phi(T)$ such that $T' \not\leq_T T'_1$, we have $T'_2 \leq_T T''$. Then the union of all such T'_2 is also a subtype of T'', which shows that $T \circ T'$ s a subtype of T'' as well. □

Lemma B.50. *Let T and T' be type frames such that $T \leq_T \mathbb{0} \to \mathbb{1}$ and $T' \leq_T \mathrm{dom}(T)$. If T is strongly polarized and T' is strongly negatively polarized, then $T \circ T'$ is strongly polarized.*

Proof. If T is strongly polarized, then, by Lemma B.39, $\mathsf{UDNF}(T)$ is strongly polarized too. We can check on the definition of $T \circ T'$ that subterms of $\mathsf{UDNF}(T)$ in it are all in positive position and they were in positive position also in $\mathsf{UDNF}(T)$. □

B.7 Normal Forms and Operators on Gradual Types

Definition B.51 (Grounding). *For every types $\tau, \tau' \in \mathcal{T}_\tau$ such that $\tau' \preccurlyeq \tau$, we define the* grounding *of τ with respect to τ', noted ${}^{\tau}/_{\tau'}$ as follows:*

$$\begin{array}{rclcrcll}
\tau_1 \vee \tau_2 / \tau'_1 \vee \tau'_2 & = & \tau_1/\tau'_1 \vee \tau_2/\tau'_2 & & \neg\tau/\neg\tau' & = & \neg(\tau/\tau') & \\
\tau_1 \vee \tau_2 / ? & = & \tau_1/? \vee \tau_2/? & & \neg\tau/? & = & \neg(\tau/?) & \\
\tau_1 \to \tau_2 / ? & = & ? \to ? & & \tau_1 \times \tau_2 / ? & = & ? \times ? & \\
b/? & = & b & & \mathbb{0}/? & = & \mathbb{0} & \\
\alpha/? & = & \alpha & & \tau/\tau' & = & \tau' & \textit{otherwise}
\end{array}$$

Proposition B.52. *For all types τ, τ' such that $\tau' \preccurlyeq \tau$, it holds that $\tau' \preccurlyeq {}^{\tau}/_{\tau'} \preccurlyeq \tau$.*

Proof. By induction on the pair (τ', τ), and by cases on τ'.

- $\tau' = ?$. Note that $\tau' \preccurlyeq {}^{\tau}/_{\tau'}$ always holds. We then reason by cases on τ to prove the second materialization.
  - $\tau = ?$. The result is immediate since ${}^{\tau}/_{\tau'} = ?$.
  - $\tau = \alpha$. Once again, ${}^{\tau}/_{\tau'} = \alpha$.
  - $\tau = b$. Once again, ${}^{\tau}/_{\tau'} = b$.
  - $\tau = \tau_1 \times \tau_2$. Then ${}^{\tau}/_{\tau'} = ? \times ?$, and it holds that $? \times ? \preccurlyeq \tau_1 \times \tau_2$.
  - $\tau = \tau_1 \to \tau_2$. Then ${}^{\tau}/_{\tau'} = ? \to ?$, and it holds that $? \to ? \preccurlyeq \tau_1 \to \tau_2$.
  - $\tau = \tau_1 \vee \tau_2$. Then ${}^{\tau}/_{\tau'} = \tau_1/? \vee \tau_2/?$. For every $i \in \{1, 2\}$, we have $? \preccurlyeq \tau_i$ thus by induction hypothesis, ${}^{\tau_i}/_{?} \preccurlyeq \tau_i$. Finally, ${}^{\tau_1}/_{?} \vee {}^{\tau_2}/_{?} \preccurlyeq \tau_1 \vee \tau_2$.
  - $\tau = \neg\tau_0$. Then ${}^{\tau}/_{?} = \neg({}^{\tau_0}/_{?})$. By induction hypothesis, ${}^{\tau_0}/_{?} \preccurlyeq \tau_0$ thus $\neg({}^{\tau_0}/_{?}) \preccurlyeq \neg\tau_0$.
  - $\tau = \mathbb{0}$. Then ${}^{\tau}/_{\tau'} = \mathbb{0}$ and the result is immediate.
- $\tau' = \tau'_1 \vee \tau'_2$. Then necessarily $\tau = \tau_1 \vee \tau_2$, with $\tau'_i \preccurlyeq \tau_i$ for every $i \in \{1, 2\}$. By induction hypothesis, $\tau'_i \preccurlyeq {}^{\tau_i}/_{\tau'_i} \preccurlyeq \tau_i$, and the result follows.
- $\tau' = \neg\tau'_0$. Then necessarily $\tau = \neg\tau_0$ with $\tau'_0 \preccurlyeq \tau_0$. By induction hypothesis, $\tau'_0 \preccurlyeq {}^{\tau_0}/_{\tau'_0} \preccurlyeq \tau_0$ and the result follows.
- Otherwise, ${}^{\tau}/_{\tau'} = \tau'$ and since $\tau' \preccurlyeq \tau$ the result is immediate.

□

Lemma B.53. *For all types τ, τ' that do not contain type connectives, if $\tau \preccurlyeq \tau'$ and ${}^{\tau'}/_{\tau} \neq \tau$ then $\tau = ?$.*

Proof. Eliminating all cases involving connectives in Definition B.51 as well as the case ${}^{\tau'}/_{\tau} = \tau$, the only remaining cases are those were $\tau = ?$. □

Lemma B.54. *For all types τ, τ' that do not contain type connectives such that ${}^{\tau'}/_{\tau} = \tau'$, then $\tau' = \tau$ or $\tau = ?$.*

Proof. Suppose that $\tau \neq ?$. Eliminating all cases involving connectives or where $\tau = ?$ in Definition B.51, the only remaining case is ${\tau'}/_{\tau} = \tau$. However, by hypothesis, ${\tau'}/_{\tau} = \tau'$ therefore $\tau = \tau'$. □

Lemma B.55. *For all types τ, τ' such that ${\tau'}/_{\tau} = \tau'$, the following holds:*

$$\forall d^L \in [\![\tau'^{\textcircled{\scriptsize A}}]\!], d^{L\cup\{X_1\}\setminus\{X_0\}} \in [\![\tau^{\textcircled{\scriptsize A}}]\!]$$

$$\forall d^L \notin [\![\tau'^{\textcircled{\scriptsize V}}]\!], d^{L\cup\{X_1\}\setminus\{X_0\}} \notin [\![\tau^{\textcircled{\scriptsize V}}]\!]$$

Proof. The two results are proved simultaneously by induction over the pair (d^L, τ).

- $\tau = ?$. Since $X_1 \in \mathrm{tags}(d^{L\cup\{X_1\}\setminus\{X_0\}})$, it is immediate that $d^{L\cup\{X_1\}\setminus\{X_0\}} \in [\![\tau^{\textcircled{\scriptsize A}}]\!]$. Moreover, $X_0 \notin \mathrm{tags}(d^{L\cup\{X_1\}\setminus\{X_0\}})$ hence $d^{L\cup\{X_1\}\setminus\{X_0\}} \notin [\![\tau^{\textcircled{\scriptsize V}}]\!]$.
- $\tau = \alpha$. By hypothesis, we have $\tau = \tau' = \alpha$. Therefore, for every $d^L \in [\![\tau'^{\textcircled{\scriptsize A}}]\!]$, it holds that $\alpha \in L$. Thus $\alpha \in \mathrm{tags}(d^{L\cup\{X_1\}\setminus\{X_0\}})$, hence the first result. The second result is proved using the same reasoning.
- $\tau = b$. By hypothesis, since ${\tau'}/_{\tau} = \tau'$, we have $\tau = \tau' = b$, and the result is immediate since the interpretation of a constant contains all possible sets of labels.
- $\tau = \tau_1 \times \tau_2$. Since ${\tau'}/_{\tau} = \tau'$, necessarily $\tau = \tau'$ and the result is immediate for the same reason as the previous case.
- $\tau = \tau_1 \to \tau_2$. Once again, necessarily $\tau = \tau'$ and the result is immediate.
- $\tau = \tau_1 \vee \tau_2$. Let $d^L \in [\![\tau^{\textcircled{\scriptsize A}}]\!]$. There exists $i \in \{1, 2\}$ such that $d^L \in [\![\tau_i^{\textcircled{\scriptsize A}}]\!]$. Thus, by induction hypothesis, it holds that $d^{L\cup\{X_1\}\setminus\{X_0\}} \in [\![\tau_i^{\textcircled{\scriptsize A}}]\!]$. Therefore, we have $d^{L\cup\{X_1\}\setminus\{X_0\}} \in [\![\tau^{\textcircled{\scriptsize A}}]\!]$, which is the result. The same reasoning can be done for the second case.
- $\tau = \neg\tau'$. Let $d^L \in [\![\tau^{\textcircled{\scriptsize A}}]\!]$. By definition, $d^L \notin [\![\tau'^{\textcircled{\scriptsize V}}]\!]$. By induction hypothesis, we therefore have $d^{L\cup\{X_1\}\setminus\{X_0\}} \notin [\![\tau'^{\textcircled{\scriptsize V}}]\!]$. Thus, $d^{L\cup\{X_1\}\setminus\{X_0\}} \in [\![\tau^{\textcircled{\scriptsize A}}]\!]$. The same reasoning can be done for the second result.

□

Corollary B.56. *For all types τ, τ' such that ${\tau'}/_{\tau} = \tau'$, and all types τ_l, τ_r such that $\tau \leq \tau_l \to \tau_r$, then $\tau' \leq \tau_l \to \tau_r$.*

Proof. Let $d^L \in [\![\tau'^{\textcircled{\scriptsize A}}]\!]$. By Lemma B.55, we know that $d^{L\cup\{X_1\}\setminus\{X_0\}} \in [\![\tau^{\textcircled{\scriptsize A}}]\!]$. Moreover, by hypothesis, $\tau \leq \tau_l \to \tau_r$, thus, by Proposition B.28, $d^{L\cup\{X_1\}\setminus\{X_0\}} \in [\![(\tau_l \to \tau_r)^{\textcircled{\scriptsize A}}]\!]$. However, the fact that an element of $\mathcal{D}$ belongs to $[\![(\tau_l \to \tau_r)^{\textcircled{\scriptsize A}}]\!]$ is independent of its set of labels, therefore $d^L \in [\![(\tau_l \to \tau_r)^{\textcircled{\scriptsize A}}]\!]$. Thus, we obtain that $\tau'^{\textcircled{\scriptsize A}} \leq_T (\tau_l \to \tau_r)^{\textcircled{\scriptsize A}}$ and the result follows by Proposition B.28. □

Corollary B.57. *For all types τ, τ' such that ${\tau'}/_{\tau} = \tau'$, and all types τ_l, τ_r such that $\tau \leq \tau_l \times \tau_r$, then $\tau' \leq \tau_l \times \tau_r$.*

Proof. Let $d^L \in [\![\tau'^{\textcircled{\scriptsize A}}]\!]$. By Lemma B.55, we know that $d^{L\cup\{X_1\}\setminus\{X_0\}} \in [\![\tau^{\textcircled{\scriptsize A}}]\!]$. Moreover, by hypothesis, $\tau \leq \tau_l \times \tau_r$, thus, by Proposition B.28, $d^{L\cup\{X_1\}\setminus\{X_0\}} \in [\![(\tau_l \times \tau_r)^{\textcircled{\scriptsize A}}]\!]$. However, the fact that an element of $\mathcal{D}$ belongs to $[\![(\tau_l \times \tau_r)^{\textcircled{\scriptsize A}}]\!]$ is independent of its set of labels, therefore $d^L \in [\![(\tau_l \times \tau_r)^{\textcircled{\scriptsize A}}]\!]$. Thus, we obtain that $\tau'^{\textcircled{\scriptsize A}} \leq_T (\tau_l \times \tau_r)^{\textcircled{\scriptsize A}}$ and the result follows by Proposition B.28. □

Definition B.58. *We define the function m recursively on $\mathcal{D}$ as follows:*

$$\begin{aligned} m : \mathcal{D} \cup \{\Omega\} &\to \mathcal{D} \cup \{\Omega\} \\ m(\Omega) &= \Omega \\ m(c^L) &= c^{L \cup \{X_0\} \setminus \{X_1\}} \\ m((d_l, d_r)^L) &= (m(d_l), m(d_r))^{L \cup \{X_0\} \setminus \{X_1\}} \\ m((d_1, d_1'), \dots, (d_n, d_n')^L) &= (m(d_1), m(d_1')), \dots, (m(d_n), m(d_n'))^{L \cup \{X_0\} \setminus \{X_1\}} \end{aligned}$$

Lemma B.59. *For all types τ, τ' such that $\tau \preccurlyeq \tau'$, the following holds:*

$$\forall d \in [\![\tau'^{\langle\!\!\wedge\!\!\rangle}]\!], m(d) \in [\![\tau^{\langle\!\!\wedge\!\!\rangle}]\!]$$

$$\forall d \notin [\![\tau'^{\langle\!\!\vee\!\!\rangle}]\!], m(d) \notin [\![\tau^{\langle\!\!\vee\!\!\rangle}]\!]$$

Proof. The two results are proved simultaneously by induction on the pair (d, τ).

- $\tau = ?$. For every $d \in \mathcal{D}$, $X_0 \in \text{tags}(m(d))$ by Definition B.58. Therefore $m(d) \in [\![X_0]\!] = [\![\tau^{\langle\!\!\wedge\!\!\rangle}]\!]$. Similarly, $X_1 \notin \text{tags}(m(d))$, thus $m(d) \notin [\![X_1]\!] = [\![\tau^{\langle\!\!\vee\!\!\rangle}]\!]$.
- $\tau = \alpha$. Immediate since, by hypothesis, $\tau \preccurlyeq \tau'$ therefore $\tau = \tau' = \alpha$.
- $\tau = b$. Immediate since, by hypothesis, $\tau \preccurlyeq \tau'$ therefore $\tau = \tau' = b$.
- $\tau = \tau_1 \times \tau_2$. By hypothesis, $\tau' = \tau_1' \times \tau_2'$ with $\tau_1 \preccurlyeq \tau_1'$ and $\tau_2 \preccurlyeq \tau_2'$. Let $d \in [\![\tau'^{\langle\!\!\wedge\!\!\rangle}]\!]$. Since $[\![\tau'^{\langle\!\!\wedge\!\!\rangle}]\!] = [\![\tau_1'^{\langle\!\!\wedge\!\!\rangle} \times \tau_2'^{\langle\!\!\wedge\!\!\rangle}]\!]$, $d = (d_1, d_2)^L$ for some $d_1, d_2 \in \mathcal{D}$, where $d_i \in [\![\tau_i'^{\langle\!\!\wedge\!\!\rangle}]\!]$, for every $i \in \{1, 2\}$. By induction, it holds that $m(d_i) \in [\![\tau_i^{\langle\!\!\wedge\!\!\rangle}]\!]$, thus $(m(d_1), m(d_2))^{L'} \in [\![\tau_1^{\langle\!\!\wedge\!\!\rangle} \times \tau_2^{\langle\!\!\wedge\!\!\rangle}]\!]$ for every set of tags L'. Hence $m(d) \in [\![\tau^{\langle\!\!\wedge\!\!\rangle}]\!]$.
  Similarly, let $d \notin [\![\tau'^{\langle\!\!\vee\!\!\rangle}]\!]$. If $d \neq (d_1, d_2)^L$ for some $d_1, d_2 \in \mathcal{D}$, then it is immediate that $m(d) \notin [\![\tau^{\langle\!\!\vee\!\!\rangle}]\!]$ since it only contains pairs. Otherwise, if $d = (d_1, d_2)^L$ for some $d_1, d_2 \in \mathcal{D}$, then $d_i \notin [\![\tau_i'^{\langle\!\!\vee\!\!\rangle}]\!]$ for some $i \in \{1, 2\}$. By induction, it holds that $m(d_i) \notin [\![\tau_i^{\langle\!\!\vee\!\!\rangle}]\!]$. Therefore, $(m(d_1), m(d_2))^{L'} \notin [\![\tau_1^{\langle\!\!\vee\!\!\rangle} \times \tau_2^{\langle\!\!\vee\!\!\rangle}]\!]$ for every set of tags L', hence the result.
- $\tau = \tau_1 \to \tau_2$. By hypothesis, $\tau' = \tau_1' \to \tau_2'$ with $\tau_1 \preccurlyeq \tau_1'$ and $\tau_2 \preccurlyeq \tau_2'$. For every $d \in [\![\tau'^{\langle\!\!\wedge\!\!\rangle}]\!]$, d is a relation $(d_1, d_1'), \dots, (d_n, d_n')^L$. Let $i \in \{1, n\}$ such that $m(d_i) \in [\![\tau_1^{\langle\!\!\vee\!\!\rangle}]\!]$. According to the contrapositive of the second induction hypothesis, $d_i \in [\![\tau_1'^{\langle\!\!\vee\!\!\rangle}]\!]$. Therefore, by definition of $[\![\tau'^{\langle\!\!\wedge\!\!\rangle}]\!]$, $d_i' \in [\![\tau_2'^{\langle\!\!\wedge\!\!\rangle}]\!]$. Applying the first induction hypothesis, $m(d_i') \in [\![\tau_2^{\langle\!\!\wedge\!\!\rangle}]\!]$.
  To summarize, $m(d_i) \in [\![\tau_1^{\langle\!\!\vee\!\!\rangle}]\!] \implies m(d_i') \in [\![\tau_2^{\langle\!\!\wedge\!\!\rangle}]\!]$.
  Therefore, $(m(d_1), m(d_1')), \dots, (m(d_n), m(d_n'))^{L'} \in [\![\tau^{\langle\!\!\wedge\!\!\rangle}]\!]$ for every set of tags L, hence the first result.
  Similarly, for every $d \notin [\![\tau'^{\langle\!\!\vee\!\!\rangle}]\!]$, if d is not a relation then it is immediate that $m(d) \notin [\![\tau^{\langle\!\!\vee\!\!\rangle}]\!]$ since $m(d)$ is also not a relation and $[\![\tau^{\langle\!\!\vee\!\!\rangle}]\!]$ only contains relations. Otherwise, if $d = (d_1, d_1'), \dots, (d_n, d_n')^L$, then, by definition of $[\![\tau'^{\langle\!\!\vee\!\!\rangle}]\!]$, there exists a $i \in \{1, n\}$ such that $d_i \in \tau_1'^{\langle\!\!\wedge\!\!\rangle}$ and $d_i' \notin \tau_2'^{\langle\!\!\vee\!\!\rangle}$. The induction hypothesis yields $m(d_i) \in \tau_1^{\langle\!\!\wedge\!\!\rangle}$ and $m(d_i') \notin \tau_2^{\langle\!\!\vee\!\!\rangle}$. Thus, $d \notin [\![\tau^{\langle\!\!\vee\!\!\rangle}]\!]$ independently of its set of tags, hence the result.
- $\tau = \tau_1 \vee \tau_2$. By hypothesis, $\tau' = \tau_1' \vee \tau_2'$ with $\tau_1 \preccurlyeq \tau_1'$ and $\tau_2 \preccurlyeq \tau_2'$. For every $d \in [\![\tau'^{\langle\!\!\wedge\!\!\rangle}]\!]$, $d \in [\![\tau_i'^{\langle\!\!\wedge\!\!\rangle}]\!]$ for some $i \in \{1, 2\}$. Thus, by induction hypothesis, $m(d) \in [\![\tau_i^{\langle\!\!\wedge\!\!\rangle}]\!] \subset [\![\tau^{\langle\!\!\wedge\!\!\rangle}]\!]$, hence the result.
  Similarly, for every $d \notin [\![\tau'^{\langle\!\!\vee\!\!\rangle}]\!]$, $d \notin [\![\tau_i'^{\langle\!\!\vee\!\!\rangle}]\!]$ for every $i \in \{1, 2\}$. Thus, by induction hypothesis, $m(d) \notin [\![\tau_i^{\langle\!\!\vee\!\!\rangle}]\!]$ for every $i \in \{1, 2\}$, hence the result.

- $\tau = \neg\tau_0$. By hypothesis, $\tau' = \neg\tau_0'$ with $\tau_0 \preccurlyeq \tau_0'$. Let $d \in [\![\tau'^{\textcircled{\scriptsize\Delta}}]\!]$. By definition, $d \notin [\![\tau_0'^{\textcircled{\scriptsize\nabla}}]\!]$. By induction, we have $m(d) \notin [\![\tau_0^{\textcircled{\scriptsize\nabla}}]\!]$, hence $m(d) \in [\![\tau^{\textcircled{\scriptsize\Delta}}]\!]$. We can do the same reasoning for $d \notin [\![\tau'^{\textcircled{\scriptsize\nabla}}]\!]$, which concludes this proof.

□

Corollary B.60. *For all types τ, τ' such that $\tau \preccurlyeq \tau'$, if $\tau \leq \mathbb{0} \to \mathbb{1}$ then $\tau' \leq \mathbb{0} \to \mathbb{1}$.*

Proof. Let $d \in [\![\tau'^{\textcircled{\scriptsize\Delta}}]\!]$. By Lemma B.59, it holds that $m(d) \in [\![\tau^{\textcircled{\scriptsize\Delta}}]\!]$. Since $\tau \leq \mathbb{0} \to \mathbb{1}$, Lemma B.28 yields $\tau^{\textcircled{\scriptsize\Delta}} \leq_T \mathbb{0} \to \mathbb{1}$. Thus, $m(d) \in [\![\mathbb{0} \to \mathbb{1}]\!]$, which implies that $m(d) = R^L$ for some relation $R \subset \mathcal{D} \times \mathcal{D} \cup \{\Omega\}$.

By inversion of Definition B.58, $d = R'^L$ for some relation $R' \subset \mathcal{D} \times \mathcal{D} \cup \{\Omega\}$. Thus $d \in [\![\mathbb{0} \to \mathbb{1}]\!]$, which yields that $\tau'^{\textcircled{\scriptsize\Delta}} \leq_T \mathbb{0} \to \mathbb{1}$. Lemma B.28 then gives the result. □

Corollary B.61. *For all types τ, τ' such that $\tau \preccurlyeq \tau'$, if $\tau \leq \mathbb{1} \times \mathbb{1}$ then $\tau' \leq \mathbb{1} \times \mathbb{1}$.*

Proof. Let $d \in [\![\tau'^{\textcircled{\scriptsize\Delta}}]\!]$. By Lemma B.59, it holds that $m(d) \in [\![\tau^{\textcircled{\scriptsize\Delta}}]\!]$. Since $\tau \leq \mathbb{1} \times \mathbb{1}$, Lemma B.28 yields $\tau^{\textcircled{\scriptsize\Delta}} \leq_T \mathbb{1} \times \mathbb{1}$. Thus, $m(d) \in [\![\mathbb{1} \times \mathbb{1}]\!]$, which implies that $m(d) = (d_l, d_r)^L$ for some $d_l, d_r \in \mathcal{D}$.

