Gradual Typing with Union and Intersection Types

GIUSEPPE CASTAGNA, CNRS - Université Paris Diderot
VICTOR LANVIN, École Normale Supérieure de Cachan

We propose a type system for functional languages with gradual types and set-theoretic type connectives and prove its soundness. In particular, we show how to lift the definition of the domain and result type of an application from non-gradual types to gradual ones and likewise for the subtyping relation. We also show that deciding subtyping for gradual types can be reduced in linear time to deciding subtyping for non-gradual types and that the same holds true for all subtyping-related decision problems that must be solved for type inference. More generally, this work not only enriches gradual type systems with unions and intersections and with the type precision that arise from their use, but also proposes and advocates a new style of gradual types programming where union and intersection types are used by programmers to instruct the system to perform fewer dynamic checks.

CCS Concepts: • **Theory of computation → Type theory**; • **Software and its engineering → Functional languages**;

Additional Key Words and Phrases: Gradual typing, set-theoretic types, union types, intersection types, negation types

1 INTRODUCTION

A static type system can be an extremely powerful tool for a programmer, providing early error detection, and offering strong compile-time guarantees on the behavior of a program. However, compared to dynamic typing, static typing often comes at the expense of development speed and flexibility, as statically-typed code might be more difficult to adapt to changing requirements. Gradual typing is a recent and promising approach that tries to get the best of both worlds [Siek and Taha 2006]. The idea behind this approach is to integrate an *unknown* type, usually denoted by "**?**", which informs the compiler that additional type checks may have to be performed at run time. Therefore, the programmer can *gradually* add type annotations to a program and controls precisely how much checking is done statically versus dynamically. Gradual typing thus allows the programmer to finely tune the distribution of dynamic and static checking over a program. However, gradualization of single expressions has more limited breadth. We argue that adding full-fledged union and intersection types to a gradual type system makes the transition between dynamic typing and static typing smoother and finer grained, giving even more control to the programmer. In particular, we are interested in developing gradual typing for the *semantic subtyping* approach [Frisch et al. 2008], where types are interpreted as sets of values. In this approach union and intersection types are naturally interpreted as the corresponding set-theoretic operations, and the subtyping relation is defined as set-containment, whence the name of *set-theoretic types*. This yields an intuitive and powerful type system in which several important constructions — *eg*, branching, pattern-matching, and overloading— can be typed very precisely. Set-theoretic types, however, exacerbate the shortcomings of static typing. In particular, type reconstruction for intersection type systems is, in general, undecidable. The consequence is that programmers have to add complete type annotations for every variable, which may hinder their development speed; all the more so given that union and intersection type annotations can be syntactically heavy. Adding

2017. 2475-1421/2017/1-ART1 $15.00
https://doi.org/

gradual typing to set-theoretic types may help to alleviate this issue by providing a way to relax the rigidity of certain type annotations via the addition of a touch of dynamic typing, while keeping the full power of static types for critical parts of code.

We said that adding set-theoretic types to a gradual type system makes the transition between dynamic typing and static typing smoother. This is for example the case for function parameters that are to be bound to values of basic types: in the current setting, the only way to gradualize their type is to use "?", while with union and intersection types more precise gradualizations become possible. We illustrate this fact in an ML-like language by progressively refining the following example that we borrow from Siek and Vachharajani [2008].

```
let succ : Int -> Int = ...
let not : Bool -> Bool = ...

let f (condition : Bool) (x : ?) : ? =
  if condition then
    succ x
  else
    not x
```

This example cannot be typed using only simple types: the type of `x` as well as the return type of `f` change depending on the value of `condition`. However, this piece of code is perfectly valid in a gradual type system, the compiler will simply add dynamic checks to ensure that the value bound to `x` can be passed as an argument to `succ` or `not` according to the case. Moreover, it will also add checks to ensure that the value returned by `f` is used correctly. Nevertheless, there are some flaws in this piece of code. For example, it is possible to pass a value of any type as the second argument of `f` (the type system ensures that the first argument will always be a Boolean). In particular, if one applies the function `f` to (a Boolean and) a value of type `string`, then the application will always fail, independently from the value of `condition`, and despite the fact that the application is statically considered well-typed. This problem can be avoided by set-theoretic types, in particular by using the union type `Int | Bool` to type the parameter `x` of the function so as to ensure that every second argument of `f` that is neither an integer nor a Boolean will be statically rejected by the type checker. This is obtained by the following code

```
let f (condition : Bool) (x : (Int | Bool)) : (Int | Bool) =
  if condition then
    succ ((Int) x)
  else
    not ((Bool) x)
```

The code above shows that the use of union types fixes the shortcoming we pointed out. However in order to ensure that the applications of `succ` and `not` are both well typed we also had to add two *type casts*[1] that check at run-time whether the argument has the required type and raise an exception otherwise.

Remark. *The use of explicit type-casts should not surprise the reader. Any language with union types needs an operation to dynamically check the type of values that are given union types. Such operations may have different forms and be implemented in different ways and at different degrees: they range from the simple check of constructors/tags (*eg*, when unions are restricted*

[1] Although we used a C-like syntax these type casts do not perform any type conversion or promotion: they are just dynamic type-checks.

*to datatypes/variant-types), to value inspection primitives such as those of Typed Racket [Tobin-Hochstadt and Felleisen 2008] (*eg, `procedure?`, `number?`, *etc.), till the unconstrained check of the type of a value returned by an expression e, that is,*

$$\texttt{match } e \texttt{ with } t_1 \texttt{ -> } e_1 \texttt{ | .... | } t_n \texttt{ -> } e_n$$

which is used in languages like CDuce [Benzaken et al. 2003] where unions are full fledged.[2] *This article studies the latter case and therefore a type cast* `(t)e` *is just syntactic sugar for*[3]

$$\texttt{match } e \texttt{ with } t \texttt{ -> } e \texttt{ | \_ -> raise "Cast error".}$$

□

In this second definition of `f` we have ensured, thanks to union types, that every application of `f` has now a chance to succeed. However, this is obtained at the expenses of the programmer who has now the burden to insert in the code the type-cases/type-casts necessary to ensure safety (in the sense established by Wright and Felleisen [1994]). By using set-theoretic types in conjunction with gradual types, it is possible both to ensure that `f` will only be applied to booleans or integers *and* to delegate the insertion of type casts to the system. This is shown by the following piece of code that our system will compile into the previous one.

```
let f (condition : Bool) (x : (Int | Bool) & ?) : (Int | Bool) =
  if condition then
    succ x
  else
    not x
```

In this example, the variable `x` is of type `((Int | Bool) & ?)`, where "`&`" denotes an intersection. This indicates that `x` has both type `(Int | Bool)` *and* type `?`. Intuitively, this means that the function `f` accepts as a second argument a value of any type (which is indicated by `?`), as long as this value is also an integer *or* a Boolean. The effect of having added "`& ?`" in the type of the parameter is that the programmer is no longer required to add the explicit casts in the body of the function: the system will take care of it at compile time. The combination 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 the *dynamic* checking, where/if necessary, for each case in the union; and while adding explicit casts in a five-line example such as the above is quite straightforward, in general (*eg*, in thousand-line modules), it is not always so, whence the interest of having a system that adds all *and only* the casts that are necessary to ensure type safety. Finally, note that the return type of `f` is no longer gradual (as it was in the first definition), since union types allow us to define it without any loss of precision. This allows the system to statically reject all cases in which the value expected from `f` is neither an integer nor a Boolean and which, with the first definition, would be detected only at run-time.

In all the examples above the return type of the function `f` can be easily and automatically deduced and could, therefore, be omitted. But there are cases in which providing the return type

[2]The cost of run-time "deep" type-cases is frequently misunderstood and overestimated. This is occasionally expressed by the belief that having only a restricted set of type inspection primitives would greatly improve runtime performance in terms of time and/or space. Frisch [2004] shows that since most of the values already encode their type information, then *in strict languages* the overhead of deep dynamic type-cases can be made small: lambda-abstractions must keep their type annotations at runtime but no other subexpression needs to be type-decorated (though in some languages one might need to differentiate constants of different types, e.g. booleans from integers in OCaml). Unconstrained dynamic type-check can then be implemented very efficiently by using the static type information available at compile time (Frisch [2004] proves that it is possible to implement it optimally for every given exploration strategy for values): in practice, it often turns out to be a check of the topmost constructor of a value.

[3]A proper definition is `match e with t & x -> x | _ -> raise "Cast error"` where *e* is evaluated just once.

of a function allows the system to deduce a better type and, thus, accept more programs. This is particularly true in conjunction with intersection types, since they allow the programmer to specify different return types for different argument configurations. Consider again the function `f` above. Since it always returns either an integer or a Boolean, we used as return type `(Int | Bool)`. By using an intersection type it is possible to give `f` a more precise type in which the return type of `f` depends on the type of `x`:

```
let f : (Bool -> (Int & ?) -> Int)  &  (Bool -> (Bool & ?) -> Bool) =
    λcondition. λx.
     if condition then
       succ x
     else
       not x
```

This time, in the body of `f`, the variable `x` has type `(Int & ?) | (Bool & ?)`. This type is equivalent[4] to `((Int | Bool) & ?)`. Hence, the function can be defined with the same body as before, and it accepts as arguments the same values. However, the return type of `f` now directly depends on the type of `x` (more precisely, on the type of the value bound to `x`): if it is of type `Int`, then the function necessarily returns an integer (that is, if the application does not fail), and the same goes for an argument of type `Bool`.

Having a return type that depends on the type of the input is reminiscent of the typing of overloaded functions (also known as "*ad hoc* polymorphism"). This correspondence is indeed a strong one, since intersections of arrow types can be used to type overloaded functions (*eg*, see Benzaken et al. [2003]; Castagna et al. [1995]; see also Forsythe [Reynolds 1996] which uses a limited form of overloading known as *coherent overloading*). As a matter of fact, our function `f` is just a curried function that when applied to a Boolean argument returns an overloaded function. This can better be seen by considering type equivalences: the type we declared above for `f`

```
(Bool -> (Int & ?) -> Int) & (Bool -> (Bool & ?) -> Bool)
```

is equivalent to the type

```
Bool -> ( (Int & ?) -> Int) & ((Bool & ?) -> Bool) )
```

Therefore an equivalent way to define `f` would have been

```
let f (condition : Bool) : ((Int & ?) -> Int) & ((Bool & ?) -> Bool)) =
    λx.if condition then
         succ x
       else
         not x
```

which shows in a clear way that the application of `f` to a (necessarily Boolean) argument returns an overloaded function whose result type depends on the type of its argument: the two occurrences of "?" in the input types of the overloaded function indicate that, in both cases (*ie*, whatever the type of the argument is), some dynamic cast *may* be needed in the body of the function and, thus, may have to be added at compile-time.

We want to conclude this introduction by mentioning singleton types, that we do not consider in this work but that can be straightforwardly added to our theory without any further modification.

[4] Although we did not formally define any equivalence relation, the reader will easily recognize here a classic distributivity rule of set-theoretic unions and intersections. Slightly more formally, in this introduction we consider two types to be equivalent if they are both one subtype of the other (according to Frisch et al. [2008]) and where "?" is considered as some distinguished base type that intersects all the other types.

In a language such as ℂDuce every value also denotes the singleton type that contains only that value. In particular ℂDuce features two separate types for `true` and `false` and the type `Bool` is then simply defined as the union `true | false`. In this language, it is then possible to define the most precise type for `f` without using neither gradual types nor explicit type casts:[5]

```
let f : (true -> Int -> Int) & (false -> Bool -> Bool) =
  fun condition x -> if condition then succ x else not x
```

Likewise, in Typed Racket, a similar typing is obtained by "`case->`", which provides a limited form of intersection:

```
(: f : (case-> (-> True Integer Integer) (-> False Boolean Boolean)))
(define (f condition x) (if condition (add1 x) (not x)))
```

with the caveat that partial applications are not allowed. Of course this works in both cases only because `Bool` is a finite type, but in general it is not possible to replace gradual types or (equivalently) explicit type casts by using singleton types.

1.1 Overview

The examples given so far provide a brief outline of the characteristics of the system we are going to study. In a nutshell, this work develops a theory for gradual set-theoretic types, that is, types that besides the usual type constructors —*eg*, arrows, products, integers, ...— include a gradual "?" basic type and set-theoretic type connectives: union, intersection, *and* negation (in the set-theoretic type approach negations are indissociable from unions and intersections). This amounts to defining and deciding their subtyping relation, using them to type a core functional language, and defining a compilation scheme that inserts all and only the type casts required to ensure that every non-diverging well-typed expression will either return a value or raise a cast error. To that end, we proceed as follows. In Section 2, we define the syntax and semantics of the types we are interested in. In particular, we use abstract interpretation [Cousot and Cousot 1977] to define the semantics of our gradual types, following a technique introduced by Garcia et al. [2016]. Section 3 presents a gradually-typed language and its associated type system. This language is an explicitly-typed lambda calculus with a typecase, the latter included to fully exploit set-theoretic types. Section 4 presents the target language, a gradually-typed lambda calculus with explicit casts. In particular, we define its type system and operational semantics and prove its soundness. Finally, section 5 describes a compilation procedure that automatically inserts casts guaranteeing that every well-typed term of our gradually typed language is compiled into a term that if it converges, then it reduces either to a value or to a cast error. A conclusion with directions for future work ends this presentation.

For space reasons, lemmas, proofs, and a few definitions are provided as supplemental material in an appendix available on-line as supplemental material.

1.2 Contributions

The main contribution of this work is the definition of the static and the dynamic semantics of a language with gradual types and set-theoretic type connectives and the proof of its soundness. In particular, we show how to lift the definition of domain and result type of an application from set-theoretic types to gradual types and likewise for the subtyping relation. We also show that

[5] In ℂDuce "`if` e `then` e_1 `else` e_2" is just syntactic sugar for "`match` e `with true ->` e_1 `| false ->` e_2". The current public release of ℂDuce requires all function parameters, such as `condition` and `x`, to be explicitly typed and, thus, it does not accept the code below. The code below is executable on a experimental prototype version (available at http://www.cduce.org/ocaml/bi) based on the work by Castagna et al. [2016] and that uses a bi-directional type system to infer the types of the parameters of functions.

deciding subtyping for gradual types can be reduced in linear time to deciding subtyping on set-theoretic types and that the same holds true for all subtyping-related decision problems needed for type inference (notably, computing domains and result types). More generally, this work not only enriches gradual type systems with unions and intersections and with the type precision that arises from their use, but also proposes and advocates a new style of programming with gradual types where union and intersection types are used by the programmer to instruct the system to perform fewer dynamic checks.

1.3 Related work

Our work combines set-theoretic types with gradual typing. The part on set-theoretic types is based on the *semantic subtyping* framework, as presented by Frisch et al. [2008], while for what concerns the addition of gradual typing we followed and adapted the technique based on abstract interpretation by Garcia et al. [2016] called "Abstracting Gradual Typing" (AGT). However, due to the specific needs of our type system and the difficulty of finding a suitable Galois connection (a standard construction for abstract interpretation), we could apply directly the AGT approach only to "lift" subtyping. In particular, the approach proposed by Garcia et al. [2016] focuses on consistent subtyping, whereas dealing with set-theoretic types requires more precise properties —most notably for lemmas related to term substitution— hence the need for new operators and for a specific dynamic semantics.

There exist other attempts at integrating gradual typing with union and/or intersection types, but none is as general as the one presented here, insofar as they just consider either a partial set of type connectives or limited forms thereof. Siek and Tobin-Hochstadt [2016] study gradual typing for a language with type-case, union types, and recursive types. Intersection and negation types are not considered and union types are in a restricted version, since it is not possible to form the union of any two types but just of types having different top-most constructors: so for instance it is not possible to union two arrows. This limitation is reflected in the type-case expression which can only check the topmost constructor of a value (*eg*, integer, product, arrow, ...) but not its type. The different cases of the type-case construction of Siek and Tobin-Hochstadt [2016] are functions that, if selected, are applied to the matched value; this allows a form of *occurrence typing* [Tobin-Hochstadt and Felleisen 2008]. Our type-case is more general than the one by Siek and Tobin-Hochstadt [2016] since expressions can be checked against any type, and occurrence typing can be encoded (see Footnote 13 later on). For the sake of simplicity we considered neither product nor recursive types (though all their theory is already developed by Frisch et al. [2008]) and disregarded blame, but otherwise our work subsumes the one by Siek and Tobin-Hochstadt [2016].

Jafery and Dunfield [2017] present a type system that contains both refinement sums and gradual sums. Similarly to our approach, they define a gradually-typed source language and a type-directed translation to a target language that contains casts. However, their approach is very different from ours: their sums are disjoint unions in which elements are explicitly injected by a constructor; as such, their sums do not have the set-theoretic property of unions (*eg*, they are neither idempotent, nor commutative, nor satisfy usual distribution laws). Also, gradual typing is confined to sum types, since the motivation of their work is to allow the programmer to gradually add refinements that make the enforcement of exhaustive pattern matching possible, a problem that does not subsists in our work or, more generally, in languages with set-theoretic types where exhaustiveness of pattern

matching is easily verifiable.[6] Finally, Jafery and Dunfield [2017] leave intersection types as future work.

Our approach also relates to a recent work by Lehmann and Tanter [2017], which presents a way to combine gradual typing with full-fledged refinement types, where formulae can contain unions, intersections, and negations. They encounter problems similar to ours, most notably when trying to find a suitable Galois connection to use for AGT. However, our work focuses on set-theoretic types which behave differently from refinement types, in particular when it comes to subtyping, function types, or when evaluating casts.

For what concerns programming languages, there have been few attempts at providing a language with union and intersection type connectives and *some form* of gradual typing, whether it be via generics or using a specific unknown type. Facebook's Flow [Chaudhuri 2014] is an impressive gradually-typed version of JavaScript that provides union and intersection types, occurrence typing, as well as a type "any" that behaves as our unknown type "?": however any is used in Flow only to shunt off the type system, but not to insert casts. By inserting by hand appropriate type annotations and casts it is possible to make the examples of our introduction type-check in Flow. Therefore, it would be interesting to see how to use our system to give the programmer the option to let Flow add the necessary casts and annotations. Also, even though an implementation is available, there is no detailed (let alone formal) definition of Flow's type system: our work certainly provides a good starting point for it. Typed Racket is another example that provides sound gradual typing (in a coarser-grained form than the one obtained by using explicit gradual types) as well as true union types. It does not support full intersection types (just the limited form we showed to be provided by the `case->` construct in which types are considered in order from the first to the last), it includes recursive types and polymorphic function, performs a limited form of type reconstruction, and, above all, is a full-fledged programming language. It also features occurrence typing [Tobin-Hochstadt and Felleisen 2008], which refines the types of some variable depending on the result of a conditional, and that can be encoded by our typecase construct. As it is the case for Flow, in Typed Racket the examples of our introduction are accepted only after the insertion of explicit type inspection primitives by the programmer. Contrary to Flow, however, using our system to insert these casts in Typed Racket does not seem appropriate, both for the absence in Typed Racket of an explicit "unknown" type and, more generally, for the general spirit of the approach.

2 TYPES

In this section, we define the syntax of set-theoretic types (that in this work we call "static types") and extend them to obtain gradual types. We define the semantics of the latter in terms of the former. This semantic is then used to lift to gradual types relevant definitions given for set-theoretic types, notably the subtyping relation.

2.1 Type Syntax

Definition 1. *(Types) The set* STypes *of* static types *and the set* GTypes *of* gradual types *are inductively generated by the following grammars:*

$$t \in \text{STypes} ::= b \mid t \to t \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1}$$
$$\tau \in \text{GTypes} ::= ? \mid b \mid \tau \to \tau \mid \tau \vee \tau \mid \tau \wedge \tau \mid \neg t \mid \mathbb{0} \mid \mathbb{1}$$

[6] This is due to the specific definition of set-theoretic patterns which combine singleton types and set-theoretic connectives in such a way that checking exhaustiveness amounts to checking a subtyping relation: see, for instance, [Castagna et al. 2016, sect. 4.2].

*where b ranges over the set of base types (*eg, `Int`, `Bool`, *...). We also single out the* atomic (gradual) types *as follows:*

$$a \in \text{Atom} ::= b \mid t \to t \qquad \alpha \in \text{GAtom} ::= \mathbf{?} \mid b \mid \tau \to \tau$$

The set STypes of static types, ranged over by $s, t, \ldots$, is formed by basic types, the type *constructor* "$\to$" for function types, type *connectives* for union, intersection and negation types, as well as $\mathbb{0}$ and $\mathbb{1}$ which denote respectively the bottom and the top type. The set GTypes of gradual types, ranged over by $\sigma, \tau, \ldots$, is obtained by adding to the static types the unknown type "**?**", which stands for the absence of type information (not to be confused with $\mathbb{0}$ or $\mathbb{1}$). As anticipated, we also included negation types, which correspond to the set-theoretic complement (*ie*, a well-typed expression has type $\neg t$ if and only if it *does not* have type t). These play an important role in our theory insofar as all subtyping-related algorithms as well as the type-inference of type-case expressions fundamentally rely on them.[7] Notice however that we do not allow negations of gradual types. (the reason is that we could neither conceive any reasonable interpretation for a type such as "$\neg$ **?**", nor we were able to deduce it from our formalization). We use the standard convention that connectives (*ie*, $\wedge$, $\vee$, and $\neg$) are given a higher precedence than constructors (*ie*, $\to$). We also use the symbol "$\setminus$" to denote the difference of a (possibly gradual) type with another, this being defined in the usual set-theoretic way, that is, $\tau \setminus t \stackrel{\text{def}}{=} \tau \wedge \neg t$.

We suppose that static types come equipped with the subtyping relation $\leq$ defined by Frisch et al. [2008] (our static types are a strict subset of those defined in [Frisch et al. 2008]). As hinted in the introduction, this subtyping relation is defined by interpreting static types as sets of *values* (*ie*, either constants or λ-abstractions) that have that type, and then defining subtyping as set containment (*ie*, a static type s is a subtype of a static type t if and only if t contains all the values of type s). More precisely: `Int` is the set of all integers; `Bool` is the set {`true`, `false`}; $\mathbb{0}$ is the empty set; $\mathbb{1}$ is the set of all (well-typed) values; type connectives are interpreted as the corresponding set-theoretic operators (*eg*, $s \vee t$ is the union of the values of the two types, while $\neg t$ is the set of all values that are not in/of type t); $s \to t$ is set of all λ-abstractions that when applied to a value of type s return only results of type t. In particular, $\mathbb{0} \to \mathbb{1}$ is the set of all functions[8], which is why we call every subtype of $\mathbb{0} \to \mathbb{1}$ a *function type*. We use $\simeq$ to denote the equivalence relation induced by the subtyping relation (*ie*, $t_1 \simeq t_2$ if and only if $t_1 \leq t_2$ and $t_2 \leq t_1$): intuitively, two static types are equivalent if and only if they denote the same set of values.

In the Introduction we described the intuitive semantics of gradual types. In the next section we formalize this intuition. But before that let us stress again that **?** must not be confused with $\mathbb{0}$ or $\mathbb{1}$: for instance, a function of type **?**$\to$ `Int` can take an argument of a certain unknown type and return an integer, but, contrary to $\mathbb{1} \to$ `Int`, this application might fail if the unknown type turns out at runtime not to be a super-type of the type of the argument; likewise, while the application of a function in `Int` $\to$**?** to an integer may return some result (whose type is unknown), the application to an integer of a function in `Int` $\to \mathbb{0}$ always diverges (since, if it converged, then it ought to return a value belonging to the empty type, which is impossible).

2.2 Semantics of Types

Since STypes $\subset$ GTypes, then any definition on gradual types can be implicitly restricted to static types. However, this containment is strict and, thus, the converse is not so straightforward. In the rest of this section we present a way to do this reverse operation, namely, to *lift* the definitions of

[7] The use of negation types becomes interesting for the programmer mainly in the presence of polymorphism —*cf.* [Castagna et al. 2015, 2014]— which is why in this work we put the emphasis essentially on unions and intersections.
[8] Actually, for every type t, all types of the form $\mathbb{0} \to t$ are equivalent and each of them denotes the set of all functions.

relations (foremost, subtyping) and operators (*eg*, domain and result type) given on static types to gradual types. Our approach uses abstract interpretation [Cousot and Cousot 1977] to interpret gradual types as sets of static types, as first proposed by Garcia et al. [2016].