By inversion of Definition B.58, $d = (d_l', d_r')^L$ for some $d_l', d_r' \in \mathcal{D}$. Thus $d \in [\![\mathbb{1} \times \mathbb{1}]\!]$, which yields that $\tau'^{\textcircled{\scriptsize\Delta}} \leq_T \mathbb{1} \times \mathbb{1}$. Lemma B.28 then gives the result. □

Corollary B.62. *For all types τ, τ' such that $\tau \preccurlyeq \tau'$, if $\tau' \not\leq \mathbb{0}$ then $\tau \not\leq \mathbb{0}$.*

Proof. Since $\tau' \not\leq \mathbb{0}$, by Lemma B.28, it holds that $\tau'^{\textcircled{\scriptsize\Delta}} \not\leq_T \mathbb{0}$. Thus, there exists $d \in [\![\tau'^{\textcircled{\scriptsize\Delta}}]\!]$. Applying Lemma B.59 yields $m(d) \in [\![\tau^{\textcircled{\scriptsize\Delta}}]\!]$, therefore $\tau^{\textcircled{\scriptsize\Delta}} \not\leq_T \mathbb{0}$. Lemma B.28, then yields the result. □

We now extend the previous definition of atoms to gradual types. That is, we refer to a gradual type of the form b, $\tau_1 \times \tau_2$, or $\tau_1 \to \tau_2$ as an atom. We write $\mathcal{A}^?_{\text{basic}}$, $\mathcal{A}^?_{\text{prod}}$, and $\mathcal{A}^?_{\text{fun}}$ for the set of gradual types of the forms b, $\tau_1 \times \tau_2$, and $\tau_1 \to \tau_2$, respectively.

In the following, the metavariable a ranges over the set $\mathcal{A}^?_{\text{basic}} \cup \mathcal{A}^?_{\text{prod}} \cup \mathcal{A}^?_{\text{fun}} \cup \mathcal{V} \cup \{?\}$.

Definition B.63 (Uniform gradual normal form). *A* uniform gradual (disjunctive) normal form *(UGDNF) is a gradual type τ of the form*

$$\bigvee_{i \in I} \left(\bigwedge_{a \in P_i} a \wedge \bigwedge_{a \in N_i} \neg a \right)$$

such that, for all $i \in I$, one of the following three condition holds:

- $P_i \cap \mathcal{A}^?_{\text{basic}} \neq \varnothing$ *and* $(P_i \cup N_i) \cap (\mathcal{A}^?_{\text{prod}} \cup \mathcal{A}^?_{\text{fun}}) = \varnothing$;
- $P_i \cap \mathcal{A}^?_{\text{prod}} \neq \varnothing$ *and* $(P_i \cup N_i) \cap (\mathcal{A}^?_{\text{basic}} \cup \mathcal{A}^?_{\text{fun}}) = \varnothing$;
- $P_i \cap \mathcal{A}^?_{\text{fun}} \neq \varnothing$ *and* $(P_i \cup N_i) \cap (\mathcal{A}^?_{\text{basic}} \cup \mathcal{A}^?_{\text{prod}}) = \varnothing$;

For every type τ, we define $\mathsf{UGDNF}(\tau) = (\mathsf{UDNF}(\tau^\oplus))^\dagger$.

Lemma B.64. *For every type τ, $\mathsf{UGDNF}(\tau)$ is in uniform gradual normal form and $\mathsf{UGDNF}(\tau) \simeq \tau$.*

Proof. We define $\tau' = \mathsf{UGDNF}(\tau)$. From the definition of UGDNF, it is immediate that τ' is in uniform gradual normal form.

Lemma B.39 ensures that $\mathsf{UDNF}(\tau^\oplus) \simeq_T \tau^\oplus$. Moreover, since UDNF preserves the strong polarization, $\mathsf{UDNF}(\tau^\oplus)$ is strongly polarized. By unicity of the strong polarization, $((\mathsf{UDNF}(\tau^\oplus))^\dagger)^\oplus = \mathsf{UDNF}(\tau^\oplus) \simeq_T \tau^\oplus$. Lemma B.28 then yields that $(\mathsf{UDNF}(\tau^\oplus))^\dagger \simeq \tau$, that is, $\tau' \simeq \tau$. □

Lemma B.65. *For every pair of types τ, τ' such that $\tau \,/\, \tau' = \tau'$, the following results hold:*

- $\tau' \in \mathcal{A}^?_{\text{basic}} \iff \tau \in \mathcal{A}^?_{\text{basic}}$
- $\tau' \in \mathcal{A}^?_{\text{fun}} \iff \tau \in \mathcal{A}^?_{\text{fun}}$
- $\tau' \in \mathcal{A}^?_{\text{prod}} \iff \tau \in \mathcal{A}^?_{\text{prod}}$
- $\tau' = ? \iff \tau = ?$
- $\tau' \in \mathcal{V} \iff \tau \in \mathcal{V}$

Proof. The result follows immediately from the definition of ${}^{\tau}/_{\tau'}$. □

We now prove the following lemma about the function $\mathcal{N}$ defined in Subsection B.6.

Lemma B.66. *For every pair of type frames T, T' such that ${}^{T^\dagger}/_{T'^\dagger} = T'^\dagger$, the following holds:*

$$\mathcal{N}(T)^\dagger / \mathcal{N}(T')^\dagger = \mathcal{N}(T')^\dagger$$

$$\mathcal{N}'(T)^\dagger / \mathcal{N}'(T')^\dagger = \mathcal{N}'(T')^\dagger$$

Proof. By induction on the pair (T, T'), and by cases on T'. Since ${}^{\neg\tau}/_{\neg\tau'} = \neg({}^{\tau}/_{\tau'})$, most of the cases are proved similarly for $\mathcal{N}$ and $\mathcal{N}'$, and may be omitted.

- $T' = X'$. Then $T'^\dagger = ?$. Thus, by hypothesis, $T^\dagger = ?$ and therefore $T = X$. In this case $\mathcal{N}(T) = X$ and $\mathcal{N}(T') = X'$, and the result follows.
- $T' = \alpha$. Then necessarily $T = \alpha$ and $\mathcal{N}$ leaves T and T' unchanged, and the result is immediate.
- $T' = b$. Same as previous case.
- $T' = T'_1 \times T'_2$. By hypothesis, T is of the form $T_1 \times T_2$. Thus $\mathcal{N}$ leaves T and T' unchanged, and the result follows.
- $T' = T'_1 \to T'_2$. Same as previous case.
- $T' = T'_1 \vee T'_2$. By hypothesis, T is of the form $T_1 \vee T_2$ where for every $i \in \{1, 2\}$, ${}^{T_i}/_{T'_i} = T'_i$. By induction and definition of $\mathcal{N}$, the first result is immediate.
  For the second result, consider $k \in \{1, 2\}$. We have $\mathcal{N}'(T_k) = \bigvee_{i \in I_k} \left(\bigwedge_{p \in P_i} a_p \wedge \bigwedge_{n \in N_i} \neg a_n \right)$.
  The induction hypothesis ensures that we also have $\mathcal{N}'(T'_k) = \bigvee_{i \in I_k} \left(\bigwedge_{p \in P_i} a'_p \wedge \bigwedge_{n \in N_i} \neg a'_n \right)$
  where for every $i \in I_k$, for every $p \in P_i$, ${}^{a^\dagger_p}/_{a'^\dagger_p} = a'^\dagger_p$ and similarly for every $n \in N_i$.
  Thus, for every pair $(i_1, i_2) \in (I_1 \times I_2)$, noting $T_I = \bigwedge_{p \in P_{i_1} \cup P_{i_2}} a_p \wedge \bigwedge_{n \in N_{i_1} \cup N_{i_2}} \neg a_n$ and $T'_I = \bigwedge_{p \in P_{i_1} \cup P_{i_2}} a'_p \wedge \bigwedge_{n \in N_{i_1} \cup N_{i_2}} \neg a'_n$, we have ${}^{T^\dagger_I}/_{T'^\dagger_I} = T'^\dagger_I$. Taking the union over all pairs $(i_1, i_2) \in (I_1 \times I_2)$ yields the result.
- $T' = \neg T'_0$. By hypothesis, T is of the form $\neg T_0$, where ${}^{T^\dagger_0}/_{T'^\dagger_0} = T'^\dagger_0$. By induction hypothesis, $\mathcal{N}'(T_0)^\dagger / \mathcal{N}'(T'_0)^\dagger = \mathcal{N}'(T'_0)^\dagger$. Thus $\mathcal{N}(\neg T_0)^\dagger / \mathcal{N}(\neg T'_0)^\dagger = \mathcal{N}(\neg T'_0)^\dagger$, which yields the result. The same reasoning can be done with $\mathcal{N}'$.
- $T' = \mathbb{0}$. Necessarily $T = \mathbb{0}$, thus $\mathcal{N}$ leaves T and T' unchanged, and the result follows.

□

Proposition B.67. *For every pair of types τ, τ' such that ${}^{\tau}/_{\tau'} = \tau'$, ${}^{\mathsf{UGDNF}(\tau)}/_{\mathsf{UGDNF}(\tau')} = \mathsf{UGDNF}(\tau')$.*

Proof. Let τ, τ' be two types such that ${}^{\tau}/_{\tau'} = \tau'$. Then applying Lemma B.66 to $\tau^\oplus$ and $\tau'^\oplus$ immediately yields that $\mathcal{N}(\tau^\oplus)^\dagger / \mathcal{N}(\tau'^\oplus)^\dagger = \mathcal{N}(\tau'^\oplus)^\dagger$.

Now, assuming that

$$\mathcal{N}(\tau'^{\oplus}) = \bigvee_{i\in I} \Big(\underbrace{\bigwedge_{p\in P_i} a'_p \wedge \bigwedge_{n\in N_i} \neg a'_n}_{\mathcal{I}'_i} \Big)$$

By definition of the grounding operation, we have

$$\mathcal{N}(\tau^{\oplus}) = \bigvee_{i\in I} \Big(\underbrace{\bigwedge_{p\in P_i} a_p \wedge \bigwedge_{n\in N_i} \neg a_n}_{\mathcal{I}_i} \Big)$$

where for every $i \in I$, for every $(p, n) \in P_i \times N_i$, $a_p^\dagger / {a'_p}^\dagger = {a'_p}^\dagger$ and $a_n^\dagger / {a'_n}^\dagger = {a'_n}^\dagger$.

Lemma B.65 then guarantees that for every $i \in I$, $a_p \in P_i \cap (\mathcal{A}_{\text{basic}} \cup \mathcal{V}) \iff a'_p \in P_i \cap (\mathcal{A}_{\text{basic}} \cup \mathcal{V})$. Therefore, following the definitions of Subsection B.6, posing $T_1 = \mathcal{I}_i^{\text{basic}}$ and $T_2 = \mathcal{I}_i'^{\text{basic}}$, it holds that $T_1^\dagger / T_2^\dagger = T_2^\dagger$. The same reasoning can be done for the product and function intersections, yielding $\mathsf{UDNF}(\tau^{\oplus})^\dagger / \mathsf{UDNF}(\tau'^{\oplus})^\dagger = \mathsf{UDNF}(\tau'^{\oplus})^\dagger$, and the result follows by definition of UGDNF. □

Definition B.68 (Function Cast Approximation). *For every pair of types τ, τ' such that $\tau' \leq \mathbb{0} \to \mathbb{1}$, and every type σ, if*

$$\mathsf{UGDNF}(\tau) = \bigvee_{i\in I} \underbrace{\bigwedge_{p\in P_i} a_p \wedge \bigwedge_{n\in N_i} \neg a_n}_{\mathcal{I}_i}$$

$$\mathsf{UGDNF}(\tau') = \bigvee_{i\in I} \underbrace{\bigwedge_{p\in P_i} a'_p \wedge \bigwedge_{n\in N_i} \neg a'_n}_{\mathcal{I}'_i}$$

$$\forall i \in I, \mathcal{I}_i \not\leq \mathbb{0} \implies \mathcal{I}'_i \not\leq \mathbb{0}$$

$$\forall i \in I, \forall p \in P_i, a_p \in \mathcal{A}^?_{\text{fun}} \iff a'_p \in \mathcal{A}^?_{\text{fun}}$$

then we define the approximation of $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle$ applied to σ, noted $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle \circ \sigma$ as follows.

$$\langle \tau \overset{p}{\Rightarrow} \tau' \rangle \circ \sigma = \Bigg\langle \bigwedge_{\substack{i\in I \\ \mathcal{I}'_i \not\leq \mathbb{0}}} \bigwedge_{\sigma \leq \bigvee_{p\in S} \sigma'_p} \bigvee_{p\in S} \sigma_p \;\to\; \bigvee_{\substack{i\in I \\ \mathcal{I}'_i \not\leq \mathbb{0}}} \bigvee_{\sigma \not\leq \bigvee_{p\in S} \sigma'_p} \bigwedge_{p\in \bar{P}_i \setminus S} \tau_p \overset{p}{\Rightarrow} \bigwedge_{\substack{i\in I \\ \mathcal{I}'_i \not\leq \mathbb{0}}} \bigwedge_{\sigma \leq \bigvee_{p\in S} \sigma'_p} \bigvee_{p\in S} \sigma'_p \;\to\; \bigvee_{\substack{i\in I \\ \mathcal{I}'_i \not\leq \mathbb{0}}} \bigvee_{\sigma \not\leq \bigvee_{p\in S} \sigma'_p} \bigwedge_{p\in \bar{P}_i \setminus S} \tau'_p \Bigg\rangle$$

where, to ease the notation, we pose $\bar{P}_i = \{p \in P_i \mid a_p \in \mathcal{A}^?_{\text{fun}}\} = \{p \in P_i \mid a'_p \in \mathcal{A}^?_{\text{fun}}\}$ and for every $p \in \bar{P}_i$, $a_p = \sigma_p \to \tau_p$ and $a'_p = \sigma'_p \to \tau'_p$.

Otherwise, $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle \circ \sigma$ is undefined.

In the future, we use $P_i \cap \mathcal{A}^?_{\text{fun}}$ as a shorthand for both $\{p \in P_i \mid a_p \in \mathcal{A}^?_{\text{fun}}\}$ and $\{p \in P_i \mid a'_p \in \mathcal{A}^?_{\text{fun}}\}$, provided the fourth condition of the above definition holds.

Lemma B.69. *For every pair of types τ, τ', and every type σ, if $\langle \tau \xRightarrow{p} \tau' \rangle \circ \sigma = \langle \tau_1 \to \tau_2 \xRightarrow{p} \tau_1' \to \tau_2' \rangle$ then the following holds:*

$$\tau / \tau' = \tau' \implies \forall i, \tau_i / \tau_i' = \tau_i'$$
$$\tau' / \tau = \tau \implies \forall i, \tau_i' / \tau_i = \tau_i$$

Proof. Given two types τ, τ' such that $\tau / \tau' = \tau'$, and any type σ, Proposition B.67 ensures that

$$\mathsf{UGDNF}(\tau) = \bigvee_{i \in I} \bigwedge_{p \in P_i} a_p \wedge \bigwedge_{n \in N_i} \neg a_n$$

and

$$\mathsf{UGDNF}(\tau') = \bigvee_{i \in I} \bigwedge_{p \in P_i} a_p' \wedge \bigwedge_{n \in N_i} \neg a_n'$$

where, for every $i \in I$ and every $p \in P_i$, $a_p / a_p' = a_p'$. Therefore, for every $p \in P_i \cap \mathcal{A}^?_{\mathsf{fun}}$, we know that $a_p = \sigma_p \to \tau_p$ and $a_p' = \sigma_p' \to \tau_p'$, and we have by definition of the grounding operator $\tau_p / \tau_p' = \tau_p'$ and $\sigma_p / \sigma_p' = \sigma_p'$. The result then immediately follows from Definition B.68, and from the definition of the grounding operator. The same reasoning can be done for $\tau' / \tau = \tau$. □

Lemma B.70. *For every pair of types τ, τ', and every type σ, if $\tau / \tau' = \tau'$ and $\tau' \leq \mathbb{0} \to \mathbb{1}$, then $\langle \tau \xRightarrow{p} \tau' \rangle \circ \sigma$ is well-defined.*

Proof. Since $\tau' \leq \mathbb{0} \to \mathbb{1}$ and $\tau' \preccurlyeq \tau$, Corollary B.60 yields that $\tau \leq \mathbb{0} \to \mathbb{1}$. Proposition B.67 then immediately ensures that

$$\mathsf{UGDNF}(\tau) = \bigvee_{i \in I} \underbrace{\bigwedge_{p \in P_i} a_p \wedge \bigwedge_{n \in N_i} \neg a_n}_{\mathcal{I}_i}$$

and

$$\mathsf{UGDNF}(\tau') = \bigvee_{i \in I} \underbrace{\bigwedge_{p \in P_i} a_p' \wedge \bigwedge_{n \in N_i} \neg a_n'}_{\mathcal{I}_i'}$$

We now prove the third condition of Definition B.68, that is,

$$\forall i \in I, \mathcal{I}_i \not\leq \mathbb{0} \implies \mathcal{I}_i' \not\leq \mathbb{0}$$

Let $i \in I$. By hypothesis, $\mathcal{I}_i / \mathcal{I}_i' = \mathcal{I}_i'$, which implies that $\mathcal{I}_i' \preccurlyeq \mathcal{I}_i$. Applying Corollary B.62 then yields the result.

For the fourth condition of Definition B.68, knowing that $\tau / \tau' = \tau'$, we have for every $i \in I$ and every $p \in P_i$, $a_p / a_p' = a_p'$. The result then follows by definition of the grounding operator. □

Lemma B.71. *For every pair of types τ, τ', and every type σ, if $\langle \tau \xRightarrow{p} \tau' \rangle \circ \sigma = \langle \tau_1 \to \tau_2 \xRightarrow{p} \tau_1' \to \tau_2' \rangle$ then the following holds:*

(1) $\sigma \leq \tau_1'$
(2) $\tau_2' = \min\{\tau \mid \tau' \leq \sigma \to \tau\}$
(3) $\tau \leq \tau_1 \to \tau_2$

Proof. In all the following, we pose

$$\mathsf{UGDNF}(\tau) = \bigvee_{i\in I} \underbrace{\bigwedge_{p\in P_i} a_p \wedge \bigwedge_{n\in N_i} \neg a_n}_{\mathcal{I}_i}$$

and

$$\mathsf{UGDNF}(\tau') = \bigvee_{i\in I} \underbrace{\bigwedge_{p\in P_i} a'_p \wedge \bigwedge_{n\in N_i} \neg a'_n}_{\mathcal{I}'_i}$$

as well as $\bar{P}_i = P_i \cap \mathcal{A}^?_{\mathsf{fun}}$ and for every $p \in \bar{P}_i$, $a_p = \sigma_p \to \tau_p$ and $a'_p = \sigma'_p \to \tau'_p$.

Moreover, we know that, by hypothesis,

$$\forall i \in I, \mathcal{I}_i \not\leq \mathbb{0} \implies \mathcal{I}'_i \not\leq \mathbb{0}$$

(1) Immediate by definition of τ'_1, since it is an intersection of supertypes of σ.

(2) Let τ_0 a type such that $\tau' \leq \sigma \to \tau_0$. By Proposition B.28, we have $\tau'^{\oplus} \leq_T \sigma^{\ominus} \to \tau_0^{\oplus}$. By Lemma B.49, this implies that $\tau'^{\oplus} \circ \sigma^{\ominus} \leq_T \tau_0^{\oplus}$. Plugging in the definition of the result type, this gives:

$$\bigvee_{\substack{i\in I\\ \mathcal{I}'_i \not\leq_T \mathbb{0}}} \bigvee_{\substack{S\subsetneq \bar{P}_i\\ \sigma^{\ominus} \not\leq_T \bigvee_{p\in S} \sigma'^{\ominus}_p}} \bigwedge_{p\in \bar{P}_i\setminus S} \tau'^{\oplus}_p \leq_T \tau_0^{\oplus}$$

According to Proposition B.28, the condition $\sigma^{\ominus} \not\leq_T \bigvee_{p\in S} \sigma'^{\ominus}_p$ is equivalent to $\sigma \not\leq \bigvee_{p\in S} \sigma'_p$. Applying Proposition B.28 a second time to the whole inequality then yields

$$\bigvee_{\substack{i\in I\\ \mathcal{I}'_i \not\leq \mathbb{0}}} \bigvee_{\substack{S\subsetneq \bar{P}_i\\ \sigma \not\leq \bigvee_{p\in S} \sigma'_p}} \bigwedge_{p\in \bar{P}_i\setminus S} \tau'_p \leq \tau_0$$

that is, $\tau'_2 \leq \tau_0$, hence the result.

(3) • We first prove that $\tau \leq \tau_1 \to \mathbb{1}$. Let $i \in I$ and $S \subseteq \bar{P}_i$. It holds that $\bigvee_{p\in S} \sigma_p \leq \bigvee_{p\in \bar{P}_i} \sigma_p$ since the union in the left hand side contains fewer elements. This implies that

$$\bigwedge_{\substack{S\subseteq \bar{P}_i\\ \sigma \leq \bigvee_{p\in S} \sigma'_p}} \bigvee_{p\in S} \sigma_p \leq \bigvee_{p\in \bar{P}_i} \sigma_p$$

Thus taking the intersection for all $i \in I$ where $\mathcal{I}'_i \not\leq \mathbb{0}$,

$$\bigwedge_{\substack{i\in I\\ \mathcal{I}'_i \not\leq \mathbb{0}}} \bigwedge_{\substack{S\subseteq \bar{P}_i\\ \sigma \leq \bigvee_{p\in S} \sigma'_p}} \bigvee_{p\in S} \sigma_p \leq \bigwedge_{\substack{i\in I\\ \mathcal{I}'_i \not\leq \mathbb{0}}} \bigvee_{p\in \bar{P}_i} \sigma_p$$

which is

$$\tau_1 \leq \bigwedge_{\substack{i\in I\\ \mathcal{I}'_i \not\leq \mathbb{0}}} \bigvee_{p\in \bar{P}_i} \sigma_p$$

Moreover, since $\forall i \in I, \mathcal{I}_i \not\leq \mathbb{0} \implies \mathcal{I}'_i \not\leq \mathbb{0}$, we have

$$\tau_1 \leq \bigwedge_{\substack{i\in I\\ \mathcal{I}'_i \not\leq \mathbb{0}}} \bigvee_{p\in \bar{P}_i} \sigma_p \leq \bigwedge_{\substack{i\in I\\ \mathcal{I}_i \not\leq \mathbb{0}}} \bigvee_{p\in \bar{P}_i} \sigma_p$$

since the intersection on the left hand side contains more elements. Now applying Proposition B.28 and remarking that the right hand side of the previous inequality corresponds to the definition of the domain operator, we get $\tau_1^{\ominus} \leq_T \mathrm{dom}(\tau^{\oplus})$. Lemma B.47 then yields $\tau^{\oplus} \leq_T \tau_1^{\ominus} \to \mathbb{1}$, and the result follows from Proposition B.28.

- We then show that, for every $i \in I$, and every $S \subsetneq \bar{P}_i$,

$$\tau_1 \not\leq \bigvee_{p \in S} \sigma_p \implies \sigma \not\leq \bigvee_{p \in S} \sigma'_p \qquad (*)$$

Suppose that the left hand side holds. Plugging in the definition of τ_1, we have

$$\bigwedge_{\substack{i \in I \\ \mathcal{I}'_i \not\leq \mathbb{0}}} \bigwedge_{\substack{S \subsetneq \bar{P}_i \\ \sigma \leq \bigvee_{p \in S} \sigma'_p}} \bigvee_{p \in S} \sigma_p \not\leq \bigvee_{p \in S} \sigma_p$$

This inequality must also hold for every term of the intersections on the left hand side, and in particular for i and S:

$$\sigma \leq \bigvee_{p \in S} \sigma'_p \implies \bigvee_{p \in S} \sigma_p \not\leq \bigvee_{p \in S} \sigma_p$$

Since the right hand side is always false, the left hand side cannot hold thus $\sigma \not\leq \bigvee_{p \in S} \sigma'_p$.