Concretization. The first step in lifting the semantics of static types to gradual types is to define the *concretization* of a gradual type as a set of static types. This is based on the intuition that the gradual type **?** can turn out to be any type, that is, the set of its possible interpretations is STypes. Formally, we define the *concretization* function γ such that $\gamma(\tau)$ returns the set of static types obtained by replacing each occurrence of **?** in τ by some static type.

Definition 2. *(Concretization) The* concretization function $\gamma : \text{GTypes} \to \mathscr{P}(\text{STypes})$ *is defined as follows:*

$$\begin{aligned} \gamma(?) &= \text{STypes} & \gamma(\tau_1 \to \tau_2) &= \{t_1 \to t_2 \mid t_i \in \gamma(\tau_i)\} \\ \gamma(\tau_1 \vee \tau_2) &= \{t_1 \vee t_2 \mid t_i \in \gamma(\tau_i)\} & \gamma(b) &= \{b\} \\ \gamma(\tau_1 \wedge \tau_2) &= \{t_1 \wedge t_2 \mid t_i \in \gamma(\tau_i)\} & \gamma(\mathbb{0}) &= \{\mathbb{0}\} \\ \gamma(\neg t) &= \{\neg t\} & \gamma(\mathbb{1}) &= \{\mathbb{1}\} \end{aligned}$$

In our set-theoretic framework, the concretization of a gradual type has several interesting properties. In particular, for every gradual type τ, the set of its concretizations $\gamma(\tau)$ has a maximum and a minimum with respect to the subtyping relation $\leq$. That is, $(\gamma(\tau), \leq)$ is a closed sublattice of $(\text{STypes}, \leq)$. This is a direct consequence of the fact that the set of static set-theoretic types is a complete lattice, bounded by $\mathbb{1}$ and $\mathbb{0}$.

Proposition 1. *(Gradual Extrema) For every gradual type $\tau \in \text{GTypes}$, there exist in $\gamma(\tau)$ two static types, noted $\tau^{\Uparrow}$ and $\tau^{\Downarrow}$, such that for every type $t \in \gamma(\tau)$, $\tau^{\Downarrow} \leq t \leq \tau^{\Uparrow}$.*
*Moreover, $\tau^{\Uparrow}$ (resp. $\tau^{\Downarrow}$) is obtained from τ by replacing all covariant occurrences of **?** by $\mathbb{1}$ (resp. $\mathbb{0}$) and all contravariant occurrences of **?** by $\mathbb{0}$ (resp. $\mathbb{1}$).*

Subtyping. Subtyping is a binary relation on static types. We want to define a conservative extension of this predicate to gradual types. More generally, given any predicate $P(t_1, \cdots, t_n)$ on static types, we want to define a predicate $\widetilde{P}(\tau_1, \cdots, \tau_n)$ on gradual types such that $\widetilde{P}$ coincides with P on static types (conservative extension) and encompasses the intuition of gradual typing. This is done by using the concretization function as follows:

Definition 3. *(Predicate lifting [Garcia et al. 2016]) For every predicate $P \in \text{STypes}^n$, we define its* consistent lifting $\widetilde{P} \in \text{GTypes}^n$ *as:*

$$\widetilde{P}(\tau_1, \cdots, \tau_n) \overset{\text{def}}{\iff} \exists (t_1, \cdots, t_n) \in \gamma(\tau_1) \times \cdots \times \gamma(\tau_n) \quad s.t. \quad P(t_1, \cdots, t_n)$$

Notice that, since $\gamma(t) = \{t\}$ for every static type t, then $\widetilde{P}$ coincides with P on static types.

Intuitively, a predicate is true for some gradual type if and only if the static counterpart of the predicate is true for some particular concretization of these types. If we apply lifting to the subtyping relation, we obtain that, for every pair of gradual types σ and τ, $\sigma \mathrel{\widetilde{\leq}} \tau$ if and only if there exist two types $(s, t) \in \gamma(\sigma) \times \gamma(\tau)$ such that $s \leq t$ (the relation $\mathrel{\widetilde{\leq}}$ is usually referred to as "consistent subtyping"). However, this definition can be simplified using the fact, stated by Proposition 1, that $(\gamma(\tau), \leq)$ is a sublattice of $(\text{STypes}, \leq)$.

Proposition 2. *(Consistent subtyping) For every pair of gradual types σ, τ, the following equivalence holds:*

$$\sigma \mathrel{\widetilde{\leq}} \tau \iff \sigma^{\Downarrow} \leq \tau^{\Uparrow}$$

This proposition is important since it implies that the subtyping problem for gradual types can be linearly reduced (as explained in Proposition 1) to the same problem on static types and, thus, that

we can re-use the algorithms that already exist for static types. In particular, since $\tau^{\Uparrow}$ and $\tau^{\Downarrow}$ are of the same size as τ, then the decision problems of $\widetilde{\leq}$ and $\leq$ have the same algorithmic complexity. This property also illustrates the (well-known) fact that consistent subtyping $\widetilde{\leq}$ *is not* transitive. Indeed, **?** is a consistent subtype of every type (as $?^{\Downarrow} = \mathbb{0}$) but also a consistent super-type of every type (as $?^{\Uparrow} = \mathbb{1}$). Thus, if consistent subtyping were to be transitive, it would collapse to the trivial full relation where $\sigma \,\widetilde{\leq}\, \tau$ for every gradual types σ and τ. We think that gradual quasi-subtyping would have been more appropriate a name for this relation, but we preferred to adopt the accepted terminology.

2.3 Operators on types

Lifting subtyping from static to gradual types was easy: we just adapted standard techniques of abstract interpretation on the lines of Garcia et al. [2016]. Lifting type operators, instead, is far more difficult and constitutes one of the hardest technical achievements of our work. Operators on types are necessary to define type inference. In particular, in order to type any functional language (notably, to type its applications) one needs at least two operators: (*i*) the *dom*(.) operator that given a *function type* t (*ie*, a subtype of $\mathbb{0} \to \mathbb{1}$, that is, a type that only contains values that are λ-abstractions) returns the domain $dom(t)$ of the functions in that type and (*ii*) the infix binary application operator $\circ$, called the *result type* operator, that given a function type t and a type s subtype of $dom(t)$ returns the type $t \circ s$ of all values resulting from the application of a function of type t to an argument of type s. To explain why lifting these operators to gradual types is challenging let us consider the *dom*(.) operator (but a similar argument holds for the result type operator too).

If we do not have any type connective, then, in general, all function types are of the form $s \to t$. Therefore defining the domain and result operators is trivial: $dom(s \to t)$ is s (the type on the left of the arrow) and if $s' \leq s$, then $(s \to t) \circ s'$ is t (the type on the right of the arrow). Adding type connectives makes such definitions more difficult since one must determine what is the domain of, say, the function type $((\texttt{Int} \to \texttt{Bool}) \wedge \neg\texttt{Int}) \vee (\neg(\texttt{Bool} \to \texttt{Int}) \wedge (\texttt{Int} \to \texttt{Int}))$ or what is the type of the result when a function of this type is applied to some argument. The solution to this problem is given by Frisch et al. [2008] and consists in transforming every function type into an equivalent —with respect to $\simeq$, the equivalence relation induced by $\leq$— type in disjunctive normal form, which is a type formed by unions of intersections of literals (*ie*, of atoms or their negation). For such a type it is then easy to define *dom*() (*eg*, the domain of an intersection of arrows is the union of the domains of the arrows; the domain of a union of arrows is the intersection of their domains) and, to a lesser extent, $\circ$. Unfortunately it is not possible to adapt this technique to gradual types: since the transitive closure of $\widetilde{\leq}$ is the trivial full binary relation on GTypes, then it is unsound to consider, as with static types, the equivalence relation induced by $\widetilde{\leq}$ (since it is the full relation too). The solution we propose is to define a different concretization function for gradual types called *applicative concretization*[9] that is tailored to the definition of the two operators at issue. The resulting definitions are quite technical and barely intuitive but they have the properties we seek for, since they allow us to define in Section 3 a type-inference system that enjoys the subject-reduction and progress properties. We first give the formal definitions, followed by an intuitive explanation, and then give an *a posteriori* justification of our definitions by showing that our new concretization function corresponds to transforming gradual types into gradual normal forms (that are the natural gradual extension of the normal forms for static types) by applying a

[9]Strictly speaking, what we are going to define is not a concretization function, in the sense of abstract interpretation, but rather a function that distills the functional characteristics of a given type.

rewriting system that preserves the semantics of $dom(.)$ and $\circ$. For space reasons we just hint at this *a posteriori* justification: all details can be found in the appendix.

The applicative concretization is defined as follows (where $\mathscr{P}_f$ denotes the set of finite subsets):

Definition 4. *(Applicative Concretization) The* applicative concretization $\gamma^+_{\mathscr{A}}$ *of a gradual type* τ *is defined as:*

$$\gamma^+_{\mathscr{A}} : \text{GTypes} \to \mathscr{P}_f(\mathscr{P}_f(\text{GTypes}))$$
$$\gamma^+_{\mathscr{A}}(\tau_1 \vee \tau_2) = \gamma^+_{\mathscr{A}}(\tau_1) \cup \gamma^+_{\mathscr{A}}(\tau_2)$$
$$\gamma^+_{\mathscr{A}}(\tau_1 \wedge \tau_2) = \{T_1 \cup T_2 \mid T_i \in \gamma^+_{\mathscr{A}}(\tau_i) \text{ for } i = 1,2\}$$
$$\gamma^+_{\mathscr{A}}(\sigma \to \tau) = \{\{\sigma \to \tau\}\}$$
$$\gamma^+_{\mathscr{A}}(\neg t) = \gamma^-_{\mathscr{A}}(t)$$
$$\gamma^+_{\mathscr{A}}(\mathbb{0}) = \emptyset$$
$$\gamma^+_{\mathscr{A}}(\mathbb{1}) = \{\emptyset\}$$
$$\gamma^+_{\mathscr{A}}(b) = \{\emptyset\}$$
$$\gamma^+_{\mathscr{A}}(?) = \{\{? \to ?\}\}$$

$$\gamma^-_{\mathscr{A}} : \text{STypes} \to \mathscr{P}_f(\mathscr{P}_f(\text{STypes}))$$
$$\gamma^-_{\mathscr{A}}(t_1 \vee t_2) = \{T_1 \cup T_2 \mid T_i \in \gamma^-_{\mathscr{A}}(t_i) \text{ for } i = 1,2\}$$
$$\gamma^-_{\mathscr{A}}(t_1 \wedge t_2) = \gamma^-_{\mathscr{A}}(t_1) \cup \gamma^-_{\mathscr{A}}(t_2)$$
$$\gamma^-_{\mathscr{A}}(s \to t) = \{\emptyset\}$$
$$\gamma^-_{\mathscr{A}}(\neg t) = \gamma^+_{\mathscr{A}}(t)$$
$$\gamma^-_{\mathscr{A}}(\mathbb{0}) = \{\emptyset\}$$
$$\gamma^-_{\mathscr{A}}(\mathbb{1}) = \emptyset$$
$$\gamma^-_{\mathscr{A}}(b) = \{\emptyset\}$$

Notice that that both functions produce sets of (respectively, gradual and static) arrow types.

Next we use the applicative concretization to define the domain and result operators on gradual types:

Definition 5. *(Gradual Type Operators) Let* τ *and* σ *be two gradual types such that* $\tau \mathrel{\widetilde{\leq}} \mathbb{0} \to \mathbb{1}$ *and* $\sigma \mathrel{\widetilde{\leq}} \widetilde{dom}(\tau)$*. The* gradual domain *of* τ*, noted* $\widetilde{dom}(\tau)$ *and the* gradual result type *of the application of* τ *to* σ*, noted* $\tau \mathbin{\widetilde{\circ}} \sigma$ *are respectively defined as follows:*

$$\widetilde{dom}(\tau) \stackrel{def}{=} \bigwedge_{S \in \gamma^+_{\mathscr{A}}(\tau)} \bigvee_{\rho \to \rho' \in S} \rho^{\Uparrow}$$

$$\tau \mathbin{\widetilde{\circ}} \sigma \stackrel{def}{=} \bigvee_{S \in \gamma^+_{\mathscr{A}}(\tau)} \bigvee_{\substack{Q \subsetneq S \\ \sigma \mathrel{\widetilde{\not\leq}} \bigvee_{(\rho \to \rho') \in Q} \rho \\ \sigma^{\Uparrow} \wedge \bigvee_{(\rho \to \rho') \in S \setminus Q} \rho^{\Uparrow} \mathrel{\widetilde{\not\leq}} \mathbb{0}}} \bigwedge_{(\rho \to \rho') \in S \setminus Q} \rho'$$

In order to explain and justify the definitions above, let us recall the details of how the operators of domain and result are defined for set-theoretic types [Frisch et al. 2008] and, thus, for our static types. Given a static function type t (*ie*, a type t such that $t \leq \mathbb{0} \to \mathbb{1}$) its domain $dom(t)$ is semantically defined in [Frisch et al. 2008] as the greatest type t' such that $t \leq t' \to \mathbb{1}$. Such a type is uniquely defined modulo the equivalence relation $\simeq$ and its definition is preserved by this equivalence, that is, two equivalent types have the same domain. Given a static type t, there exists a type equivalent to t that is in *disjunctive normal form*, that is, a type which is a union of *uniform intersections*[10] of atoms or their negations. In particular if t is a *function type*, then all its disjunctive normal forms have the following shape:

$$t \simeq \bigvee_{f \in F} \bigwedge_{j \in P_f} s_j \to t_j \wedge \bigwedge_{n \in N_f} \neg(s_n \to t_n) \tag{1}$$

Frisch et al. [2008] prove that the domain of the disjunctive normal form above is equivalent to

$$\bigwedge_{f \in F} \bigvee_{j \in P_f} s_j \tag{2}$$

and since the domain is preserved by $\simeq$, then the type in (2) is also equivalent to $dom(t)$, the domain of t. Since for every type t it is possible effectively to compute a disjunctive normal form equivalent

[10] Uniform means that the atoms in the intersections are either all arrows or all basic types

to it, then the domain of a function type can be effectively computed, too. Frisch et al. [2008] proceed similarly for the result type: if t is a functional type with a disjunctive normal form as in (1) and s is a subtype of $dom(t)$, then the result type $t \circ s$ is defined semantically (as the least type t' such that $t \leq s \to t'$), it is preserved by equivalence, and it is proved to be equivalent to:

$$t \circ s \simeq \bigvee_{f \in F} \bigvee_{\substack{Q \subsetneq P_f \\ s \not\leq \bigvee_{q \in Q} s_q}} \bigwedge_{p \in P_f \setminus Q} t_p \tag{3}$$

We do not explain the formula above: the details are given by Frisch et al. [2008] (a simpler and more detailed presentation is given by Castagna [2015, sect. 4.4.3]). We just invite the reader to recognize in (2) and (3) the patterns of the formulæ given in Definition 5 and see that what the applicative concretization of Definition 4 does is nothing but to transform a gradual type into a particular disjunctive normal form represented as a finite set (the union) of finite sets (the intersections) of arrow types. This correspondence is not just intuitive but is formalized in the appendix. Here we just outline its main ideas.

First, notice that neither (2), the definition of domain, nor (3), the definition of result, depend on the negated arrows of the normal form: the set N_f (of *N*egative atoms) does not occur in them, just the set P_f (of *P*ositive atoms) is used. Since $\gamma^+_{\mathscr{A}}$ yields a normal form tailored for these operators, then it does not produce any negated arrow (they are erased by the case $\gamma^-_{\mathscr{A}}(s \to t) = \{\emptyset\}$). This becomes evident when one applies $\gamma^+_{\mathscr{A}}$ to a static functional type t like in (1): $\gamma^+_{\mathscr{A}}(t) = \{\{s_j \to t_j \mid j \in P_f\} \mid f \in F\}$, which represents $\bigvee_{f \in F} \bigwedge_{j \in P_f} s_j \to t_j$, namely, the positive atoms of the normal form.

Second, for what concerns the definition of domain it is easy to see that this precisely corresponds to the definition given in (2) where $\gamma^+_{\mathscr{A}}$ gives the decomposition in disjunctive normal form and where the domain of an arrow $\rho \to \rho'$ is assumed to be the set of all the values in all possible concretizations of ρ, that is $\rho^{\Uparrow}$.

Third, there is the interpretation of the unknown type **?**. Since we are only interested in possible interpretations of gradual types *as functions*, we interpret **?** in the same way as **?** $\to$ **?**, hence the definition of $\gamma^+_{\mathscr{A}}(?)$.

Fourth and last, the formula for the result type in Definition 5 *nearly* corresponds to the formula in (3) where $\gamma^+_{\mathscr{A}}$ gives the decomposition in disjunctive normal form. There are however two differences, both in the definition of the sets Q on which the inner union ranges over. The first one is straightforward: since we are working with gradual types we replaced the relation $\not\leq$ by its consistent lifting $\not\widetilde{\leq}$. The second is much subtler, it is a consequence of the first difference, and requires to understand how the formula in (3) works. Once again, for a detailed explanation of this formula we invite the reader to consult the one given by Castagna [2015, sect. 4.4.3]; for this work it suffices to say that when we remove from P_f the arrows that are in Q, then the domain of (the intersection of) the arrows that remain has a non-empty intersection with the type of the argument (*ie*, these remaining arrows form a set of arrows that *may* handle the argument). This is so because Q always contains arrows that alone *cannot* completely handle the argument (*ie*, the type of the argument is not a subtype of the domain of the arrows in Q: this is condition $s \not\leq \bigvee_{q \in Q} s_q$). The consequence of replacing the gradual relation $\not\widetilde{\leq}$ for its non-gradual counterpart $\not\leq$ is that now Q contains arrows that alone *may not* completely handle the argument. We must ensure that we are not removing too many arrows, that is, that the arrows we remove cannot turn out (at run-time) to be exactly those that were supposed to handle the argument. In other terms, we have to ensure that, as for the formula in (3), the domain of the arrows that remain has a non-empty intersection with the type of the argument. This is exactly what the condition $\sigma^{\Uparrow} \wedge \bigvee_{(\rho \to \rho') \in S \setminus Q} \rho^{\Uparrow} \not\leq \mathbb{0}$ in Definition 5 does.

As an example, consider the function type $(? \to \texttt{Bool}) \wedge (\texttt{Int} \to \texttt{Int})$ applied to the type Int. Intuitively, the result type must be Int: either (*i*) the domain of the first arrow turns out to be incompatible with Int, and thus only the second arrow contributes to the result by returning an Int, or (*ii*) also the first arrow can handle the argument, but then the result must be of type $\texttt{Bool} \wedge \texttt{Int}$, that is, the empty set of values, which means that the application cannot return any value and thus it must diverge; so if a result is returned by this application, then it will be of type Int. Since the applicative concretization of the function type is the singleton $\{\ \{? \to \texttt{Bool}; \texttt{Int} \to \texttt{Int}\}\ \}$, then the only set S to consider when computing the result type of the application is $\{? \to \texttt{Bool}; \texttt{Int} \to \texttt{Int}\}$. Moreover, $\texttt{Int} \not\lesssim ?$ holds, but $\texttt{Int} \not\lesssim \texttt{Int}$ does not. As such, the only two choices for $Q \subsetneq S$ are $Q = \emptyset$ (which yields the summand $\texttt{Bool} \wedge \texttt{Int}$ of the result type) and $Q = \{? \to \texttt{Bool}\}$ (which yields the summand Int of the result type), hence the result type $\texttt{Int} \vee (\texttt{Bool} \wedge \texttt{Int})$ which is the result we expected, as it is equivalent to Int.

To justify the need for the second condition, consider this time the same function type but applied to an argument of type Bool. Intuitively, the only possible result type is Bool, since a function of type $\texttt{Int} \to \texttt{Int}$ cannot be applied to a Boolean. However, since $\texttt{Bool} \not\lesssim ?$ and $\texttt{Bool} \not\lesssim \texttt{Int}$, then without the second condition, there would be three possible choices for $Q \subsetneq S$, namely, $Q = \{? \to \texttt{Bool}\}$, $Q = \{\texttt{Int} \to \texttt{Int}\}$, and $Q = \emptyset$. This would give the result type $\texttt{Bool} \vee \texttt{Int} \vee (\texttt{Bool} \wedge \texttt{Int})$, which is equivalent to $\texttt{Bool} \vee \texttt{Int}$. This return type, although sound, is not correct since the application cannot return a value of type Int. The problem comes from the fact that $Q = \{? \to \texttt{Bool}\}$ is not a valid choice, since the function types that remain in $S \backslash Q$, that is just $\texttt{Int} \to \texttt{Int}$, cannot handle an argument of type Bool. Hence the need for the second condition, which excludes the previous case and ensures that the functions in $S \backslash Q$ will always be able to handle at least some values of the argument type.

The technical justification of Definitions 4 and 5 is that they allow us to define in the next sections a type system that satisfies the subject reduction and progress properties. We can however also give a less technical and more semantic justification of these definitions by showing that they correspond to transposing to gradual types the definitions given for set-theoretic types. In particular, if we transpose the notion of disjunctive normal forms from set-theoretic types to gradual types we obtain gradual types of the following form:

$$\begin{array}{lll} \bigvee_{i \in I_{f,1}} \bigwedge_{j \in J_i} \sigma_j \to \tau_j \wedge \bigwedge_{j \in J_n} \neg(s_j \to t_j) & \vee & \bigvee_{i \in I_{f,2}} \bigwedge_{j \in J_i} \sigma_j \to \tau_j \wedge \bigwedge_{j \in J_n} \neg(s_j \to t_j) \wedge ? \\ \vee \bigvee_{i \in I_{b,1}} \bigwedge_{j \in J_i} b_j \wedge \bigwedge_{j \in J_n} \neg b_j & \vee & \bigvee_{i \in I_{b,2}} \bigwedge_{j \in J_i} b_j \wedge \bigwedge_{j \in J_n} \neg b_j \wedge ? \end{array}$$

It is possible to define a rewriting system that transforms every gradual type into a gradual disjunctive normal form (see its definition in the Appendix, Definition 19). This system is strongly normalizing but it is not confluent. This is expected: as a static type is equivalent to several distinct disjunctive normal forms, so the rewriting system maps a gradual type in distinct gradual disjunctive normal forms. This was not a problem with static types since the (semantically defined) domain and result operators were preserved by type equivalence. We cannot have the same result for gradual types since the equivalence relation induced by consistent subtyping is trivial, but we can easily transpose it since we can show (cf. Section A.1.5 in the Appendix) that the rewriting system preserves the gradual domain and result operators of Definition 5, thus providing a more "semantic" justification for them. In other terms, the equivalence relation induced by the rewriting systems is a sound approximation of the semantic relation for gradual types, the one induced by the lifting of subtyping being too coarse to be of any use.

3 GRADUALLY-TYPED LANGUAGE

We next define a language that uses the types of the previous section and give the corresponding typing rules.

3.1 Language Syntax

3.1.1 Grammar. The gradually typed language we consider here is a typed lambda calculus tailored to set-theoretic types.

Definition 6. *(Gradually Typed Lambda Calculus) The terms constituting the gradually typed lambda calculus are defined by the following grammar:*

$$\begin{array}{rll} \textbf{\textit{Terms}} & e ::= x \mid c \mid \lambda^{\mathbb{I}} x.\, e \mid e\ e \mid (e \in t)?e : e & \\ \textbf{\textit{Values}} & v ::= x \mid c \mid \lambda^{\mathbb{I}} x.\, e & \\ \textbf{\textit{Interfaces}} & \mathbb{I} ::= \{\sigma_i \to \tau_i \mid i \in I\} & (I \textit{ a finite set}) \end{array}$$

In an ML-like language, union types are always tagged, meaning they can be eliminated by pattern matching on their constructors. However, our set-theoretic type system allows the use of untagged unions (for example $\texttt{Int} \vee \texttt{Bool}$), which can only be eliminated dynamically. Thus, we need to keep a dynamic representation of types, and allow for dynamic type tests which are noted $(e \in t)?e_1 : e_2$. Such a test will reduce to expression e_1 if *the result* of the evaluation of e has type t, or to expression e_2 otherwise. Moreover, note that only static types are allowed to be dynamically checked against. The reason for this is that checking a value v against a gradual type τ would intuitively amount to checking that the type of v is a subtype of τ, but then it would be the same as checking v against either $\tau^{\Uparrow}$ or $\tau^{\Downarrow}$ (according to the semantics we want to give to the type case), that is, in both cases a static type (which is why we chose to let the programmer unambiguously specify the static type the expression must be checked against).

The second difference from the usual lambda calculus is the use of explicit *interfaces* in lambda expressions. An interface is simply a set of arrows, which stands for the set of all the types of the function. For instance, in this syntax the fourth definition of `f` we gave in the introduction becomes $\lambda^{\{\texttt{Bool}\to\texttt{Int}\wedge ?\to\texttt{Int}\,;\,\texttt{Bool}\to\texttt{Bool}\wedge ?\to\texttt{Bool}\}} \textit{constant}\,.\ldots$ Giving explicit function types to abstractions allows us to have more precise types. For example, consider the function types $\tau_1 = (\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool})$ and $\tau_2 = (\texttt{Int} \vee \texttt{Bool}) \to (\texttt{Int} \vee \texttt{Bool})$. Both types have the same domain (that is, $\texttt{Int} \vee \texttt{Bool}$), and, in fact, every function of type τ_1 is also of type τ_2, but not vice-versa. Indeed, τ_1 is more precise than τ_2: when applying a function of type τ_1 to an argument of type `Int`, we can deduce statically that the result will be of type `Int`. Whereas, when applying a function of type τ_2 to the same argument, the only information we can deduce is that the result will be of type $\texttt{Int} \vee \texttt{Bool}$. Thus, the former type conveys more information than the latter, but it cannot be expressed if only function parameters are explicitly typed.

3.1.2 Interfaces. With static types, the type of a function is the intersection of the types in its interface, as it is a value of each of those types. For example, the type of a well-typed function having the interface $\{\texttt{Int} \to \texttt{Int}; \texttt{Bool} \to \texttt{Bool}\}$ is the intersection $(\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool})$. However, with gradual types this interpretation is less clear-cut since, for instance, there are two possible ways to understand the interface $\{\texttt{Int} \to \texttt{Int}; ? \to ?\}$. One could say that a function with this interface returns a value of type `Int` when applied to an argument of type `Int`, and returns something else when applied to an argument that is not of type `Int`. Or, one could interpret the interface as stating that the type of this function is the intersection $(\texttt{Int} \to \texttt{Int}) \wedge (? \to ?)$ which, according to the definition of the result type given in the previous section, means that it returns a result of type $\texttt{Int} \wedge ?$ when given an argument of type `Int`.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ (T_x) \qquad \frac{}{\Gamma \vdash c : B(c)}\ (T_c) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \mathrel{\widetilde{\leq}} \mathbb{0}\to\mathbb{1} \quad \tau_2 \mathrel{\widetilde{\leq}} \widetilde{dom}(\tau_1)}{\Gamma \vdash e_1\ e_2 : \tau_1 \widetilde{\circ} \tau_2}\ (T_{app})$$

$$\frac{\forall(\sigma \to \tau) \in \mathbb{I}, \quad \Gamma, x : \sigma \vdash e : \tau' \quad \tau' \mathrel{\widetilde{\leq}} \tau}{\Gamma \vdash \lambda^{\mathbb{I}} x.\ e : \mathtt{TypeOf}(\mathbb{I})}\ (T_\lambda) \qquad \frac{\Gamma \vdash e : \tau \quad \begin{cases} \tau \not\mathrel{\widetilde{\leq}} \neg t & \implies \Gamma \vdash e_1 : \sigma_1 \\ \tau \not\mathrel{\widetilde{\leq}} t & \implies \Gamma \vdash e_2 : \sigma_2 \end{cases}}{\Gamma \vdash ((e \in t)?e_1 : e_2) : \sigma_1 \vee \sigma_2}\ (T_{case})$$

Fig. 1. Typing rules for the gradually typed language

For this work we chose the first approach, since it seems more intuitive and closer to the spirit of gradual types. However, to keep the intuition that the type of a function is the intersection of the types in its interface (and thus ease the formalization), we decided to put a restriction on the interfaces: all the domains (left part) of the arrows of an interface have to be pairwise distinct. Formally, in what follows, we will only consider valid the interfaces $\{\sigma_i \to \tau_i \mid i \in I\}$ such that $\forall (i,j) \in I^2, i \neq j \implies (\sigma_i \wedge \sigma_j)^{\Uparrow} \leq \mathbb{0}$. Moreover, we also impose the condition $\forall i \in I, \sigma_i^{\Uparrow} \not\leq \mathbb{0}$, since an arrow of an interface is only meaningful if its domain is not empty. For instance, the interface $\{\mathtt{Int} \to \mathtt{Int}; ?\to ?\}$ is not a valid interface (because $\mathtt{Int} \wedge ? \neq \mathbb{0}$), but $\{\mathtt{Int} \to \mathtt{Int}; (?\setminus \mathtt{Int}) \to ?\}$ is. This definition is not restrictive, as the transformation of an arbitrary interface to a valid interface can be done statically, although this can lead to an exponential blow-up on the size of an interface. For example, the invalid interface $\{\mathtt{Nat} \to \mathtt{Nat}; \mathtt{Even} \to \mathtt{Even}; ?\to ?\}$ can be converted statically into the (intuitively) equivalent interface

$$\{\ (\mathtt{Nat}\setminus\mathtt{Even}) \to \mathtt{Nat}\ ;\ (\mathtt{Nat}\wedge\mathtt{Even}) \to (\mathtt{Nat}\wedge\mathtt{Even})\ ;\ (\mathtt{Even}\setminus\mathtt{Nat}) \to \mathtt{Even}\ ;\ (?\setminus(\mathtt{Nat}\wedge\mathtt{Even})) \to ?\ \}$$

Moreover, interfaces that contain several overlapping gradual types (that is, gradual types that have some concretizations in common) can be rewritten using the same method. For example, the interface $\{(?\to \mathtt{Int}; ?\to \mathtt{Bool}\}$ can simply be converted into the interface $\{? \to \mathtt{Int} \vee \mathtt{Bool}\}$.

According to this new definition, we will use $\mathtt{TypeOf}(\mathbb{I})$ to denote the type associated to a valid interface $\mathbb{I}$, that is, the intersection of its types. Formally, we define the operator $\mathtt{TypeOf}$ as follows:

$$\mathtt{TypeOf}(\mathbb{I}) = \bigwedge_{(\sigma\to\tau)\in\mathbb{I}} \sigma \to \tau$$

3.2 Typing

Having defined the syntax of the language and the type of lambda expressions, we now use the operators defined in the previous section to provide typing rules for this language. The rules are presented in Figure 1 and assume that we are given a function B that associates to every constant c its static type $B(c)$.

The typing rules (T_x) and (T_c) are the usual rules for typing variables and constants respectively, while the typing rule for lambda expressions (T_λ) formalizes the explanations we gave earlier. Notice that, as in any gradual type system, adding a subsumption rule would make the system unsound (since the transitive closure of subtyping is the full relation, then by two consecutive applications of subsumption it would be possible to give any type to every well-typed term). Therefore our systems uses a more "algorithmic presentation" in which the checks of the subtyping relation are distributed over the rules. In particular the rule (T_λ), checks that the inferred return type is a subtype of the type specified by the programmer in the interface (*ie*, $\tau' \mathrel{\widetilde{\leq}} \tau$ which is equivalent to $\tau'^{\Downarrow} \leq \tau^{\Uparrow}$). This is used to type functions such as $\lambda^{\{\mathtt{Int}\to ?\}} x.\ x$ or, even simpler, $\lambda^{\{\mathtt{Nat}\to\mathtt{Int}\}} x.\ x$.

The typing rule for applications, (T_{app}) is also straightforward, using the definitions of Section 2. Given an application e_1e_2, we just need to ensure that the type τ_1 of e_1 is a function type (*ie*, a subtype of $\mathbb{0} \to \mathbb{1}$, which is equivalent to checking $\tau_1^{\Downarrow} \leq \mathbb{0} \to \mathbb{1}$), and that the type of e_2 is a consistent subtype of the domain of e_1 (equivalently, $\tau_2^{\Downarrow} \leq \widetilde{dom}(\tau_1)$ since the domain always is a static type). The return type of the application is then defined using the result type operator $\widetilde{\circ}$.

To explain the rule (T_{case}), let us first remind the definition of its static counterpart, as defined by Frisch et al. [2008].

$$\frac{\Gamma \vdash e : t' \quad \begin{cases} t' \not\leq \neg t & \Longrightarrow \Gamma \vdash e_1 : s \\ t' \not\leq t & \Longrightarrow \Gamma \vdash e_2 : s \end{cases}}{\Gamma \vdash ((e \in t)?e_1 : e_2) : s}$$

The intuition behind this rule is that if we can statically prove that a branch will not be evaluated, then it does not need to be typed.[11] Given a type-case $(e \in t)?e_1 : e_2$, we know that the set of values that e can reduce to is given by the type t' of e. Therefore, saying that the first branch cannot be evaluated amounts to saying that $t' \wedge t \simeq \mathbb{0}$, which is set-theoretically equivalent to $t' \leq \neg t$. Thus, the branch e_1 should be evaluated only if $t' \not\leq \neg t$, hence the condition. Naturally, the same reasoning can be done with expression e_2, providing the second case of this rule.

To deduce the gradual equivalent of this rule, we use the definition of predicate lifting to lift the operator $\not\leq$, which yields the following definition: $\tau \mathrel{\widetilde{\not\leq}} \sigma \iff \exists (t,s) \in \gamma(\tau) \times \gamma(\sigma), t \not\leq s$. As for subtyping, this definition can be simplified using the extrema of concretizations: $\tau \mathrel{\widetilde{\not\leq}} \sigma \iff \tau^{\Uparrow} \not\leq \sigma^{\Downarrow}$. Notice that this definition—*ie*, the lifting of $\not\leq$— *is not* equivalent to the negation of $\widetilde{\leq}$ —the lifting of $\leq$—, whose definition is $\neg(\tau \mathrel{\widetilde{\leq}} \sigma) \iff \forall (t,s) \in \gamma(\tau) \times \gamma(\sigma), t \not\leq s \iff \tau^{\Downarrow} \not\leq \sigma^{\Uparrow}$. The condition $t' \not\leq \neg t$ then lifts to the condition $\tau \mathrel{\widetilde{\not\leq}} \neg t$ presented in rule (T_{case}) which is equivalent to $\tau^{\Uparrow} \not\leq \neg t$ (since $\neg t^{\Downarrow} = \neg t$). Likewise for $t' \not\leq t$ and $\tau \mathrel{\widetilde{\not\leq}} t$

Rationale: The language we defined in this section is an abstraction of the language the programmer is supposed to program with. Notice that we did not define the semantics of this language and *a fortiori* we did not prove any soundness or safety property of the type system defined above. The semantics of this language will be given by translating its terms into the "cast language" we define in the next section. The translation is defined in Section 5 where we also prove (cf. Theorem 3) that every well-typed term of this language is translated into a well-typed term of the cast language. The soundness of the cast language's type systems (cf. Theorem 2) implies safety (as expressed in Corollary 1) of the type system of Figure 1.

4 CAST LANGUAGE

In so-called "sound" gradual typing the compiler must insert dynamic checks into gradually-typed programs to ensure that they do not get stuck at execution. In the previous section we defined our gradually-typed language. In this section we present the target language that includes casts, and give its static and dynamic semantics.

4.1 Syntax

The target language we consider is closely related to the gradually-typed lambda calculus that was defined in the previous section.

[11] Actually, it *must not* be typed, otherwise every type-case expression would always be typed by the union of the types of its two cases and the only typeable overloaded functions would be the *coherent* ones, as in Forsythe [Reynolds 1996]. For instance, it would not be possible to deduce that $\lambda x.(x \in \texttt{Int})?(-x):\texttt{not}(x)$ has type $(\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool})$: see §3.3 in [Frisch et al. 2008] for a detailed explanation.

Definition 7. *(Cast Language) The terms constituting the cast language are defined by the following grammar:*

$$\begin{aligned}
\textbf{\textit{Terms}}^{\langle\rangle} \quad & e ::= x \mid c \mid \lambda^{\mathbb{I}}_{\langle\tau\rangle}\, x.\, e \mid e\ e \mid (e \in t)?e : e \mid \langle\tau\rangle\ e \\
\textbf{\textit{Values}}^{\langle\rangle} \quad & v ::= c \mid \lambda^{\mathbb{I}}_{\langle\tau\rangle}\, x.\, e \\
\textbf{\textit{Interfaces}} \quad & \mathbb{I} ::= \{\sigma_i \to \tau_i \mid i \in I\} \\
\textbf{\textit{Errors}} \quad & \mathscr{E} ::= \texttt{CastError}
\end{aligned}$$

As before, every interface $\mathbb{I} = \{\sigma_i \to \tau_i \mid i \in I\}$ must satisfy the conditions $\forall (i,j) \in I^2, (\sigma_i \wedge \sigma_j)^{\Uparrow} \leq \mathbb{0}$, and $\forall i \in I, \sigma_i^{\Uparrow} \not\leq \mathbb{0}$.

The most important addition of this language to the gradually-typed language of the previous section is the presence of the *cast* expression $\langle\tau\rangle\ e$ which verifies whether the value resulting from the evaluation of the expression e has type τ and returns the value itself or `CastError`, accordingly.

The other difference is that lambda-abstractions now include cast annotations. The notation $\lambda^{\mathbb{I}}_{\langle\tau\rangle}\, x.\, e$ denotes the cast of the function $\lambda^{\mathbb{I}} x.\, e$ to the type τ. The idea behind this notation is that a lambda expression "stores" a cast in order to evaluate it, *only* when the function is applied. This lazy evaluation of casts allows us to avoid unnecessary but costly η-expansions which, as pointed out by Takikawa et al. [2016], hinder the practical interest of sound gradual typing. Moreover, to ease the formalization, we do not include uncast lambda-abstractions of the form $\lambda^{\mathbb{I}} x.\, e$ in the grammar. This has two implications. First, this obviously comes at a performance cost, since we may add unnecessary casts to certain functions (most notably, fully statically-typed functions). Second, as a consequence, the compilation procedure will always add casts, even in fully statically-typed programs. However, these casts should never result in a cast error. To ease the notation, we may write $\lambda^{\mathbb{I}} x.\, e$ as syntactic sugar for an abstraction that stores an identity cast, that is, $\lambda^{\mathbb{I}}_{\langle \texttt{TypeOf}(\mathbb{I})\rangle}\, x.\, e$.

Rationale: Our main objective in this section is to prove the soundness of the cast language, which will allow us to prove that the execution of the compilation of every well-typed term of the gradually-typed language is sound. To achieve this, we start by defining a set of typing rules for the cast language, before giving its operational semantics. We must proceed in this order because the presence of casts and type-cases makes the operational semantics depend on type inference.

4.2 Typing

The intuition of the type system in the previous section was that a term is well typed if there exist some concretizations of the gradual types occurring in it for which the execution would succeed. A well-typed term, then, will be compiled into a term of the cast language above by inserting in it all the dynamic checks necessary to ensure the progress property, that is, that its execution will either diverge,[12] or converge to a value, or raise a cast error. To prove this property of the compilation of a well-typed term, we define a type system for the cast language satisfying the above progress property —*ie*, in which every well-typed converging term yields either a value or a cast error— and then prove that well-typed terms of the gradual language are compiled into well-typed terms of the cast language.

The key case for the type system of the cast language is, as expected, the typing of applications. Consider, for example, an application $e_1\ e_2$, where e_1 is of type **?→?** and e_2 is of type `Int`. This application is well-typed, since e_1 is known to be a function that can be applied *to any* argument, of any type: that fact that e_1 is well typed with type **?→?** ensures that e_1 contains all the casts necessary to guarantee that its argument (of type **?**) will not be misused. Consider now the same

[12] An example of well-typed term that diverges is $\omega\omega$ where $\omega \stackrel{\text{def}}{=} \lambda^{\{(?\to?)\wedge?\to?\}}.xx$.

application but where e_1 is of type ?. This time, this application *must not* be considered well-typed, since e_1 is *not necessarily* a function: at run-time its gradual type may turn out to be, say, `Int` (*eg*, if e_1 reduces to 42) and the application would thus reduce to a stuck term. Correcting this application would require adding a cast: for example, $(\langle ?\to ?\rangle\ e_1)\ e_2$, which is well-typed since it is an instance of the previous case (the function $(\langle ?\to ?\rangle\ e_1)$ is well typed with type ?→?, provided that e_1 is well typed).

In the absence of type connectives the typing of an application is easily solved: whenever the type e_1 is not an arrow, the application is not well typed (*eg*, see the typing of the cast languages in [Siek and Taha 2006; Wadler and Findler 2009]). Type connectives make this issue more difficult. Consider again the application $e_1\ e_2$ and suppose that the type of e_1 is $(\texttt{Int} \to \texttt{Int}) \wedge ?$. In the gradually-typed language of Section 3 this application is well-typed since the domain of this function is $\mathbb{1}$: this function can be applied to any argument. However, in the cast language the situation is quite different, since whether this application is to be considered well-typed or not depends on the type of e_2. If for instance e_2 is of type `Int`, then we know that this application cannot fail since the type $(\texttt{Int} \to \texttt{Int}) \wedge ?$ guarantees that e_2 is a function that can (at least) be applied to integer arguments. If instead e_2 is, say, of type `Bool`, then this application may fail, notably in the case that ? does not turn out to be a subtype of $\texttt{Bool} \to \mathbb{1}$. Therefore this application is not a well-typed term of the cast language, even though it could be transformed into one at compile time by a suitable cast of the function (*eg*, $(\langle \texttt{Bool} \to \mathbb{1}\rangle\ e_1)\ e_2$). This example shows that, even if the domain of a function of type $(\texttt{Int} \to \texttt{Int}) \wedge ?$ is $\mathbb{1}$, we are guaranteed that the application of such a function will not fail only if the argument is of type `Int`. We say that `Int` is the *safe domain* of the functions of type $(\texttt{Int} \to \texttt{Int}) \wedge ?$, that is, what we call safe domain is the type of all the arguments for which the application of these functions cannot get stuck (*ie*, all the arguments for which the functions need not to be cast). In particular, while the domain of ? is $\mathbb{1}$, the safe domain of ? is $\mathbb{0}$, since it is not possible to guarantee that the application of a "function" of type ? will not fail, whatever argument we use. Likewise, the domain and safe domain of ?→? are both $\mathbb{1}$: a function of type ?→? contains in its body all the casts necessary to ensure that the execution will not get stuck, whatever argument the function is applied to. To define the *safe domain* operator, we replicate the method used in Section 2 to define the domain operator, and start by giving a new concretization function.

Definition 8. *(Safe Applicative Concretization) For every gradual type τ, we define the* safe applicative concretization $\gamma^+_{\mathscr{S}}$ *of τ as follows:*

$$\gamma^+_{\mathscr{S}} : \text{GTypes} \to \mathscr{P}_f(\mathscr{P}_f(\text{GTypes}))$$

$$\gamma^+_{\mathscr{S}}(\tau_1 \vee \tau_2) = \gamma^+_{\mathscr{S}}(\tau_1) \cup \gamma^+_{\mathscr{S}}(\tau_2)$$
$$\gamma^+_{\mathscr{S}}(\tau_1 \wedge \tau_2) = \{T_1 \cup T_2 \mid T_i \in \gamma^+_{\mathscr{S}}(\tau_i) \text{ for } i = 1, 2\}$$
$$\gamma^+_{\mathscr{S}}(\sigma \to \tau) = \{\{\sigma \to \tau\}\}$$
$$\gamma^+_{\mathscr{S}}(\neg t) = \gamma^-_{\mathscr{S}}(t)$$
$$\gamma^+_{\mathscr{S}}(\mathbb{0}) = \emptyset$$
$$\gamma^+_{\mathscr{S}}(\mathbb{1}) = \{\emptyset\}$$
$$\gamma^+_{\mathscr{S}}(b) = \{\emptyset\}$$
$$\gamma^+_{\mathscr{S}}(?) = \{\{\mathbb{0} \to ?\}\}$$

$$\gamma^-_{\mathscr{S}} : \text{STypes} \to \mathscr{P}_f(\mathscr{P}_f(\text{STypes}))$$

$$\gamma^-_{\mathscr{S}}(t_1 \vee t_2) = \{T_1 \cup T_2 \mid T_i \in \gamma^-_{\mathscr{S}}(t_i) \text{ for } i = 1, 2\}$$
$$\gamma^-_{\mathscr{S}}(t_1 \wedge t_2) = \gamma^-_{\mathscr{S}}(t_1) \cup \gamma^-_{\mathscr{S}}(t_2)$$
$$\gamma^-_{\mathscr{S}}(s \to t) = \{\emptyset\}$$
$$\gamma^-_{\mathscr{S}}(\neg t) = \gamma^+_{\mathscr{S}}(t)$$
$$\gamma^-_{\mathscr{S}}(\mathbb{0}) = \{\emptyset\}$$
$$\gamma^-_{\mathscr{S}}(\mathbb{1}) = \emptyset$$
$$\gamma^-_{\mathscr{S}}(b) = \{\emptyset\}$$

The only difference between $\gamma^+_{\mathscr{S}}$ and $\gamma^+_{\mathscr{S}}$ is the concretization of ? which the latter concretizes using only its safe domain. Next, we use this definition to define the *safe domain operator*, similarly to the domain operator.

Definition 9. *(Safe Gradual Domain) For every gradual type τ verifying $\tau \mathrel{\widetilde{\leq}} \mathbb{0} \to \mathbb{1}$, we define the* safe gradual domain *of τ, noted $\widetilde{dom}_{\mathscr{G}}(\tau)$, as follows:*

$$\widetilde{dom}_{\mathscr{G}}(\tau) = \begin{cases} \bigwedge_{S \in \gamma^+_{\mathscr{G}}(\tau)} \bigvee_{\sigma \to \rho \in S} \sigma^{\Uparrow} & \text{if } \tau^{\Uparrow} \leq \mathbb{0} \to \mathbb{1} \\ \mathbb{0} & \text{otherwise} \end{cases}$$

This definition is the same as the definition of the domain given in Section 2 except an important difference: if $\tau^{\Uparrow} \not\leq \mathbb{0} \to \mathbb{1}$, then the safe domain of τ is defined as $\mathbb{0}$. This formalizes the intuition that, if τ is *not always* a function type (*ie*, there exists a concretization of τ that is not a function type), then its safe domain is empty.