- Now consider $i \in I$. It holds that

$$\bigvee_{\substack{S \subsetneq \bar{P}_i \\ \tau_1 \not\leq \bigvee_{p \in S} \sigma_p}} \bigwedge_{p \in \bar{P}_i \setminus S} \tau_p \leq \bigvee_{\substack{S \subsetneq \bar{P}_i \\ \sigma \not\leq \bigvee_{p \in S} \sigma'_p}} \bigwedge_{p \in \bar{P}_i \setminus S} \tau_p$$

since, according to $(*)$, the union on the right contains more elements. Now, taking the union on both sides for every $i \in I$ such that $\mathcal{I}'_i \not\leq \mathbb{0}$,

$$\bigvee_{\substack{i \in I \\ \mathcal{I}'_i \not\leq \mathbb{0}}} \bigvee_{\substack{S \subsetneq \bar{P}_i \\ \tau_1 \not\leq \bigvee_{p \in S} \sigma_p}} \bigwedge_{p \in \bar{P}_i \setminus S} \tau_p \leq \bigvee_{\substack{i \in I \\ \mathcal{I}'_i \not\leq \mathbb{0}}} \bigvee_{\substack{S \subsetneq \bar{P}_i \\ \sigma \not\leq \bigvee_{p \in S} \sigma'_p}} \bigwedge_{p \in \bar{P}_i \setminus S} \tau_p = \tau_2$$

Using the condition $\forall i \in I, \mathcal{I}_i \not\leq \mathbb{0} \implies \mathcal{I}'_i \not\leq \mathbb{0}$, the union on the left contains more elements than the same union on $\mathcal{I}_i \not\leq \mathbb{0}$, yielding

$$\bigvee_{\substack{i \in I \\ \mathcal{I}_i \not\leq \mathbb{0}}} \bigvee_{\substack{S \subsetneq \bar{P}_i \\ \tau_1 \not\leq \bigvee_{p \in S} \sigma_p}} \bigwedge_{p \in \bar{P}_i \setminus S} \tau_p \leq \tau_2$$

Using Proposition B.28 and remarking that the left hand side corresponds to the definition of the result operator, we deduce that $\tau^{\oplus} \circ \tau_1^{\ominus} \leq_T \tau_2^{\oplus}$, thus $\tau^{\oplus} \leq_T \tau_1^{\ominus} \to \tau^{\oplus} \circ \tau_1^{\ominus} \leq_T \tau_1^{\ominus} \to \tau_2^{\oplus}$, and applying Proposition B.28 yields the result.

□

Definition B.72 (Cast Projection). *For every pair of types τ, τ' such that $\tau' \leq \mathbb{1} \times \mathbb{1}$ if*

(1) $\mathsf{UGDNF}(\tau) = \bigvee_{i \in I} \underbrace{\bigwedge_{p \in P_i} a_p \wedge \bigwedge_{n \in N_i} \neg a_n}_{\mathcal{I}_i}$

(2) $\mathsf{UGDNF}(\tau') = \bigvee_{i \in I} \underbrace{\bigwedge_{p \in P_i} a'_p \wedge \bigwedge_{n \in N_i} \neg a'_n}_{\mathcal{I}'_i}$

(3) $\forall i \in I, \mathcal{I}_i \not\leq \mathbb{0} \implies \mathcal{I}'_i \not\leq \mathbb{0}$

(4) $\forall j \in I, \forall N \subseteq \bar{N}_j, \forall i \in \{1,2\}, \pi_i\,(\tau^j_N) \not\leq \mathbb{0} \implies \pi_i\,(\tau'^j_N) \not\leq \mathbb{0}$

(5) $\forall i \in I, \forall p \in P_i, a_p \in \mathcal{A}^?_{\mathsf{prod}} \iff a'_p \in \mathcal{A}^?_{\mathsf{prod}}$

(6) $\forall i \in I, \forall n \in N_i, a_n \in \mathcal{A}^?_{\mathsf{prod}} \iff a'_n \in \mathcal{A}^?_{\mathsf{prod}}$

then we define the i-th projection of $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle$, noted $\pi_i\,(\langle \tau \overset{p}{\Rightarrow} \tau' \rangle)$ as follows.

$$\pi_i\,(\langle \tau \overset{p}{\Rightarrow} \tau' \rangle) = \left\langle \bigvee_{\substack{j \in I \\ \mathcal{I}'_j \not\leq \mathbb{0}}} \bigvee_{\substack{N \subseteq \bar{N}_j \\ \pi_1(\tau^{ij}_N) \not\leq \mathbb{0} \\ \pi_2(\tau^{ij}_N) \not\leq \mathbb{0}}} \pi_i\,(\tau^j_N) \overset{p}{\Rightarrow} \bigvee_{\substack{j \in I \\ \mathcal{I}'_j \not\leq \mathbb{0}}} \bigvee_{\substack{N \subseteq \bar{N}_j \\ \pi_1(\tau'^{ij}_N) \not\leq \mathbb{0} \\ \pi_2(\tau'^{ij}_N) \not\leq \mathbb{0}}} \pi_i\,(\tau'^j_N) \right\rangle$$

where

$$\bar{P}_i = \{p \in P_i \mid a_p \in \mathcal{A}^?_{\mathsf{prod}}\} = \{p \in P_i \mid a'_p \in \mathcal{A}^?_{\mathsf{prod}}\}$$

$$\bar{N}_i = \{n \in N_i \mid a_n \in \mathcal{A}^?_{\mathsf{prod}}\} = \{n \in N_i \mid a'_n \in \mathcal{A}^?_{\mathsf{prod}}\}$$

$$\tau^i_N = \left(\bigwedge_{\substack{p \in \bar{P}_i \\ a_p = \tau_1 \times \tau_2}} \tau_1 \wedge \bigwedge_{\substack{n \in N \\ a_n = \tau_1 \times \tau_2}} \neg\tau_1,\ \bigwedge_{\substack{p \in \bar{P}_i \\ a_p = \tau_1 \times \tau_2}} \tau_2 \wedge \bigwedge_{\substack{n \in N_i \setminus N \\ a_n = \tau_1 \times \tau_2}} \neg\tau_2 \right)$$

$$\tau'^i_N = \left(\bigwedge_{\substack{p \in P_i \\ a'_p = \tau'_1 \times \tau'_2}} \tau'_1 \wedge \bigwedge_{\substack{n \in N \\ a'_n = \tau'_1 \times \tau'_2}} \neg\tau'_1,\ \bigwedge_{\substack{p \in P_i \\ a'_p = \tau'_1 \times \tau'_2}} \tau'_2 \wedge \bigwedge_{\substack{n \in N_i \setminus N \\ a'_n = \tau'_1 \times \tau'_2}} \neg\tau'_2 \right)$$

otherwise, $\pi_i\,(\langle \tau \overset{p}{\Rightarrow} \tau' \rangle)$ is undefined.

In the future, we use $P_i \cap \mathcal{A}^?_{\mathsf{prod}}$ as a shorthand for both $\{p \in P_i \mid a_p \in \mathcal{A}^?_{\mathsf{prod}}\}$ and $\{p \in P_i \mid a'_p \in \mathcal{A}^?_{\mathsf{prod}}\}$, provided the fourth condition of the above definition holds; and similarly for $N_i \cap \mathcal{A}^?_{\mathsf{prod}}$ provided the fifth condition above holds.

Lemma B.73. *For every pair of types τ, τ', if $\pi_i\,\langle \tau \overset{p}{\Rightarrow} \tau' \rangle = \langle \tau_i \overset{p}{\Rightarrow} \tau'_i \rangle$ then the following holds:*

$$\tau / \tau' = \tau' \implies \tau_i / \tau'_i = \tau'_i$$

$$\tau' / \tau = \tau \implies \tau'_i / \tau_i = \tau_i$$

Proof. Given two types τ, τ' such that $\tau / \tau' = \tau'$, Proposition B.67 ensures that

$$\mathsf{UGDNF}(\tau) = \bigvee_{i \in I} \bigwedge_{p \in P_i} a_p \wedge \bigwedge_{n \in N_i} \neg a_n$$

and

$$\mathsf{UGDNF}(\tau') = \bigvee_{i \in I} \bigwedge_{p \in P_i} a'_p \wedge \bigwedge_{n \in N_i} \neg a'_n$$

where, for every $i \in I$ and every $p \in P_i$, $a_p / a'_p = a'_p$, and for every $n \in N_i$, $a_n / a'_n = a'_n$.

Therefore, for every $p \in P_i \cap \mathcal{A}^?_{\text{prod}}$, we know that $a_p = \tau_1 \times \tau_2$ and $a'_p = \tau'_1 \to \tau'_2$, and we have by definition of the grounding operator $\tau_1 / \tau'_1 = \tau'_1$ and $\tau_2 / \tau'_2 = \tau'_2$. The result then immediately follows from Definition B.72, and from the definition of the grounding operator. The same reasoning can be done for $\tau' / \tau = \tau$. □

Lemma B.74. *For every pair of types τ, τ', if $\tau / \tau' = \tau'$ and $\tau' \leq \mathbb{1} \times \mathbb{1}$, then $\pi_i \langle \tau \overset{p}{\Rightarrow} \tau' \rangle$ is well-defined.*

Proof. Since $\tau' \leq \mathbb{1} \times \mathbb{1}$ and $\tau' \preccurlyeq \tau$, Corollary B.61 yields that $\tau \leq \mathbb{1} \times \mathbb{1}$. Proposition B.67 then immediately ensures that

$$\mathsf{UGDNF}(\tau) = \bigvee_{i \in I} \underbrace{\bigwedge_{p \in P_i} a_p \wedge \bigwedge_{n \in N_i} \neg a_n}_{\mathcal{I}_i}$$

and

$$\mathsf{UGDNF}(\tau') = \bigvee_{i \in I} \underbrace{\bigwedge_{p \in P_i} a'_p \wedge \bigwedge_{n \in N_i} \neg a'_n}_{\mathcal{I}'_i}$$

We now prove the third condition of Definition B.72, that is,

$$\forall i \in I, \mathcal{I}_i \not\leq \mathbb{0} \implies \mathcal{I}'_i \not\leq \mathbb{0}$$

Let $i \in I$. By hypothesis, $\mathcal{I}_i / \mathcal{I}'_i = \mathcal{I}'_i$, which implies that $\mathcal{I}'_i \preccurlyeq \mathcal{I}_i$. Applying Corollary B.62 then yields the result.

The fourth condition is proven similarly, by remarking that for every $j \in I$ and every $N \subseteq \bar{N}_j$, $\pi_i (\tau'^j_N) \preccurlyeq \pi_i (\tau^j_N)$.

For the fifth condition of Definition B.68, knowing that $\tau / \tau' = \tau'$, we have for every $i \in I$ and every $p \in P_i$, $a_p / a'_p = a'_p$. The result then follows by definition of the grounding operator.

The sixth condition can be proven using the same reasoning. □

Lemma B.75. *For every pair of types τ, τ' such that $\tau \leq \mathbb{1} \times \mathbb{1}$, if $\pi_1 \langle \tau \overset{p}{\Rightarrow} \tau' \rangle = \langle \tau_1 \overset{p}{\Rightarrow} \tau'_1 \rangle$ then the following holds:*

(1) $\tau \leq (\tau_1 \times \mathbb{1})$
(2) $\tau'_1 = \min\{\tau \mid \tau' \leq \tau \times \mathbb{1}\}$

Proof. In all the following, we pose

$$\mathsf{UGDNF}(\tau) = \bigvee_{i \in I} \underbrace{\bigwedge_{p \in P_i} a_p \wedge \bigwedge_{n \in N_i} \neg a_n}_{\mathcal{I}_i}$$

and

$$\mathsf{UGDNF}(\tau') = \bigvee_{i \in I} \underbrace{\bigwedge_{p \in P_i} a'_p \wedge \bigwedge_{n \in N_i} \neg a'_n}_{\mathcal{I}'_i}$$

We also pose $\bar{P}_i = P_i \cap \mathcal{A}^?_{\text{prod}}$ and $\bar{N}_i = N_i \cap \mathcal{A}^?_{\text{prod}}$.

Finally, as in Definition B.72, we pose, for every $i \in I$ and every set $N \subseteq N_i$:

$$\tau_N^i = \Big(\bigwedge_{\substack{p \in \bar{P}_i \\ a_p = \tau_1 \times \tau_2}} \tau_1 \wedge \bigwedge_{\substack{n \in N \\ a_n = \tau_1 \times \tau_2}} \neg\tau_1, \bigwedge_{\substack{p \in \bar{P}_i \\ a_p = \tau_1 \times \tau_2}} \tau_2 \wedge \bigwedge_{\substack{n \in N_i \setminus N \\ a_n = \tau_1 \times \tau_2}} \neg\tau_2 \Big)$$

$$\tau_N'^i = \Big(\bigwedge_{\substack{p \in P_i \\ a'_p = \tau'_1 \times \tau'_2}} \tau'_1 \wedge \bigwedge_{\substack{n \in N \\ a'_n = \tau'_1 \times \tau'_2}} \neg\tau'_1, \bigwedge_{\substack{p \in P_i \\ a'_p = \tau'_1 \times \tau'_2}} \tau'_2 \wedge \bigwedge_{\substack{n \in N_i \setminus N \\ a'_n = \tau'_1 \times \tau'_2}} \neg\tau'_2 \Big)$$

(1) Since $\tau \leq \mathbb{1}\times\mathbb{1}$, Proposition B.28 yields $\tau^\oplus \leq_T \mathbb{1}\times\mathbb{1}$. Thus, Lemma B.42 gives $\tau^\oplus \leq_T \pi_1(\tau^\oplus)\times\mathbb{1}$. Plugging in the definition of π_1 on type frames, we obtain:

$$\tau^\oplus \leq_T \Big(\bigvee_{\substack{i \in I \\ I_i \not\leq \mathbb{0}}} \bigvee_{\substack{N \subseteq \bar{N}_i \\ \pi_1(\tau_N^{\oplus i}) \not\leq_T \mathbb{0} \\ \pi_2(\tau_N^{\oplus i}) \not\leq_T \mathbb{0}}} \pi_1(\tau_N^{\oplus i}) \Big) \times \mathbb{1}$$

Now, remarking that $\pi_1(\tau_N^{\oplus i}) = (\pi_1(\tau_N^i))^\oplus$ and applying Proposition B.28, we obtain that $\pi_1(\tau_N^{\oplus i}) \not\leq_T \mathbb{0} \iff \pi_1(\tau_N^i) \not\leq \mathbb{0}$. Condition (4) of Definition B.72 then yields $\pi_1(\tau_N^{\oplus i}) \not\leq_T \mathbb{0} \implies \pi_1(\tau_N'^i) \not\leq \mathbb{0}$. The same reasoning for the second projection yields $\pi_2(\tau_N^{\oplus i}) \not\leq_T \mathbb{0} \implies \pi_2(\tau_N'^i) \not\leq \mathbb{0}$. Using this and Condition (3) of Definition B.72, we deduce

$$\Big(\bigvee_{\substack{i \in I \\ I_i \not\leq \mathbb{0}}} \bigvee_{\substack{N \subseteq \bar{N}_i \\ \pi_1(\tau_N^{\oplus i}) \not\leq_T \mathbb{0} \\ \pi_2(\tau_N^{\oplus i}) \not\leq_T \mathbb{0}}} \pi_1(\tau_N^{\oplus i}) \Big) \leq_T \Big(\bigvee_{\substack{i \in I \\ I'_i \not\leq \mathbb{0}}} \bigvee_{\substack{N \subseteq \bar{N}_i \\ \pi_1(\tau_N'^i) \not\leq \mathbb{0} \\ \pi_2(\tau_N'^i) \not\leq \mathbb{0}}} (\pi_1(\tau_N^i))^\oplus \Big)$$

Since the unions on the right contain more elements than the unions on the left. Finally, we have

$$\tau^\oplus \leq_T \Big(\bigvee_{\substack{i \in I \\ I'_i \not\leq \mathbb{0}}} \bigvee_{\substack{N \subseteq \bar{N}_i \\ \pi_1(\tau_N'^i) \not\leq \mathbb{0} \\ \pi_2(\tau_N'^i) \not\leq \mathbb{0}}} (\pi_1(\tau_N^i))^\oplus \Big) \times \mathbb{1}$$

And applying Proposition B.28 yields

$$\tau \leq \Big(\bigvee_{\substack{i \in I \\ I'_i \not\leq \mathbb{0}}} \bigvee_{\substack{N \subseteq \bar{N}_i \\ \pi_1(\tau_N'^i) \not\leq \mathbb{0} \\ \pi_2(\tau_N'^i) \not\leq \mathbb{0}}} \pi_1(\tau_N^i) \Big) \times \mathbb{1}$$

which is the result.

(2) Let τ_0 such that $\tau' \leq \tau_0 \times \mathbb{1}$. We show that $\tau'_1 \leq \tau_0$.
By Proposition B.28, we have $\tau'^\oplus \leq_T \tau_0^\oplus \times \mathbb{1}$. Thus, by Lemma B.42, we have $\pi_1((\tau')^\oplus) \leq_T \tau_0^\oplus$. Plugging in the definition of the projection of a type frame yields:

$$\bigvee_{\substack{i \in I \\ I'_i \not\leq \mathbb{0}}} \bigvee_{\substack{N \subseteq \bar{N}_i \\ \pi_1(\tau_N'^{\oplus i}) \not\leq_T \mathbb{0} \\ \pi_2(\tau_N'^{\oplus i}) \not\leq_T \mathbb{0}}} \pi_1(\tau_N'^{\oplus i}) \leq_T \tau_0^\oplus$$

Remarking that, for every $i \in \{1,2\}$ and every $j \in I$, $\pi_i(\tau_N'^{\oplus j}) = (\pi_i(\tau_N'^j))^\oplus$, and applying Proposition B.28, we obtain

$$\pi_i(\tau_N'^{\oplus j}) \not\leq_T \mathbb{0} \iff (\pi_i(\tau_N'^j))^\oplus \not\leq_T \mathbb{0} \iff \pi_i(\tau_N'^j) \not\leq \mathbb{0}$$

$$[\textsc{Var}]\ \frac{}{\Gamma \vdash x\colon \forall\vec{\alpha}.\,\tau}\ \Gamma(x) = \forall\vec{\alpha}.\,\tau \qquad [\textsc{Const}]\ \frac{}{\Gamma \vdash c\colon b_c}$$

$$[\textsc{Abstr}]\ \frac{\Gamma, x\colon \tau' \vdash E\colon \tau}{\Gamma \vdash (\lambda^{\tau'\to\tau} x.\,E)\colon \tau' \to \tau} \qquad [\textsc{App}]\ \frac{\Gamma \vdash E_1\colon \tau' \to \tau \qquad \Gamma \vdash E_2\colon \tau'}{\Gamma \vdash E_1\,E_2\colon \tau}$$

$$[\textsc{Pair}]\ \frac{\Gamma \vdash E_1\colon \tau_1 \qquad \Gamma \vdash E_2\colon \tau_2}{\Gamma \vdash (E_1, E_2)\colon \tau_1 \times \tau_2} \qquad [\textsc{Proj}]\ \frac{\Gamma \vdash E\colon \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\,E\colon \tau_i}$$

$$[\textsc{Let}]\ \frac{\Gamma \vdash E_1\colon \forall\vec{\alpha}.\,\tau_1 \qquad \Gamma, x\colon \forall\vec{\alpha}.\,\tau_1 \vdash E_2\colon \tau}{\Gamma \vdash (\text{let } x = E_1 \text{ in } E_2)\colon \tau}$$

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

$$[\textsc{Cast}]\ \frac{\Gamma \vdash E\colon \tau'}{\Gamma \vdash E\langle \tau' \overset{p}{\Rightarrow} \tau\rangle\colon \tau}\ \begin{cases} p = l \implies \tau' \preccurlyeq \tau \\ p = \bar{l} \implies \tau \preccurlyeq \tau' \end{cases} \qquad [\textsc{Subsume}]\ \frac{\Gamma \vdash e\colon \tau'}{\Gamma \vdash e\colon \tau}\ \tau' \le \tau$$

Fig. 13. Full Typing Rules for the Set-Theoretic Cast Calculus

Thus we have:

$$\bigvee_{\substack{i\in I\\ \mathcal{I}'_i \not\le \mathbb{0}}} \bigvee_{\substack{N\subseteq \bar{N}_i\\ \pi_1(\tau'^i_N)\not\le\mathbb{0}\\ \pi_2(\tau'^i_N)\not\le\mathbb{0}}} (\pi_1(\tau'^i_N))^{\oplus} \le_T \tau_0^{\oplus}$$

Then, applying Proposition B.28 yields

$$\bigvee_{\substack{i\in I\\ \mathcal{I}'_i \not\le \mathbb{0}}} \bigvee_{\substack{N\subseteq \bar{N}_i\\ \pi_1(\tau'^i_N)\not\le\mathbb{0}\\ \pi_2(\tau'^i_N)\not\le\mathbb{0}}} \pi_1(\tau'^i_N) \le \tau_0$$

Remarking that the left hand side corresponds to the definition of τ'_1 yields the result: $\tau'_1 \le \tau_0$. □

Lemma B.76. *For every pair of types τ, τ', if $\tau \le \mathbb{1}\times\mathbb{1}$ and $\pi_2\,\langle \tau \overset{p}{\Rightarrow} \tau'\rangle = \langle \tau_2 \overset{p}{\Rightarrow} \tau'_2\rangle$ then the following holds:*

(1) $\tau \le (\mathbb{1}\times\tau_2)$
(2) $\tau'_2 = \min\{\tau \mid \tau' \le \mathbb{1}\times\tau\}$

Proof. Same proof as Lemma B.75. □

B.8 Cast Calculus with Set-Theoretic Types

The full typing rules for the cast calculus with set-theoretic types are defined in Figure 13.

The values of the cast language are defined by the following grammar.

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

Definition B.77 (Value Type Operator). *We define the operator* type *on values of the cast language (except type abstractions) as follows:*

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

Lemma B.78. *For every value V that is not a type abstraction, $\varnothing \vdash V : \mathrm{type}(V)$.*

Proof. By cases on V.

- $V = c$. $\mathrm{type}(V) = b_c$. By typing rule [Const], $\varnothing \vdash V : b_c$.
- $V = \lambda^{\tau_1\to\tau_2} x.\, E$. $\mathrm{type}(V) = \tau_1 \to \tau_2$. By typing rule [Abstr], $\varnothing \vdash V : \tau_1 \to \tau_2$.
- $V = (V_1, V_2)$. $\mathrm{type}(V) = \mathrm{type}(V_1) \times \mathrm{type}(V_2)$. By induction, for every $i \in \{1, 2\}$, $\varnothing \vdash V_i : \mathrm{type}(V_i)$. Then by typing rule [Pair], $\varnothing \vdash V : \mathrm{type}(V_1) \times \mathrm{type}(V_2)$.
- $V = V'\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$. $\mathrm{type}(V) = \tau_2$. By typing rule [Cast], $\varnothing \vdash V : \tau_2$.

□

Lemma B.79 (Progress for Cast Values). *For every value V, every label p, and all types τ_1, τ_2, if $\varnothing \vdash V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle : \tau_2$, then one of the following cases holds:*

- *$V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$ is a value*
- *there exists a term E such that $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow E$*
- *$V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow \mathtt{blame}\ p$*

Proof. By hypothesis, $\varnothing \vdash V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle : \tau_2$. Therefore, by inversion of the typing rules, it holds that $\varnothing \vdash V : \tau_1$ and we distinguish two main cases: $\tau_1 \preccurlyeq \tau_2$ or $\tau_2 \preccurlyeq \tau_1$. The proof is then done by subcases over τ_2/τ_1 or τ_1/τ_2. The case where $\tau_1 = \tau_2$ is a particular case that is handled separately.