The definition of safe domain allows us to type applications in the cast language, but there still is a class of terms of the cast language that poses a problem: the lambda-abstractions. The problem comes from the use of interfaces and can be illustrated by considering the rule (T_λ) in Figure 1 which shows that once we have inferred the type of the body of the function, we still have to check that this type is a consistent subtype of the type the programmer wrote in the interface, insofar as the latter is the one used to type the function. The situation is similar for the abstractions in the cast language, but in this case consistent subtyping is not strict enough to ensure progress. Consider for instance a lambda-abstraction of the cast language of the form $\lambda^{\{\texttt{Int}\to\texttt{Int}\to\texttt{Int}\}}x.\, e$ (for the sake of simplicity we omitted the cast in the lambda abstraction: consider it to be the identity cast). If this lambda expression is well-typed, then so is the double application $((\lambda^{\{\texttt{Int}\to\texttt{Int}\to\texttt{Int}\}}x.\, e)3)\ 5$. However requiring that the type of the body e is a consistent subtype of $\texttt{Int} \to \texttt{Int}$ does not suffice to ensure progress. Take for e the expression $\langle ? \rangle \texttt{x}$: this expression has type ? and $? \mathrel{\widetilde{\leq}} \texttt{Int}\to\texttt{Int}$, nevertheless the application above would return $(3)5$, a stuck expression. Therefore, to compare the type of the body with the one recorded in the interface we need a relation $\sqsubseteq$ that is strictly finer than consistent subtyping. The problem can be reframed into a more general one, namely, that consistent subtyping does not satisfy the substitution property: replacing in a given context a term of some gradual type by a term of a consistent subtype does not preserve progress. Hence the need for a stronger relation $\sqsubseteq$ on types which enjoys the substitution property and is preserved during the evaluation of a term: for instance, in the example $\lambda^{\{\texttt{Int}\to\texttt{Int}\to\texttt{Int}\}}x.\, e$ above we want that whatever value e reduces to, it will preserve progress in a context where an expression of type $\texttt{Int} \to \texttt{Int}$ is expected. In particular, this relation has to be transitive (which is not the case of consistent subtyping), and must retain enough information about function types so that reducing under an application still produces a well-typed term. Consider a term e of type τ that reduces into a term e' of type τ'. First of all, we want to make sure that we did not lose information when reducing e, or that τ' is more precise than τ. This amounts to saying that every concretization of τ' is also a concretization of τ, which can be written as $\tau'^{\Uparrow} \leq \tau^{\Uparrow}$. Note that this relation is transitive, and it implies subtyping. Secondly, if τ is always a function type (*ie*, $\tau^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$), then although the previous condition implies that τ' is also a function type, it does not say anything about the applicative concretizations of τ and τ'. Thus we do not know anything about the relation between the domains and results of τ and τ'. We want to make sure that the subtyping relation between τ and τ' also translates to their applicative concretizations. Formally, this can be written as the condition:

$$\gamma^+_{\mathscr{G}}(\tau') \subset \gamma^+_{\mathscr{G}}(\tau)$$

Using the intuition that the applicative concretization is representing a normal form, this amounts to saying that if τ is a union of intersections, then τ' is a union of fewer intersections than τ. The

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ (T_x^{\langle\rangle}) \qquad \frac{\mathsf{Type0f}(\mathbb{I}) \mathrel{\widetilde{\leq}} \tau \qquad \forall(\sigma \to \rho) \in \mathbb{I}, \quad \Gamma, x : \sigma \vdash e : \rho' \quad \rho' \sqsubseteq \rho}{\Gamma \vdash \lambda^{\mathbb{I}}_{\langle\tau\rangle}\, x.\, e : \tau}\ (T_\lambda^{\langle\rangle})$$

$$\frac{}{\Gamma \vdash c : B(c)}\ (T_c^{\langle\rangle}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1^{\Uparrow} \leq \mathbb{0} \to \mathbb{1} \quad \tau_2^{\Uparrow} \leq \widetilde{dom}_{\mathscr{D}}(\tau_1)}{\Gamma \vdash e_1\ e_2 : \tau_1 \widetilde{\circ} \tau_2}\ (T_{app}^{\langle\rangle})$$

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \langle\tau\rangle\, e : \tau}\ (T_{cast}^{\langle\rangle}) \qquad \frac{\Gamma \vdash e : \tau \quad \begin{cases} \tau \mathrel{\widetilde{\not\leq}} \neg t & \Longrightarrow \Gamma \vdash e_1 : \sigma_1 \\ \tau \mathrel{\widetilde{\not\leq}} t & \Longrightarrow \Gamma \vdash e_2 : \sigma_2 \end{cases}}{\Gamma \vdash ((e \in t)?e_1 : e_2) : \sigma_1 \vee \sigma_2}\ (T_{case}^{\langle\rangle})$$

Fig. 2. Typing rules for the cast language

same reasoning can be done with the result types, which gives the same condition but with $\gamma^+_{\mathscr{A}}$. Hence the definition of the applicative subtyping.

Definition 10. *(Applicative Subtyping) For every gradual types σ and τ such that $\sigma^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$ and $\tau^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$, we define the applicative subtyping relation as follows:*

$$\sigma \leq \tau \iff \begin{cases} \gamma^+_{\mathscr{D}}(\sigma) \subset \gamma^+_{\mathscr{D}}(\tau) \\ \gamma^+_{\mathscr{A}}(\sigma) \subset \gamma^+_{\mathscr{A}}(\tau) \end{cases}$$

Note that the subset relation used in this definition uses the syntactic equality over types. That is, $\{\{\mathtt{Int} \to \mathtt{Int}\}\} \subset \{\{\mathtt{Int} \to \mathtt{Int}\}; \{\mathtt{Bool} \to \mathtt{Bool}\}\}$ but $\{\{\mathtt{Int} \to \mathtt{Int}\}\} \not\subset \{\{\mathtt{Nat} \to \mathtt{Int}\}\}$ even if $\mathtt{Int} \to \mathtt{Int} \leq \mathtt{Nat} \to \mathtt{Int}$. We are only interested in finding a possible subtyping relation that has the required properties (substitution property and preservation during evaluation), not in finding the most general one. As such, we chose to only consider the syntactic equality between types since it makes the relation general enough and easier to use than a subtyping-based definition.

We can then define the compatible subtyping, which possesses all the properties we are looking for, as it will be formalized in Theorem 1.

Definition 11. *(Compatible Subtyping) For every gradual types σ and τ, we define the compatible subtyping $\sqsubseteq$ as follows:*

$$\sigma \sqsubseteq \tau \iff \begin{cases} \sigma^{\Uparrow} \leq \tau^{\Uparrow} \\ \tau^{\Uparrow} \leq \mathbb{0} \to \mathbb{1} \implies \sigma \leq \tau \end{cases}$$

The typing rules for the cast language are presented in Figure 2, and are directly derived from the rules of the gradually-typed language presented in Section 3. The differences are the addition of a rule $(T_{cast}^{\langle\rangle})$ for the typing of cast expressions and the modifications of the rules $(T_{app}^{\langle\rangle})$ and $(T_\lambda^{\langle\rangle})$ in the way we just outlined. In particular, the example we discussed above, $\lambda^{\{\mathtt{Int}\to\mathtt{Int}\to\mathtt{Int}\}}x.\langle ?\rangle x$, is not well typed because $? \not\leq \mathtt{Int}\to\mathtt{Int}$.

The rule $(T_{cast}^{\langle\rangle})$ is straightforward and states that the expression $\langle\tau\rangle\, e$ has type τ, independently of the type of e. The reason for that is that if e converges, then either e will reduce to a value of type τ —and so the whole expression will— or a cast error will be raised: in either cases the expression will not be stuck.

The rule $(T_{app}^{\langle\rangle})$ is modified to ensure that the type on the left-hand side of an application is *always* a function type, that is, all of its concretizations are a subtype of $\mathbb{0} \to \mathbb{1}$. This translates

to the condition $\tau_1^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$. Moreover, we also want to ensure that the type of the argument is always accepted by the function, which amounts to saying that all its concretizations are in the safe domain of the function. Hence the condition $\tau_2^{\Uparrow} \leq \widetilde{dom}_{\gamma}(\tau_1)$.

Finally, rule $(T_\lambda^{\langle\rangle})$ returns the type of the cast embedded in the lambda abstraction and uses the stricter compatible subtyping relation $\sqsubseteq$ to check that the body of the abstraction is compatible with the one written in the interface.

As a result, this type system enjoys a substitution property with respect to the compatible subtyping, which is formalized in the following theorem.

Theorem 1. *(Substitution) For every terms of the cast language $e, e' \in \boldsymbol{Terms}^{\langle\rangle}$ and every typing context Γ, if $\Gamma, x : \sigma \vdash e : \tau$ and $\Gamma \vdash e' : \sigma'$ where $\sigma' \sqsubseteq \sigma$, then $\Gamma \vdash e[x := e'] : \tau'$ and $\tau' \sqsubseteq \tau$.*

Finally, notice that the inference rules of Figure 2 are deterministic: the system infers a unique type for every well-typed expression. This property is important since type inference is used in the next subsection to define the operational semantics which, thanks to this property, will be deterministic, too.

4.3 Operational Semantics

The small-step reduction rules for the cast language are presented in Figure 3 and are divided in four categories: cast, type-case, application, and context.

The definition of the reduction contexts at the bottom of the figure corresponds to a leftmost outermost weak reduction strategy which is applied by the two context reduction rules, (R_E) and (R_{E-fail}), which respectively propagate the computation and raise the cast errors at top-level.

The reduction rules for type casts applied to constants, (R_{cast-c}) and $(R_{cast-c-fail})$, are straightforward: if the type of the constant is a subtype of the target type, then the cast succeeds and the constant is returned; otherwise, the cast fails and the expression reduces to a cast error. The cast rules for lambda-abstractions $(R_{cast-\lambda})$ and $(R_{cast-\lambda-fail})$ look similar. However, there is an important difference, that is, that casts for lambda-abstractions are evaluated lazily at the moment of their application. So instead of performing an η-expansion, as it is customary in cast languages for gradual typing, the rule $(R_{cast-\lambda})$ "stores" the cast in the lambda-expression to evaluate it later. Note that if a lambda expression is preceded by multiple casts, then the rule $(R_{cast-\lambda})$ erases all of them, except for the last one. Indeed, storing all the successive casts of a function would introduce a chain of casts when computing the result of an application, where the only relevant cast would be the outermost one (as it represents the type that is expected by the rest of the program). Therefore, removing these casts preserves the soundness of the evaluation while reducing the number of possible cast errors. Notice that if we can already statically determine that a given cast will always fail —*ie*, because the cast type is not a consistent subtype of the interface—, then the $(R_{cast-\lambda-fail})$ rule raises outright the cast error.

The rule (R_{case-L}) is slightly less intuitive. If v is a value, then the semantics of the expression $((v \in t)?e_1 : e_2)$ in a language without gradual types consists in checking whether the type of v is a subtype of t (since, by subsumption, this implies that the value has type t) and executing e_1 if so, and e_2 otherwise. As we explained in Footnote 2, this test can be efficiently implemented by using the type information available at compile time. In the presence of gradual types, checking the subtyping relation between the type τ of v and t (*ie*, $\tau^{\Downarrow} \leq t$), although sound, is not the best choice for selecting e_1, since it might yield the insertion of casts that will always fail. Instead we have to select e_1 only if the type of v is *always* a subtype of τ, that is, whatever concretization its type will turn out to be (*ie*, $\tau^{\Uparrow} \leq t$). This is done by the rule (R_{case-L}) , while rule (R_{case-R}) corresponds to

Cast

$$\frac{B(c) \mathrel{\widetilde{\leq}} \tau}{\langle \tau \rangle\, c \mapsto c}\ (R_{cast-c}) \qquad \frac{\neg(B(c) \mathrel{\widetilde{\leq}} \tau)}{\langle \tau \rangle\, c \mapsto \texttt{CastError}}\ (R_{cast-c-fail})$$

$$\frac{\texttt{TypeOf}(\mathbb{I}) \mathrel{\widetilde{\leq}} \tau}{\langle \tau \rangle\, \lambda^{\mathbb{I}}_{\langle \tau' \rangle}\, x.\, e \mapsto \lambda^{\mathbb{I}}_{\langle \tau \rangle}\, x.\, e}\ (R_{cast-\lambda}) \qquad \frac{\neg(\texttt{TypeOf}(\mathbb{I}) \mathrel{\widetilde{\leq}} \tau)}{\langle \tau \rangle\, \lambda^{\mathbb{I}}_{\langle \tau' \rangle}\, x.\, e \mapsto \texttt{CastError}}\ (R_{cast-\lambda-fail})$$

Type-case

$$\frac{\emptyset \vdash v : \tau \quad \tau^{\Uparrow} \leq t}{((v \in t)?e_1 : e_2) \mapsto e_1}\ (R_{case-L}) \qquad \frac{\emptyset \vdash v : \tau \quad \tau^{\Uparrow} \not\leq t}{((v \in t)?e_1 : e_2) \mapsto e_2}\ (R_{case-R})$$

Application

$$\frac{\exists(\sigma_i \to \tau_i) \in \mathbb{I} \quad B(c) \mathrel{\widetilde{\leq}} \sigma_i}{(\lambda^{\mathbb{I}}_{\langle \tau \rangle}\, x.\, e)c \mapsto \langle \tau \mathrel{\widetilde{\circ}} B(c) \rangle\, e[x := \langle \sigma_i \rangle\, c]}\ (R_{app-c}) \qquad \frac{\nexists(\sigma_i \to \tau_i) \in \mathbb{I} \quad B(c) \leq \sigma_i^{\Uparrow}}{(\lambda^{\mathbb{I}}_{\langle \tau \rangle}\, x.\, e)c \mapsto \texttt{CastError}}\ (R_{app-c-fail})$$

$$\frac{\nexists(\sigma_i \to \tau_i) \in \mathbb{I} \quad B(c) \leq \sigma_i^{\Uparrow}}{(\lambda^{\mathbb{I}}_{\langle \tau \rangle}\, x.\, e)c \mapsto \texttt{CastError}}\ (R_{app-c-fail}) \qquad \frac{\nexists(\sigma_i \to \tau_i) \in \mathbb{I} \quad \texttt{TypeOf}(\mathbb{I}')^{\Downarrow} \leq \sigma_i^{\Uparrow}}{(\lambda^{\mathbb{I}}_{\langle \tau \rangle}\, x.\, e)(\lambda^{\mathbb{I}'}_{\langle \tau' \rangle}\, y.\, e') \mapsto \texttt{CastError}}\ (R_{app-\lambda-fail})$$

Context

$$\frac{e \mapsto e'}{E[e] \mapsto E[e']}\ (R_E) \qquad \frac{e \mapsto \texttt{CastError}}{E[e] \mapsto \texttt{CastError}}\ (R_{E-fail})$$

$$E ::= \square \mid Ee \mid vE \mid (E \in t)?e : e \mid \langle \tau \rangle\, E$$

Fig. 3. Small-step reduction semantics for the cast language

the other case. To understand the need of this stricter check consider the following expression

$$(\,(\,f = \lambda^{\{?\to?\}} x.(\,1 + \langle \texttt{Int} \rangle x\,) \in (\texttt{Bool} \to \texttt{Bool})\,)\ ?\ (f\ \texttt{true}) : \texttt{false}$$

that checks whether f is of type $\texttt{Bool} \to \texttt{Bool}$, applies it to `true` if this holds, and return `false` otherwise. In this term f is bound to a dynamically-typed version of the successor function, since it is given type **?→?**. This type is a consistent subtype of $\texttt{Bool} \to \texttt{Bool}$. However, since $(?\to?)^{\Uparrow} = \mathbb{0} \to \mathbb{1}$ is not a subtype of $\texttt{Bool} \to \texttt{Bool}$, then the reduction rule (R_{case-R}) applies and the second branch is selected, which is arguably the correct choice. Selecting the first branch, as the simpler subtyping check would do, would yield to a term that always fails since it would try to cast the argument of f, that is `true`, to `Int`.[13]

The rules (R_{app-*}) for application are the most difficult ones. First of all, remember that we evaluate function casts in a lazy way and, in particular, when the functions are applied. So the application rules perform two operations at once: the substitution corresponding to the beta

[13]Notice that for this example we used an extended syntax of type case in which the tested term is also bound to a variable. As explained in Section E in the Appendix of [Castagna et al. 2014] this extended syntax can be encoded in our syntax and provides a finer-grained typing for the type-case expression, similar to the occurrence typing of Tobin-Hochstadt and Felleisen [2008].

reduction, and the evaluation of the cast embedded in the function. The latter consists in η-expanding the function to apply the appropriate casts to its argument and to its result, and we perform it directly on the reductum of the beta-reduction. More precisely, when evaluating the expression $(\lambda^{\mathbb{I}}_{\langle\tau\rangle}\, x.\, e)v$ the lambda-expression expects an argument whose type is a subtype of its domain, not of $\widetilde{dom}(\tau)$ (which is the domain of the cast function). As such, x cannot simply be substituted by v in the expression e: an additional check must be performed on v. Hence, we look in the interface for an arrow $\sigma_i \to \tau_i$ whose input type σ_i is compatible with the type of v and we substitute for x the expression $\langle\sigma_i\rangle\, v$ (or its reductum). Two observations: first, this target type σ_i, if it exists, is unique thanks to the property that interfaces have pairwise disjoint input types. Second, if the argument is a lambda-abstraction, then the cast it contains is erased by the cast to the target type, and thus the former does not dictate how the argument can be used in the body of the function. Therefore, the choice of the target type should not depend on the cast contained by the argument (since it is erased) but on the interface contained in the argument, which explains why we had to split the application rule in two subcases, one for constants (which uses the type of the argument to select the target type) and the other for abstractions (which uses the interfaces of the argument to select the target type). Notice that the target type may also not exist (rules $R_{app-c-fail}$ and $R_{app-\lambda-fail}$), which corresponds to a case in which the η-expansion would (later) raise a cast error, and so this is what the rules ($R_{app-c-fail}$) and ($R_{app-\lambda-fail}$) do: for an example, consider an application such as $(\lambda^{\texttt{Int}\to\texttt{Int}}_{\langle ?\to\texttt{Int}\rangle} x.1 + x)\ true$ that is well-typed —the function is cast to $?\to \texttt{Int}$ and thus it can be applied to any argument— but obviously reduces to a cast error, since there is no domain in the interface that is compatible with the type of the argument.

4.4 Soundness.

To prove the soundness of the cast language, we prove the usual lemmas, starting with subject reduction. Since we defined the compatible subtyping relation to be preserved during the evaluation and to enjoy a substitution property (see Theorem 1), the subject reduction lemma can be directly worded as follows.

Lemma 1. *(Subject Reduction) — For every terms $e_1, e_2 \in \boldsymbol{Terms}^{\langle\rangle}$ and every typing context Γ, if $e_1 \mapsto e_2$ and $\Gamma \vdash e_1 : \tau_1$, then $\Gamma \vdash e_2 : \tau_2$ and $\tau_2 \sqsubseteq \tau_1$.*

We then prove the progress lemma, stating that every well-typed term that is not a value can be reduced. In our case, it can be reduced either to another term or to a cast error.

Lemma 2. *(Progress) — For every term $e \in \boldsymbol{Terms}^{\langle\rangle}$, if $\emptyset \vdash e : \tau$ then $e \in \boldsymbol{Values}^{\langle\rangle}$ or $\exists e' \in \boldsymbol{Terms}^{\langle\rangle}, e \mapsto e'$ or $e \mapsto \texttt{CastError}$.*

Finally, the soundness of the cast language is a direct consequence of the two previous lemmas.

Theorem 2. *(Soundness) — For every term $e \in \boldsymbol{Terms}^{\langle\rangle}$, if $\emptyset \vdash e : \tau$ then either e diverges or $\exists v \in \boldsymbol{Values}^{\langle\rangle}$ such that $e \mapsto^* v$ or $e \mapsto^* \texttt{CastError}$.*

This soundness property will allow us to prove, in the next section, that the dynamic semantics of the gradually-typed lambda calculus resulting from its compilation to the cast language is sound with respect to the typing rules presented in Section 3.

5 COMPILATION

In this section, we define the compilation procedure of the gradually-typed lambda calculus to the cast calculus. By proving that the compilation rules map well-typed terms of the gradually-typed calculus into well-typed terms of the cast calculus and using the soundness of the latter, we can prove that the execution of compiled well-typed gradually-typed terms is sound.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x \rightsquigarrow x : \tau}\ (C_x) \qquad \frac{\Gamma \vdash e \rightsquigarrow e' : \tau \quad \begin{cases} \tau \mathrel{\widetilde{\not\leq}} \neg t & \Longrightarrow \Gamma \vdash e_1 \rightsquigarrow e_1' : \sigma_1 \\ \tau \mathrel{\widetilde{\not\leq}} t & \Longrightarrow \Gamma \vdash e_2 \rightsquigarrow e_2' : \sigma_2 \end{cases}}{\Gamma \vdash ((e \in t)?e_1 : e_2) \rightsquigarrow ((e' \in t)?e_1' : e_2') : \sigma_1 \vee \sigma_2}\ (C_{case})$$

$$\frac{}{\Gamma \vdash c \rightsquigarrow c : B(c)}\ (C_c) \qquad \frac{\begin{array}{ll} \Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1 & \tau_2^{\Uparrow} \leq \widetilde{dom}_{\mathscr{S}}(\tau_1) \\ \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_2 & \tau_1^{\Uparrow} \leq \mathbb{0} \to \mathbb{1} \end{array}}{\Gamma \vdash e_1\ e_2 \rightsquigarrow e_1'\ e_2' : \tau_1 \widetilde{\circ} \tau_2}\ (C_{app-1})$$

$$\frac{\begin{array}{ll} \Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1 & \tau_2^{\Uparrow} \not\leq \widetilde{dom}_{\mathscr{S}}(\tau_1) \\ \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_2 & \tau_2^{\Downarrow} \leq \widetilde{dom}_{\mathscr{S}}(\tau_1) \quad \tau_1^{\Uparrow} \leq \mathbb{0} \to \mathbb{1} \end{array}}{\Gamma \vdash e_1\ e_2 \rightsquigarrow \langle \tau_1 \widetilde{\circ} \tau_2 \rangle\ (e_1'\ \langle \widetilde{dom}_{\mathscr{S}}(\tau_1) \rangle\ e_2') : \tau_1 \widetilde{\circ} \tau_2}\ (C_{app-2})$$

$$\frac{\begin{array}{ll} \Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1 & \tau_1^{\Uparrow} \not\leq \mathbb{0} \to \mathbb{1} \quad \text{or} \quad \tau_2^{\Downarrow} \not\leq \widetilde{dom}_{\mathscr{S}}(\tau_1) \\ \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_2 & \end{array}}{\Gamma \vdash e_1\ e_2 \rightsquigarrow (\langle \tau_2 \to (\tau_1 \widetilde{\circ} \tau_2) \rangle\ e_1')\ e_2' : \tau_1 \widetilde{\circ} \tau_2}\ (C_{app-3})$$

$$\frac{\begin{array}{ll} \forall \sigma_i \to \tau_i \in \mathbb{I}, & e_i' = \begin{cases} e_i & \text{if } \tau_i' \sqsubseteq \tau_i \\ \langle \tau_i \rangle\ e_i & \text{otherwise} \end{cases} \\ \Gamma, x : \sigma_i \vdash e \rightsquigarrow e_i : \tau_i' & \end{array}}{\Gamma \vdash \lambda^{\mathbb{I}} x.\ e \rightsquigarrow (\lambda^{\mathbb{I}} x.(x \in \sigma_1^{\Uparrow})?\ e_1' : \cdots : (x \in \sigma_{i-1}^{\Uparrow})?\ e_{i-1}' : e_i') : \mathtt{TypeOf}(\mathbb{I})}\ (C_\lambda)$$

Fig. 4. Compilation rules for the gradually-typed language

5.1 Compilation rules

The compilation rules for the gradually-typed language are presented in Figure 4. They are directed by the type system of the cast language, since we want to translate terms of the gradually-typed lambda calculus to well-typed terms of the cast language. For this reason the judgments in the rules have the form $\Gamma \vdash e \rightsquigarrow e' : \tau$, meaning that under the typing hypothesis Γ the term e of the gradually-typed language is translated into the term e' of the cast language and that the latter has type τ.

The compilation rules for constants (C_c), variables (C_x) and typecases (C_{case}) are straightforward: the first two are the identity translation while the last one is just the component-wise translation of all sub-expressions.

There are three rules to compile applications, denoted by (C_{app-i}). This comes from the fact that there are three possible ways to compile an application: either (*i*) the application can be compiled "as is", or (*ii*) the right-hand side of the application needs to be cast to a type compatible with the type of the function, or (*iii*) the left-hand side needs to be cast to a function type compatible with the type of the argument.

The first case corresponds to the rule (C_{app-1}) and is a direct consequence of the typing rules of the cast language. That is, if the two sides of the application compile respectively to e_1' and e_2' such that $e_1'\ e_2'$ is already a well-typed term of the cast language, then no cast is needed.