- $\tau_1 = \tau_2$. In this case, $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow V$ by rule [CastId].
- $\tau_1 \preccurlyeq \tau_2$ and $\tau_1 \neq \tau_2$. We distinguish the following subcases:
  - $\tau_2/\tau_1 = \tau_1$. Then $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$ is a value.
  - $\tau_2/\tau_1 = \tau_2$. We proceed by case disjunction over V:
    - $V = V'\langle \tau_1' \overset{q}{\Rightarrow} \tau_2'\rangle$ where $\tau_1'/\tau_2' = \tau_1'$. If $\tau_1' \leq \tau_2$ then $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow V'$ by rule [Collapse]. Otherwise, if $\tau_1' \not\leq \tau_2$ then $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow \mathtt{blame}\ p$ by rule [Blame].
    - $V = V'\langle \tau_1' \overset{q}{\Rightarrow} \tau_2'\rangle$ where $\tau_2'/\tau_1' = \tau_1'$. This case is identical to the previous one. If $\tau_1' \leq \tau_2$ then $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow V'$ by rule [Collapse]. Otherwise, if $\tau_1' \not\leq \tau_2$ then $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow \mathtt{blame}\ p$ by rule [Blame].
    - $V = V'\langle \tau_1' \overset{q}{\Rightarrow} \tau_2'\rangle$ where $\tau_1'/\tau_2' = \tau_2'$. If $\tau_2' \leq \tau_2$ then $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow V$ by rule [UpSimpl]. Otherwise, if $\tau_2' \not\leq \tau_2$ then $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow \mathtt{blame}\ p$ by rule [UpBlame].
    - V is unboxed. If $\mathrm{type}(V) \leq \tau_2$ then $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow V$ by rule [UnboxSimpl]. Otherwise, $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow \mathtt{blame}\ p$ by rule [UnboxBlame].
  - $\forall i \in \{1, 2\}, \tau_2/\tau_1 \neq \tau_i$. In this case, $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2/\tau_1\rangle\langle \tau_2/\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$ by rule [ExpandR].
- $\tau_2 \preccurlyeq \tau_1$ and $\tau_1 \neq \tau_2$. We distinguish the following subcases:
  - $\tau_1/\tau_2 = \tau_1$. In this case, $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$ is a value.
  - $\tau_1/\tau_2 = \tau_2$. In this case, $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$ is a value.
  - $\forall i, \tau_1/\tau_2 \neq \tau_i$. In this case, $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow V\langle \tau_1 \overset{p}{\Rightarrow} \tau_1/\tau_2\rangle\langle \tau_1/\tau_2 \overset{p}{\Rightarrow} \tau_2\rangle$ by rule [ExpandL].

□

Lemma B.80 (Progress). *For every term E such that $\varnothing \vdash E : S$, one of the following cases holds:*

Cast Reductions.

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

$(*)$ to ease the notation and to avoid redundant conditions, the rule [CastId] takes precedence over the following ones. All other casts are therefore considered to be non-identity casts.

Standard Reductions.

$$
\begin{array}{rrcll}
\text{[CastApp]} & V\langle \tau \overset{p}{\Rightarrow} \tau'\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{if } \tau'/\tau = \tau \text{ or } \tau/\tau' = \tau' \\
 & & & \text{where } \langle \tau \overset{p}{\Rightarrow} \tau'\rangle \circ \mathrm{type}(V') = \langle \tau_1 \to \tau_2 \overset{p}{\Rightarrow} \tau_1' \to \tau_2'\rangle & \\
\text{[CastProj]} & \pi_i\ (V\langle \tau \overset{p}{\Rightarrow} \tau'\rangle) & \hookrightarrow & (\pi_i\ V)\langle \tau_i \overset{p}{\Rightarrow} \tau_i'\rangle & \text{if } \tau'/\tau = \tau \text{ or } \tau/\tau' = \tau' \\
 & & & \text{where } \langle \tau_i \overset{p}{\Rightarrow} \tau_i'\rangle = \pi_i\ (\langle \tau \overset{p}{\Rightarrow} \tau'\rangle) & \\
\text{[FailApp]} & V\langle \tau \overset{p}{\Rightarrow} \tau'\rangle V' & \hookrightarrow & \texttt{blame p} & \text{if } \langle \tau \overset{p}{\Rightarrow} \tau'\rangle \circ \mathrm{type}(V') \text{ undef.} \\
\text{[FailProj]} & \pi_i\ (V\langle \tau \overset{p}{\Rightarrow} \tau'\rangle) & \hookrightarrow & \texttt{blame p} & \text{if } \pi_i\ (\langle \tau \overset{p}{\Rightarrow} \tau'\rangle) \text{ undef.} \\
\text{[SimplApp]} & V\langle \tau \overset{p}{\Rightarrow} \tau'\rangle V' & \hookrightarrow & V\ V' & \text{if } \tau/\tau' = \tau \\
\text{[SimplProj]} & \pi_i\ (V\langle \tau \overset{p}{\Rightarrow} \tau'\rangle) & \hookrightarrow & \pi_i\ V & \text{if } \tau/\tau' = \tau \\
\text{[App]} & (\lambda^{\tau_1 \to \tau_2} x.\, E) V & \hookrightarrow & E\{x := V\} & \\
\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]} & \mathsf{let}\ x = V\ \mathsf{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. 14. Full Reductions for the Cast Calculus

- *there exists a value V such that $E = V$*
- *there exists a term E' such that $E \hookrightarrow E'$*
- *there exists a label p such that $E \hookrightarrow \texttt{blame}\ p$*

Proof. By complete induction over the expression E.

- Case x. Impossible by hypothesis since a single variable cannot be well-typed in the empty environment.
- Case c. Immediate since c is a value.
- Case $\lambda^{\tau_1 \to \tau_2} x.\, E$. Immediate since $\lambda^{\tau_1 \to \tau_2} x.\, E$ is a value.
- Case $E_1\ E_2$. By inversion of the typing rules, we deduce that $\varnothing \vdash E_1 : \tau_1 \to \tau_2$ and $\varnothing \vdash E_2 : \tau_1$. We can thus apply the induction hypothesis on both E_1 and E_2, which yields the following subcases.

  - $\exists E_2'$ such that $E_2 \hookrightarrow E_2'$. By rule [Context] and since $E_1\ \square$ is a valid reduction context, $E_1\ E_2 \hookrightarrow E_1\ E_2'$.
  - $E_2 \hookrightarrow \texttt{blame}\ p$. By rule [CtxBlame] and since $E_1\ \square$ is a valid reduction context, $E_1\ E_2 \hookrightarrow \texttt{blame}\ p$.
  - E_2 is a value and $\exists E_1'$ such that $E_1 \hookrightarrow E_1'$. Since E_2 is a value, $\square\ E_2$ is a valid reduction context, thus $E_1\ E_2 \hookrightarrow E_1'\ E_2$ by rule [Context].
  - E_2 is a value and $E_1 \hookrightarrow \texttt{blame}\ p$. Since E_2 is a value, $\square\ E_2$ is a valid reduction context, thus $E_1\ E_2 \hookrightarrow \texttt{blame}\ p$ by rule [CtxBlame].
  - Both E_1 and E_2 are values. Reasoning by case analysis on E_1 and not considering ill-typed cases:
    * $E_1 = \lambda^{\tau_1' \to \tau_2'} x.\ E_1'$ where $\tau_1' \to \tau_2' \leq \tau_1 \to \tau_2$. In this case, $E_1\ E_2$ reduces to $E_1'\{x \coloneqq E_2\}$ by rule [App].
    * $E_1 = V\langle \tau_1' \overset{p}{\Rightarrow} \tau_2' \rangle$ where $\tau_2' \leq \tau_1 \to \tau_2$ and $\tau_2' / \tau_1' = \tau_1'$ or $\tau_1' / \tau_2' = \tau_2'$. If $\tau_1' \leq \mathbb{0} \to \mathbb{1}$ then $E_1 E_2$ reduces to $(V\, E_2 \langle \tau_l' \overset{\bar{p}}{\Rightarrow} \tau_l \rangle) \langle \tau_r \overset{p}{\Rightarrow} \tau_r' \rangle$ where $\langle \tau_1' \overset{p}{\Rightarrow} \tau_2' \rangle \circ \mathrm{type}(E_2) = \langle \tau_l \to \tau_r \overset{p}{\Rightarrow} \tau_l' \to \tau_r' \rangle$ by rule [CastApp].
      Otherwise, if $\tau_1' \not\leq \mathbb{0} \to \mathbb{1}$, then $E_1\ E_2 \hookrightarrow \texttt{blame}\ p$ by rule [FailApp].
    * $E_1 = V\langle \tau_1' \overset{p}{\Rightarrow} \tau_2' \rangle$ where $\tau_2' \leq \tau_1 \to \tau_2$ and $\tau_1' / \tau_2' = \tau_1'$. Then $E_1\ E_2 \hookrightarrow V\ E_2$ by rule [SimplApp].
- Case $\Lambda\vec{\alpha}.\ E$. Immediate since $\Lambda\vec{\alpha}.\ E$ is a value.
- Case $E\ [\vec{t}]$. By inversion of the typing rule [TApp], we deduce that $\varnothing \vdash E : \forall\vec{\alpha}.\tau$. We can thus apply the induction hypothesis on E which yields the following subcases:
  - $E \hookrightarrow E'$. Since $\square\ [\vec{t}]$ is a valid reduction context, $E\ [\vec{t}]$ reduces to $E'\ [\vec{t}]$ by [Context].
  - $E \hookrightarrow \texttt{blame}\ p$. Since $\square\ [\vec{t}]$ is a valid reduction context, $E\ [\vec{t}]$ also reduces to $\texttt{blame}\ p$ by [CtxBlame].
  - E is a value. In this case, by inversion of the typing rules, E is necessarily of the form $\Lambda\vec{\alpha}.\ E'$. Therefore, $E\ [\vec{t}] \hookrightarrow E'\{\vec{\alpha} \coloneqq \vec{t}\}$ by [TypeApp], concluding this case.
- Case (E_1, E_2). By inversion of the typing rule [Pair], we deduce that $\varnothing \vdash E_i : \tau_i$, for $i \in \{1, 2\}$. Thus, we can apply the induction hypothesis on both E_1 and E_2, yielding the following subcases:
  - $E_2 \hookrightarrow E_2'$. Since $(E_1, \square)$ is a valid reduction context, $(E_1, E_2) \hookrightarrow (E_1, E_2')$ by rule [Context].
  - $E_2 \hookrightarrow \texttt{blame}\ p$. Since $(E_1, \square)$ is a valid reduction context, $(E_1, E_2) \hookrightarrow \texttt{blame}\ p$ by rule [CtxBlame].
  - E_2 is a value and $E_1 \hookrightarrow E_1'$. Since E_2 is a value, $(\square, E_2)$ is a valid reduction context, thus $(E_1, E_2) \hookrightarrow (E_1', E_2)$ by rule [Context].
  - E_2 is a value and $E_1 \hookrightarrow \texttt{blame}\ p$. Since E_2 is a value, $(\square, E_2)$ is a valid reduction context, thus $(E_1, E_2) \hookrightarrow \texttt{blame}\ p$ by rule [CtxBlame].
  - Both E_1 and E_2 are values. In this case, (E_1, E_2) is itself a value, concluding this case.
- Case $\pi_i\ E$. By inversion of the typing rule [Proj], we deduce that $\varnothing \vdash E : \tau_1 \times \tau_2$. Thus, we can apply the induction hypothesis to E, yielding the following subcases:
  - $E \hookrightarrow E'$. Since $\pi_i\ \square$ is a valid reduction context, $\pi_i\ E \hookrightarrow \pi_i\ E'$ by rule [Context].
  - $E \hookrightarrow \texttt{blame}\ p$. Since $\pi_i\ \square$ is a valid reduction context, $\pi_i\ E$ reduces to $\texttt{blame}\ p$ by rule [CtxBlame].
  - E is a value. By cases on E, not considering the ill-typed cases:
    * $E = (V_1, V_2)$. In this case, $\pi_i\ E$ reduces to V_i by rule [Proj].

  - $E = V\langle \tau_1' \overset{p}{\Rightarrow} \tau_2' \rangle$ where $\tau_2' \leq \tau_1 \times \tau_2$ and $\tau_2' / \tau_1' = \tau_1'$ or $\tau_1' / \tau_2' = \tau_2'$. In this case, if $\tau_1' \leq \mathbb{1} \times \mathbb{1}$ then $\pi_i E$ reduces to $(\pi_i V)\langle \tau_p \overset{p}{\Rightarrow} \tau_p' \rangle$ where $\langle \tau_p \overset{p}{\Rightarrow} \tau_p' \rangle = \pi_i(\langle \tau_1' \overset{p}{\Rightarrow} \tau_2' \rangle)$ by rule [CastProj]. Otherwise, if $\tau_1' \not\leq \mathbb{1} \times \mathbb{1}$, then $\pi_i\ E \hookrightarrow \texttt{blame}\ p$ by rule [FailProj].
  - $E = V\langle \tau_1' \overset{p}{\Rightarrow} \tau_2' \rangle$ where $\tau_2' \leq \tau_1 \times \tau_2$ and $\tau_1' / \tau_2' = \tau_1'$. Then $E \hookrightarrow \pi_i\ V$ by rule [SimplProj].
- Case let $x = E_1$ in E_2. By inversion of the typing rule [Let], we deduce that $\varnothing \vdash E_1 : \tau_1$. Therefore, we can apply the induction hypothesis to E_1, yielding the following subcases:
  - $E_1 \hookrightarrow E_1'$. Since let $x = \square$ in E_2 is a valid reduction context, let $x = E_1$ in $E_2 \hookrightarrow$ let $x = E_1'$ in E_2 by rule [Context].
  - $E_1 \hookrightarrow \texttt{blame}\ p$. Since let $x = \square$ in E_2 is a valid reduction context, let $x = E_1$ in $E_2 \hookrightarrow \texttt{blame}\ p$ by rule [CtxBlame].
  - E_1 is a value. We immediately deduce that let $x = E_1$ in $E_2 \hookrightarrow E_2\{x \coloneqq E_1\}$ by rule [Let].
- Case $E\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle$. By inversion of the typing rule [Cast], we deduce that $\varnothing \vdash E : \tau_1$. Therefore, we can apply the induction hypothesis to E, yielding the following subcases:
  - $E \hookrightarrow E'$. Since $\square\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle$ is a valid reduction context, $E\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \hookrightarrow E'\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle$ by rule [Context].
  - $E \hookrightarrow \texttt{blame}\ p$. Since $\square\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle$ is a valid reduction context, $E\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \hookrightarrow \texttt{blame}\ p$ by rule [CtxBlame].
  - E is a value. In this case, we can apply Lemma B.79 to $E\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle$ which yields the result and concludes the proof.

□

Lemma B.81. *If $\Gamma, x : S' \vdash E : S$, then for every expression E' such that $\Gamma \vdash E' : S'$, we have $\Gamma \vdash E\{x \coloneqq E'\} : S$.*

Proof. By induction on E.

- x. We have $S = S'$ and the result follows from $\Gamma \vdash E' : S'$ since $E\{x \coloneqq E'\} = E'$.
- y. Immediate since $E\{x \coloneqq E'\} = E$.
- c. Immediate since $E\{x \coloneqq E'\} = E$.
- $\lambda^{\tau_1 \to \tau_2} y.\, E_y$. By inversion of the typing rules, we have $\tau_1 \to \tau_2 \leq S$, and $\Gamma, x : S', y : \tau_1 \vdash E_y : \tau_2$. Thus, by induction hypothesis, $\Gamma, y : \tau_1 \vdash E_y\{x \coloneqq E'\} : \tau_2$. This implies that $\Gamma \vdash \lambda^{\tau_1 \to \tau_2} y.\, (E_y\{x \coloneqq E'\}) : \tau_1 \to \tau_2$ by rule [Abstr], and the result follows since $E\{x \coloneqq E'\} = \lambda^{\tau_1 \to \tau_2} y.\, (E_y\{x \coloneqq E'\})$.
- $E_1\ E_2$. By hypothesis, we have $\Gamma, x : S' \vdash E_1 : \tau_1 \to S$ and $\Gamma, x : S' \vdash E_2 : \tau_1$. By induction hypothesis, we deduce that $\Gamma \vdash E_1\{x \coloneqq E'\} : \tau_1 \to S$ and $\Gamma \vdash E_2\{x \coloneqq E'\} : \tau_1$. Therefore, $\Gamma \vdash (E_1\{x \coloneqq E'\})(E_2\{x \coloneqq E'\}) : S$ by rule [App], hence the result.
- (E_1, E_2). By hypothesis, we have $\Gamma, x : S' \vdash E_1 : \tau_1$ and $\Gamma, x : S' \vdash E_2 : \tau_2$, where $\tau_1 \times \tau_2 \leq S$. By induction hypothesis, we deduce that $\Gamma \vdash E_1\{x \coloneqq E'\} : \tau_1$ and $\Gamma \vdash E_2\{x \coloneqq E'\} : \tau_2$. Therefore, $\Gamma \vdash (E_1\{x \coloneqq E'\}, E_2\{x \coloneqq E'\}) : \tau_1 \times \tau_2$ by rule [Pair], and the result follows.
- $\pi_i\ E_p$. By hypothesis, we have $\Gamma, x : S' \vdash E_p : (\tau_1 \times \tau_2)$, where $\tau_i \leq S$. By induction hypothesis, we deduce that $\Gamma \vdash E_p\{x \coloneqq E'\} : (\tau_1 \times \tau_2)$. Therefore, $\Gamma \vdash \pi_i\ (E_p\{x \coloneqq E'\}) : \tau_i$ by rule [Proj], and the result follows.
- let $y = E_1$ in E_2. By hypothesis, we have $\Gamma, x : S' \vdash E_1 : \forall \vec{\alpha}.\tau_1$ and $\Gamma, x : S', y : \forall \vec{\alpha}.\tau_1 \vdash E_2 : S$. Therefore, by induction hypothesis, we deduce $\Gamma \vdash E_1\{x \coloneqq E'\} : \forall \vec{\alpha}.\tau_1$ and $\Gamma, y : \forall \vec{\alpha}.\tau_1 \vdash E_2\{x \coloneqq E'\} : S$. This yields $\Gamma \vdash$ let $y = E_1\{x \coloneqq E'\}$ in $E_2\{x \coloneqq E'\} : S$ by rule [Let], hence the result.
- $\Lambda \vec{\alpha}.\, E$. By hypothesis, $\Gamma, x : S' \vdash E : \tau$ where $\forall \vec{\alpha}.\tau \leq S$. By induction hypothesis, we deduce $\Gamma \vdash E\{x \coloneqq E'\} : \tau$. Hence, rule [TAbstr] yields $\Gamma \vdash \Lambda \vec{\alpha}.\, E\{x \coloneqq E'\} : \forall \vec{\alpha}.\tau$, hence the result.

- $E\,[\vec{t}]$. By hypothesis, $\Gamma, x : S' \vdash E : \forall\vec{\alpha}.\tau$, and $\tau\{\vec{\alpha} := \vec{t}\} \leq S$. The induction hypothesis then yields $\Gamma \vdash E\{x := E'\} : \forall\vec{\alpha}.\tau$. Applying rule [TApp] yields $\Gamma \vdash (E\{x := E'\})\,[\vec{t}] : \tau\{\vec{\alpha} := \vec{t}\}$, hence the result.
- $E\langle\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$. By hypothesis, $\Gamma, x : S' \vdash E : \tau_1$, and $\tau_2 \leq S$. The induction hypothesis then yields $\Gamma \vdash E\{x := E'\} : \tau_1$. Applying rule [Cast] gives $\Gamma \vdash (E\{x := E'\})\langle\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle : \tau_2$ and the result follows.

□

Lemma B.82. *If $\Gamma \vdash E : S$ and $\Gamma \vdash \mathcal{E}[E] : S'$ then for every expression E' such that $\Gamma \vdash E' : S$, we have $\Gamma \vdash \mathcal{E}[E'] : S'$.*

Proof. By complete induction over $\mathcal{E}$.

- $\square$. Immediate with $S = S'$.
- $E_f\ \mathcal{E}$. By hypothesis and inversion of rule [App], we have $\Gamma \vdash E_f : \tau \to S'$ and $\Gamma \vdash \mathcal{E}[E] : \tau$. By induction hypothesis, it holds that $\Gamma \vdash \mathcal{E}[E'] : \tau$. Therefore, by [App], $\Gamma \vdash E_f\ \mathcal{E}[E'] : S'$.
- $\mathcal{E}\ V$. By hypothesis and inversion of rule [App], we have $\Gamma \vdash \mathcal{E}[E] : \tau \to S'$ and $\Gamma \vdash V : \tau$. By induction hypothesis, it holds that $\Gamma \vdash \mathcal{E}[E'] : \tau \to S'$. Therefore, by [App], $\Gamma \vdash \mathcal{E}[E']\,V : S'$.
- $\mathcal{E}\,[\vec{t}]$. By hypothesis and inversion of [TApp], we have $\Gamma \vdash \mathcal{E}[E] : \forall\vec{\alpha}.\tau$ where $\tau\{\vec{\alpha} := \vec{t}\} \leq S'$. By IH, it holds that $\Gamma \vdash \mathcal{E}[E'] : \forall\vec{\alpha}.\tau$. Therefore, by rule [TApp], we have $\Gamma \vdash \mathcal{E}[E']\,[\vec{t}] : \tau\{\vec{\alpha} := \vec{t}\}$ and the result follows by [Subsume].
- $(E_l, \mathcal{E})$. By hypothesis and inversion of [Pair], we have $\Gamma \vdash E_l : \tau_1$ and $\Gamma \vdash \mathcal{E}[E] : \tau_2$ where $\tau_1 \times \tau_2 \leq S'$. By IH, we deduce $\Gamma \vdash \mathcal{E}[E'] : \tau_2$. Therefore, it holds by rule [Pair] that $\Gamma \vdash (E_l, \mathcal{E}[E']) : \tau_1 \times \tau_2$ and the result follows by rule [Subsume].
- $(\mathcal{E}, V)$. By hypothesis and inversion of [Pair], we have $\Gamma \vdash \mathcal{E}[E] : \tau_1$ and $\Gamma \vdash V : \tau_2$ where $\tau_1 \times \tau_2 \leq S'$. By IH, we deduce $\Gamma \vdash \mathcal{E}[E'] : \tau_1$. Therefore, it holds by rule [Pair] that $\Gamma \vdash (\mathcal{E}[E'], V) : \tau_1 \times \tau_2$ and the result follows by rule [Subsume].
- $\pi_i\ \mathcal{E}$. By hypothesis and inversion of [Proj], we have $\Gamma \vdash \mathcal{E}[E] : \tau_1 \times \tau_2$ where $\tau_i \leq S'$. By IH, we deduce $\Gamma \vdash \mathcal{E}[E'] : \tau_1 \times \tau_2$ thus [Proj] yields that $\Gamma \vdash \pi_i\ (\mathcal{E}[E']) : \tau_i$, and the result follows by subsumption.
- $\mathsf{let}\ x = \mathcal{E}\ \mathsf{in}\ E_l$. By hypothesis and inversion of [Let], we have $\Gamma \vdash \mathcal{E}[E] : \forall\vec{\alpha}.\tau$ and $\Gamma, x : \forall\vec{\alpha}.\tau \vdash E_l : S'$. By IH, we deduce $\Gamma \vdash \mathcal{E}[E'] : \forall\vec{\alpha}.\tau$. Therefore, it holds by rule [Let] that $\Gamma \vdash \mathsf{let}\ x = \mathcal{E}[E']\ \mathsf{in}\ E_l : S'$.
- $\mathcal{E}\langle\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$. By hypothesis and inversion of [Cast], we have $\Gamma \vdash \mathcal{E}[E] : \tau_1$ and $\tau_2 \leq S'$. By IH, it holds that $\Gamma \vdash \mathcal{E}[E'] : \tau_1$. Therefore, by rule [Cast], we have $\Gamma \vdash \mathcal{E}[E']\langle\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle : \tau_2$, and the result follows by [Subsume].

□

Lemma B.83. *If $\Gamma \vdash E : \tau$, then for every type substitution θ, $\Gamma\theta \vdash E\theta : \tau\theta$.*

Proof. By induction on the derivation of $\Gamma \vdash E : \tau$ and by case on the last rule applied.

- [Var]. We have $\Gamma \vdash x : \forall\vec{\alpha}.\,\tau$ and $\Gamma(x) = \forall\vec{\alpha}.\,\tau$. We deduce $(\Gamma\theta)(x) = \forall\vec{\alpha}.\,\tau\theta$. Since $x\theta = x$, we apply [Var] to deduce the result: $\Gamma\theta \vdash x : \forall\vec{\alpha}.\,\tau\theta$.
- [Const]. Immediate since $b_c\theta = b_c$.
- [Abstr], [App], [Pair], [Proj], [TAbstr], [TApp], [Let]. Direct application of the induction hypothesis.
- [Subsume]. By Proposition B.32, $\tau' \leq \tau$ implies $\tau'\theta \leq \tau\theta$ for any static type substitution θ, and the result follows.
- [Cast]. By Proposition B.36, $\tau' \preccurlyeq \tau$ implies $\tau'\theta \preccurlyeq \tau\theta$ for any type substitution θ, and the result follows.

□

Lemma B.84 (Subject Reduction). *For every terms E, E' and every context Γ, if $\Gamma \vdash E : S$ and $E \hookrightarrow E'$ then $\Gamma \vdash E' : S$.*

Proof. By case disjunction over the rule used in the reduction $E \hookrightarrow E'$.

- [ExpandL] $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow V\langle \tau_1 \overset{p}{\Rightarrow} \tau_1/\tau_2\rangle\langle \tau_1/\tau_2 \overset{p}{\Rightarrow} \tau_2\rangle$. By inversion of the typing rules, $\tau_2 \leq S$. By hypothesis of the reduction rule, $\tau_2 \preccurlyeq \tau_1$. By inversion of the typing rule [Cast], we deduce that $\Gamma \vdash V : \tau_1$ and $p = \bar{l}$. By Proposition B.52, we have $\tau_2 \preccurlyeq \tau_1/\tau_2 \preccurlyeq \tau_1$. Therefore, applying the typing rule [Cast] twice yields $\Gamma \vdash V\langle \tau_1 \overset{p}{\Rightarrow} \tau_1/\tau_2\rangle : \tau_1/\tau_2$ and then $\Gamma \vdash V\langle \tau_1 \overset{p}{\Rightarrow} \tau_1/\tau_2\rangle\langle \tau_1/\tau_2 \overset{p}{\Rightarrow} \tau_2\rangle : \tau_2$. Since $\tau_2 \leq S$, applying [Subsume] yields the result.
- [ExpandR]. $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2/\tau_1\rangle\langle \tau_2/\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$. By inversion of the typing rules, $\tau_2 \leq S$. By hypothesis of the reduction rule, $\tau_1 \preccurlyeq \tau_2$. By inversion of the typing rule [Cast], we deduce that $\Gamma \vdash V : \tau_1$ and $p = l$. By Proposition B.52, we have $\tau_1 \preccurlyeq \tau_2/\tau_1 \preccurlyeq \tau_2$. Therefore, applying the typing rule [Cast] twice yields $\Gamma \vdash V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2/\tau_1\rangle : \tau_2/\tau_1$ and then $\Gamma \vdash V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2/\tau_1\rangle\langle \tau_2/\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle : \tau_2$. Since $\tau_2 \leq S$, applying [Subsume] yields the result.
- [CastId] $V\langle \tau \overset{p}{\Rightarrow} \tau\rangle \hookrightarrow V$. By inversion of the typing rules, $\tau \leq S$. By inversion of the typing rule [Cast], $\Gamma \vdash V : \tau$. Applying [Subsume] yields $\Gamma \vdash V : S$.
- [Collapse] $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle\langle \tau_1' \overset{q}{\Rightarrow} \tau_2'\rangle \hookrightarrow V$. By inversion of the typing rules, $\tau_2' \leq S$. Inversing the typing rule [Cast] twice yields $\Gamma \vdash V : \tau_1$. Since, by hypothesis of [Collapse], $\tau_1 \leq \tau_2' \leq S$, applying [Subsume] yields $\Gamma \vdash V : S$.
- [UpSimpl] $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle\langle \tau_1' \overset{q}{\Rightarrow} \tau_2'\rangle \hookrightarrow V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$. By inversion of the typing rules, $\tau_2' \leq S$. Inversing the typing rule [Cast] yields $\Gamma \vdash V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle : \tau_2$. By hypothesis of the reduction rule, $\tau_2 \leq \tau_2' \leq S$. Therefore, applying [Subsume] yields $\Gamma \vdash V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle : S$.
- [UnboxSimpl] $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow V$. By inversion of the typing rules, $\tau_2 \leq S$. By Lemma B.78, $\Gamma \vdash V : \mathrm{type}(V)$. By hypothesis of [UnboxSimpl], $\mathrm{type}(V) \leq \tau_2 \leq S$, thus applying [Subsume] yields $\Gamma \vdash V : S$.
- [CastApp] $V\langle \tau \overset{p}{\Rightarrow} \tau'\rangle V' \hookrightarrow (V\ V'\langle \tau_1' \overset{\bar{p}}{\Rightarrow} \tau_1\rangle)\langle \tau_2 \overset{p}{\Rightarrow} \tau_2'\rangle$ where $\langle \tau \overset{p}{\Rightarrow} \tau'\rangle \circ \mathrm{type}(V') = \langle \tau_1 \to \tau_2 \overset{p}{\Rightarrow} \tau_1' \to \tau_2'\rangle$.

  First of all, Lemma B.69 ensures that the casts $\langle \tau_1' \overset{\bar{p}}{\Rightarrow} \tau_1\rangle$ and $\langle \tau_2 \overset{p}{\Rightarrow} \tau_2'\rangle$ are well-formed and respect the materialization conditions present in the typing rule [Cast]. Lemma B.71 then ensures $\mathrm{type}(V') \leq \tau_1'$, thus $\Gamma \vdash V'\langle \tau_1' \overset{\bar{p}}{\Rightarrow} \tau_1\rangle : \tau_1$ by rule [Cast].
  Inversing the typing rules yields $\Gamma \vdash V : \tau$ and Lemma B.71 gives $\tau \leq \tau_1 \to \tau_2$. Therefore $\Gamma \vdash V : \tau_1 \to \tau_2$ by rule [Subsume], and $\Gamma \vdash V\ V'\langle \tau_1' \overset{\bar{p}}{\Rightarrow} \tau_1\rangle : \tau_2$ by rule [App].

  Finally, we obtain $\Gamma \vdash (V\ V'\langle \tau_1' \overset{\bar{p}}{\Rightarrow} \tau_1\rangle)\langle \tau_2 \overset{p}{\Rightarrow} \tau_2'\rangle : \tau_2'$ by rule [Cast]. The last thing we need to prove is $\tau_2' \leq S$. By inversing the typing rules, and using Lemma B.78, we have $\tau' \leq \mathrm{type}(V') \to S$. The result follows by applying Lemma B.71.
- [CastProj] $\pi_i\ (V\langle \tau \overset{p}{\Rightarrow} \tau'\rangle) \hookrightarrow (\pi_i\ V)\langle \tau_i \overset{p}{\Rightarrow} \tau_i'\rangle$ where $\pi_i\ (\langle \tau \overset{p}{\Rightarrow} \tau'\rangle) = \langle \tau_i \overset{p}{\Rightarrow} \tau_i'\rangle$. First of all, Lemma B.73 ensures that the cast $\langle \tau_i \overset{p}{\Rightarrow} \tau_i'\rangle$ is well-formed and respect the materialization conditions present in the typing rule [Cast].
  Now consider $i = 1$ (the case $i = 2$ is proved in the same way). Lemma B.75 then ensures $\tau \leq (\tau_i \times \mathbb{1})$. And, by hypothesis and inversion of the typing rules, we know that $\Gamma \vdash V : \tau$. Therefore, by rule [Subsume], we have $\Gamma \vdash V : \tau_i \times \mathbb{1}$. Then, by rule [Proj], we deduce $\Gamma \vdash \pi_1\ V : \tau_i$. Finally, the rule [Cast] allows us to conclude that $\Gamma \vdash (\pi_1\ V)\langle \tau_i \overset{p}{\Rightarrow} \tau_i'\rangle : \tau_i'$.
  Now, by hypothesis, we have $\Gamma \vdash \pi_1(V\langle \tau \overset{p}{\Rightarrow} \tau'\rangle) : S$. Thus, by inversion of the typing rules and

subsumption, we deduce $\Gamma \vdash V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle : (S \times \mathbb{1})$. That is, by inversion of [Cast], $\tau' \leq S \times \mathbb{1}$. Now, the second part of Lemma B.75 yields $\tau_i' = \min\{\tau_0 \mid \tau' \leq \tau_0 \times \mathbb{1}\}$. From this, we can deduce $\tau_i' \leq S$, and we conclude that $\Gamma \vdash (\pi_1\ V)\langle \tau_i \overset{p}{\Rightarrow} \tau_i' \rangle : S$ by subsumption.

- [SimplApp] $V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle V' \hookrightarrow V\ V'$. By inversion of the typing rules, and using Lemma B.78, we have $\tau' \leq \mathrm{type}(V') \to S$. By hypothesis of the reduction rule, we also have $\tau / \tau' = \tau$. Applying Corollary B.56 therefore yields $\tau \leq \mathrm{type}(V') \to S$. Since $\Gamma \vdash V : \tau$ by inversion of the typing rules, we deduce that $\Gamma \vdash V : \mathrm{type}(V') \to S$ by rule [Subsume]. We can then conclude by applying rule [App].
- [SimplProj] $\pi_i\ (V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle) \hookrightarrow \pi_i\ V$. By inversion of the typing rules, we have $\tau' \leq \tau_1 \times \tau_2$ where $\tau_i \leq S$. By hypothesis of the reduction rule, we also have $\tau / \tau' = \tau$. Applying Corollary B.56 therefore yields $\tau \leq \tau_1 \times \tau_2$. Since $\Gamma \vdash V : \tau$ by inversion of the typing rules, we deduce that $\Gamma \vdash V : \tau_1 \times \tau_2$ by rule [Subsume]. We can then conclude by applying rule [Proj] and [Subsume].
- [App] $(\lambda^{\tau_1 \to \tau_2} x.\, E) V \hookrightarrow E\{x := V\}$. By inversion of the typing rules, we have $\Gamma \vdash V : \tau_1$, $\tau_2 \leq S$, as well as $\Gamma, x : \tau_1 \vdash E : \tau_2$. Lemma B.81 immediately yields that $\Gamma \vdash E\{x := V\} : \tau_2$, and the result follows by [Subsume].
- [Proj] $\pi_i\ (V_1, V_2) \hookrightarrow V_i$. By inversion of the typing rules, we have $\Gamma \vdash (V_1, V_2) : \tau_1 \times \tau_2$ and $\tau_i \leq S$. Inversing the typing rules a second time yields $\Gamma \vdash V_1 : \tau_1$ and $\Gamma \vdash V_2 : \tau_2$, therefore, by [Subsume] we obtain $\Gamma \vdash V_i : S$.
- [TypeApp] $(\Lambda \vec{\alpha}.\, E)\ [\vec{t}] \hookrightarrow E\{\vec{\alpha} := \vec{t}\}$. We have, by hypothesis, $\Gamma \vdash \Lambda \vec{\alpha}.\, E : \forall \vec{\alpha}.\tau$ where $\Gamma \vdash E : \tau$ and $\tau\{\vec{\alpha} := \vec{t}\} \leq S$. Applying Lemma B.83 yields $\Gamma\{\vec{\alpha} := \vec{t}\} \vdash E\{\vec{\alpha} := \vec{t}\} : \tau\{\vec{\alpha} := \vec{t}\}$. However, by hypothesis of the typing rules, $\vec{\alpha} \notin \Gamma$. Therefore, $\Gamma\{\vec{\alpha} := \vec{t}\} = \Gamma$, and we have $\Gamma \vdash E\{\vec{\alpha} := \vec{t}\} : \tau\{\vec{\alpha} := \vec{t}\}$, which is the result.
- [Let] $\mathtt{let}\ x = V\ \mathtt{in}\ E \hookrightarrow E\{x := V\}$. By hypothesis, we have $\Gamma \vdash V : \forall \vec{\alpha}.\tau$, and $\Gamma, x : \forall \vec{\alpha}.\tau \vdash E : \tau'$ where $\tau' \leq S$. Lemma B.81 immediately yields that $\Gamma \vdash E\{x := V\} : \tau'$, and the result follows by [Subsume].
- [Context] $\mathcal{E}[E] \hookrightarrow \mathcal{E}[E']$ where $E \hookrightarrow E'$. Immediate by Lemma B.82.

□

Theorem B.85 (Soundness). *For every term E such that $\varnothing \vdash E : S$, one of the following cases holds:*

- *there exists a value V such that $E \hookrightarrow^* V$*
- *there exists a label p such that $E \hookrightarrow^* \mathtt{blame}\ p$*
- *E diverges*

Proof. Immediate corollary of Lemma B.84 and Lemma B.80. □

Theorem B.86 (Blame Safety). *For every term E such that $\varnothing \vdash E : S$, and every blame label l, $E \not\hookrightarrow^* \mathtt{blame}\ \bar{l}$.*

Proof. Given Lemma B.84, and by induction over E, it is sufficient to prove the result for reductions of length one. The proof is then done by case disjunction over the reduction rules that can produce a blame.

- [Blame] $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \langle \tau_1' \overset{q}{\Rightarrow} \tau_2' \rangle \hookrightarrow \mathtt{blame}\ q$. By hypothesis of the reduction rule, $\tau_1' \not\preccurlyeq \tau_2'$. Thus, by inversion of the typing rule [Cast], q is a positive label.
- [UpBlame] $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \langle \tau_1' \overset{q}{\Rightarrow} \tau_2' \rangle \hookrightarrow \mathtt{blame}\ q$. By hypothesis of the reduction rule, $\tau_1' \not\preccurlyeq \tau_2'$. Thus, by inversion of the typing rule [Cast], q is a positive label.
- [UnboxBlame] $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \hookrightarrow \mathtt{blame}\ p$. By hypothesis of the reduction rule, $\tau_1 \not\preccurlyeq \tau_2$. Thus, by inversion of the typing rule [Cast], p is a positive label.

- [FailApp] $V\langle \tau \xRightarrow{p} \tau' \rangle V' \hookrightarrow \texttt{blame}\ p$. By hypothesis of the reduction rule, the two possible cases are ${\tau'}/\tau = \tau$ and $\tau/\tau' = \tau'$. Moreover, by inversion of the typing rules, $\tau' \leq \mathbb{0} \to \mathbb{1}$. Thus, by contradiction, if $\tau/\tau' = \tau'$, then $\langle \tau \xRightarrow{p} \tau' \rangle \circ \mathrm{type}(V')$ would be well-defined according to Lemma B.70. Therefore, we necessarily have ${\tau'}/\tau = \tau$, and by inversion of the typing rule [Cast], p is a positive label.
- [FailProj] $\pi_i\,(V\langle \tau \xRightarrow{p} \tau' \rangle) \hookrightarrow \texttt{blame}\ p$. By hypothesis of the reduction rule, the two possible cases are ${\tau'}/\tau = \tau$ and $\tau/\tau' = \tau'$. Moreover, by inversion of the typing rules, $\tau' \leq \mathbb{1} \to \mathbb{1}$. Thus, by contradiction, if $\tau/\tau' = \tau'$, then $\pi_i\,(\langle \tau \xRightarrow{p} \tau' \rangle)$ would be well-defined according to Lemma B.74. Therefore, we necessarily have ${\tau'}/\tau = \tau$, and by inversion of the typing rule [Cast], p is a positive label.
- [CtxBlame] $\mathcal{E}[E] \hookrightarrow \texttt{blame}\ p$. Immediate by induction.

□

Theorem B.87 (Conservativity). *For every term E such that $\varnothing \vdash_{\text{Sub}} E : \tau$, $E \hookrightarrow_{\text{Sub}} E' \iff E \hookrightarrow_{\text{Set}} E'$ and $E \hookrightarrow_{\text{Sub}} \texttt{blame}\ p \iff E \hookrightarrow_{\text{Set}} \texttt{blame}\ p$.*

Proof. By cases over the rule used in the reduction of E, and induction on E.

(1) First implication, (Sub) $\implies$ (Set).
- [ExpandL] $V\langle \tau \xRightarrow{p} ? \rangle \hookrightarrow_{\text{Sub}} V\langle \tau \xRightarrow{p} \tau/? \rangle \langle \tau/? \xRightarrow{p} ? \rangle$. By hypothesis of the reduction rule, $\tau/? \neq \tau$, and $\tau \neq ?$. Therefore, $\tau/? \neq ?$ (since only ${?}/? = ?$). Thus, the rule [ExpandL] can be applied in Set to yield $V\langle \tau \xRightarrow{p} ? \rangle \hookrightarrow_{\text{Set}} V\langle \tau \xRightarrow{p} \tau/? \rangle \langle \tau/? \xRightarrow{p} ? \rangle$.
- [ExpandR] $V\langle ? \xRightarrow{p} \tau \rangle \hookrightarrow_{\text{Sub}} V\langle ? \xRightarrow{p} \tau/? \rangle \langle \tau/? \xRightarrow{p} \tau \rangle$. By hypothesis of the reduction rule, $\tau/? \neq \tau$, and $\tau \neq ?$. Therefore, $\tau/? \neq ?$ for the same reason as before. Thus, rule [ExpandR] can be applied in Set to yield $V\langle ? \xRightarrow{p} \tau \rangle \hookrightarrow_{\text{Sub}} V\langle ? \xRightarrow{p} \tau/? \rangle \langle \tau/? \xRightarrow{p} \tau \rangle$.
- [CastId] $V\langle \tau \xRightarrow{p} \tau \rangle \hookrightarrow_{\text{Sub}} V$. Immediate since [CastId] is unchanged in Set.
- [Collapse] $V\langle \rho \xRightarrow{p} ? \rangle \langle ? \xRightarrow{q} \rho' \rangle \hookrightarrow_{\text{Sub}} V$. By hypothesis, $\rho \leq \rho'$. Moreover, by definition of ground types, we have $\rho/? = \rho$ and ${\rho'}/? = \rho'$. All the hypothesis of rule [Collapse] in Set are therefore valid, and the rule can be applied to deduce $V\langle \rho \xRightarrow{p} ? \rangle \langle ? \xRightarrow{q} \rho' \rangle \hookrightarrow_{\text{Set}} V$.
- [Blame] $V\langle \rho \xRightarrow{p} ? \rangle \langle ? \xRightarrow{q} \rho' \rangle \hookrightarrow_{\text{Sub}} \texttt{blame}\ q$. We have the same hypothesis as before except $\rho \not\leq \rho'$. Therefore, we can apply rule [Blame] in Set to deduce $V\langle \rho \xRightarrow{p} ? \rangle \langle ? \xRightarrow{q} \rho' \rangle \hookrightarrow_{\text{Set}} \texttt{blame}\ q$.
- [CastApp] $V\langle \tau_1 \to \tau_2 \xRightarrow{p} \tau_1' \to \tau_2' \rangle V' \hookrightarrow_{\text{Sub}} V(V'\langle \tau_1' \xRightarrow{\bar{p}} \tau_1 \rangle)\langle \tau_2 \xRightarrow{p} \tau_2' \rangle$. We pose $\tau = \tau_1 \to \tau_2$ and $\tau' = \tau_1' \to \tau_2'$. τ and τ' are trivially in disjunctive normal form, and both are not empty (since an arrow cannot be empty). Thus, the cast $\langle \tau \xRightarrow{p} \tau' \rangle \circ \mathrm{type}(V')$ is well-defined (satisfies the conditions of Definition B.68). Moreover, by hypothesis, we know that either $\tau \preccurlyeq \tau'$ or $\tau' \preccurlyeq \tau$. By definition of the grounding operator, we then either have ${\tau'}/\tau = \tau$ or $\tau/\tau' = \tau'$. Thus, all the hypothesis of the rule [CastApp] in Set are valid.
  Finally, by inversion of the typing rule [App], we deduce that $\mathrm{type}(V') \leq \tau_1'$. A simple application of Definition B.68 (case were I and P_i are singletons) shows that $\langle \tau \xRightarrow{p} \tau' \rangle \circ \mathrm{type}(V') = \langle \tau \xRightarrow{p} \tau' \rangle$, hence the result.
- [App] $(\lambda^{\tau_1 \to \tau_2} x.\, E)V \hookrightarrow_{\text{Sub}} E\{x := V\}$. Immediate since [App] is unchanged in Set.
- [ProjCast] $\pi_i\,(V\langle \tau_1 \times \tau_2 \xRightarrow{p} \tau_1' \times \tau_2' \rangle) \hookrightarrow_{\text{Sub}} (\pi_i\, V)\langle \tau_i \xRightarrow{p} \tau_i' \rangle$. Let $\tau = \tau_1 \times \tau_2$ and $\tau' = \tau_1' \times \tau_2'$. τ and τ' are trivially in disjunctive normal form, and both are not empty (since a product cannot be empty in Sub, as both sides cannot be empty). Thus, the cast $\pi_i\,(\langle \tau \xRightarrow{p} \tau' \rangle)$ is well-defined as it satisfies all the conditions of Definition B.72. Moreover, by hypothesis (inversion of typing rule [Cast]), we know that either $\tau \preccurlyeq \tau'$ or $\tau' \preccurlyeq \tau$. By definition of

the grounding operator, this yields that either $\tau'/\tau = \tau$ or $\tau/\tau' = \tau'$ respectively. Thus, all the hypothesis of [CastProj] in Set are verified.
Finally, a simple application of Definition B.72 (case were I and P_i are singletons, and $N_i = \varnothing$) shows that $\pi_i\,(\langle \tau \overset{p}{\Rightarrow} \tau' \rangle) = \langle \tau_i \overset{p}{\Rightarrow} \tau_i' \rangle$, hence the result.

- [Proj] $\pi_i\,(V_1, V_2) \hookrightarrow_{\text{Sub}} V_i$. Immediate since [Proj] is unchanged in Set.
- [TypeApp] $(\Lambda\vec{\alpha}.\,E)\,[\vec{t}] \hookrightarrow_{\text{Sub}} E\{\vec{\alpha} := \vec{t}\}$. Immediate since [TypeApp] is unchanged in Set.
- [Let] $\texttt{let}\ x = V\ \texttt{in}\ E \hookrightarrow_{\text{Sub}} E\{x := V\}$. Immediate since [Let] is unchanged in Set.
- [Context] $\mathcal{E}[E] \hookrightarrow_{\text{Sub}} \mathcal{E}[E']$ where $E \hookrightarrow_{\text{Sub}} E'$. By induction hypothesis, $E \hookrightarrow_{\text{Set}} E'$. Thus, by rule [Context] in the Set system, $\mathcal{E}[E] \hookrightarrow_{\text{Set}} \mathcal{E}[E']$.
- [CtxBlame] $\mathcal{E}[E] \hookrightarrow_{\text{Sub}} \texttt{blame}\ p$ where $E \hookrightarrow_{\text{Sub}} \texttt{blame}\ p$. By induction hypothesis, $E \hookrightarrow_{\text{Set}} \texttt{blame}\ p$. Thus, by rule [CtxBlame] in the Set system, $\mathcal{E}[E] \hookrightarrow_{\text{Set}} \texttt{blame}\ p$.