Otherwise, given an application $e_1\ e_2$ whose two sides respectively compile into $e_1' : \tau_1$ and $e_2' : \tau_2$ we have to decide whether to cast the function e_1' (to an arrow whose domain is τ_2) or its argument

e_2' (to the safe domain of τ_1). There is one case for which no doubt is possible, namely when τ_1 is *not* a function type (*ie*, $\tau_1^{\Uparrow} \not\leq \mathbb{0} \to \mathbb{1}$). In this case we must dynamically verify that e_1' returns a lambda-abstraction and therefore we apply the cast to the function. This is done by the rule (C_{app-3}). If instead τ_1 is a function type, then we have to ensure that, at runtime, the type of the function is compatible with the type of the argument. Unfortunately there does not exist a recipe that works in all circumstances: it is not difficult to find two types τ_1 and τ_2 in which neither the choice of casting the argument nor the choice of casting the function is always better.[14] Note however that if we cast the function, then we do it to an arrow type, while the cast of the argument does not impose any particular constraint on the type in the cast. Thus casting the function looks like a stronger requirement than casting the argument: by casting the function to an arrow we loose all the information given by the use of connectives in its type and we flatten its domain and its safe domain into a unique type. This is the reason why, in our compilation rules, we privilege to cast the argument to the safe domain of the function, but only if such a cast has a chance to succeed, that is, only if the type of the argument is a consistent subtype of the safe domain (*ie*, $\tau_2^{\Downarrow} \leq \widetilde{dom}_{\diamond}(\tau_1)$): this is done by rule (C_{app-2}). The condition $\tau_2^{\Downarrow} \leq \widetilde{dom}_{\diamond}(\tau_1)$ ensures that we will not try to apply to the argument a cast that is statically known to always fail: for instance, if we apply a function of type $(\mathtt{Int} \to \mathtt{Int}) \wedge \mathtt{?}$ to an argument of type `Bool` it would be completely useless to cast this argument to `Int`, the safe domain of the function, since the cast would always fail. Thus, in this case, even though e_1' has a functional type, we will instead cast the function to the type $\mathtt{Bool} \to \mathtt{?}$. This last case is handled by the rule (C_{app-3}): if $\tau_2^{\Downarrow} \not\leq \widetilde{dom}_{\diamond}(\tau_1)$, then the rule ($C_{app-3}$) is used again, and the cast is applied to the function. Since $\tau_2 \to (\tau_1 \widetilde{\circ} \tau_2) \mathrel{\widetilde{\leq}} \tau_1$, we statically know that this cast may succeed.

Finally, there remains the translation rule for lambda abstractions. Here the difficulty comes from the use of interfaces and, more generally, of intersection types. When a lambda abstraction is given an interface containing more than one type —i.e., it is typed by an intersection type—, then the function is type-checked several times, once for each type in the interface. The compilation is driven by the type system and during the typing-checking of the expression, casts are inserted to ensure soundness. The consequence of type-checking the same expression several times is that each time different casts may be added, and these may be incompatible from one pass to the other. Let us show this by an example: consider the following identity function with a bizarre but correct interface.

$$\lambda^{\{\mathtt{Int}\to\mathtt{Int}\,;\,(\mathtt{?}\setminus\mathtt{Int})\to\mathtt{Bool}\}} x.x$$

The interface states that when the function is applied to an integer then the application returns an integer and when it is applied to any other argument then it returns a Boolean, provided it does not fail. How should we compile this function? If the function is applied to an integer, then no cast is necessary, while for all other arguments the body needs to be cast to `Bool`. But since this cast would be incorrect with an integer argument, the question arises whether we must insert this cast or not. The solution is to compile the body of the function by a type-case that distinguishes the two argument configurations, that is:

$$\lambda^{\{\mathtt{Int}\to\mathtt{Int}\,;\,(\mathtt{?}\setminus\mathtt{Int})\to\mathtt{Bool}\}} x.(x{\in}\mathtt{Int})\ \mathtt{?}\ x : \langle \mathtt{Bool} \rangle\, x \tag{4}$$

the body of the function is duplicated in all the branches but the cast to `Bool` is applied only if the argument is not of type `Int`.

[14] Consider $\tau_1 = ((\mathtt{Bool} \vee \mathtt{?}) \to \mathtt{?}) \vee ((\mathtt{Int} \to \mathtt{Int}) \wedge \mathtt{?}))$ and $\tau_2 = \mathtt{Bool} \vee \mathtt{?}$. The application is well-typed. However, if e_1' is the successor function and e_2' is an integer, then casting the function (to $\mathtt{Bool} \vee \mathtt{?} \to \mathtt{?}$) would fail while casting the argument (to the safe domain `Int`) would not; taking instead for e_1' and e_2' the Boolean "not" function and true would fail for a cast on the argument and succeed if the cast is on the function.

This is exactly what the (C_λ) rule does. In particular it transforms the body of a lambda-abstraction into a type case with as many branches as arrows in the interface, and puts into each branch the translation of the body obtained under the hypothesis that the parameter of the function has the input type of the arrow under consideration (the type case is not inserted if the interface is composed of a single arrow). Remember that, given an abstraction $\lambda^{\mathbb{I}} x.\, e$, its interface $\mathbb{I} = \{\sigma_i \to \tau_i \mid i \in I\}$ verifies $\forall i, j \in I,\ i \neq j \implies \sigma_i^{\Uparrow} \wedge \sigma_j^{\Uparrow} \leq \mathbb{0}$ and its domain is $\widetilde{dom}(\mathtt{TypeOf}(\mathbb{I})) = \bigvee_{i \in I} \sigma_i^{\Uparrow}$. This means that the different cases of the typecase in the translation cover the whole domain of the function and are not overlapping. Thus, the order in which they are considered does not matter. As a consequence, for every value of type $\sigma \mathrel{\widetilde{\leq}} \widetilde{dom}(\mathtt{TypeOf}(\mathbb{I}))$, there exists a unique $i \in I$ such that $\sigma^{\Uparrow} \leq \sigma_i^{\Uparrow}$. This is the essence of the rule (C_λ): since the compilation of the body e of the abstraction depends on the type of x, it checks the variable x against every possible type $\sigma_i^{\Uparrow}$ and branches to the corresponding compiled expression e_i (where $e \rightsquigarrow e_i$ under the hypothesis that x has type σ_i). Finally, to ensure that the compiled function is well-typed (in the type system of the cast language) with respect to its interface, the rule also add casts to branches that require it. For instance, in our example in (4) no cast is inserted in the first branch (since if x : `Int` then the body has type `Int` which is exactly the type in the interface) but a cast is inserted in the second (since x, and thus the body, has type `? \Int` and `? \Int` $\not\leq$ `Bool`).

5.2 Safety and Soundness

The compilation procedure compiles gradually-typed terms to terms of the cast language, and thus implicitly defines the reduction semantics of the gradually-typed language. The safety property for this language states that the compilation of every well-typed term reduces to a value, a cast error, or diverges. However, for this statement to be meaningful, the compilation must be exhaustive; that is, it should be possible to compile every well-typed term of the gradually-typed language. This is formally stated as follows.

Lemma 3. *(Exhaustiveness of Compilation) — For every term $e \in \boldsymbol{Terms}$ and every typing context Γ, if $\Gamma \vdash e : \tau$ then $\Gamma \vdash e \rightsquigarrow e' : \tau$ where $e' \in \boldsymbol{Terms}^{\langle\rangle}$.*

Moreover, we can prove that the compilation procedure preserves the type of the compiled term.

Lemma 4. *(Type Preservation by Compilation) — For every term $e \in \boldsymbol{Terms}$ and every typing context Γ, if $\Gamma \vdash e \rightsquigarrow e' : \tau$ then $\Gamma \vdash e' : \tau$.*

These two lemmas can be summarized as the following theorem, proving the soundness of the compilation procedure with respect to both type systems.

Theorem 3. *(Soundness of Compilation) — For every term $e \in \boldsymbol{Terms}$ and every typing context Γ, if $\Gamma \vdash e : \tau$ then $\Gamma \vdash e \rightsquigarrow e' : \tau$, where $e' \in \boldsymbol{Terms}^{\langle\rangle}$ and $\Gamma \vdash e' : \tau$.*

The safety property of the cast language stated in Theorem 2 can then be "lifted" to the gradually-typed lambda calculus.

Corollary 1. *(Safety of the Gradually-Typed Language) — For every term $e \in \boldsymbol{Terms}$, if $\emptyset \vdash e : \tau$, then $e \rightsquigarrow e' : \tau$ where $e' \in \boldsymbol{Terms}^{\langle\rangle}$ and either e' diverges, or $\exists v \in \boldsymbol{Values}^{\langle\rangle}$ such that $e' \mapsto^* v$ and $\emptyset \vdash v : \tau' \mathrel{\widetilde{\leq}} \tau$, or $e' \mapsto^* \mathtt{CastError}$.*

6 CONCLUSION

In this work, we presented a foundational study for gradually-typed languages with union and intersection types and argued that the combination of the former with the latter allows a smoother and finer-grained transition between static and dynamic typing, than what is possible with the

current state of the art. As the use of the unknown type "?" allows the programmer to gradually add type information in selected parts of the program, so the use of unions and intersections allows the same programmer to gradually refine the type information about a given part of the program or a single expression, thus enabling a new style of gradual programming. While adding unions and intersections *inside arrows* brings real benefits at no cost, we have also seen that for adding intersections *on arrows* there is instead a price to pay: intersections of arrows allow the programmer to write overloaded functions, but when these are combined with gradual types their code needs to be specialized to insert casts that depend on the type of the argument. This happens both in the semantics of the cast language (where the cast to apply to the argument is different in the case of interfaces with multiple arrows) and in the compilation rules (where the body of a λ-abstraction is compiled into a type-case when the interface contains multiple arrows). Further research is needed to study how to alleviate this cost.

The language presented in this work is minimalist and if we want our study to scale to real world functional languages, then it is necessary to extend it in several directions, foremost by adding advanced features, such as pattern matching, type recursion, and new type constructors (in particular, products and records). Equally important will be the addition of polymorphic types, an addition that we plan to pursue by drawing ideas from both the work on polymorphic set-theoretic types by Castagna et al. [2015, 2014] and the work on polymorphic gradual types by Siek and Vachharajani [2008] and Garcia and Cimini [2015]. This work will be followed by the study of some forms of type reconstruction and local type inference on the basis of what was already done on set-theoretic types by Castagna et al. [2016]. Only then we will be able to see whether and how the techniques whose study started here apply to languages such as JavaScript and how they compare and/or whether they may bring any contribution to approaches such as the one of the language Flow that we described in Section 1.3 on related work.

From a more theoretical point of view, we saw that using type connectives increases the discretionary of adding casts in an application. Our choice, embodied by the compilation rule (C_{app-2}), was to push casts on the argument part of an application as much as possible. However, it may be worth exploring whether by a "threesome" approach as proposed by Siek and Wadler [2010] one could find more expressive intermediate solutions. Another property that may also be worth verifying for our system is the *gradual guarantee* [Siek et al. 2015], which states that a program that runs without errors still does with less precise type annotations.

We completely disregarded error-message generation, both at static and at dynamic time. In particular, we did not considered the so-called blame. As pointed out by an anonymous referee, a "blame theorem", as presented by Wadler and Findler [2009], is not worth proving for our calculus, since the wrong casts might be blamed. The reason for that lies in the fact that we chose to collapse casts on functions (see rule $R_{cast-\lambda}$ in Figure 3). While this choice makes the calculus simpler without hindering soundness, it yields a formalism unfit to finger culprits. In particular, complete monitoring [Dimoulas et al. 2012] fails for a calculus that omit casts in this way since it cannot keep track of every owner of a term (see Dimoulas et al. [2012] for a general discussion). We are currently studying a different cast language that will make the study of blame and error-message generation possible, so that whenever the execution of a program results in a cast error, then it is the dynamically-typed part of the program that must be blamed for that. Finally, we would like to explore how this study applies when one tries to extend it with generalized abstract data types.

ACKNOWLEDGMENTS

The authors would like to thank Tommaso Petrucciani, Jeremy Siek, Éric Tanter, and the anonymous reviewers for their remarks and suggestions which allowed us to significantly improve this work.

REFERENCES

Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. 2003. CDuce: an XML-centric general-purpose language. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP '03)*. ACM, 51–63.

Giuseppe Castagna. 2015. Covariance and Contravariance: a fresh look at an old issue (a primer in advanced type systems for learning functional programmers). (2015). Unpublished manuscript, available at the author's web page.

Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. 1995. A calculus for overloaded functions with subtyping. *Information and Computation* 117, 1 (1995), 115–135.

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, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. 2014. Polymorphic Functions with Set-Theoretic Types. Part 1: Syntax, Semantics, and Evaluation. In *Proceedings of the 41st ACM Symposium on Principles of Programming Languages (POPL '14)*. ACM, 5–17.

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 '16)*. ACM, 378–391.

Avik Chaudhuri. 2014. *Flow: A static type checker for JavaScript*. Facebook. https://flowtype.org.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*. ACM, 238–252.

Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete Monitors for Behavioral Contracts. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP'12*. 214–233.

Alain Frisch. 2004. Regular Tree Language Recognition with Static Information. In *IFIP 18th World Computer Congress TC1, 3rd International Conference on Theoretical Computer Science (TCS2004) (IFIP)*, Vol. 155. Kluwer/Springer, 661–674.

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.

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.

Khurram A. Jafery and Joshua Dunfield. 2017. Sums of Uncertainty: Refinements Go Gradual. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. ACM, 804–817.

Nico Lehmann and Éric Tanter. 2017. Gradual refinement types. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. ACM, 18–20.

John C. Reynolds. 1996. *Design of the Programming Language Forsythe*. Technical Report CMU-CS-96-146. Carnegie Mellon University.

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 Sam Tobin-Hochstadt. 2016. The Recursive Union of Some Gradual Types. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella (Eds.). Springer, 388–410.

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

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

Jeremy G. Siek and Philip Wadler. 2010. Threesomes, with and Without Blame. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, 365–376.

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.

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, 395–406.

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can'T Be Blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems (LNCS)*. Springer, 1–16.

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994), 38 – 94.

A APPENDIX

In this appendix, we present full definitions of the language and type system we studied throughout this paper, along with complete proofs of all the results.

A.1 Gradual Types

A.1.1 Extreme Concretizations and Subtyping. For every gradual type τ, we define its maximal concretization by substituting every positive occurrence of ? by $\mathbb{1}$ and every negative occurrence of ? by $\mathbb{0}$. We do the opposite to define its minimal concretization. The following definition formalizes this.

Definition 12. *(Extreme Concretizations) For every type $\tau \in$ GTypes, we mutually define the maximal (resp. minimal) concretization of τ, noted $\tau^{\Uparrow}$ (resp. $\tau^{\Downarrow}$) as follows:*

$$
\begin{array}{cc}
.^{\Uparrow} : \text{GTypes} \to \text{STypes} & .^{\Downarrow} : \text{GTypes} \to \text{STypes} \\
?^{\Uparrow} = \mathbb{1} & ?^{\Downarrow} = \mathbb{0} \\
(\tau_1 \vee \tau_2)^{\Uparrow} = \tau_1^{\Uparrow} \vee \tau_2^{\Uparrow} & (\tau_1 \vee \tau_2)^{\Downarrow} = \tau_1^{\Downarrow} \vee \tau_2^{\Downarrow} \\
(\tau_1 \wedge \tau_2)^{\Uparrow} = \tau_1^{\Uparrow} \wedge \tau_2^{\Uparrow} & (\tau_1 \wedge \tau_2)^{\Downarrow} = \tau_1^{\Downarrow} \wedge \tau_2^{\Downarrow} \\
(\sigma \to \tau)^{\Uparrow} = \sigma^{\Downarrow} \to \tau^{\Uparrow} & (\sigma \to \tau)^{\Downarrow} = \sigma^{\Uparrow} \to \tau^{\Downarrow} \\
t^{\Uparrow} = t & t^{\Downarrow} = t
\end{array}
$$

Based on this definition, we can now formalize the intuition that the extreme concretizations of a gradual type τ belong to its concretization $\gamma(\tau)$ and are the extreme elements of this set.

Theorem 4. *(Gradual Extrema) For every type $\tau \in$ GTypes, the following properties hold:*
(1) $\forall t \in \gamma(\tau), t \leq \tau^{\Uparrow}$
(2) $\forall t \in \gamma(\tau), \tau^{\Downarrow} \leq t$
(3) $\tau^{\Uparrow} \in \gamma(\tau)$
(4) $\tau^{\Downarrow} \in \gamma(\tau)$

Proof. Let $\tau \in$ GTypes. The four properties must be proved all at once, by induction over the type τ. We will only prove properties (1) and (3), as properties (2) and (4) are extremely similar.

- $\tau = \tau_1 \vee \tau_2$. By definition of the maximal concretization, we have $\tau^{\Uparrow} = \tau_1^{\Uparrow} \vee \tau_2^{\Uparrow}$.
  By induction hypothesis, it holds that $\tau_1^{\Uparrow} \in \gamma(\tau_1)$ and $\tau_2^{\Uparrow} \in \gamma(\tau_2)$. And, by definition of γ:
  $$\gamma(\tau) = \{t_1 \vee t_2 \mid t_1 \in \gamma(\tau_1), t_2 \in \gamma(\tau_2)\}$$
  Therefore, $\tau^{\Uparrow} \in \gamma(\tau)$, which proves property (3).
  Now, let t be any type of $\gamma(\tau)$. By definition of γ, there exist two types $t_1 \in \gamma(\tau_1)$ and $t_2 \in \gamma(\tau_2)$ such that $t = t_1 \vee t_2$. By induction hypothesis, it holds that $t_1 \leq \tau_1^{\Uparrow}$ and $t_2 \leq \tau_2^{\Uparrow}$. Since the set-theoretic union is monotonic with respect to set containment (a.k.a. subtyping), we obtain that $t_1 \vee t_2 \leq \tau_1^{\Uparrow} \vee \tau_2^{\Uparrow}$. This proves property (1).
- $\tau = \tau_1 \wedge \tau_2$. This case is similar to the previous one and is proved in the same way, since the set-theoretic intersection is also monotonic with respect to set containment.
- $\tau = \mu \to \sigma$. By definition of the maximal concretization, we have $\tau^{\Uparrow} = \mu^{\Downarrow} \to \sigma^{\Uparrow}$.
  By induction hypothesis, it holds that $\mu^{\Downarrow} \in \gamma(\mu)$ and $\sigma^{\Uparrow} \in \gamma(\sigma)$. And, by definition of γ:
  $$\gamma(\tau) = \{u \to s \mid u \in \gamma(\mu), s \in \gamma(\sigma)\}$$
  Therefore, $\tau^{\Uparrow} \in \gamma(\tau)$, which proves property (3).

Let t be any type of $\gamma(\tau)$. By definition of γ, there exist two types $u \in \gamma(\mu)$ and $s \in \gamma(\sigma)$ such that $t = u \to s$. Moreover, by induction hypothesis, it holds that $\mu^{\Downarrow} \leq u$ and $s \leq \sigma^{\Uparrow}$. Hence the following inequalities:

$$\begin{aligned} t = u \to s &\leq u \to \sigma^{\Uparrow} && \text{By covariance of the codomain} \\ &\leq \mu^{\Downarrow} \to \sigma^{\Uparrow} && \text{By contravariance of the domain} \\ &= \tau^{\Uparrow} \end{aligned}$$

This proves property (1).

- $\tau = ?$. By definition of the maximal concretization, we have $?^{\Uparrow} = \mathbb{1}$. Since $\gamma(?) = \text{STypes}$, it is obvious that $?^{\Uparrow} \in \gamma(?)$, hence property (3).
  Moreover, for every static type t, it holds that $t \leq \mathbb{1}$, hence property (1).
- $\tau \in \text{STypes}$. This case is trivial given that, for every static type t, $t^{\Uparrow} = t$, and $\gamma(t) = \{t\}$. Hence, it is obvious that $t \leq t^{\Uparrow}$ and that $t^{\Uparrow} \in \gamma(t)$. This last case concludes the proof by induction.

□

Definition 13. *(Subtyping of Gradual Types) Using the concretization function γ, subtyping is defined on gradual types as follows:*

$$\forall(\sigma, \tau) \in \text{GTypes}^2, \sigma \mathrel{\widetilde{\leq}} \tau \iff \exists (s, t) \in \gamma(\sigma) \times \gamma(\tau), s \leq t$$

Proposition 3. *(Reduction of Consistent Subtyping) Deciding consistent subtyping is equivalent to deciding static subtyping. In particular, it holds that:*

$$\forall(\sigma, \tau) \in \text{GTypes}^2, \sigma \mathrel{\widetilde{\leq}} \tau \iff \sigma^{\Downarrow} \leq \tau^{\Uparrow}$$

Proof. Let σ, τ be two gradual types.

- Assume that $\sigma \mathrel{\widetilde{\leq}} \tau$. By definition of $\mathrel{\widetilde{\leq}}$, there exists two static types $(s, t) \in \gamma(\sigma) \times \gamma(\tau)$ such that $s \leq t$.
  By Theorem 4, it holds that $\sigma^{\Downarrow} \leq s$ and $t \leq \tau^{\Uparrow}$. By transitivity of static subtyping, we deduce that $\sigma^{\Downarrow} \leq \tau^{\Uparrow}$.
- Now, assume that $\sigma^{\Downarrow} \leq \tau^{\Uparrow}$. According to Theorem 4, it holds that $\sigma^{\Downarrow} \in \gamma(\sigma)$ and $\tau^{\Uparrow} \in \gamma(\tau)$. Hence the proposition.

□

Definition 14. *(Equivalence of Gradual Types) We define the equivalence relation $\cong$ on gradual types such that for all types $\sigma, \tau \in \text{GTypes}$,*

$$\sigma \cong \tau \iff \begin{cases} \sigma^{\Downarrow} \simeq \tau^{\Downarrow} \\ \sigma^{\Uparrow} \simeq \tau^{\Uparrow} \end{cases}$$

where $\simeq$ denotes the equivalence of two static types (subtyping wise).

A.1.2 Static Domains and Results. When considering non-gradual, set-theoretic types, the first step to define the domain and result type of a function is usually to define a disjunctive normal form for types.

Definition 15. *(Static Disjunctive Normal Form) A static type t is said to be in disjunctive normal form (DNF) if it is of the form:*

$$t \equiv \bigvee_{i \in I} \bigwedge_{j \in J_i} l_j$$

where l_j are static literals, that is, static atoms or negation of static atoms:

$$l ::= a \mid \neg a$$

and atoms are defined as:

$$a ::= s \to t \mid \mathbb{0} \mid \mathbb{1} \mid b$$

It is possible to go even further by defining a uniform disjunctive normal form, that is, a normal form that does not contain heterogeneous intersections (ie. intersection of an arrow and a base type).

Definition 16. *(Static Uniform Disjunctive Normal Form) A static type t is said to be in uniform disjunctive normal form (uDNF) if it is of the form:*

$$t \equiv \bigvee_{f\in F}\bigwedge_{j\in J_f} s_j \to t_j \wedge \bigwedge_{n\in N_f} \neg(s_n \to t_n) \\ \bigvee_{p\in P}\bigwedge_{j\in J_p} b_j \qquad \wedge \bigwedge_{n\in N_p} \neg b_n$$

Having defined this, it is then possible to show that every type is equivalent (w.r.t. static subtyping) to a type in uniform disjunctive normal form.

Proposition 4. *(Equivalence to Static uDNF) Every static type t is equivalent to a type in uniform disjunctive normal form:*

$$t \simeq \bigvee_{f\in F}\bigwedge_{j\in J_f} s_j \to t_j \wedge \bigwedge_{n\in N_f} \neg(s_n \to t_n) \\ \bigvee_{p\in P}\bigwedge_{j\in J_p} b_j \qquad \wedge \bigwedge_{n\in N_p} \neg b_n$$

In particular, if t is a function type (that is, $t \leq \mathbb{0} \to \mathbb{1}$), then

$$t \simeq \bigvee_{f\in F}\bigwedge_{j\in J_f} s_j \to t_j \wedge \bigwedge_{n\in N_f} \neg(s_n \to t_n)$$

The domain and result type of a function type are then defined on disjunctive normal forms as follows.

Definition 17. *(Static Domain) For every static type $t \leq \mathbb{0} \to \mathbb{1}$ in uDNF, we define the domain of t as follows, using the notations of Definition 16:*

$$dom(t) = \bigwedge_{f\in F}\bigvee_{j\in J_f} s_j$$

Definition 18. *(Static Result) For every static type $t \leq \mathbb{0} \to \mathbb{1}$ in uDNF, and every static type s, we define the result type of the application of t to s as follows, using the notations of Definition 16:*

$$t \circ s = \bigvee_{f\in F} \bigvee_{\substack{Q\subsetneq J_f \\ s\not\leq \bigvee_{q\in Q} s_q}} \bigwedge_{p\in J_f\setminus Q} t_p$$

Since every static type is equivalent to a type in disjunctive normal form (according to Proposition 4), the set of static types can be partitioned into equivalence classes for $\simeq$ such that every class contains *at least* one type in disjunctive normal form. Fortunately, the definitions of the domain and result are invariant by equivalence (that is, if two normal forms are equivalent for $\simeq$, then they have the same domain and result). Thus, it is possible to define the domain and result on every equivalence class (that is, for every static type), independently of the choice of the representative in disjunctive normal form.