(2) Second implication, (Set) $\implies$ (Sub). We omit the trivial cases where the same rule is present in both systems.

- [ExpandL] $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \hookrightarrow V\langle \tau_1 \overset{p}{\Rightarrow} \tau_1/\tau_2 \rangle \langle \tau_1/\tau_2 \overset{p}{\Rightarrow} \tau_2 \rangle$. By hypothesis of the reduction rule, $\tau_2 \preccurlyeq \tau_1$ and $\tau_1/\tau_2 \neq \tau_2$. Therefore, by Lemma B.53, we deduce that $\tau_2 = ?$. Since $\tau_1/\tau_2 \neq \tau_2$, we have $\tau_1 \neq ?$, and by hypothesis of [ExpandL] $\tau_1/\tau_2 \neq \tau_1$ therefore all the conditions of [ExpandL] in Sub are verified, and the result follows.
- [ExpandR] $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \hookrightarrow V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2/\tau_1 \rangle \langle \tau_2/\tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle$. By hypothesis of the reduction rule, $\tau_1 \preccurlyeq \tau_2$ and $\tau_2/\tau_1 \neq \tau_1$. Therefore, by Lemma B.53, we deduce that $\tau_1 = ?$. Since $\tau_2/\tau_1 \neq \tau_1$, we have $\tau_2 \neq ?$, and by hypothesis of [ExpandR] $\tau_2/\tau_1 \neq \tau_2$ therefore all the conditions of [ExpandR] in Sub are verified, and the result follows.
- [Collapse] $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \langle \tau_1' \overset{q}{\Rightarrow} \tau_2' \rangle \hookrightarrow V$. By hypothesis of the reduction rule, $\tau_2'/\tau_1' = \tau_2'$ and $\tau_2' \neq \tau_1'$ (by precedence of [CastId]). Thus, by Lemma B.54, we have $\tau_1' = ?$. By typing hypothesis, we also have $\tau_2 \leq \tau_1'$. By definition of subtyping on non-set-theoretic types, we deduce $\tau_2 = ?$. And by hypothesis of the reduction rule, we finally have $\tau_1 \leq \tau_2'$ and $\tau_1/\tau_2 = \tau_1$ (the case $\tau_2/\tau_1 = \tau_1$ being only possible if $\tau_2 = \tau_1 = ?$). Thus, all the conditions for the rule [Collapse] in Sub are verified, and the result follows.
- [Blame] $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \langle \tau_1' \overset{q}{\Rightarrow} \tau_2' \rangle \hookrightarrow \texttt{blame}\ q$. Same reasoning as before, except this time $\tau_1 \not\leq \tau_2'$ which allows us to apply rule [Blame] in Sub.
- [UpSimpl] $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \langle \tau_1' \overset{q}{\Rightarrow} \tau_2' \rangle \hookrightarrow V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle$. By hypothesis, $\tau_2'/\tau_1' = \tau_2'$ and $\tau_1' \neq \tau_2'$. By Lemma B.54, $\tau_1' = ?$. Moreover, by typing hypothesis, $\tau_2 \leq \tau_1'$ thus $\tau_2 = ?$ by definition of subtyping. By hypothesis, $\tau_1/\tau_2 = \tau_2$ but since $\tau_2 = ?$, we necessarily have $\tau_1 = ?$ by Definition B.51. Thus, we have a contradiction since $\tau_1 \neq \tau_2$ by hypothesis, and this rule cannot be applied.
- [UpBlame] $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \langle \tau_1' \overset{q}{\Rightarrow} \tau_2' \rangle \hookrightarrow \texttt{blame}\ q$. Same reasoning as before, this rule cannot be applied.
- [UnboxSimpl] $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \hookrightarrow V$. By hypothesis, $\tau_2/\tau_1 = \tau_2$ and $\tau_1 \neq \tau_2$. Applying Lemma B.54 yields $\tau_1 = ?$. However, by hypothesis, $\varnothing \vdash V : \tau_1$. A simple case disjunction on V shows that this cannot hold, thus we have a contradiction and this rule cannot be applied.
- [UnboxBlame] $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \hookrightarrow \texttt{blame}\ p$. Same reasoning as before, this rule cannot be applied.
- [CastApp] $V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle V' \hookrightarrow (V\ V'\langle \tau_1' \overset{\bar{p}}{\Rightarrow} \tau_1 \rangle) \langle \tau_2 \overset{p}{\Rightarrow} \tau_2' \rangle$ where $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle \circ \text{type}(V') = \langle \tau_1 \to \tau_2 \overset{p}{\Rightarrow} \tau_1' \to \tau_2' \rangle$. By hypothesis and inversion of rule [App], $\tau' \leq \mathbb{0} \to \mathbb{1}$. Since τ' does not contain connectives, $\tau' = \sigma_1' \to \sigma_2'$ for some σ_1', σ_2'. Moreover, by hypothesis of the reduction

we either have $\tau/\tau' = \tau'$ or $\tau'/\tau = \tau$, thus necessarily $\tau = \sigma_1 \to \sigma_2$ for some σ_1, σ_2 by Definition B.51. Moreover, by hypothesis, we have $\mathrm{type}(V') \leq \sigma_1'$. A simple application of Definition B.68 then yields $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle \circ \mathrm{type}(V') = \langle \sigma_1 \to \sigma_2 \overset{p}{\Rightarrow} \sigma_1' \to \sigma_2' \rangle = \langle \tau_1 \to \tau_2 \overset{p}{\Rightarrow} \tau_1' \to \tau_2' \rangle$. Applying rule [App] in Sub yields $V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle V' \hookrightarrow (V\ V' \langle \sigma_1' \overset{\bar{p}}{\Rightarrow} \sigma_1 \rangle) \langle \sigma_2 \overset{p}{\Rightarrow} \sigma_2' \rangle$, hence the result.

- [CastProj] $\pi_i\ (V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle) \hookrightarrow (\pi_i\ V)\langle \tau_i \overset{p}{\Rightarrow} \tau_i' \rangle$ where $\langle \tau_i \overset{p}{\Rightarrow} \tau_i' \rangle = \pi_i\ (\langle \tau \overset{p}{\Rightarrow} \tau' \rangle)$. Same reasoning with product types and Definition B.72.
- [FailApp] $V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle V' \hookrightarrow \mathtt{blame}\ p$. Using the same reasoning as for [CastApp], we deduce $\tau = \tau_1 \to \tau_2$ and $\tau' = \tau_1' \to \tau_2'$. In particular, both τ and τ' are trivially in disjunctive normal form and are non-empty, and thus verify all the conditions of Definition B.68. Therefore, this rule cannot be applied.
- [FailProj] $\pi_i\ (V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle) \hookrightarrow \mathtt{blame}\ p$. Same reasoning as before but with product types. This rule cannot be applied.
- [SimplApp] $V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle V' \hookrightarrow V\ V'$. By hypothesis, $\tau/\tau' = \tau$, and $\tau \neq \tau'$. Therefore, by Lemma B.54, we have $\tau' = ?$. But $? \not\leq \tau_1 \to \tau_2$ for every τ_1 and τ_2, therefore the reducee cannot be well-typed, and this rule cannot be applied.
- [SimplProj] $\pi_i\ (V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle) \hookrightarrow \pi_i\ V$. Same reasoning as before, this rule cannot be applied.

□

B.9 Type Inference with Set-Theoretic Types

Lemma B.88 (Stability of typing under type substitution). *If $\Gamma \vdash e \rightsquigarrow E : \tau$, then, for every static type substitution θ, we have $\Gamma\theta \vdash e\theta \rightsquigarrow E\theta : \tau\theta$.*

Proof. By induction on the derivation of $\Gamma \vdash e \rightsquigarrow E : \tau$ and by case on the last rule applied.

Case: [Var]

	$\Gamma \vdash x \rightsquigarrow x\ [\vec{t}] : \tau\{\vec{\alpha} := \vec{t}\}$	Given
	$\Gamma(x) = \forall \vec{\alpha}.\ \tau$	Given
	$(\Gamma\theta)(x) = \forall \vec{\alpha}.\ \tau\theta$	since, by α-renaming, $\vec{\alpha} \sharp \theta$
(1)	$\Gamma\theta \vdash x \rightsquigarrow x\ [\vec{t}\theta] : \tau\theta\{\vec{\alpha} := \vec{t}\theta\}$	by [Var], since the $\vec{t}\theta$ are all static
(2)	$\tau\theta\{\vec{\alpha} := \vec{t}\theta\} = \tau\{\vec{\alpha} := \vec{t}\}\theta$	since $\vec{\alpha} \sharp \theta$, $\forall \alpha \in \mathrm{var}(\tau).\ \alpha\theta\{\vec{\alpha} := \vec{t}\theta\} = \alpha\{\vec{\alpha} := \vec{t}\}\theta$
	$\Gamma\theta \vdash x \rightsquigarrow x\ [\vec{t}]\theta : \tau\{\vec{\alpha} := \vec{t}\}\theta$	by (1) and (2)

Case: [Const]
Straightforward, since $b_c\theta = b_c$.

Case: [Abstr], [AAbstr], [App], [Pair], [Proj]
Direct application of the induction hypothesis. For [Abstr], note that $t\theta$ is always static.

Case: [Subsume]
By Proposition B.32, $\tau' \leq \tau$ implies $\tau'\theta \leq \tau\theta$ for any static type substitution θ.

Case: [Materialize]
By Proposition B.36, $\tau' \preccurlyeq \tau$ implies $\tau'\theta \preccurlyeq \tau\theta$ for any type substitution θ.

Case: [Let]

$\Gamma \vdash (\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2) \rightsquigarrow (\mathsf{let}\ x = \Lambda\vec{\alpha},\vec{\beta}.\ E_1\ \mathsf{in}\ E_2) : \tau$ Given

By inversion of [Let]:

(1) $\Gamma \vdash e_1 \rightsquigarrow E_1 : \tau_1$

(2) $\Gamma, x : \forall \vec{\alpha},\vec{\beta}.\, \tau_1 \vdash e_2 \rightsquigarrow E_2 : \tau$

(3) $\vec{\alpha},\vec{\beta} \sharp \Gamma$ and $\vec{\beta} \sharp e_1$

Let $\vec{\alpha}_1$ and $\vec{\beta}_1$ be vectors of distinct variables chosen outside $\mathrm{var}(\Gamma)$, $\mathrm{var}(e_1)$, $\mathrm{dom}(\theta)$, and $\mathrm{var}(\theta)$. Let $\rho = \{\vec{\alpha} \coloneqq \vec{\alpha}_1\} \cup \{\vec{\beta} \coloneqq \vec{\beta}_1\}$.

$\Gamma\rho \vdash e_1\rho \rightsquigarrow E_1\rho : \tau_1\rho$ by IH from (1), since ρ is static

(4) $\Gamma \vdash e_1\{\vec{\alpha} \coloneqq \vec{\alpha}_1\} \rightsquigarrow E_1\rho : \tau_1\rho$ by (3)

(5) $\Gamma\theta \vdash e_1\{\vec{\alpha} \coloneqq \vec{\alpha}_1\}\theta \rightsquigarrow E_1\rho\theta : \tau_1\rho\theta$ by IH from (4)

(6) $\Gamma\theta, x : (\forall \vec{\alpha},\vec{\beta}.\, \tau_1)\theta \vdash e_2\theta \rightsquigarrow E_2\theta : \tau\theta$ by IH from (2)

(7) $\Gamma\theta, x : (\forall \vec{\alpha}_1,\vec{\beta}_1.\, \tau_1\rho)\theta \vdash e_2\theta \rightsquigarrow E_2\theta : \tau\theta$ by α-renaming from (6)

(8) $\Gamma\theta, x : (\forall \vec{\alpha}_1,\vec{\beta}_1.\, \tau_1\rho\theta) \vdash e_2\theta \rightsquigarrow E_2\theta : \tau\theta$ from (7) since $\vec{\alpha}_1,\vec{\beta}_1 \sharp \theta$

$\Gamma\theta \vdash (\mathsf{let}\ \vec{\alpha}_1\ x = e_1\{\vec{\alpha} \coloneqq \vec{\alpha}_1\}\theta\ \mathsf{in}\ e_2\theta) \rightsquigarrow$
$(\mathsf{let}\ x = \Lambda\vec{\alpha}_1,\vec{\beta}_1.\, E_1\rho\theta\ \mathsf{in}\ E_2\theta) : \tau\theta$ by [Let] from (5) and (8)

This concludes the proof because $\mathsf{let}\ \vec{\alpha}_1\ x = e_1\{\vec{\alpha} \coloneqq \vec{\alpha}_1\}\theta\ \mathsf{in}\ e_2\theta$ and $(\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2)\theta$ are equivalent by α-renaming, as are $\mathsf{let}\ x = \Lambda\vec{\alpha}_1,\vec{\beta}_1.\, E_1\rho\theta\ \mathsf{in}\ E_2\theta$ and $(\mathsf{let}\ x = \Lambda\vec{\alpha},\vec{\beta}.\, E_1\ \mathsf{in}\ E_2)\theta$. □

Given two type schemes S_1 and S_2, we write $S_1 \leq S_2$ when the schemes have the same quantified variables and their types are in the subtyping relation: that is, $\forall\vec{\alpha}.\, \tau_1 \leq \forall\vec{\alpha}.\, \tau_2$ if and only if $\tau_1 \leq \tau_2$. We write $\Gamma_1 \leq \Gamma_2$ when $\mathrm{dom}(\Gamma_1) = \mathrm{dom}(\Gamma_2)$ and, for all $x \in \mathrm{dom}(\Gamma_1)$, $\Gamma_1(x) \leq \Gamma_2(x)$.

Lemma B.89 (Weakening). *If* $\Gamma_2 \vdash e \rightsquigarrow E : \tau$ *and* $\Gamma_1 \leq \Gamma_2$, *then* $\Gamma_1 \vdash e \rightsquigarrow E : \tau$.

Proof. By induction on the derivation of $\Gamma_2 \vdash e \rightsquigarrow E : \tau$ and by case on the last rule applied.

- [Var]: we have $\Gamma_2 \vdash x \rightsquigarrow x\,[\vec{t}] : \tau\{\vec{\alpha} \coloneqq \vec{t}\}$, where $\Gamma_2(x) = \forall\vec{\alpha}.\, \tau$. By definition of $\Gamma_1 \leq \Gamma_2$, we have $\Gamma_1(x) \leq \Gamma_2(x)$, therefore $\Gamma_1(x) = \forall\vec{\alpha}.\, \tau'$ and $\tau' \leq \tau$. By [Var] we derive $\Gamma_1 \vdash x \rightsquigarrow x\,[\vec{t}] : \tau'\{\vec{\alpha} \coloneqq \vec{t}\}$; then by [Subsume] we derive $\Gamma_1 \vdash x \rightsquigarrow x\,[\vec{t}] : \tau\{\vec{\alpha} \coloneqq \vec{t}\}$ since $\tau'\{\vec{\alpha} \coloneqq \vec{t}\} \leq \tau\{\vec{\alpha} \coloneqq \vec{t}\}$ (by Proposition B.32, subtyping is preserved by static type substitutions).
- [Const]: straightforward.
- [Abstr], [AAbstr], [App], [Pair], [Proj], [Subsume], [Materialize]: we conclude by direct application of the induction hypothesis. For [Abstr] and [AAbstr], note that $\Gamma_1 \leq \Gamma_2$ implies $(\Gamma_1, x : \tau) \leq (\Gamma_2, x : \tau)$ for every τ.
- [Let]: we have derived $\Gamma_2 \vdash (\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2) \rightsquigarrow (\mathsf{let}\ x = \Lambda\vec{\alpha},\vec{\beta}.\, E_1\ \mathsf{in}\ E_2) : \tau$ from the premises

  $$\Gamma_2 \vdash e_1 \rightsquigarrow E_1 : \tau_1 \qquad \Gamma_2, x : \forall\vec{\alpha},\vec{\beta}.\, \tau_1 \vdash e_2 \rightsquigarrow E_2 : \tau \qquad \vec{\alpha},\vec{\beta} \sharp \Gamma_2 \text{ and } \vec{\beta} \sharp e_1\,.$$

  Let $\vec{\alpha}_1$ and $\vec{\beta}_1$ be vectors of variables chosen outside $\mathrm{var}(\Gamma_1)$ and $\mathrm{var}(e_1)$. Let $\rho = \{\vec{\alpha} \coloneqq \vec{\alpha}_1\} \cup \{\vec{\beta} \coloneqq \vec{\beta}_1\}$. Since ρ is a static type substitution, we can apply Lemma B.88 to derive $\Gamma_2\rho \vdash e_1\rho \rightsquigarrow E_1\rho : \tau_1\rho$, which is $\Gamma_2 \vdash e_1\rho \rightsquigarrow E_1\rho : \tau_1\rho$ because the $\vec{\alpha}$ and $\vec{\beta}$ variables do not occur in Γ_2.
  By induction, we derive $\Gamma_1 \vdash e_1\rho \rightsquigarrow E_1\rho : \tau_1\rho$ and $\Gamma_1, x : \forall\vec{\alpha},\vec{\beta}.\, \tau_1 \vdash e_2 \rightsquigarrow E_2 : \tau$. By α-renaming, $\forall\vec{\alpha},\vec{\beta}.\, \tau_1$ is equivalent to $\forall\vec{\alpha}_1,\vec{\beta}_1.\, \tau_1\rho$. Note that the $\vec{\beta}_1$ variables do not occur in

$e_1\rho$, because they do not occur in e_1 and they are introduced by ρ only on variables which themselves do not occur in e_1. Therefore, we have

$$\Gamma_1 \vdash e_1\rho \rightsquigarrow E_1\rho\colon \tau_1\rho \qquad \Gamma_1, x\colon \forall\vec{\alpha}_1,\vec{\beta}_1.\,\tau_1\rho \vdash e_2 \rightsquigarrow E_2\colon \tau \qquad \vec{\alpha}_1,\vec{\beta}_1 \sharp \Gamma_1 \text{ and } \vec{\beta}_1 \sharp e_1\rho$$

from which we derive $\Gamma_1 \vdash (\text{let } \vec{\alpha}_1\ x = e_1\rho \text{ in } e_2) \rightsquigarrow (\text{let } x = \Lambda\vec{\alpha}_1,\vec{\beta}_1.\,E_1\rho \text{ in } E_2)\colon \tau$, which is the result we need since, by α-renaming, let $\vec{\alpha}\ x = e_1$ in e_2 and let $\vec{\alpha}_1\ x = e_1\rho$ in e_2 are equivalent, as are (let $x = \Lambda\vec{\alpha},\vec{\beta}.\,E_1$ in E_2) and (let $x = \Lambda\vec{\alpha}_1,\vec{\beta}_1.\,E_1\rho$ in E_2). □

We assume that there exists a function $\text{tally}_{(\cdot)}(\cdot)$ such that, when $\overline{t^1 \mathrel{\dot\leq} t^2}$ is a set of constraints of the form $t^1 \mathrel{\dot\leq} t^2$ and Δ is a finite subset of $\mathcal{V}^\alpha$, $\text{tally}_\Delta(\overline{t^1 \mathrel{\dot\leq} t^2})$ is a finite (possibly empty) set of type substitutions mapping both α and X variables to type frames.

When $\theta \in \text{tally}_\Delta(\overline{t^1 \mathrel{\dot\leq} t^2})$, we have:

- for every $(t^1 \mathrel{\dot\leq} t^2) \in \overline{t^1 \mathrel{\dot\leq} t^2}$, $t^1\theta \leq_T t^2\theta$;
- $\text{dom}(\theta) \subseteq \text{var}(\overline{t^1 \mathrel{\dot\leq} t^2}) \setminus \Delta$.

Tallying generates some fresh variables, so $\text{var}(\theta) \not\subseteq \text{var}(\overline{t^1 \mathrel{\dot\leq} t^2})$. We can assume that the variables in $\text{var}(\theta) \setminus \text{var}(\overline{t^1 \mathrel{\dot\leq} t^2}$ are drawn from any given set of fresh variables (e.g., we can assume that they are disjoint from Δ).

Tallying also has a completeness property. We omit it because the type constraint solving algorithm we build using it is incomplete anyway.

We define an algorithm based on tallying that also handles equality constraints between type frames and variables, of the form $(T \doteq \alpha)$. We are only interested in using this when the equality constraints are those we will generate from materialization constraints. Therefore, we give an algorithm tailored to this situation, which fails unless certain conditions are satisfied. In practice, these conditions should never occur when solving the constraints we generate in our system. However, we do not prove this: proving would only be needed to show completeness of the solving algorithm, but completeness does not hold for the algorithm anyway.

We define as follows the algorithm $\text{tally}^{\doteq}_\Delta(\{\,(t^1_i \mathrel{\dot\leq} t^2_i) \mid i \in I\,\} \cup \{\,(T_j \mathrel{\dot\leq} \alpha_j) \mid j \in J\,\})$:

(1) If any of the following conditions holds, return $\varnothing$:
   - there exist j_1 and j_2 in J such that $\alpha_{j_1} = \alpha_{j_2}$ and $T_{j_1} \neq T_{j_2}$;
   - there exist j_1 and j_2 in J such that $\alpha_{j_1} \in \text{var}(T_{j_2})$;
   - there exists $j \in J$ such that $\alpha_j \in \Delta$.

(2) Compute $\Theta = \text{tally}_\Delta(\{\,(t^1_i\{\alpha_j \coloneqq T_j\}_{j\in J} \mathrel{\dot\leq} t^2_i\{\alpha_j \coloneqq T_j\}_{j\in J}) \mid i \in I\,\})$.

(3) Return $\{\,\theta_0 \cup \{\alpha_j \coloneqq T_j\theta_0\}_{j\in J} \mid \theta_0 \in \Theta\,\}$.

As anticipated, in step (1) the algorithm fails if some conditions are met. These never occur in the way we use the algorithm, because α in a constraint $(\tau \mathrel{\dot\leq} \alpha)$ (which will become $(T \doteq \alpha)$) is always chosen fresh. We do not prove this because it would only be needed for a proof of completeness of solve, which fails for the difficulties in dealing with recursive types.

In step (3), the union of the two substitutions is well-defined because θ_0 is not defined on the α_j, since they do not appear in the constraints given to tally.

The algorithm satisfies the following property.

Proposition B.90.

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

Proof. Let $\theta \in \mathsf{tally}^{\doteq}_{\Delta}\Big(\overline{t^1 \mathrel{\dot\leq} t^2} \cup \overline{T \doteq \alpha}\Big)$, with

$$\overline{t^1 \mathrel{\dot\leq} t^2} = \{\, (t_i^1 \mathrel{\dot\leq} t_i^2) \mid i \in I \,\} \qquad \overline{T \doteq \alpha} = \{\, (T_j \doteq \alpha_j) \mid j \in J \,\}$$

By definition of $\mathsf{tally}^{\doteq}$, we have:

$$\theta_0 \in \mathsf{tally}_{\Delta}\Big(\Big\{\, (t_i^1\{\alpha_j \coloneqq T_j\}_{j\in J} \mathrel{\dot\leq} t_i^2\{\alpha_j \coloneqq T_j\}_{j\in J}) \;\Big|\; i \in I \,\Big\}\Big) \qquad \theta = \theta_0 \cup \{\alpha_j \coloneqq T_j\theta_0\}_{j\in J}$$

Let $i \in I$. We must show $t_i^1\theta \leq_T t_i^2\theta$.

By the properties of tallying, $t_i^1\{\alpha_j \coloneqq T_j\}_{j\in J}\theta_0 \leq_T t_i^2\{\alpha_j \coloneqq T_j\}_{j\in J}\theta_0$. We have

$$t_i^1\{\alpha_j \coloneqq T_j\}_{j\in J}\theta_0 = t_i^1\theta \qquad t_i^2\{\alpha_j \coloneqq T_j\}_{j\in J}\theta_0 = t_i^2\theta$$

and therefore $t_i^1\theta \leq_T t_i^2\theta$.

Let $j \in J$. We must show $T_j\theta = \alpha_j\theta$. We have $\alpha_j\theta = T_j\theta_0$. We also have $T_j\theta = T_j\theta_0$ because $\mathsf{var}(T_j) \cap \{\, \alpha_j \mid j \in J \,\} = \varnothing$ (this is checked in step (1) of the algorithm).

Finally, by the properties of tallying,

$$\mathsf{dom}(\theta_0) \subseteq \mathsf{var}\Big(\Big\{\, (t_i^1\{\alpha_j \coloneqq T_j\}_{j\in J} \mathrel{\dot\leq} t_i^2\{\alpha_j \coloneqq T_j\}_{j\in J}) \;\Big|\; i \in I \,\Big\}\Big) \setminus \Delta$$

and, as a consequence,

$$\mathsf{dom}(\theta) \subseteq \mathsf{dom}(\theta_0) \cup \{\, \alpha_j \mid j \in J \,\} \subseteq \mathsf{var}\big(\overline{t^1 \mathrel{\dot\leq} t^2} \cup \overline{T \doteq \alpha}\big) \setminus \Delta\,. \qquad \square$$

Constraint solving is then defined as shown in the main text.

Proposition B.91. *If $\theta \in \mathsf{solve}_{\Delta}(D)$, then $\theta \Vdash_{\Delta} D$ and $\mathsf{dom}(\theta) \subseteq \mathsf{var}(D)$.*

Proof. Let

$$D = \{\, (t_i^1 \mathrel{\dot\leq} t_i^2) \mid i \in I \,\} \cup \{\, (\tau_j \mathrel{\dot\preccurlyeq} \alpha_j) \mid j \in J \,\} \cup \{\, (\alpha_k \mathrel{\dot\preccurlyeq} \alpha_k) \mid k \in K \,\}$$

(where we assume, for all $j \in J$, that $\tau_j \neq \alpha_j$).

Let $\theta \in \mathsf{solve}_{\Delta}(D)$. Then, by definition of solve, we have the following:

$$\theta = (\theta_0\theta_0')^{\dagger}|_{\mathcal{V}^{\alpha}} \qquad \theta_0 \in \mathsf{tally}^{\doteq}_{\Delta}(\{\, (t_i^1 \mathrel{\dot\leq} t_i^2) \mid i \in I \,\} \cup \overline{T \doteq \alpha}) \qquad \theta_0' = \{\vec{X} \coloneqq \vec{\alpha}'\} \cup \{\vec{\alpha} \coloneqq \vec{X}\}$$
$$\overline{T \doteq \alpha} = \{\, (T_j \doteq \alpha_j) \mid j \in J \,\} \qquad \forall j \in J.\ T_j^{\dagger} = \tau_j$$
$$\overline{A} = \mathsf{var}_{\dot\preccurlyeq}(D)\theta_0 \cup \textstyle\bigcup_{i\in I}(\mathsf{var}^{\pm}(t_i^1\theta_0) \cup \mathsf{var}^{\pm}(t_i^2\theta_0))$$
$$\vec{X} = \mathcal{V}^X \cap \overline{A} \qquad \vec{\alpha} = \mathsf{var}(D) \setminus (\Delta \cup \mathsf{dom}(\theta_0) \cup \overline{A}) \qquad \vec{\alpha}' \text{ and } \vec{X} \text{ fresh}$$

We must show the following results:

$$\forall i \in I.\ t_i^1\theta \leq t_i^2\theta \qquad \forall j \in J.\ \tau_j\theta \preccurlyeq \alpha_j\theta$$
$$\mathsf{static}(\theta, \textstyle\bigcup_{j\in J}\mathsf{var}(\tau_j) \cup \{\, \alpha_j \mid j \in J \,\}) \qquad \mathsf{dom}(\theta) \subseteq \mathsf{var}(D) \setminus \Delta$$

To show $\forall i \in I.\ t_i^1\theta \leq t_i^2\theta$, consider an arbitrary $i \in I$. By Proposition B.90, we have $t_i^1\theta_0 \leq_T t_i^2\theta_0$. Then, by Proposition B.6, we have $t_i^1\theta_0\theta_0' \leq_T t_i^2\theta_0\theta_0'$. We show that $t_i^1\theta_0\theta_0'$ and $t_i^2\theta_0\theta_0'$ are polarized, which implies that $(t_i^1\theta_0\theta_0')^{\dagger} \leq (t_i^2\theta_0\theta_0')^{\dagger}$ since every polarized type frame T is such that $T \in \star^{\mathsf{pol}}(T^{\dagger})$. Consider an arbitrary $j \in \{1, 2\}$: we must show $\mathsf{var}^+_X(t_i^j\theta_0\theta_0') \cap \mathsf{var}^-_X(t_i^j\theta_0\theta_0') = \varnothing$. By contradiction, assume $X \in \mathsf{var}^+_X(t_i^j\theta_0\theta_0') \cap \mathsf{var}^-_X(t_i^j\theta_0\theta_0')$. Since the variables in $\vec{\alpha}'$ and $\vec{X}'$ are all distinct, θ_0' does not map different variables to the same variable. Moreover, note that $\mathsf{var}(\theta_0') \not\sharp \mathsf{var}(t_i^j)$. Therefore, there are two cases:

- $X \in \mathsf{var}^+_X(t_i^j\theta_0) \cap \mathsf{var}^-_X(t_i^j\theta_0)$ and $X \notin \mathsf{dom}(\theta_0')$;
- there exists an $A \in \mathsf{var}^+(t_i^j\theta_0) \cap \mathsf{var}^-(t_i^j\theta_0)$ such that $A\theta_0' = X$.

In the first case, the first condition implies $X \in \overline{A}$: but then $X \notin \mathsf{dom}(\theta_0')$ is impossible. In the second case, we would have $A \in \overline{A}$: therefore, $A\theta_0' = X$ is impossible. Finally, $(t_i^1\theta_0\theta_0')^\dagger \leq (t_i^2\theta_0\theta_0')^\dagger$ implies $t_i^1\theta \mathrel{\dot\leq} t_i^2\theta$ because $\mathsf{var}(t_i^1) \cup \mathsf{var}(t_i^2) \subseteq \mathcal{V}^\alpha$.

To show $\forall j \in J.\ \tau_j\theta \preccurlyeq \alpha_j\theta$, consider an arbitrary $j \in J$. By Proposition B.90, we have $T_j\theta_0 = \alpha_j\theta_0$. Moreover,

$$\tau_j\theta = (\tau_j\theta_0\theta_0')^\dagger = (T_j^\dagger\theta_0\theta_0')^\dagger \qquad \alpha_j\theta = (\alpha_j\theta_0\theta_0')^\dagger = (T_j\theta_0\theta_0')^\dagger$$

We have $\theta_j\theta \preccurlyeq \alpha_j\theta$ because, for every $\alpha \in \mathsf{var}_\alpha(T_j)$, $(\alpha^\dagger\theta_0\theta_0')^\dagger = (\alpha\theta_0\theta_0')^\dagger$.

To show $\mathsf{dom}(\theta) \subseteq \mathsf{var}(D) \setminus \Delta$, consider $\alpha \notin \mathsf{var}(D) \setminus \Delta$: we show $\alpha\theta = \alpha$. (Note that, trivially, $X\theta = X$ for every X.) By Proposition B.90, we have

$$\mathsf{dom}(\theta_0) \subseteq \mathsf{var}\big(\{\,(t_i^1 \mathrel{\dot\leq} t_i^2) \mid i \in I\,\} \cup \overline{T \doteq \alpha}\big) \setminus \Delta$$

Since $\mathsf{var}_\alpha\big(\{\,(t_i^1 \mathrel{\dot\leq} t_i^2) \mid i \in I\,\} \cup \overline{T \doteq \alpha}\big) \subseteq \mathsf{var}(D)$, we have $\alpha\theta_0 = \alpha$. Then, $\alpha\theta_0' = \alpha$ since $\mathsf{dom}(\theta_0') \cap \mathcal{V}^\alpha \subseteq \mathsf{var}(D)$.

Finally, to show $\mathsf{static}(\theta, \bigcup_{j\in J}\mathsf{var}(\tau_j) \cup \{\,\alpha_j \mid j \in J\,\})$, consider an arbitrary $\alpha \in \bigcup_{j\in J}\mathsf{var}(\tau_j) \cup \{\,\alpha_j \mid j \in J\,\}$: we show that $\alpha\theta$ is static, that is, that $\mathsf{var}_X(\alpha\theta_0\theta_0') = \varnothing$. Note that $\alpha \in \mathsf{var}_{\dot\preccurlyeq}(D)$. We have $\mathsf{var}(\alpha\theta_0) \subseteq \mathsf{var}_{\dot\preccurlyeq}(D)\theta_0$ and $\mathsf{var}(\alpha\theta_0\theta_0') = \bigcup_{A\in\mathsf{var}(\alpha\theta_0)}\mathsf{var}(A\theta_0')$. Therefore, if there existed $X \in \mathsf{var}(\alpha\theta_0\theta_0')$, there should exist $A \in \mathsf{var}(\alpha\theta_0)$ such that $X \in \mathsf{var}(A\theta_0')$. By definition of θ_0', we would need $A \in \vec\alpha$ or $A \in \mathcal{V}^X \setminus \mathsf{dom}(\theta_0')$: but $\vec\alpha$ is disjoint from $\mathsf{var}_{\dot\preccurlyeq}(D)\theta_0$, and $\mathcal{V}^X \cap \mathsf{var}_{\dot\preccurlyeq}(D)\theta_0 \subseteq \mathsf{dom}(\theta_0')$. □

Lemma B.92. *Let $\mathcal{D}$ be a derivation of $\Gamma;\Delta \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D$. Then:*

- *if $e = x$, then $\Gamma(x) = \forall\vec\alpha.\,\tau$ and $D = \{(\tau\{\vec\alpha := \vec\beta\} \mathrel{\dot\preccurlyeq} \alpha), (\alpha \mathrel{\dot\leq} t)\}$ (for some τ, α, $\vec\alpha$, $\vec\beta$);*
- *if $e = c$, then $D = \{b_c \mathrel{\dot\leq} t\}$;*
- *if $e = \lambda x.\,e'$, then $\mathcal{D}$ contains a sub-derivation of $(\Gamma, x : \alpha_1);\Delta \vdash \langle\!\langle e' : \alpha_2 \rangle\!\rangle \rightsquigarrow D'$, and $D = D' \cup \{(\alpha_1 \mathrel{\dot\preccurlyeq} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot\leq} t)\}$;*
- *if $e = \lambda x : \tau.\,e'$, then $\mathcal{D}$ contains a sub-derivation of $(\Gamma, x : \tau);\Delta \vdash \langle\!\langle e' : \alpha_2 \rangle\!\rangle \rightsquigarrow D'$, and $D = D' \cup \{(\tau \mathrel{\dot\preccurlyeq} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot\leq} t)\}$;*
- *if $e = e_1\ e_2$, then $\mathcal{D}$ contains two sub-derivations of $\Gamma;\Delta \vdash \langle\!\langle e_1 : \alpha \to t \rangle\!\rangle \rightsquigarrow D_1$ and $\Gamma;\Delta \vdash \langle\!\langle e_2 : \alpha \rangle\!\rangle \rightsquigarrow D_2$ (for some α, D_1, and D_2), and $D = D_1 \cup D_2$;*
- *if $e = (e_1, e_2)$, then $\mathcal{D}$ contains two sub-derivations of $\Gamma;\Delta \vdash \langle\!\langle e_1 : \alpha_1 \rangle\!\rangle \rightsquigarrow D_1$ and $\Gamma;\Delta \vdash \langle\!\langle e_2 : \alpha_2 \rangle\!\rangle \rightsquigarrow D_2$ (for some α_1, α_2, D_1, and D_2), and $D = D_1 \cup D_2 \cup \{\alpha_1 \times \alpha_2 \mathrel{\dot\leq} t\}$;*
- *if $e = \pi_i\,e'$, then $\mathcal{D}$ contains a sub-derivation of $\Gamma;\Delta \vdash \langle\!\langle e' : \alpha_1 \times \alpha_2 \rangle\!\rangle \rightsquigarrow D'$, and $D = D' \cup \{\alpha_i \mathrel{\dot\leq} t\}$;*
- *if $e = (\mathsf{let}\ \vec\alpha\ x = e_1\ \mathsf{in}\ e_2)$, then $\mathcal{D}$ contains two sub-derivations of $\Gamma;\Delta \cup \vec\alpha \vdash \langle\!\langle e_1 : \alpha \rangle\!\rangle \rightsquigarrow D_1$ and $(\Gamma, x : \forall\vec\alpha,\vec\beta.\,\alpha\theta_1);\Delta \vdash \langle\!\langle e_2 : t \rangle\!\rangle \rightsquigarrow D_2$, and the following hold:*

$$D = D_2 \cup \mathsf{equiv}(\theta_1, D_1) \qquad \theta_1 \in \mathsf{solve}_{\Delta\cup\vec\alpha}(D_1)$$

$$\vec\alpha \sharp \mathsf{var}(\Gamma\theta_1) \qquad \vec\beta = \mathsf{var}(\alpha\theta_1) \setminus (\mathsf{var}(\Gamma\theta_1) \cup \vec\alpha \cup \mathsf{var}(e_1))$$

Proof. Straightforward, since the constraint simplification rules are syntax-directed. □

Lemma B.93. *If $\Gamma;\Delta \vdash C \rightsquigarrow D$, then $\mathsf{var}(\Gamma) \cap \mathsf{var}(D) \subseteq \mathsf{var}(C) \cup \mathsf{var}_{\dot\preccurlyeq}(D)$.*

Proof. By induction on C (the form of C determines the derivation).

Case: $C = (t_1 \mathrel{\dot\leq} t_2)$ or $C = (\tau \mathrel{\dot\preccurlyeq} \alpha)$ We have $\mathsf{var}(D) \subseteq \mathsf{var}(C)$.

Case: $C = (\tau \mathrel{\dot\preccurlyeq} \alpha)$ We have $\mathsf{var}(D) \subseteq \mathsf{var}_{\dot\preccurlyeq}(D) \cup \{\alpha\}$ and $\alpha \in \mathsf{var}(C)$.

Case: $C = (\mathsf{def}\ x : \tau\ \mathsf{in}\ C')$ By IH, $\mathsf{var}(\Gamma, x : \tau) \cap \mathsf{var}(D) \subseteq \mathsf{var}(C') \cup \mathsf{var}_{\dot\preccurlyeq}(D)$. This directly yields the result since $\mathsf{var}(C') \subseteq \mathsf{var}(C)$.

Case: $C = (\exists \vec{\alpha}.\, C')$ By IH, $\mathsf{var}(\Gamma) \cap \mathsf{var}(D) \subseteq \mathsf{var}(C') \cup \mathsf{var}_{\dot{\preccurlyeq}}(D)$. The side condition on the rule imposes $\vec{\alpha} \sharp \Gamma$. Then, $\mathsf{var}(\Gamma) \cap \mathsf{var}(D) \subseteq \mathsf{var}(C) \cup \mathsf{var}_{\dot{\preccurlyeq}}(D)$ since $\mathsf{var}(C) = \mathsf{var}(C') \setminus \vec{\alpha}$.

Case: $C = (C_1 \wedge C_2)$ By IH, for both i, $\mathsf{var}(\Gamma) \cap \mathsf{var}(D_i) \subseteq \mathsf{var}(C_i) \cup \mathsf{var}_{\dot{\preccurlyeq}}(D_i)$. This directly implies $\mathsf{var}(\Gamma) \cap \mathsf{var}(D_1 \cup D_2) \subseteq \mathsf{var}(C_1 \wedge C_2) \cup \mathsf{var}_{\dot{\preccurlyeq}}(D_1 \cup D_2)$.

Case: $C = (\mathsf{let}\, x \colon \forall \vec{\alpha}; \alpha[C_1]^{\vec{\alpha}_1}.\, \alpha \;\mathsf{in}\; C_2)$ By IH,

$$\mathsf{var}(\Gamma) \cap \mathsf{var}(D_1) \subseteq \mathsf{var}(C_1) \cup \mathsf{var}_{\dot{\preccurlyeq}}(D_1)$$

$$\mathsf{var}(\Gamma, x \colon \forall \vec{\alpha}, \vec{\beta}.\, \alpha\theta_1) \cap \mathsf{var}(D_2) \subseteq \mathsf{var}(C_2) \cup \mathsf{var}_{\dot{\preccurlyeq}}(D_2)$$

We have

$$\begin{aligned} D &= D_2 \cup \mathsf{equiv}(\theta_1, D_1) \\ \mathsf{var}(D) &= \mathsf{var}(D_2) \cup \mathsf{var}(D_1)\theta_1 \cup \mathsf{var}_{\dot{\preccurlyeq}}(D_1) \cup S \cup S\theta_1 \\ \mathsf{var}_{\dot{\preccurlyeq}}(D) &= \mathsf{var}_{\dot{\preccurlyeq}}(D_2) \cup \mathsf{var}(D_1)\theta_1 \cup \mathsf{var}_{\dot{\preccurlyeq}}(D_1) \\ \mathsf{var}(C) &= (\mathsf{var}(C_1) \setminus (\vec{\alpha} \cup \{\alpha\})) \cup \mathsf{var}(C_2) \end{aligned}$$

where $S = \{\, \alpha \in \mathsf{dom}(\theta_1) \mid \alpha\theta_1 \text{ static} \,\}$.

Consider an arbitrary $\beta \in \mathsf{var}(\Gamma) \cap \mathsf{var}(D)$.

Case: $\beta \in \mathsf{var}(D_2)$ Then $\beta \in \mathsf{var}(C_2) \cup \mathsf{var}_{\dot{\preccurlyeq}}(D_2)$ and hence $\beta \in \mathsf{var}(C) \cup \mathsf{var}_{\dot{\preccurlyeq}}(D)$.

Case: $\beta \in \mathsf{var}(D_1)\theta_1 \cup \mathsf{var}_{\dot{\preccurlyeq}}(D_1)$ Then $\beta \in \mathsf{var}_{\dot{\preccurlyeq}}(D)$.

Case: $\beta \in S$

Then $\beta \in \mathsf{dom}(\theta_1)$. By Proposition B.91, $\beta \in \mathsf{var}(D_1)$.

Since $\beta \in \mathsf{var}(\Gamma) \cap \mathsf{var}(D_1)$, we have $\beta \in \mathsf{var}(C_1) \cup \mathsf{var}_{\dot{\preccurlyeq}}(D_1)$. Since $\beta \in \mathsf{var}(\Gamma)$, by the side conditions of the rule we know $\beta \neq \alpha$ and $\beta \notin \vec{\alpha}$. Therefore, $\beta \in \mathsf{var}(C) \cup \mathsf{var}_{\dot{\preccurlyeq}}(D)$.

Case: $\beta \in S\theta_1$

Then $\beta \in \mathsf{var}(\gamma\theta_1)$ for some $\gamma \in \mathsf{dom}(\theta_1)$ such that $\gamma\theta_1$ is static.

By Proposition B.91, $\gamma \in \mathsf{var}(D_1)$. Then $\beta \in \mathsf{var}(D_1)\theta_1 \subseteq \mathsf{var}_{\dot{\preccurlyeq}}(D)$. □

Lemma B.94. *Let θ and θ' be two type substitutions such that $\theta \Vdash_\Delta D$ and $\mathsf{static}(\theta', \mathsf{var}(D)\theta)$. If $(t_1 \mathrel{\dot{\leq}} t_2) \in D$, then $t_1\theta\theta' \leq t_2\theta\theta'$.*

Proof. By definition of $\theta \Vdash_\Delta D$, we have $t_1\theta \leq t_2\theta$. Since $\mathsf{var}(t_1) \cup \mathsf{var}(t_2) \subseteq \mathsf{var}(D)$, we have $\mathsf{var}(t_1\theta) \cup \mathsf{var}(t_2\theta) \subseteq \mathsf{var}(D)\theta$. Because $\mathsf{static}(\theta', \mathsf{var}(D)\theta)$, the restriction of θ' to $\mathsf{var}(t_1\theta) \cup \mathsf{var}(t_2\theta)$ is a static substitution. By Proposition B.32, $t_1\theta\theta' \leq t_2\theta\theta'$. □

Lemma B.95. *Let θ and θ' be two type substitutions such that $\theta \Vdash_\Delta D$ and $\mathsf{static}(\theta', \mathsf{var}(D)\theta)$. If $(\tau \mathrel{\dot{\preccurlyeq}} \alpha) \in D$, then $\tau\theta\theta' \preccurlyeq \alpha\theta\theta'$.*

Proof. By definition of $\theta \Vdash_\Delta D$, we have $\tau\theta \preccurlyeq \alpha\theta$. Then, $\tau\theta\theta' \preccurlyeq \alpha\theta\theta'$ follows by Proposition B.36. □

Lemma B.96.

$$\forall \Gamma, \Delta, e, \alpha, D, \theta. \left.\begin{array}{l} \Gamma; \Delta \vdash \langle\!\langle e \colon \alpha \rangle\!\rangle \rightsquigarrow D \\ \theta \in \mathsf{solve}_\Delta(D) \\ \mathsf{var}(e) \subseteq \Delta \\ \alpha \notin \mathsf{var}(\Gamma) \end{array}\right\} \implies \mathsf{static}(\theta, \mathsf{var}(\Gamma))$$

Proof. Consider an arbitrary $\beta \in \mathsf{var}(\Gamma)$. We show that $\beta\theta$ is static.

Case: $\beta \notin \mathsf{dom}(\theta)$ Then $\beta\theta = \beta$, which is static.