A.1.3 Applicative Concretization. Unfortunately, the same reasoning cannot be applied directly to gradual types, without a suitable equivalence relation. Indeed, the subtyping-based equivalence presented in Definition 14 is not precise enough. For example, consider the two following gradual types, that can be considered to be in disjunctive normal form:

$$\tau = ? \wedge ((\texttt{Int} \wedge ?) \to \texttt{Bool})$$
$$\sigma = ? \wedge ((\texttt{Bool} \wedge ?) \to \texttt{Bool})$$

It holds that $\tau \cong \sigma$. However, they are not *intuitively* the same. In particular, applying the first one to an argument of type `Int` returns a value of type $\texttt{Bool} \vee (\texttt{Bool} \wedge ?) \cong \texttt{Bool}$ whereas applying the second one to the same argument returns a value of type ?.

Hence the need for a stronger equivalence relation that encompasses the properties of function types. The solution we propose consists in defining new concretization functions that correspond to the transformation of a type into its disjunctive normal form. This leads to the definition of the applicative concretization given in Definition 4.

A.1.4 Gradual Domains and Results. Using the intuition that $\gamma^+_{\mathscr{D}}(\tau)$ represents a disjunctive normal form of τ, we can now define the domain and the result type of a gradual type by analogy with static types. The definition of the gradual domain, presented in Definition 5 is simply the gradual equivalent of its static counterpart, presented in Definition 17. Unfortunately, lifting the result type operator is not as easy, due to the criterion $s \not\leq \bigvee_{q \in Q} s_q$ that appears in Definition 18. This requires lifting the predicate $\not\leq$, which is not the same as negating the predicate $\widetilde{\leq}$. Indeed, by definition of predicate lifting operations, we have for every gradual types τ and σ,

$$\tau \widetilde{\not\leq} \sigma \iff \exists (t, s) \in \gamma(\tau) \times \gamma(\sigma), t \not\leq s$$

whereas the negation of $\widetilde{\leq}$ is defined by

$$\neg(\tau \mathrel{\widetilde{\leq}} \sigma) \iff \forall (t, s) \in \gamma(\tau) \times \gamma(\sigma), t \not\leq s$$

Thankfully, we can once again give a simple definition of the predicate $\widetilde{\not\leq}$, using the same technique as for $\widetilde{\leq}$:

$$\begin{aligned} \forall \tau, \sigma \in \text{GTypes}, \tau \mathrel{\widetilde{\not\leq}} \sigma &\iff \exists (t, s) \in \gamma(\tau) \times \gamma(\sigma), t \not\leq s \\ &\iff \tau^{\Uparrow} \not\leq \sigma^{\Downarrow} \end{aligned}$$

Thus, the gradual lifting of the criterion $s \not\leq \bigvee_{q \in Q} s_q$ is $\sigma^{\Uparrow} \not\leq \left(\bigvee_{(\rho \to \rho') \in Q} \rho\right)^{\Downarrow}$. Hence the definition of the gradual result given in Definition 5.

A.1.5 Gradual Disjunctive Normal Forms. In this section, we give some *a posteriori* intuition for the applicative concretization functions. We first define a rewriting system that rewrites any gradual type into an equivalent disjunctive normal form (with respect to $\cong$), prove its termination, and show that the applicative concretization is preserved by rewriting. Thus, any operator defined on

disjunctive normal forms (such as the domain or the result) can also be defined using the applicative concretization. We start by giving the definition of the rewriting system.

Definition 19. *(DNF Rewriting System) — We define the following rewriting system, noted $\mathscr{R}_{DNF}$, on gradual types:*

$$
\begin{aligned}
&\mathscr{R}_{DNF} : \text{GTypes} \to \text{GTypes} \\
&\quad \sigma \wedge (\tau_1 \vee \tau_2) \mapsto_{\mathscr{R}} (\sigma \wedge \tau_1) \vee (\sigma \wedge \tau_2) && (\mathscr{R}^{1a}_{DNF}) \\
&\quad (\tau_1 \vee \tau_2) \wedge \sigma \mapsto_{\mathscr{R}} (\tau_1 \wedge \sigma) \vee (\tau_2 \wedge \sigma) && (\mathscr{R}^{1b}_{DNF}) \\
&\quad \neg(\tau_1 \vee \tau_2) \mapsto_{\mathscr{R}} (\neg\tau_1) \wedge (\neg\tau_2) && (\mathscr{R}^{2}_{DNF}) \\
&\quad \neg(\tau_1 \wedge \tau_2) \mapsto_{\mathscr{R}} (\neg\tau_1) \vee (\neg\tau_2) && (\mathscr{R}^{3}_{DNF}) \\
&\quad \neg\neg\tau \mapsto_{\mathscr{R}} \tau && (\mathscr{R}^{4}_{DNF}) \\
&\quad E(\tau) \mapsto_{\mathscr{R}} E(\tau') \text{ if } \tau \mapsto_{\mathscr{R}} \tau' && (\mathscr{R}^{5}_{DNF})
\end{aligned}
$$

where the rewriting contexts are defined as:

$$E ::= \tau \vee E \mid E \vee \tau \mid \tau \wedge E \mid E \wedge \tau \mid \neg E \mid \square$$

We now verify that the rewriting system $\mathscr{R}_{DNF}$ preserves the previously defined relation $\approxeq$.

Proposition 5. *(Preservation of equivalence) — For every gradual types σ and τ, if $\sigma \mapsto^*_{\mathscr{R}} \tau$ then $\sigma \approxeq \tau$.*

Proof. Since $\approxeq$ is an equivalence relation (and is thus transitive), the proof is simply done by cases over a one-step reduction $\sigma \mapsto_{\mathscr{R}} \tau$ and is then generalized by transitivity to an arbitrary reduction.

- $(\mathscr{R}^{1a}_{DNF})$: $\sigma \wedge (\tau_1 \vee \tau_2) \mapsto_{\mathscr{R}} (\sigma \wedge \tau_1) \vee (\sigma \wedge \tau_2)$
  By definition of the maximal and minimal interpretations of a type, we have:

  $$(\sigma \wedge (\tau_1 \vee \tau_2))^{\Uparrow} = \sigma^{\Uparrow} \wedge (\tau_1^{\Uparrow} \vee \tau_2^{\Uparrow})$$
  $$((\sigma \wedge \tau_1) \vee (\sigma \wedge \tau_2))^{\Uparrow} = (\sigma^{\Uparrow} \wedge \tau_1^{\Uparrow}) \vee (\sigma^{\Uparrow} \wedge \tau_2^{\Uparrow})$$

  Since De Morgan's laws hold for static types, it is clear that:

  $$\sigma^{\Uparrow} \wedge (\tau_1^{\Uparrow} \vee \tau_2^{\Uparrow}) \simeq (\sigma^{\Uparrow} \wedge \tau_1^{\Uparrow}) \vee (\sigma^{\Uparrow} \wedge \tau_2^{\Uparrow})$$

  Hence the preservation of the maximal interpretation. The same goes for the minimal interpretation, which proves that the rule $(\mathscr{R}^{1a}_{DNF})$ preserves the equivalence $\approxeq$.
- $(\mathscr{R}^{1b}_{DNF})$: Unions and intersections are evidently commutative for the relation $\approxeq$. This case is therefore a direct consequence of the previous one by commutativity of the intersection constructor.
- $(\mathscr{R}^{2}_{DNF})$, $(\mathscr{R}^{3}_{DNF})$, $(\mathscr{R}^{4}_{DNF})$: these three cases are actually immediate since only static types can be negated, and are therefore simple applications of De Morgan's laws.
- $(\mathscr{R}^{5}_{DNF})$: $E(\tau) \mapsto_{\mathscr{R}} E(\tau')$ if $\tau \mapsto_{\mathscr{R}} \tau'$
  This case is proved by induction over the rewriting context E:
  – $E = \sigma \vee E'$, in which case we have $\sigma \vee E'(\tau) \mapsto_{\mathscr{R}} \sigma \vee E'(\tau')$.

By induction hypothesis, we know that $E'(\tau) \approxeq E'(\tau')$. Hence the following equivalences:

$$\begin{aligned}(E(\tau))^{\Uparrow} &\equiv (\sigma \vee E'(\tau))^{\Uparrow} \\ &\equiv \sigma^{\Uparrow} \vee (E'(\tau))^{\Uparrow} \\ &\simeq \sigma^{\Uparrow} \vee (E'(\tau'))^{\Uparrow} \quad \text{(by induction hypothesis)} \\ &\equiv (\sigma \vee E'(\tau'))^{\Uparrow} \\ &\equiv (E(\tau'))^{\Uparrow}\end{aligned}$$

Naturally, the same reasoning holds for the minimal interpretation of $E(\tau)$, which proves this first case.

- The three other union and intersection cases are similar and are proved in the same way.
- $E = \neg E'$, in which case we have $\neg E'(\tau) \mapsto_{\mathscr{R}} \neg E'(\tau')$.
  Once again, since only purely static types can be negated, and given that $\simeq$ is a congruence on static types, this case is immediate. This concludes the last case for the rewriting context as well as the last rewriting rule, hence the proposition.

□

We also state the following results, formalizing the intuition behind the applicative concretization.

Proposition 6. *(Preservation of equivalence by $\gamma^+_{\mathscr{A}}$) — For every gradual types σ and τ, if $\sigma \mapsto^*_{\mathscr{R}} \tau$ then $\gamma^+_{\mathscr{A}}(\sigma) = \gamma^+_{\mathscr{A}}(\tau)$.*

Proof. Immediate by definition of $\gamma^+_{\mathscr{A}}$, which is preserved by every rule of $\mathscr{R}_{DNF}$. □

Now that we have proved that the rewriting system preserves the equivalence of two gradual types, we need to prove that it correctly produces types in disjunctive normal form. Thus, we start by defining what it means for a gradual type to be in disjunctive normal form.

Definition 20. *(Gradual Disjunctive Normal Form) A gradual type σ is said to be in disjunctive normal form (DNF) if it is of the form:*

$$\sigma \equiv \bigvee_{i \in I} \bigwedge_{j \in J_i} \gamma_j$$

where γ_j are gradual literals, that is, gradual atoms or negation of static atoms:

$$\gamma ::= \alpha \mid \neg a$$

Proposition 7. *(Soundness of rewriting) — For every gradual types τ and σ, if $\tau \mapsto^*_{\mathscr{R}} \sigma$ and $\sigma \not\mapsto_{\mathscr{R}}$, then σ is in disjunctive normal form.*

Proof. This proof is done by contradiction. Let τ and σ be two gradual types such that $\tau \mapsto^*_{\mathscr{R}} \sigma$. Let us assume that σ is not in disjunctive normal form. It is then possible to prove, by complete induction on σ, that it can be reduced using $\mathscr{R}_{DNF}$.

- $\sigma \equiv \sigma_1 \vee \sigma_2$.
  A union of two types in disjunctive normal form is a type in disjunctive normal form. Therefore, at least one of the two types σ_1 and σ_2 is not in DNF.
  Suppose that σ_1 is not in DNF. By induction hypothesis, σ_1 can be rewritten to a gradual type σ_1'. Rewriting contexts allow us to rewrite on the left of a union. Therefore, $\sigma \equiv \sigma_1 \vee \sigma_2 \mapsto_{\mathscr{R}} \sigma_1' \vee \sigma_2$, hence the contradiction.
  Similarly, if σ_2 is not in DNF, the same argument holds since we can apply a rewriting rule to the right of a union.

- $\sigma \equiv \sigma_1 \wedge \sigma_2$.
  An intersection of two intersections of literals is a type in disjunctive normal form. Therefore, at least one of the two types σ_1 and σ_2 is not an intersection of literals.
  Suppose that σ_1 is not an intersection of literals. This means that σ_1 contains at least one union that is not under an arrow, or a union or intersection below a negation. Reasoning on the topmost constructor of σ_1, there are three possible subcases:
  – The topmost constructor of σ_1 is a union, that is, $\sigma_1 \equiv \sigma_{1,1} \vee \sigma_{1,2}$. In this case, we have $\sigma \equiv (\sigma_{1,1} \vee \sigma_{1,2}) \wedge \sigma_2$. Therefore, σ can be rewritten using the rule $(\mathscr{R}^{1b}_{DNF})$ to $(\sigma_{1,1} \wedge \sigma_2) \vee (\sigma_{1,2} \wedge \sigma_2)$
  – The topmost constructor of σ_1 is an intersection. Since σ_1 is strictly smaller than σ, it is possible to apply the induction hypothesis to σ_1 which states that σ_1 can be rewritten to a type σ_1'.
    By definition of rewriting contexts, it is possible to rewrite to the left of an intersection, hence $\sigma \equiv \sigma_1 \wedge \sigma_2 \mapsto_{\mathscr{R}} \sigma_1' \wedge \sigma_2$.
  – The topmost constructor of σ_1 is a negation, that is, $\sigma_1 \equiv \neg t$. Since σ_1 is not a literal, it is not in disjunctive normal form. By induction hypothesis, σ_1 rewrites to a types σ_1'.
    Once again, by definition of rewriting contexts, it is possible to rewrite to the left of an intersection, hence $\sigma \equiv \sigma_1 \wedge \sigma_2 \mapsto_{\mathscr{R}} \sigma_1' \wedge \sigma_2$.
  The case of σ_2 is proved symmetrically: using $(\mathscr{R}^{1a}_{DNF})$ or by rewriting to the right of an intersection.
- $\sigma \equiv \neg s$.
  Since σ is not in disjunctive normal form, s is not an atom. Therefore, the topmost constructor of s is necessarily a union, an intersection, or a negation.
  – $s = s_1 \vee s_2$.
    In this case, $\sigma \equiv \neg(s_1 \vee s_2) \mapsto_{\mathscr{R}} \neg s_1 \wedge \neg s_2$ by the rule $\mathscr{R}^2_{DNF}$.
  – $s = s_1 \wedge s_2$.
    In this case, $\sigma \equiv \neg(s_1 \wedge s_2) \mapsto_{\mathscr{R}} \neg s_1 \vee \neg s_2$ by the rule $\mathscr{R}^3_{DNF}$.
  – $s = \neg t$.
    In this case, $\sigma \equiv \neg\neg t \mapsto_{\mathscr{R}} t$ by the rule $\mathscr{R}^4_{DNF}$.
  In all three cases, σ can be rewritten, hence the contradiction, and the last case of the proof.

□

We now state that the rewriting system is terminating, thus effectively rewriting any type to an equivalent type in disjunctive normal form.

Proposition 8. *(Termination of Rewriting) — $\mathscr{R}_{DNF}$ is terminating, that is, there is no infinite chain of the form*

$$\tau_1 \mapsto_{\mathscr{R}} \tau_2 \mapsto_{\mathscr{R}} \cdots$$

Proof. This is a common result in rewriting theory. It is proven by defining the *multiset path ordering* based on the following ordering on connectives: $\neg \; > \; \wedge \; > \; \vee$. This ordering is, by construction, strictly decreasing for every rewriting rule. □

As a consequence, we deduce the following proposition:

Proposition 9. *(Existence of DNF) — For every gradual type τ, there exists a gradual type σ such that $\tau \approxeq \sigma$ and σ is in disjunctive normal form.*

Proof. Let τ be any gradual type. Consider any rewriting chain starting from τ: $\tau \mapsto_{\mathscr{R}} \tau_1 \mapsto_{\mathscr{R}} \cdots$. According to Proposition 8, such a chain is necessarily finite. Let τ_n be its last element. Formally, $\tau \mapsto^*_{\mathscr{R}} \tau_n \not\mapsto_{\mathscr{R}}$.

Now, according to Proposition 7, τ_n is in disjunctive normal form. Moreover, by Proposition 5, it holds that $\tau \approxeq \tau_n$, hence the proposition. □

Remark: It is clear that union and intersection constructors are commutative for the equivalence relation $\approxeq$. Thus, the disjunctive normal form of a given type is not unique. In particular, types below a union or an intersection can be reordered as necessary without altering the equivalence property.

We now prove that, similarly to non-gradual set-theoretic types, every gradual type is equivalent to a type in uniform disjunctive normal form, that is, a type whose intersections contain only arrows or only base types.

Definition 21. *(Gradual Uniform Disjunctive Normal Form) — A gradual type σ is said to be in uniform disjunctive normal form (uDNF) if it is of the form:*

$$\begin{aligned}\sigma \equiv & \bigvee_{i\in I_{f,1}} \bigwedge_{j\in J_i} \sigma_j \to \tau_j \wedge \bigwedge_{j\in N_i} \neg(s_j \to t_j) \\ \vee & \bigvee_{i\in I_{f,2}} \bigwedge_{j\in J_i} \sigma_j \to \tau_j \wedge \bigwedge_{j\in N_i} \neg(s_j \to t_j) \wedge ? \\ \vee & \bigvee_{i\in I_{b,1}} \bigwedge_{j\in J_i} b_j \wedge \bigwedge_{j\in N_i} \neg b_j \\ \vee & \bigvee_{i\in I_{b,2}} \bigwedge_{j\in J_i} b_j \wedge \bigwedge_{j\in N_i} \neg b_j \wedge ?\end{aligned}$$

Proposition 10. *(Existence of uDNF) — For every gradual type τ, there exists a gradual type σ such that $\tau \approxeq \sigma$ and σ is in uniform disjunctive normal form.*

Proof. Let τ be any gradual type. According to Proposition 9, there exists a gradual type σ in disjunctive normal form such that $\tau \approxeq \sigma$:

$$\sigma \equiv \bigvee_{i\in I} \bigwedge_{j\in J_i} \alpha_j \wedge \bigwedge_{n\in N_i} \neg a_n$$

First of all, it is clear that $\neg\mathbb{0} \simeq \mathbb{1}$ and $\neg\mathbb{1} \simeq \mathbb{0}$. Therefore, any negation of $\mathbb{1}$ or $\mathbb{0}$ can be replaced by the opposite type. Moreover, we also have the following equivalences, for every gradual type τ:

$$\begin{aligned}\tau \wedge \mathbb{1} &\approxeq \tau \\ \tau \vee \mathbb{1} &\approxeq \mathbb{1} \\ \tau \wedge \mathbb{0} &\approxeq \mathbb{0} \\ \tau \vee \mathbb{0} &\approxeq \tau\end{aligned}$$

These are direct consequences of the properties of the union and intersection of static types, and the definitions of $\mathbb{1}$ and $\mathbb{0}$.

Therefore, given that $\approxeq$ is a congruence, any occurrence of $\mathbb{1}$ or $\mathbb{0}$ can be removed from σ while preserving the equivalence of τ and σ.

We now prove that we can safely remove any *heterogeneous intersection* from σ. Consider the following cases:

- There exists an intersection $i \in I$ that contains two terms $(j, j') \in J_i^2$ such that $\alpha_j = \sigma' \to \tau'$ and $\alpha_{j'} = b$.

Developing the maximal interpretation of $\alpha_j \wedge \alpha_{j'}$, we obtain:

$$\begin{aligned}(\alpha_j \wedge \alpha_{j'})^{\Uparrow} &= \alpha_j^{\Uparrow} \wedge \alpha_j'^{\Uparrow} \\ &= (\sigma'^{\Downarrow} \to \tau'^{\Uparrow}) \wedge b \\ &\simeq \mathbb{0}\end{aligned}$$

The same reasoning shows that $(\alpha_j \wedge \alpha_{j'})^{\Downarrow} \simeq \mathbb{0}$. Therefore, it holds that $\alpha_j \wedge \alpha_{j'} \cong \mathbb{0}$. Thus, the whole intersection i is equivalent to the empty type and can be removed from σ.

- $\exists i \in I, \exists j \in J_i, \exists n \in N_i, \alpha_j = \sigma' \to \tau'$ and $a_n = b$.
  Developing the maximal interpretation of $\alpha_j \wedge \neg a_n$ gives:

$$\begin{aligned}(\alpha_j \wedge \neg a_n)^{\Uparrow} &= \alpha_j^{\Uparrow} \wedge \neg a_n^{\Downarrow} \\ &= (\sigma'^{\Downarrow} \to \tau'^{\Uparrow}) \wedge \neg b \\ &\simeq \sigma'^{\Downarrow} \to \tau'^{\Uparrow} \\ &= (\sigma' \to \tau')^{\Uparrow}\end{aligned}$$

  The same reasoning shows that

$$(\alpha_j \wedge \neg a_n)^{\Downarrow} \simeq (\sigma' \to \tau')^{\Downarrow}$$

  Therefore, it holds that

$$\alpha_j \wedge \neg a_n \cong \sigma' \to \tau' = \alpha_j$$

  Thus, we can safely remove the atom a_n from the intersection i while preserving the equivalence of σ and τ.
- $\exists i \in I, \exists j \in J_i, \exists n \in N_i, \alpha_j = b$ and $a_n = s \to t$.
  Once again, we develop the maximal interpretation of $\alpha_j \wedge \neg a_n$:

$$\begin{aligned}(\alpha_j \wedge \neg a_n)^{\Uparrow} &= \alpha_j^{\Uparrow} \wedge \neg a_n^{\Downarrow} \\ &= b \wedge \neg(s^{\Downarrow} \to t^{\Uparrow}) \\ &\simeq b = \alpha_j^{\Uparrow}\end{aligned}$$

  The same reasoning shows that $(\alpha_j \wedge \neg a_n)^{\Downarrow} = b = \alpha_j^{\Downarrow}$. Hence, we have $\alpha_j \wedge \neg a_n \cong \alpha_j$. Therefore, the atom a_n can once again be safely removed from the intersection i.
- There is only one case of heterogeneous intersection left, which is the case of an intersection containing the negation of an arrow as well as the negation of a base type. However, if this intersection also contains a positive arrow or base type, then it falls into one of the two previous cases. Therefore, there are only two subcases left: either the intersection does not contain any positive literal, or the only positive literal it contains is **?**.
  These two cases are handled in the same, following way. Consider such an intersection $i \in I$:

$$\tau_i = \bigwedge_{j \in (J_i \cup N_i)} \gamma_j$$

  This intersection is effectively equivalent to the following one:

$$\tau_i' = \bigwedge_{j \in (J_i \cup N_i)} \gamma_j \wedge \mathbb{1}$$

Remarking that $\mathbb{1} \approxeq \mathbb{1}_{\to} \vee \mathbb{1}_B$ where $\mathbb{1}_B$ is the union of all base types and $\mathbb{1}_{\to} = \mathbb{0} \to \mathbb{1}$ is the set of all functional values, we obtain the following equivalence:

$$\begin{aligned}\tau_i' &\approxeq \bigwedge_{j \in (J_i \cup N_i)} \gamma_j \wedge (\mathbb{1}_{\to} \vee \mathbb{1}_B) \\ &\approxeq \left(\bigwedge_{j \in (J_i \cup N_i)} \gamma_j \wedge (\mathbb{0} \to \mathbb{1}) \right) \vee \bigvee_{b \in B} \bigwedge_{j \in J_i} \gamma_j \wedge b\end{aligned}$$

This type is in disjunctive normal form and is such that every intersection contains at least one positive literal that is not **?**. Therefore, it can be handled using the previous cases. Hence, τ_i is equivalent to a type in uniform disjunctive normal form $\tau_{i,DNF}$.
Now, it is easy to see that, given a type in disjunctive normal form, replacing any of its intersections by a type in DNF still produces a type in DNF.
Using this fact, we can safely replace the intersection τ_i by $\tau_{i,DNF}$ in σ, producing a type σ' in disjunctive normal form such that $\sigma \approxeq \sigma'$. This concludes the last case.

In conclusion, τ is equivalent to a type in disjunctive normal form that does not contain heterogeneous intersections. Reordering intersections and separating those which contain an occurrence of **?** gives the form presented in Definition 21, which concludes the proof. □

We immediately deduce the following corollaries:

Corollary 2. *(uDNF for Function Types) Every gradual type τ such that $\tau^{\Downarrow} \leq \mathbb{0} \to \mathbb{1}$ can be written in disjunctive normal form as:*

$$\begin{aligned}\tau \approxeq &\bigvee_{i \in I_{f,1}} \bigwedge_{j \in J_i} \sigma_j \to \tau_j \wedge \bigwedge_{j \in J_n} \neg(s_j \to t_j) \\ \vee &\bigvee_{i \in I_{f,2}} \bigwedge_{j \in J_i} \sigma_j \to \tau_j \wedge \bigwedge_{j \in J_n} \neg(s_j \to t_j) \wedge ? \\ \vee &\bigvee_{i \in I_b} \bigwedge_{j \in J_i} b_j \wedge \bigwedge_{j \in J_n} \neg b_j \wedge ?\end{aligned}$$

As a remark, note that the definition of the uniform disjunctive normal form can be simplified when dealing with safe function types (that is, gradual types that are *always* a subtype of $\mathbb{0} \to \mathbb{1}$).

Corollary 3. *(uDNF for Safe Function Types) Every gradual type τ such that $\tau^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$ can be written in disjunctive normal form as:*

$$\begin{aligned}\tau \approxeq &\bigvee_{i \in I_{f,1}} \bigwedge_{j \in J_i} \sigma_j \to \tau_j \wedge \bigwedge_{j \in J_n} \neg(s_j \to t_j) \\ \vee &\bigvee_{i \in I_{f,2}} \bigwedge_{j \in J_i} \sigma_j \to \tau_j \wedge \bigwedge_{j \in J_n} \neg(s_j \to t_j) \wedge ?\end{aligned}$$

where for every $i \in I_{f,2}$, $J_i \neq \emptyset$.

A.2 Cast Language

A.2.1 Syntax.

Definition 22. *(Cast Language) The terms constituting the cast language are defined by the following grammar:*

$$\begin{aligned}
\textbf{\textit{Terms}}^{\langle\rangle} \quad & e ::= x \mid c \mid \lambda^{\mathbb{I}}_{\langle\tau\rangle}\, x.\, e \mid e\ e \mid (e \in t)?e : e \mid \langle\tau\rangle\ e \\
\textbf{\textit{Values}}^{\langle\rangle} \quad & v ::= c \mid \lambda^{\mathbb{I}}_{\langle\tau\rangle}\, x.\, e \quad \textit{where } \tau^{\Uparrow} \not\leq \mathbb{0} \\
\textbf{\textit{Interfaces}} \quad & \mathbb{I} ::= \{\sigma_i \to \tau_i \mid i \in I\} \\
\textbf{\textit{Errors}} \quad & \mathscr{E} ::= \texttt{CastError}
\end{aligned}$$

A.2.2 Operators.

We define the *Safe Applicative Concretization* on gradual types as presented in Definition 8. We also define the *Safe Gradual Domain* as presented in Definition 9, following the same intuition as for the gradual domain defined in Definition 5. Using these operators, we define the subtyping relations presented in Definition 10 and Definition 11. We then prove several properties on these relations, ultimately leading to a substitution property.

Proposition 11. *The relation $\preceq$ is reflexive and transitive.*

Proof. Trivial by definition of $\preceq$. □

Proposition 12. *The relation $\sqsubseteq$ is reflexive and transitive.*

Proof. The result is immediate by Proposition 11 and since the relation defined by $\sigma^{\Uparrow} \leq \tau^{\Uparrow}$ is symmetric and transitive. □

Proposition 13. *Denoting by $\sqsupseteq\sqsubseteq$ the equivalence relation defined as the symmetric closure of $\sqsubseteq$, the following results hold, for every gradual types σ and τ:*

$$\begin{aligned}
\sigma \vee \tau &\sqsupseteq\sqsubseteq \tau \vee \sigma \\
\sigma \wedge \tau &\sqsupseteq\sqsubseteq \tau \wedge \sigma \\
\sigma \wedge \mathbb{1} &\sqsupseteq\sqsubseteq \sigma \\
\sigma \vee \mathbb{0} &\sqsupseteq\sqsubseteq \sigma \\
\sigma \wedge \sigma &\sqsupseteq\sqsubseteq \sigma \\
\sigma \vee \sigma &\sqsupseteq\sqsubseteq \sigma
\end{aligned}$$

Proof. Once again, the result is immediate since that, given any of these equations, the applicative concretizations of both of its sides are equal. □

It particular, this proposition allows us to "reorganize" any intersection or union of any number of terms while preserving the relation $\sqsubseteq$.

Lemma 5. *For every gradual type τ, it holds that*

$$\mathbb{0} \sqsubseteq \tau$$

Proof. Let τ be any gradual type. Since $\mathbb{0}^{\Uparrow} = \mathbb{0} \leq t$ for any type t, it holds that $\mathbb{0}^{\Uparrow} \leq \tau^{\Uparrow}$. Moreover, if $\tau \leq \mathbb{0} \to \mathbb{1}$, since $\gamma^+_{\mathscr{D}}(\mathbb{0}) = \gamma^+_{\mathscr{A}}(\mathbb{0}) = \emptyset$, we immediately deduce $\mathbb{0} \preceq \tau$. □

Lemma 6. *For every gradual types σ_1, σ_2 and τ_1, τ_2, if $\sigma_1 \sqsubseteq \tau_1$ and $\sigma_2 \sqsubseteq \tau_2$ then*

$$\sigma_1 \vee \sigma_2 \sqsubseteq \tau_1 \vee \tau_2$$

Proof. Let $\sigma_1, \sigma_2, \tau_1, \tau_2$ be any gradual types such that $\sigma_1 \sqsubseteq \tau_1$ and $\sigma_2 \sqsubseteq \tau_2$. We prove the two points of Definition 11.

- By definition of compatible subtyping, it holds that $\sigma_1^{\Uparrow} \leq \tau_1^{\Uparrow}$ and $\sigma_2^{\Uparrow} \leq \tau_2^{\Uparrow}$. Hence, according to the set-theoretic definitions of the union and of static subtyping, it holds that $\sigma_1^{\Uparrow} \leq \tau_1^{\Uparrow} \vee \tau_2^{\Uparrow}$, and $\sigma_2^{\Uparrow} \leq \tau_1^{\Uparrow} \vee \tau_2^{\Uparrow}$. Thus, according to the same definitions, it holds that $\sigma_1^{\Uparrow} \vee \sigma_2^{\Uparrow} \leq \tau_1^{\Uparrow} \vee \tau_2^{\Uparrow}$. Finally, according to the definition of the maximal concretization, we deduce $(\sigma_1 \vee \sigma_2)^{\Uparrow} \leq (\tau_1 \vee \tau_2)^{\Uparrow}$.
- Suppose that $(\tau_1 \vee \tau_2)^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$. Therefore, it holds that $\tau_1^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$ and $\tau_2^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$. Thus, by hypothesis, we have $\sigma_1 \preceq \tau_1$ and $\sigma_2 \preceq \tau_2$.
  Let $S \in \gamma_{\mathscr{G}}^+(\sigma_1 \vee \sigma_2)$. We want to show that $S \in \gamma_{\mathscr{G}}^+(\tau_1 \vee \tau_2)$. By definition of $\gamma_{\mathscr{G}}^+$, we know that $S \in \gamma_{\mathscr{G}}^+(\sigma_1)$ or $S \in \gamma_{\mathscr{G}}^+(\sigma_2)$. We suppose $S \in \gamma_{\mathscr{G}}^+(\sigma_1)$, the second case is proved in the same way. Since $\sigma_1 \preceq \tau_1$, we know that $S \in \gamma_{\mathscr{G}}^+(\tau_1)$. By definition of $\gamma_{\mathscr{G}}^+$, it holds that $S \in \gamma_{\mathscr{G}}^+(\tau_1 \vee \tau_2)$, hence the result. The same reasoning can be done with $\gamma_{\mathscr{A}}^+$, which proves the proposition.

□

Corollary 4. *For every gradual types σ, τ and ρ, if $\sigma \sqsubseteq \tau$ then the following result holds:*

$$\sigma \sqsubseteq \tau \vee \rho$$

Proof. This is an immediate corollary of Lemma 6 with $\sigma_2 = \mathbb{0}$, using Lemma 5. □

Lemma 7. *For every gradual types σ and τ verifying $\tau^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$, it holds that:*

$$\sigma \sqsubseteq \tau \implies \widetilde{dom}_{\mathscr{G}}(\tau) \leq \widetilde{dom}_{\mathscr{G}}(\sigma)$$

Proof. Let σ and τ be any gradual types such that $\tau^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$ and $\sigma \sqsubseteq \tau$.
Let $S \in \gamma_{\mathscr{G}}^+(\sigma)$. Since $\sigma \preceq \tau$, we know that $S \in \gamma_{\mathscr{G}}^+(\tau)$. Since it holds by reflexivity that

$$\bigvee_{\rho \to \rho' \in S} \rho^{\Uparrow} \leq \bigvee_{\rho \to \rho' \in S} \rho^{\Uparrow}$$

we can take the intersection of the left hand side for every $S = T \in \gamma_{\mathscr{G}}^+(\tau)$, yielding

$$\bigwedge_{T \in \gamma_{\mathscr{G}}^+(\tau)} \bigvee_{\rho \to \rho' \in T} \rho^{\Uparrow} \leq \bigvee_{\rho \to \rho' \in S} \rho^{\Uparrow}$$

Since this equation does not depend on the choice of S anymore, we can take the intersection of the right hand side over $\gamma_{\mathscr{G}}^+(\sigma)$, giving

$$\bigwedge_{T \in \gamma_{\mathscr{G}}^+(\tau)} \bigvee_{\rho \to \rho' \in T} \rho^{\Uparrow} \leq \bigwedge_{S \in \gamma_{\mathscr{G}}^+(\sigma)} \bigvee_{\rho \to \rho' \in S} \rho^{\Uparrow}$$

Which is the result. □

Lemma 8. *For every gradual types σ, τ and ρ such that $\tau^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$ and $\rho^{\Uparrow} \leq \widetilde{dom}(\tau)$, it holds that:*

$$\sigma \sqsubseteq \tau \implies \sigma \mathbin{\widetilde{\circ}} \rho \sqsubseteq \tau \mathbin{\widetilde{\circ}} \rho$$

Proof. Let σ, τ and ρ be any gradual types such that $\tau^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$, $\rho^{\Uparrow} \leq \widetilde{dom}(\tau)$ and $\sigma \sqsubseteq \tau$.
To ease the notation, for any set of arrows S, we will denote by (C_S) and (D_S) the following

conditions:

$$(C_S) \qquad \rho^{\Uparrow} \not\leq \bigvee_{(\nu\to\nu')\in S} \nu^{\Downarrow}$$

$$(D_S) \qquad \rho^{\Uparrow} \wedge \bigvee_{(\nu\to\nu')\in S} \nu^{\Uparrow} \not\leq \mathbb{0}$$

By reflexivity of $\sqsubseteq$, we can immediately write:

$$\bigvee_{S\in\gamma^+_{\mathscr{A}}(\sigma)} \bigvee_{\substack{Q\subsetneq S \\ Q \text{ verifies } (C_Q) \text{ and } (D_{S\setminus Q})}} \bigwedge_{(\rho\to\rho')\in S\setminus Q} \rho' \sqsubseteq \bigvee_{S\in\gamma^+_{\mathscr{A}}(\sigma)} \bigvee_{\substack{Q\subsetneq S \\ Q \text{ verifies } (C_Q) \text{ and } (D_{S\setminus Q})}} \bigwedge_{(\rho\to\rho')\in S\setminus Q} \rho'$$

Applying Corollary 4, we deduce that:

$$\bigvee_{S\in\gamma^+_{\mathscr{A}}(\sigma)} \bigvee_{\substack{Q\subsetneq S \\ Q \text{ verifies } (C_Q) \text{ and } (D_{S\setminus Q})}} \bigwedge_{(\rho\to\rho')\in S\setminus Q} \rho' \sqsubseteq \bigvee_{S\in\gamma^+_{\mathscr{A}}(\sigma)} \bigvee_{\substack{Q\subsetneq S \\ Q \text{ verifies } (C_Q) \text{ and } (D_{S\setminus Q})}} \bigwedge_{(\rho\to\rho')\in S\setminus Q} \rho' \vee \bigvee_{S\in\gamma^+_{\mathscr{A}}(\tau)\setminus\gamma^+_{\mathscr{A}}(\sigma)} \bigvee_{\substack{Q\subsetneq S \\ Q \text{ verifies } (C_Q) \text{ and } (D_{S\setminus Q})}} \bigwedge_{(\rho\to\rho')\in S\setminus Q} \rho'$$

However, by hypothesis, we have $\sigma \preceq \tau$. Therefore, $\gamma^+_{\mathscr{A}}(\sigma) \subset \gamma^+_{\mathscr{A}}(\tau)$ and it holds that $\gamma^+_{\mathscr{A}}(\tau) = \gamma^+_{\mathscr{A}}(\sigma) \sqcup (\gamma^+_{\mathscr{A}}(\tau)\setminus\gamma^+_{\mathscr{A}}(\sigma))$. Thus, the left hand side of the equation can be rewritten yielding:

$$\bigvee_{S\in\gamma^+_{\mathscr{A}}(\sigma)} \bigvee_{\substack{Q\subsetneq S \\ Q \text{ verifies } (C_Q) \text{ and } (D_{S\setminus Q})}} \bigwedge_{(\rho\to\rho')\in S\setminus Q} \rho' \sqsubseteq \bigvee_{T\in\gamma^+_{\mathscr{A}}(\tau)} \bigvee_{\substack{Q\subsetneq T \\ Q \text{ verifies } (C_Q) \text{ and } (D_{T\setminus Q})}} \bigwedge_{(\rho\to\rho')\in T\setminus Q} \rho'$$

Which is the result. □

Lemma 9. *For every gradual types σ and τ, and every gradual type ρ such that $\rho^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$, it holds that:*

$$\sigma \sqsubseteq \tau \implies \rho\widetilde{\circ}\sigma \sqsubseteq \rho\widetilde{\circ}\tau$$

Proof. Let σ, τ and ρ be three gradual types, such that $\rho^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$ and $\sigma \sqsubseteq \tau$.
Let $S \in \gamma^+_{\mathscr{A}}(\gamma)$, and any strict subset $S' \subsetneq S$ verifying the following conditions:

$$(1) \quad \sigma^{\Uparrow} \not\leq \bigvee_{(\rho\to\rho')\in S'} \rho^{\Downarrow}$$

$$(2) \quad \sigma^{\Uparrow} \wedge \bigvee_{(\rho\to\rho')\in S\setminus S'} \rho^{\Uparrow} \not\leq \mathbb{0}$$

Since, by hypothesis, $\sigma^{\Uparrow} \leq \tau^{\Uparrow}$, we immediately deduce that $\tau^{\Uparrow}$ must verify the same conditions, that is:

$$(1') \quad \tau^{\Uparrow} \not\leq \bigvee_{(\rho\to\rho')\in S'} \rho^{\Downarrow}$$

$$(2') \quad \tau^{\Uparrow} \wedge \bigvee_{(\rho\to\rho')\in S\setminus S'} \rho^{\Uparrow} \not\leq \mathbb{0}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ (T_x^{\langle\rangle}) \qquad \frac{}{\Gamma \vdash c : B(c)}\ (T_c^{\langle\rangle})$$

$$\frac{\mathtt{TypeOf}(\mathbb{I})^{\Downarrow} \leq \tau^{\Uparrow} \qquad \forall(\sigma \to \rho) \in \mathbb{I}, \quad \Gamma, x : \sigma \vdash e : \rho' \quad \rho' \sqsubseteq \rho}{\lambda^{\mathbb{I}}_{\langle\tau\rangle}\, x.\, e : \tau}\ (T_\lambda^{\langle\rangle})$$

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \langle\tau\rangle\, e : \tau}\ (T_{cast}^{\langle\rangle}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1^{\Uparrow} \leq \mathbb{0} \to \mathbb{1} \quad \tau_2^{\Uparrow} \leq \widetilde{dom}(\tau_1)}{\Gamma \vdash e_1\ e_2 : \tau_1 \widetilde{\circ} \tau_2}\ (T_{app}^{\langle\rangle})$$

$$\frac{\Gamma \vdash e : \tau \quad \begin{cases} \tau^{\Uparrow} \not\leq \neg t & \Gamma \vdash e_1 : \sigma_1 \\ \tau^{\Uparrow} \not\leq t & \Gamma \vdash e_2 : \sigma_2 \end{cases}}{\Gamma \vdash ((e \in t)?e_1 : e_2) : \sigma_1 \vee \sigma_2}\ (T_{case-both}^{\langle\rangle})$$

$$\frac{\Gamma \vdash e : \tau \quad \begin{cases} \tau^{\Uparrow} \not\leq \neg t \\ \tau^{\Uparrow} \leq t \end{cases} \quad \Gamma \vdash e_1 : \sigma_1}{\Gamma \vdash ((e \in t)?e_1 : e_2) : \sigma_1}\ (T_{case-L}^{\langle\rangle}) \qquad \frac{\Gamma \vdash e : \tau \quad \begin{cases} \tau^{\Uparrow} \leq \neg t \\ \tau^{\Uparrow} \not\leq t \end{cases} \quad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash ((e \in t)?e_1 : e_2) : \sigma_2}\ (T_{case-R}^{\langle\rangle})$$

$$\frac{\Gamma \vdash e : \tau \quad \tau^{\Uparrow} \leq \mathbb{0}}{\Gamma \vdash ((e \in t)?e_1 : e_2) : \tau}\ (T_{case-none}^{\langle\rangle})$$

Fig. 5. Full typing rules for the cast language

Therefore, by Corollary 4, it holds that:

$$\bigvee_{\substack{S' \subsetneq S \\ \sigma^{\Uparrow} \not\leq \bigvee_{(\rho\to\rho')\in S'} \rho^{\Uparrow} \\ \sigma^{\Uparrow} \wedge \bigvee_{(\rho\to\rho')\in S\setminus S'} \rho^{\Uparrow} \not\leq \mathbb{0}}} \bigwedge_{(\rho\to\rho')\in S\setminus S'} \rho' \sqsubseteq \bigvee_{\substack{S' \subsetneq S \\ \tau^{\Uparrow} \not\leq \bigvee_{(\rho\to\rho')\in S'} \rho^{\Uparrow} \\ \tau^{\Uparrow} \wedge \bigvee_{(\rho\to\rho')\in S\setminus S'} \rho^{\Uparrow} \not\leq \mathbb{0}}} \bigwedge_{(\rho\to\rho')\in S\setminus S'} \rho'$$

as the union in the right hand side contains more terms than the union in the left hand side.

Since this is valid for every $S \in \gamma_{\mathscr{A}}^{+}(\gamma)$, using Lemma 6, we can take the union on both sides yielding:

$$\bigvee_{S\in\gamma_{\mathscr{A}}^{+}(\gamma)} \bigvee_{\substack{S' \subsetneq S \\ \sigma^{\Uparrow} \not\leq \bigvee_{(\rho\to\rho')\in S'} \rho^{\Uparrow} \\ \sigma^{\Uparrow} \wedge \bigvee_{(\rho\to\rho')\in S\setminus S'} \rho^{\Uparrow} \not\leq \mathbb{0}}} \bigwedge_{(\rho\to\rho')\in S\setminus S'} \rho' \sqsubseteq \bigvee_{S\in\gamma_{\mathscr{A}}^{+}(\gamma)} \bigvee_{\substack{S' \subsetneq S \\ \tau^{\Uparrow} \not\leq \bigvee_{(\rho\to\rho')\in S'} \rho^{\Uparrow} \\ \tau^{\Uparrow} \wedge \bigvee_{(\rho\to\rho')\in S\setminus S'} \rho^{\Uparrow} \not\leq \mathbb{0}}} \bigwedge_{(\rho\to\rho')\in S\setminus S'} \rho'$$

Which is the result. □

Lemma 10. *For every gradual types σ_1, σ_2, τ_1 and τ_2 such that $\tau_1^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$ and $\tau_2^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$, if $\sigma_1 \sqsubseteq \sigma_2$ and $\tau_1 \sqsubseteq \tau_2$ then it holds that:*

$$\tau_1 \widetilde{\circ} \sigma_1 \sqsubseteq \tau_2 \widetilde{\circ} \sigma_2$$

Proof. Immediate by transitivity of $\sqsubseteq$ and Lemmas 8 and 9. □

A.2.3 Type System. The full typing system for the cast language is presented in Figure 5.

Proposition 14. *(Uniqueness of Typing) For every term $e \in \textbf{\textit{Terms}}^{\langle\rangle}$, if $\Gamma \vdash e : \tau$ and $\Gamma \vdash e : \tau'$ then $\tau \equiv \tau'$.*

Proof. Trivial by structural induction on e since all the typing rules are deterministic in e. □

Proposition 15. *(Emptiness of $\mathbb{0}$) For every value $v \in \textbf{\textit{Values}}^{\langle\rangle}$, if $\Gamma \vdash v : \tau$ then $\tau^{\Uparrow} \not\leq \mathbb{0}$.*

Proof. Trivial by cases on v. If v is a constant, then $\tau = B(c) \not\leq \mathbb{0}$. If v is a lambda, by definition of $\textbf{Values}^{\langle\rangle}$, it holds that $\tau^{\Uparrow} \not\leq \mathbb{0}$. □

Theorem 5. *(Well-Typedness after Substitution) For every terms of the cast language $e, e' \in \textbf{\textit{Terms}}^{\langle\rangle}$ and every typing context Γ, if $\Gamma, x : \sigma \vdash e : \tau$ and $\Gamma \vdash e' : \sigma'$ where $\sigma' \sqsubseteq \sigma$ then $\Gamma \vdash e[x := e'] : \tau'$ where $\tau' \sqsubseteq \tau$.*

Proof. Let Γ be any typing context, and $e, e' \in \textbf{Terms}^{\langle\rangle}$. We have the following hypotheses:

$$(H1) \quad \Gamma, x : \sigma \vdash e : \tau$$
$$(H2) \quad \Gamma \vdash e' : \sigma'$$
$$(H3) \quad \sigma' \sqsubseteq \sigma$$

The proof is done by structural induction on e.