Case: $\beta \in \mathrm{dom}(\theta)$ Then $\beta \in \mathrm{var}(D)$ (by Proposition B.91), and therefore $\beta \in \mathrm{var}(\Gamma) \cap \mathrm{var}(D)$.
By Lemma B.93, $\beta \in \mathrm{var}(\langle\!\langle e : \alpha \rangle\!\rangle) \cup \mathrm{var}_{\dot{\preccurlyeq}}(D)$.
Case: $\beta \in \mathrm{var}(\langle\!\langle e : \alpha \rangle\!\rangle)$ This case is impossible because $\mathrm{var}(\langle\!\langle e : \alpha \rangle\!\rangle) = \mathrm{var}(e) \cup \{\alpha\}$, $\mathrm{dom}(\theta) \nsharp \mathrm{var}(e)$ (because $\mathrm{var}(e) \subseteq \Delta$), and $\alpha \notin \mathrm{var}(\Gamma)$.
Case: $\beta \in \mathrm{var}_{\dot{\preccurlyeq}}(D)$ Since $\theta \Vdash_\Delta D$, $\beta\theta$ must be static. □

Lemma B.97.

$$\forall \Gamma, \Delta, D_1, \theta_1, \rho, \theta, \theta'. \left.\begin{array}{l} \theta \Vdash_\Delta \mathrm{equiv}(\theta_1, D_1) \\ \mathrm{dom}(\rho) \nsharp \Gamma\theta_1 \\ \mathrm{static}(\theta', \mathrm{var}(\mathrm{equiv}(\theta_1, D_1))\theta) \\ \mathrm{static}(\theta_1, \mathrm{var}(\Gamma)) \end{array}\right\} \Longrightarrow \Gamma\theta\theta' \leq \Gamma\theta_1\rho\theta\theta'$$

Proof. Consider an arbitrary $x \in \mathrm{dom}(\Gamma)$. We have $\Gamma(x) = \forall\vec{\alpha}.\, \tau$. We assume by α-renaming that $\vec{\alpha} \nsharp \theta_1, \rho, \theta, \theta'$; then, $(\Gamma\theta\theta')(x) = \forall\vec{\alpha}.\, \tau\theta\theta'$ and $(\Gamma\theta_1\rho\theta\theta')(x) = \forall\vec{\alpha}.\, \tau\theta_1\rho\theta\theta'$. We must show $\tau\theta\theta' \leq \tau\theta_1\rho\theta\theta'$. We show $\forall\alpha \in \mathrm{var}(\tau).\ \alpha\theta\theta' \simeq \alpha\theta_1\rho\theta\theta'$, which implies $\tau\theta\theta' \simeq \tau\theta_1\rho\theta\theta'$ by Lemma B.37.

To show $\forall\alpha \in \mathrm{var}(\tau).\ \alpha\theta\theta' \simeq \alpha\theta_1\rho\theta\theta'$, consider an arbitrary $\alpha \in \mathrm{var}(\tau)$.

Case: $\alpha \in \vec{\alpha}$ Then (by our choice of naming) $\alpha\theta\theta' = \alpha$ and $\alpha\theta_1\rho\theta\theta' = \alpha$.
Case: $\alpha \notin \vec{\alpha}$ Then $\alpha \in \mathrm{var}(\Gamma)$ and hence: $\mathrm{var}(\alpha\theta_1) \subseteq \mathrm{var}(\Gamma\theta_1)$, and $\alpha\theta_1\rho = \alpha\theta_1$, and $\alpha\theta_1$ is static.
Case: $\alpha \notin \mathrm{dom}(\theta_1)$ Then $\alpha\theta_1 = \alpha$, $\alpha\theta_1\rho = \alpha$, and $\alpha\theta_1\rho\theta\theta' = \alpha\theta\theta'$.
Case: $\alpha \in \mathrm{dom}(\theta_1)$
Then $\{(\alpha \mathrel{\dot{\leq}} \alpha\theta_1), (\alpha\theta_1 \mathrel{\dot{\leq}} \alpha)\} \subseteq \mathrm{equiv}(\theta_1, D_1)$. Therefore, we have $\alpha\theta_1\theta \simeq \alpha\theta$ and $\mathrm{static}(\theta', \mathrm{var}(\alpha\theta) \cup \mathrm{var}(\alpha\theta_1\theta))$. By Proposition B.32, $\alpha\theta_1\theta\theta' \simeq \alpha\theta\theta'$. □

Theorem B.98. *Let $\mathcal{D}$ be a derivation of $\Gamma; \mathrm{var}(e) \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D$. Let θ be a type substitution such that $\theta \Vdash_{\mathrm{var}(e)} D$. Then, we have $\Gamma\theta \vdash e \rightsquigarrow (\!|e|\!)^{\mathcal{D}}_{\theta} : t\theta$.*

Proof. We show the following, stronger result (for all $\mathcal{D}$, Γ, Δ, e, t, D, θ, and θ'):

$$\left.\begin{array}{l} \mathcal{D} \text{ is a derivation of } \Gamma; \Delta \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D \\ \theta \Vdash_\Delta D \\ \mathrm{static}(\theta', \mathrm{var}(D)\theta) \\ \mathrm{var}(e) \subseteq \Delta \end{array}\right\} \Longrightarrow \Gamma\theta\theta' \vdash e\theta' \rightsquigarrow (\!|e|\!)^{\mathcal{D}}_{\theta}\theta' : t\theta\theta'$$

This result implies the statement: we take $\Delta = \mathrm{var}(e)$ and $\theta' = \{\}$ (the identity substitution).
The proof is by structural induction on e.

Case: $e = x$

(1) $\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle x : t \rangle\!\rangle \rightsquigarrow D$ Given
(2) $\theta \Vdash_\Delta D$ Given
(3) $\mathrm{static}(\theta', \mathrm{var}(D)\theta)$ Given

By Lemma B.92 from (1):

$\Gamma(x) = \forall\vec{\alpha}.\, \tau$

$D = \{(\tau\{\vec{\alpha} := \vec{\beta}\} \mathrel{\dot{\preccurlyeq}} \alpha), (\alpha \mathrel{\dot{\leq}} t)\}$

Then:

$(\Gamma\theta\theta')(x) = \forall\vec{\alpha}.\, \tau\theta\theta'$	assuming $\vec{\alpha} \sharp \theta, \theta'$ by α-renaming
the types $\vec{\beta}\theta\theta'$ are static	by (2) and (3)
$\forall\alpha \in \mathsf{var}(\tau).\ \alpha\theta\theta'\{\vec{\alpha} \coloneqq \vec{\beta}\theta\theta'\} = \alpha\{\vec{\alpha} \coloneqq \vec{\beta}\}\theta\theta'$	since $\vec{\alpha} \sharp \theta, \theta'$
$\tau\theta\theta'\{\vec{\alpha} \coloneqq \vec{\beta}\theta\theta'\} = \tau\{\vec{\alpha} \coloneqq \vec{\beta}\}\theta\theta'$	
$\tau\{\vec{\alpha} \coloneqq \vec{\beta}\}\theta\theta' \preccurlyeq \alpha\theta\theta'$	by Lemma B.95
$\alpha\theta\theta' \leq t\theta\theta'$	by Lemma B.94
$\Gamma\theta\theta' \vdash x \rightsquigarrow x\,[\vec{\beta}\theta\theta']\colon \tau\theta\theta'\{\vec{\alpha} \coloneqq \vec{\beta}\theta\theta'\}$	by [Var]
$\Gamma\theta\theta' \vdash x \rightsquigarrow x\,[\vec{\beta}\theta\theta']\langle\tau\{\vec{\alpha} \coloneqq \vec{\beta}\}\theta\theta' \overset{\ell}{\Rightarrow} \alpha\theta\theta'\rangle\colon t\theta\theta'$	by [Materialize] and [Subsume]

This concludes this case since $(\!|x|\!)^{\mathcal{D}}_{\theta}\theta' = x\,[\vec{\beta}\theta\theta']\langle\tau\{\vec{\alpha} \coloneqq \vec{\beta}\}\theta\theta' \overset{\ell}{\Rightarrow} \alpha\theta\theta'\rangle$.

Case: $e = c$

$\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle c\colon t\rangle\!\rangle \rightsquigarrow D$	Given
$D = \{b_c \mathrel{\dot{\leq}} t\}$	by Lemma B.92
$b_c\theta\theta' \leq t\theta\theta'$	by Lemma B.94
$\Gamma\theta\theta' \vdash c\theta\theta' \rightsquigarrow c\colon t\theta\theta'$	by [Const] and [Subsume]
$(\!|c|\!)^{\mathcal{D}}_{\theta}\theta' = c$	

Case: $e = \lambda x.\, e'$

$\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle \lambda x.\, e'\colon t\rangle\!\rangle \rightsquigarrow D$ Given

By Lemma B.92:

$\mathcal{D}' :: (\Gamma, x\colon \alpha_1); \Delta \vdash \langle\!\langle e'\colon \alpha_2\rangle\!\rangle \rightsquigarrow D'$

$D = D' \cup \{(\alpha_1 \mathrel{\dot{\preccurlyeq}} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\}$

Then:

$\alpha_1\theta\theta'$ is static	
$(\alpha_1 \to \alpha_2)\theta\theta' \leq t\theta\theta'$	by Lemma B.94
$\Gamma\theta\theta', x\colon \alpha_1\theta\theta' \vdash e'\theta\theta' \rightsquigarrow (\!|e'|\!)^{\mathcal{D}'}_{\theta}\theta'\colon \alpha_2\theta\theta'$	by IH
$\Gamma\theta\theta' \vdash (\lambda x.\, e'\theta\theta') \rightsquigarrow \lambda^{(\alpha_1\to\alpha_2)\theta\theta'}x.\, (\!|e'|\!)^{\mathcal{D}'}_{\theta}\theta'\colon (\alpha_1 \to \alpha_2)\theta\theta'$	by [Abstr]
$\Gamma\theta\theta' \vdash (\lambda x.\, e'\theta\theta') \rightsquigarrow \lambda^{(\alpha_1\to\alpha_2)\theta\theta'}x.\, (\!|e'|\!)^{\mathcal{D}'}_{\theta}\theta'\colon t\theta\theta'$	by [Subsume]
$(\!|\lambda x.\, e|\!)^{\mathcal{D}}_{\theta}\theta' = \lambda^{(\alpha_1\to\alpha_2)\theta\theta'}x.\, (\!|e'|\!)^{\mathcal{D}'}_{\theta}\theta'$	

Case: $e = \lambda x\colon \tau.\, e'$

$\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle \lambda x\colon \tau.\, e'\colon t\rangle\!\rangle \rightsquigarrow D$ Given

By Lemma B.92:

$\mathcal{D}' :: (\Gamma, x\colon \tau); \Delta \vdash \langle\!\langle e'\colon \alpha_2\rangle\!\rangle \rightsquigarrow D'$

$D = D' \cup \{(\tau \mathrel{\dot{\preccurlyeq}} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\}$

Then:

$\tau\theta\theta' \preccurlyeq \alpha_1\theta\theta'$ by Lemma B.95

$(\alpha_1 \to \alpha_2)\theta\theta' \leq t\theta\theta'$ by Lemma B.94

$\Gamma\theta\theta', x\colon \tau\theta\theta' \vdash e'\theta\theta' \rightsquigarrow (\!|e'|\!)^{\mathcal{D}'}_{\theta}\theta'\colon \alpha_2\theta\theta'$ by IH

$\Gamma\theta\theta' \vdash (\lambda x\colon \tau.\, e')\theta\theta' \rightsquigarrow \lambda^{(\tau\to\alpha_2)\theta\theta'} x.\, (\!|e'|\!)^{\mathcal{D}'}_{\theta}\theta'\colon (\tau \to \alpha_2)\theta\theta'$ by [AAbstr]

$\Gamma\theta\theta' \vdash (\lambda x\colon \tau.\, e')\theta\theta' \rightsquigarrow$
$\left(\lambda^{(\tau\to\alpha_2)\theta\theta'} x.\, (\!|e'|\!)^{\mathcal{D}'}_{\theta}\theta'\right)\langle(\tau \to \alpha_2)\theta\theta' \overset{\ell}{\Rightarrow} (\alpha_1 \to \alpha_2)\theta\theta'\rangle\colon (\alpha_1 \to \alpha_2)\theta\theta'$ by [Materialize]

$\Gamma\theta\theta' \vdash (\lambda x\colon \tau.\, e')\theta\theta' \rightsquigarrow$
$\left(\lambda^{(\tau\to\alpha_2)\theta\theta'} x.\, (\!|e'|\!)^{\mathcal{D}'}_{\theta}\theta'\right)\langle(\tau \to \alpha_2)\theta\theta' \overset{\ell}{\Rightarrow} (\alpha_1 \to \alpha_2)\theta\theta'\rangle\colon t\theta\theta'$ by [Subsume]

$(\!|\lambda x\colon \tau.\, e|\!)^{\mathcal{D}}_{\theta}\theta' =$
$\quad\left(\lambda^{(\tau\to\alpha_2)\theta\theta'} x.\, (\!|e'|\!)^{\mathcal{D}'}_{\theta}\theta'\right)\langle(\tau \to \alpha_2)\theta\theta' \overset{\ell}{\Rightarrow} (\alpha_1 \to \alpha_2)\theta\theta'\rangle$

Case: $e = e_1\ e_2$

$\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle e_1\ e_2\colon t\rangle\!\rangle \rightsquigarrow D$ Given

By Lemma B.92:

$\mathcal{D}_1 :: \Gamma; \Delta \vdash \langle\!\langle e_1\colon \alpha \to t\rangle\!\rangle \rightsquigarrow D_1$
$\mathcal{D}_2 :: \Gamma; \Delta \vdash \langle\!\langle e_2\colon \alpha\rangle\!\rangle \rightsquigarrow D_2$
$D = D_1 \cup D_2$

Then:

$\Gamma\theta\theta' \vdash e_1\theta\theta' \rightsquigarrow (\!|e_1|\!)^{\mathcal{D}_1}_{\theta}\theta'\colon (\alpha \to t)\theta\theta'$ by IH

$\Gamma\theta\theta' \vdash e_2\theta\theta' \rightsquigarrow (\!|e_2|\!)^{\mathcal{D}_2}_{\theta}\theta'\colon \alpha\theta\theta'$ by IH

$\Gamma\theta\theta' \vdash (e_1\ e_2)\theta\theta' \rightsquigarrow (\!|e_1|\!)^{\mathcal{D}_1}_{\theta}\theta'\ (\!|e_2|\!)^{\mathcal{D}_2}_{\theta}\theta'\colon t\theta\theta'$ by [Appl]

$(\!|e_1\ e_2|\!)^{\mathcal{D}}_{\theta}\theta' = (\!|e_1|\!)^{\mathcal{D}_1}_{\theta}\theta'\ (\!|e_2|\!)^{\mathcal{D}_2}_{\theta}\theta'$

Case: $e = (e_1, e_2)$

$\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle (e_1, e_2)\colon t\rangle\!\rangle \rightsquigarrow D$ Given

By Lemma B.92:

$\mathcal{D}_1 :: \Gamma; \Delta \vdash \langle\!\langle e_1\colon \alpha_1\rangle\!\rangle \rightsquigarrow D_1$
$\mathcal{D}_2 :: \Gamma; \Delta \vdash \langle\!\langle e_2\colon \alpha_2\rangle\!\rangle \rightsquigarrow D_2$
$D = D_1 \cup D_2 \cup \{\alpha_1 \times \alpha_2 \mathrel{\dot{\leq}} t\}$

Then:

$(\alpha_1 \times \alpha_2)\theta\theta' \leq t\theta\theta'$ by Lemma B.94

$\Gamma\theta\theta' \vdash e_1\theta\theta' \rightsquigarrow (\!|e_1|\!)^{\mathcal{D}_1}_{\theta}\theta' : \alpha_1\theta\theta'$ by IH

$\Gamma\theta\theta' \vdash e_2\theta\theta' \rightsquigarrow (\!|e_2|\!)^{\mathcal{D}_2}_{\theta}\theta' : \alpha_2\theta\theta'$ by IH

$\Gamma\theta\theta' \vdash (e_1, e_2)\theta\theta' \rightsquigarrow \left((\!|e_1|\!)^{\mathcal{D}_1}_{\theta}\theta', (\!|e_2|\!)^{\mathcal{D}_2}_{\theta}\theta'\right) : t\theta\theta'$ by [Pair] and [Subsume]

$(\!|(e_1, e_2)|\!)^{\mathcal{D}}_{\theta}\theta' = \left((\!|e_1|\!)^{\mathcal{D}_1}_{\theta}\theta', (\!|e_2|\!)^{\mathcal{D}_2}_{\theta}\theta'\right)$

Case: $e = \pi_i\ e'$

$\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle \pi_i\ e' : t \rangle\!\rangle \rightsquigarrow D$ Given

By Lemma B.92:

$\mathcal{D}' :: \Gamma; \Delta \vdash \langle\!\langle e' : \alpha_1 \times \alpha_2 \rangle\!\rangle \rightsquigarrow D'$

$D = D' \cup \{\alpha_i \dot{\leq} t\}$

Then:

$\alpha_i\theta\theta' \leq t\theta\theta'$ by Lemma B.94

$\Gamma\theta\theta' \vdash e'\theta\theta' \rightsquigarrow (\!|e'|\!)^{\mathcal{D}'}_{\theta}\theta' : (\alpha_1 \times \alpha_2)\theta\theta'$ by IH

$\Gamma\theta\theta' \vdash (\pi_i\ e')\theta\theta' \rightsquigarrow \pi_i\ ((\!|e'|\!)^{\mathcal{D}'}_{\theta}\theta') : t\theta\theta'$ by [Proj] and [Subsume]

$(\!|\pi_i\ e'|\!)^{\mathcal{D}}_{\theta}\theta' = (\pi_i\ (\!|e'|\!)^{\mathcal{D}'}_{\theta})\theta'$

Case: $e = (\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2)$

$\mathcal{D} :: \Gamma; \Delta \vdash \langle\!\langle \mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2 : t \rangle\!\rangle \rightsquigarrow D$ Given

By Lemma B.92:

$\mathcal{D}_1 :: \Gamma; \Delta \cup \vec{\alpha} \vdash \langle\!\langle e_1 : \alpha \rangle\!\rangle \rightsquigarrow D_1$

$\mathcal{D}_2 :: (\Gamma, x : \forall\vec{\alpha},\vec{\beta}.\ \alpha\theta_1); \Delta \vdash \langle\!\langle e_2 : t \rangle\!\rangle \rightsquigarrow D_2$

$D = D_2 \cup \mathsf{equiv}(\theta_1, D_1)$

$\theta_1 \in \mathsf{solve}_{\Delta\cup\vec{\alpha}}(D_1)$

$\vec{\alpha} \sharp \mathsf{var}(\Gamma\theta_1)$

$\vec{\beta} = \mathsf{var}(\alpha\theta_1) \setminus (\mathsf{var}(\Gamma\theta_1) \cup \vec{\alpha} \cup \mathsf{var}(e_1))$

Let $\vec{\alpha}_1$ and $\vec{\beta}_1$ be vectors of distinct variables chosen outside $\mathsf{var}(e_1)$, $\mathsf{dom}(\theta)$, $\mathsf{var}(\theta)$, $\mathsf{dom}(\theta')$, and $\mathsf{var}(\theta')$. Let $\rho = \{\vec{\alpha} := \vec{\alpha}_1\} \cup \{\vec{\beta} := \vec{\beta}_1\}$. Then:

$e\theta' = (\mathsf{let}\ \vec{\alpha}_1\ x = e_1\rho\theta'\ \mathsf{in}\ e_2\theta')$ since $\vec{\beta} \sharp e_1$ and $\vec{\alpha}_1 \sharp \theta'$

$(\!|e|\!)^{\mathcal{D}}_{\theta} = \left(\mathsf{let}\ x = (\Lambda\vec{\alpha}_1,\vec{\beta}_1.\ (\!|e_1|\!)^{\mathcal{D}_1}_{\theta_1}\rho\theta)\ \mathsf{in}\ (\!|e_2|\!)^{\mathcal{D}_2}_{\theta}\right)$

$(\!|e|\!)^{\mathcal{D}}_{\theta}\theta' = \left(\mathsf{let}\ x = (\Lambda\vec{\alpha}_1,\vec{\beta}_1.\ (\!|e_1|\!)^{\mathcal{D}_1}_{\theta_1}\rho\theta\theta')\ \mathsf{in}\ (\!|e_2|\!)^{\mathcal{D}_2}_{\theta}\theta'\right)$ since $\vec{\alpha}_1,\vec{\beta}_1 \sharp \theta'$

For e_1:

$\theta_1 \Vdash_{\Delta \cup \vec{\alpha}} D_1$	
$\mathsf{static}(\rho\theta\theta', \mathsf{var}(D_1)\theta_1)$	proven below
$\mathsf{var}(e_1) \subseteq \Delta \cup \vec{\alpha}$	
$\Gamma\theta_1\rho\theta\theta' \vdash e_1\rho\theta\theta' \rightsquigarrow (\!\lvert e_1 \rvert\!)^{\mathcal{D}_1}_{\theta_1}\rho\theta\theta' : \alpha\theta_1\rho\theta\theta'$	by IH
$e_1\rho\theta\theta' = e_1\rho\theta'$	since $\mathsf{dom}(\theta) \cap \mathsf{var}(e_1\rho) = \varnothing$
$\alpha \notin \mathsf{var}(\Gamma)$	by inversion
$\mathsf{static}(\theta_1, \mathsf{var}(\Gamma))$	by Lemma B.96
$\Gamma\theta\theta' \leq \Gamma\theta_1\rho\theta\theta'$	by Lemma B.97
$\Gamma\theta\theta' \vdash e_1\rho\theta' \rightsquigarrow (\!\lvert e_1 \rvert\!)^{\mathcal{D}_1}_{\theta_1}\rho\theta\theta' : \alpha\theta_1\rho\theta\theta'$	by Lemma B.89

For e_2:

$\theta \Vdash_{\Delta} D_2$	
$\mathsf{static}(\theta', \mathsf{var}(D_2)\theta)$	
$\mathsf{var}(e_2) \subseteq \Delta$	
$\Gamma\theta\theta', x : (\forall\vec{\alpha},\vec{\beta}.\, \alpha\theta_1)\theta\theta' \vdash e_2\theta' \rightsquigarrow (\!\lvert e_2 \rvert\!)^{\mathcal{D}_2}_{\theta}\theta' : t\theta\theta'$	by IH
$(\forall\vec{\alpha},\vec{\beta}.\, \alpha\theta_1)\theta\theta' = (\forall\vec{\alpha}_1,\vec{\beta}_1.\, \alpha\theta_1\rho\theta\theta')$	since $\vec{\alpha}_1,\vec{\beta}_1 \sharp \theta, \theta'$
$\Gamma\theta\theta', x : (\forall\vec{\alpha}_1,\vec{\beta}_1.\, \alpha\theta_1\rho\theta\theta') \vdash e_2\theta' \rightsquigarrow (\!\lvert e_2 \rvert\!)^{\mathcal{D}_2}_{\theta}\theta' : t\theta\theta'$	
$\vec{\alpha}_1,\vec{\beta}_1 \sharp \Gamma\theta\theta'$ and $\vec{\beta}_1 \sharp e_1\rho\theta'$	

Finally:

$$\Gamma\theta\theta' \vdash e\theta' \rightsquigarrow (\!\lvert e \rvert\!)^{\mathcal{D}}_{\theta}\theta' : t\theta\theta' \qquad \text{by [Let]}$$

To check $\mathsf{static}(\rho\theta\theta', \mathsf{var}(D_1)\theta_1)$, take an arbitrary $\alpha \in \mathsf{var}(D_1)\theta_1$.

- If $\alpha \in \mathsf{dom}(\rho)$, then $\alpha\rho$ is a variable in $\vec{\alpha}_1,\vec{\beta}_1$ and $\alpha\rho = \alpha\rho\theta\theta'$ (because $\vec{\alpha}_1,\vec{\beta}_1 \sharp \theta, \theta'$): hence $\alpha\rho\theta\theta'$ is static.
- If $\alpha \notin \mathsf{dom}(\rho)$, then $\alpha\rho\theta\theta' = \alpha\theta\theta'$. We have $(\alpha \mathrel{\dot{\preccurlyeq}} \alpha) \in \mathsf{equiv}(\theta_1, D_1)$. Since $\mathsf{equiv}(\theta_1, D_1) \subseteq D$, $\alpha\theta$ is static. Furthermore, $\mathsf{var}(\alpha\theta) \subseteq \mathsf{var}(D)\theta$; hence, $\alpha\theta\theta'$ is static too. □