- $e = x$. In this particular case, $\sigma = \tau$, and $e[x := e'] = e'$. By hypothesis $(H2)$, it holds that $\Gamma \vdash e[x := e'] : \sigma'$, hence the result by hypothesis $(H3)$
- $e = y \neq x$. Since x does not appear in e, by hypothesis $(H1)$, it holds that $\Gamma \vdash e : \tau$. Moreover, we have $e[x := e'] = e$, hence the result.
- $e = c$. Since x does not appear in e and $\emptyset \vdash c : B(c)$, the result is trivial.
- $e = \lambda^{\mathbb{I}}_{\langle \tau_\lambda \rangle}\, y.\, e_\lambda$. Note that if $y = x$ then the result is trivial since the term is left unchanged by substitution of x. Therefore we consider in the following that $y \neq x$.
  First of all, by inversion of $(T^{\langle\rangle}_\lambda)$ on hypothesis $(H1)$, it holds that $\tau_\lambda = \tau$, and $\texttt{TypeOf}(\mathbb{I})^{\Downarrow} \leq \tau^{\Uparrow}_\lambda$.
  We want to show that $\Gamma \vdash \lambda^{\mathbb{I}}_{\langle \tau_\lambda \rangle}\, y.\, (e_\lambda[x := e']) : \tau_\lambda$, which gives the result by reflexivity of $\sqsubseteq$.
  To apply the rule $(T^{\langle\rangle}_\lambda)$ and deduce this result, we need to prove:
  $$\forall (\sigma_y \to \rho_y) \in \mathbb{I}, \quad \Gamma, y : \sigma_y \vdash e_\lambda[x := e'] : \rho'_y \quad \rho'_y \sqsubseteq \rho_y$$
  Let $\sigma_y \to \rho_y \in \mathbb{I}$. By inversion of $(T^{\langle\rangle}_\lambda)$ and $(H1)$, it holds that
  $$\Gamma, x : \sigma, y : \sigma_y \vdash e_\lambda : \rho'_y \quad \rho'_y \sqsubseteq \rho_y$$
  By induction hypothesis, substituting x by e' in e_λ implies:
  $$\Gamma, y : \sigma_y \vdash e_\lambda[x := e'] : \rho''_y \quad \rho''_y \sqsubseteq \rho'_y$$
  By transitivity of $\sqsubseteq$, we deduce that $\rho''_y \sqsubseteq \rho_y$, hence the result.
- $e = e_1\ e_2$. By inversion of $(T^{\langle\rangle}_{app})$ in $(H1)$, we have the following hypotheses:
  $$(H1_{inv}) \quad \Gamma, x : \sigma \vdash e_1 : \tau_1$$
  $$(H2_{inv}) \quad \Gamma, x : \sigma \vdash e_2 : \tau_2$$
  $$(H3_{inv}) \quad \tau_1^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$$
  $$(H4_{inv}) \quad \tau_2^{\Uparrow} \leq \widetilde{dom}(\tau_1)$$
  $$(H5_{inv}) \quad \tau = \tau_1 \widetilde{\circ} \tau_2$$

We want to apply $(T^{\langle\rangle}_{app})$ to $e[x := e'] \equiv (e_1[x := e'])(e_2[x := e'])$, and thus we need to prove the four required premises.
Applying the induction hypothesis to $(H1_{inv})$ gives us that $\Gamma \vdash e_1[x := e'] : \tau'_1$ (1st premise) where $\tau'_1 \sqsubseteq \tau_1$ (*). By definition of compatible subtyping, it holds that $\tau'^{\Uparrow}_1 \leq \tau^{\Uparrow}_1$. By $(H3_{inv})$ and transitivity of static subtyping, we deduce the 3rd premise, $\tau'^{\Uparrow}_1 \leq \mathbb{0} \to \mathbb{1}$.
Applying the induction hypothesis to $(H2_{inv})$ gives us that $\Gamma \vdash e_2[x := e'] : \tau'_2$ (2nd premise) where $\tau'_2 \sqsubseteq \tau_2$ (**). Now, applying Lemma 7 to (*) and $(H3_{inv})$, we deduce that $\widetilde{dom}_{\mathscr{S}}(\tau_1) \leq \widetilde{dom}_{\mathscr{S}}(\tau'_1)$. By definition of compatible subtyping and (**), it holds that $\tau'^{\Uparrow}_2 \leq \tau^{\Uparrow}_2$, and by $(H4_{inv})$ and transitivity, we deduce that $\tau'^{\Uparrow}_2 \leq \widetilde{dom}_{\mathscr{S}}(\tau'_1)$, which is the last premise.
Applying $(T^{\langle\rangle}_{app})$ to $e[x := e']$ shows that $\Gamma \vdash e[x := e'] : \tau'_1 \widetilde{\circ} \tau'_2$, and by Lemma 10 with (*) and (**), $\tau'_1 \widetilde{\circ} \tau'_2 \sqsubseteq \tau_1 \widetilde{\circ} \tau_2$, hence the result.

- $e = (e_t \in t)?e_1 : e_2$. By inversion of the typing rules, we always have:

$$(H1_{inv}) \quad \Gamma, x : \sigma \vdash e_t : \tau_t$$

Moreover, by induction hypothesis on e_t, the following hypotheses hold:

$$(IH1) \quad \Gamma \vdash e_t[x := e'] : \tau'_t$$
$$(IH2) \quad \tau'_t \sqsubseteq \tau_t$$

We then distinguish four possible cases.
  - $\tau'^{\Uparrow}_t \not\leq \neg t$ and $\tau'^{\Uparrow}_t \not\leq t$. By hypothesis $(IH2)$, since static subtyping is transitive, it necessarily holds that $\tau^{\Uparrow}_t \not\leq \neg t$ and $\tau^{\Uparrow}_t \not\leq t$. Therefore, by inversion of $(T^{\langle\rangle}_{case-both})$ on $(H1)$, it holds that $\Gamma, x : \sigma \vdash e_1 : \tau_1$ and $\Gamma, x : \sigma \vdash e_2 : \tau_2$. Moreover, we have $\tau \equiv \tau_1 \vee \tau_2$.
    Applying the induction hypothesis to both e_1 and e_2 gives us that $\Gamma \vdash e_i[x := e'] : \tau'_i$ where $\tau'_i \sqsubseteq \tau_i$ (for $i \in \{1, 2\}$).
    Moreover, applying $(T^{\langle\rangle}_{case-both})$ to $e[x := e']$ gives us that $\Gamma \vdash e[x := e'] : \tau'_1 \vee \tau'_2$. Hence the result by Lemma 6.
  - $\tau'^{\Uparrow}_t \not\leq \neg t$ and $\tau'^{\Uparrow}_t \leq t$. By hypothesis $(IH2)$, since static subtyping is transitive, it necessarily holds that $\tau^{\Uparrow}_t \not\leq \neg t$. Thus, we can either inverse the rule $(T^{\langle\rangle}_{case-both})$ or $(T^{\langle\rangle}_{case-L})$ on $(H1)$, deducing that $\Gamma, x : \sigma \vdash e_1 : \tau_1$.
    Applying the induction hypothesis to e_1 gives us that $\Gamma \vdash e_1[x := e'] : \tau'_1$ and $\tau'_1 \sqsubseteq \tau_1$. We can then apply $(T^{\langle\rangle}_{case-L})$ to $e[x := e']$, which gives us $\Gamma \vdash e[x := e'] : \tau'_1$, hence the result.
  - $\tau'^{\Uparrow}_t \leq \neg t$ and $\tau'^{\Uparrow}_t \not\leq t$. This case is proved in the same way as the previous one.
  - $\tau'^{\Uparrow}_t \leq \mathbb{0}$. Applying the rule $(T^{\langle\rangle}_{case-none})$ to $e[x := e']$ yields $\Gamma \vdash e[x := e'] : \tau'_t$. Hence the result by $(IH2)$.
- $e = \langle\tau_c\rangle\, e_c$. By inversion of $(T^{\langle\rangle}_{cast})$, we have $\Gamma, x : \sigma \vdash e_c : \rho$, and $\tau = \tau_c$. By induction hypothesis, it holds that $\Gamma \vdash e_c[x := e'] : \rho'$ and $\rho' \sqsubseteq \rho$. Thus, it immediately holds that $\Gamma \vdash \langle\tau_c\rangle\, e_c[x := e'] : \tau_c$, hence the result by reflexivity of $\sqsubseteq$, which concludes the proof.

□

A.2.4 Soundness. Before proving the subject reduction lemma, we prove the following result which states that the reduction rules are deterministic, which is important in particular for rules $(R_{app-\lambda})$ and (R_{app-c}).

Proposition 16. *(Uniqueness of Reduction) — For every term $e \in \boldsymbol{Terms}^{\langle\rangle}$, if $e \mapsto e_1$ and $e \mapsto e_2$ then $e_1 = e_2$.*

$$\frac{\exists(\sigma_i \to \tau_i) \in \mathbb{I} \quad B(c) \leq \sigma_i^{\Uparrow}}{(\lambda^{\mathbb{I}}_{\langle\tau\rangle}\, x.\, e)c \mapsto \langle \tau \widetilde{\circ} B(c)\rangle\, e[x := \langle\sigma_i\rangle\, c]}\ (R_{app-c})$$

$$\frac{\exists(\sigma_i \to \tau_i) \in \mathbb{I} \quad \mathtt{TypeOf}(\mathbb{I}')^{\Downarrow} \leq \sigma_i^{\Uparrow}}{(\lambda^{\mathbb{I}}_{\langle\tau\rangle}\, x.\, e)(\lambda^{\mathbb{I}'}_{\langle\tau'\rangle}\, y.\, e') \mapsto \langle \tau \widetilde{\circ} \tau'\rangle\, e[x := \lambda^{\mathbb{I}'}_{\langle\sigma_i\rangle}\, y.\, e']}\ (R_{app-\lambda})$$

$$\frac{\emptyset \vdash v : \tau \quad \tau^{\Uparrow} \leq t}{((v \in t)?e_1 : e_2) \mapsto e_1}\ (R_{case-L}) \qquad \frac{\emptyset \vdash v : \tau \quad \tau^{\Uparrow} \not\leq t}{((v \in t)?e_1 : e_2) \mapsto e_2}\ (R_{case-R})$$

$$\frac{\mathtt{TypeOf}(\mathbb{I})^{\Downarrow} \leq \tau^{\Uparrow}}{\langle\tau\rangle\, \lambda^{\mathbb{I}}_{\langle\tau'\rangle}\, x.\, e \mapsto \lambda^{\mathbb{I}}_{\langle\tau\rangle}\, x.\, e}\ (R_{cast-\lambda}) \qquad \frac{\emptyset \vdash c : s \quad s \leq \tau^{\Uparrow}}{\langle\tau\rangle\, c \mapsto c}\ (R_{cast-c})$$

$$\frac{\mathtt{TypeOf}(\mathbb{I})^{\Downarrow} \not\leq \tau^{\Uparrow}}{\langle\tau\rangle\, \lambda^{\mathbb{I}}_{\langle\tau'\rangle}\, x.\, e \mapsto \mathtt{CastError}}\ (R_{fail-\lambda}) \qquad \frac{\emptyset \vdash c : s \quad s \not\leq \tau^{\Uparrow}}{\langle\tau\rangle\, c \mapsto \mathtt{CastError}}\ (R_{fail-c})$$

$$\frac{e \mapsto e'}{E[e] \mapsto E[e']}\ (R_E) \qquad \frac{e \mapsto \mathtt{CastError}}{E[e] \mapsto \mathtt{CastError}}\ (R_{fail-E})$$

$$\frac{\nexists(\sigma_i \to \tau_i) \in \mathbb{I} \quad B(c) \leq \sigma_i^{\Uparrow}}{(\lambda^{\mathbb{I}}_{\langle\tau\rangle}\, x.\, e)c \mapsto \mathtt{CastError}}\ (R_{app-fail-c})$$

$$\frac{\nexists(\sigma_i \to \tau_i) \in \mathbb{I} \quad \mathtt{TypeOf}(\mathbb{I}')^{\Downarrow} \leq \sigma_i^{\Uparrow}}{(\lambda^{\mathbb{I}}_{\langle\tau\rangle}\, x.\, e)(\lambda^{\mathbb{I}'}_{\langle\tau'\rangle}\, y.\, e') \mapsto \mathtt{CastError}}\ (R_{app-fail-\lambda})$$

$$E ::= \Box \mid Ee \mid vE \mid (E \in t)?e : e \mid \langle\tau\rangle\, E$$

Fig. 6. Complete small-step reduction semantics for the cast language

Proof. Let $e \in \mathbf{Terms}^{\langle\rangle}$ such that $e \mapsto e_1$ and $e \mapsto e_2$. We reason by induction over e and by cases over the rules used in both reductions. There are only several possibilities. The rules (R_{app-c}) and $(R_{app-\lambda})$, due to their existential quantification, can possibly be applied in two different ways to the same term. There are also pairs of rules, such as (R_{case-L}) and (R_{case-R}), that can be applied to the same term. We distinguish the following cases:

- Both reductions use the rule (R_{app-c}). $e = (\lambda^{\mathbb{I}}_{\langle\tau\rangle}\, x.\, e')c$. By contradiction, assume that there exist two distinct arrows $(\sigma_i \to \tau_i)$ and $(\sigma_j \to \tau_j)$ in $\mathbb{I}$ such that $B(c) \leq \sigma_i^{\Uparrow}$ and $B(c) \leq \sigma_j^{\Uparrow}$. Therefore, it holds that $B(c) \leq \sigma_i^{\Uparrow} \wedge \sigma_j^{\Uparrow}$. However, the interface criterion ensures that $\sigma_i^{\Uparrow} \wedge \sigma_j^{\Uparrow} \leq \mathbb{0}$. Therefore, by transitivity, it holds that $B(c) \leq \mathbb{0}$, which is a contradiction. Hence we necessarily have $i = j$ and $e_1 = e_2$.
- Both reductions use the rule $(R_{app-\lambda})$. $e = (\lambda^{\mathbb{I}}_{\langle\tau\rangle}\, x.\, e')(\lambda^{\mathbb{I}'}_{\langle\tau'\rangle}\, y.\, e'')$. Once again, by contradiction, suppose that there exist two distinct arrows $(\sigma_i \to \tau_i)$ and $(\sigma_j \to \tau_j)$ in $\mathbb{I}$ such that $\mathtt{TypeOf}(\mathbb{I}')^{\Downarrow} \leq \sigma_i^{\Uparrow}$ and $\mathtt{TypeOf}(\mathbb{I}')^{\Downarrow} \leq \sigma_j^{\Uparrow}$. It holds that $\mathtt{TypeOf}(\mathbb{I}')^{\Downarrow} \leq \sigma_i^{\Uparrow} \wedge \sigma_j^{\Uparrow} \leq \mathbb{0}$, according

to the interface criterion. However, since $\mathtt{TypeOf}(\mathbb{I}')^{\Downarrow}$ is an intersection of (static) arrows, it cannot be empty, hence the contradiction. Thus, we necessarily have $i = j$ and $e_1 = e_2$.
- The rules (R_{case-L}) and (R_{case-R}) cannot be used on the same term since their premises are disjoint: it is not possible to have both $\tau^{\Uparrow} \leq t$ and $\tau^{\Uparrow} \not\leq t$.
- The rules $(R_{cast-\lambda})$ and $(R_{fail-\lambda})$ cannot be used on the same term since their premises are disjoint.
- The rules (R_{cast-c}) and (R_{fail-c}) cannot be used on the same term since their premises are disjoint.
- The last case correspond to the rule (R_E) being used for both reductions. However, since the reduction contexts are defined deterministically, the result holds by induction.

□

We now prove the subject reduction lemma.

Lemma 11. *(Subject Reduction) — For every typing context* Γ*, for every terms* e_1 *and* e_2*, if* $e_1 \mapsto e_2$ *and* $\Gamma \vdash e_1 : \tau_1$ *then* $\Gamma \vdash e_2 : \tau_2$ *and* $\tau_2 \sqsubseteq \tau_1$*.*

Proof. Let Γ be any typing context, and let e_1, e_2 be two terms such that $e_1 \mapsto e_2$ and $\Gamma \vdash e_1 : \tau_1$. The proof of the theorem is done by case over the reduction rule used to reduce e_1 into e_2.

- (R_{app-c}). In this case, $e_1 = (\lambda^{\mathbb{I}}_{\langle\tau\rangle}\, x.\, e)c$ and $e_2 = \langle \tau \widetilde{\circ} B(c)\rangle\, e[x := \langle\sigma_i\rangle\, c]$, where $B(c) \leq \sigma_i^{\Uparrow}$ and $(\sigma_i \to \tau_i) \in \mathbb{I}$.
  By inversion of the typing rule $(T^{\langle\rangle}_{app})$ on e_1, it holds that $\Gamma \vdash \lambda^{\mathbb{I}}_{\langle\tau\rangle}\, x.\, e : \tau_\lambda$, and that $\Gamma \vdash e_1 : \tau_\lambda \widetilde{\circ} B(c)$. Inversing the rule $(T^{\langle\rangle}_{\lambda})$ then yields $\tau_\lambda = \tau$ and, in particular, $\Gamma, x : \sigma_i \vdash e : \tau_i'$ where $\tau_i' \sqsubseteq \tau_i$.
  Moreover, applying $(T^{\langle\rangle}_{cast})$ to $\langle\sigma_i\rangle\, c$ yields $\Gamma \vdash \langle\sigma_i\rangle\, c : \sigma_i$. Therefore, by Theorem 5, we deduce that $\Gamma \vdash e[x := \langle\sigma_i\rangle\, c] : \tau_i''$ where $\tau_i'' \sqsubseteq \tau_i' \sqsubseteq \tau_i$.
  Finally, applying $(T^{\langle\rangle}_{cast})$ to e_2 yields $\Gamma \vdash \langle \tau \widetilde{\circ} B(c)\rangle\, e[x := \langle\sigma_i\rangle\, c] : \tau \widetilde{\circ} B(c)$, hence the result by reflexivity of $\sqsubseteq$.
- $(R_{app-\lambda})$. In this case, $e_1 = (\lambda^{\mathbb{I}}_{\langle\tau\rangle}\, x.\, e)(\lambda^{\mathbb{I}'}_{\langle\tau'\rangle}\, y.\, e')$ and $e_2 = \langle \tau \widetilde{\circ} \tau'\rangle\, e[x := \lambda^{\mathbb{I}'}_{\langle\sigma_i\rangle}\, y.\, e']$, where $\mathtt{TypeOf}(\mathbb{I}')^{\Downarrow} \leq \sigma_i^{\Uparrow}$ and $(\sigma_i \to \tau_i) \in \mathbb{I}$.
  Inversing the typing rule $(T^{\langle\rangle}_{app})$ on e_1 yields the following hypotheses:

$$(H1) \quad \Gamma \vdash \lambda^{\mathbb{I}}_{\langle\tau\rangle}\, x.\, e : \tau_\lambda$$
$$(H2) \quad \Gamma \vdash \lambda^{\mathbb{I}'}_{\langle\tau'\rangle}\, y.\, e' : \tau_{\lambda'}$$
$$(H3) \quad \Gamma \vdash e_1 : \tau_\lambda \widetilde{\circ} \tau_{\lambda'}$$

  Inversing the rule $(T^{\langle\rangle}_{app})$ on $(H1)$ and $(H2)$ immediately shows that $\tau_\lambda = \tau$ and $\tau_{\lambda'} = \tau'$. Moreover, by inversion of $(T^{\langle\rangle}_{app})$, it holds in particular that $\Gamma, x : \sigma_i \vdash e : \tau_i'$ where $\tau_i' \sqsubseteq \tau_i$.
  To apply the substitution theorem, we still need to prove that $\lambda^{\mathbb{I}'}_{\langle\sigma_i\rangle}\, y.\, e'$ is well-typed, by the rule $(T^{\langle\rangle}_{\lambda})$. Since, by hypothesis, $\mathtt{TypeOf}(\mathbb{I}')^{\Downarrow} \leq \sigma_i^{\Uparrow}$, the first premise of the rule holds. Moreover, inversing the rule $(T^{\langle\rangle}_{\lambda})$ on hypothesis $(H2)$ yields that $\Gamma, x : \sigma \vdash e : \rho'$ where $\rho' \sqsubseteq \rho$ for every $(\sigma \to \rho) \in \mathbb{I}'$, which is the second required premise. We can therefore apply the rule $(T^{\langle\rangle}_{\lambda})$ to deduce that $\Gamma \vdash \lambda^{\mathbb{I}'}_{\langle\sigma_i\rangle}\, y.\, e' : \sigma_i$.
  Applying Theorem 5 then yields that $e[x := \lambda^{\mathbb{I}'}_{\langle\sigma_i\rangle}\, y.\, e']$ is well-typed, thus, by rule $(T^{\langle\rangle}_{cast})$, it holds that $\Gamma \vdash e_2 : \tau \widetilde{\circ} \tau'$, hence the result.

- (R_{case-L}). In this case, we have $e_1 = (v \in t)?e_L : e_R$ and $e_2 = e_L$. Moreover, according to the rule (R_{case-L}), the following hypotheses hold:

$$(H1) \quad \emptyset \vdash v : \tau$$
$$(H2) \quad \tau^{\Uparrow} \leq t$$

  By inversion of the typing rules, there are four possible cases:
  - By inversion of $(T^{\langle\rangle}_{case-both})$. It holds that $\Gamma \vdash e_L : \sigma_1$, and τ_1 can be written as $\sigma_1 \vee \sigma_2$. Thus, $e_2 = e_L$ is well-typed of type σ_1, and by Lemma 6 we have $\sigma_1 \sqsubseteq \tau_1$.
  - By inversion of $(T^{\langle\rangle}_{case-L})$. It holds that $\Gamma \vdash e_L : \sigma_1$, and $\tau_1 = \sigma_1$. Hence the result by reflexivity of the relation $\sqsubseteq$.
  - By inversion of $(T^{\langle\rangle}_{case-R})$. This case cannot occur since the hypothesis $(H2)$ contradicts the premises of the rule $(T^{\langle\rangle}_{case-R})$.
  - By inversion of $(T^{\langle\rangle}_{case-none})$. It holds that $\Gamma \vdash v : \mathbb{0}$. Since, by hypothesis $(H1)$, v can be typed in an empty environment, we deduce that $\emptyset \vdash v : \mathbb{0}$. However, there is no value of type $\mathbb{0}$ in an empty environment, thus this case cannot occur.
- (R_{case-R}). In this case, we have $e_1 = (v \in t)?e_L : e_R$ and $e_2 = e_R$. Moreover, the following hypotheses hold:

$$(H1) \quad \emptyset \vdash v : \tau$$
$$(H2) \quad \tau^{\Uparrow} \not\leq t$$

  Once again, by inversion of the typing rules, we distinguish four possible cases:
  - By inversion of $(T^{\langle\rangle}_{case-both})$. It holds that $\Gamma \vdash e_R : \sigma_2$, and τ_1 can be written as $\sigma_1 \vee \sigma_2$. Thus, $e_2 = e_R$ is well-typed of type σ_2, and by Lemma 6 we have $\sigma_2 \sqsubseteq \tau_1$.
  - By inversion of $(T^{\langle\rangle}_{case-L})$. This case cannot occur since the hypothesis $(H2)$ contradicts the premises of the rule $(T^{\langle\rangle}_{case-L})$.
  - By inversion of $(T^{\langle\rangle}_{case-R})$. It holds that $\Gamma \vdash e_R : \sigma_2$, and $\tau_1 = \sigma_2$. Hence the result by reflexivity of the relation $\sqsubseteq$.
  - By inversion of $(T^{\langle\rangle}_{case-none})$. It holds that $\Gamma \vdash v : \mathbb{0}$. Since, by hypothesis $(H1)$, v can be typed in an empty environment, we deduce that $\emptyset \vdash v : \mathbb{0}$. However, there is no value of type $\mathbb{0}$ in an empty environment, thus this case cannot occur.
- $(R_{cast-\lambda})$. In this case, we have $e_1 = \langle\tau\rangle\, \lambda^{\mathbb{I}}_{\langle\tau'\rangle}\, x.e$ and $e_2 = \lambda^{\mathbb{I}}_{\langle\tau\rangle}\, x.e$.

  By inversion of the rule $(T^{\langle\rangle}_{cast})$ on e_1, we obtain that $\Gamma \vdash \lambda^{\mathbb{I}}_{\langle\tau'\rangle}\, x.e : \sigma$, for a gradual type σ, and that $\Gamma \vdash e_1 : \tau$. Then, by inversion of the rule $(T^{\langle\rangle}_{\lambda})$ on this term, we deduce that $\sigma = \tau'$ and:

$$\forall(\rho_1 \to \rho_2) \in \mathbb{I}, \quad \Gamma, x : \rho_1 \vdash e : \rho_2' \quad \text{where} \quad \rho_2' \sqsubseteq \rho_2$$

  Moreover, by hypothesis of $(R_{cast-\lambda})$, it holds that $\texttt{TypeOf}(\mathbb{I})^{\Downarrow} \leq \tau^{\Uparrow}$. We can therefore apply the rule $(T^{\langle\rangle}_{\lambda})$ to e_2 and deduce $\Gamma \vdash \lambda^{\mathbb{I}}_{\langle\tau\rangle}\, x.e : \tau$. Hence the result by reflexivity of $\sqsubseteq$.
- (R_{cast-c}). In this case, we have $e_1 = \langle\tau_1\rangle\, c$ and $e_2 = c$. Moreover, according to the premises of the rule (R_{cast-c}), it holds that $\emptyset \vdash c : s$ and $s \leq \tau_1^{\Uparrow}$.

  Since s is a base type (and thus not a function type), it is not possible that $\tau_1^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$ (or else, by transitivity of static subtyping, we would have $s \leq \mathbb{0} \to \mathbb{1}$). Moreover, it holds that $s^{\Uparrow} \leq \tau_1^{\Uparrow}$, since $s^{\Uparrow} = s$. Thus, we have $s \sqsubseteq \tau_1$.

- (R_E) where $e_1 = e_L e_R$ and $e_L \mapsto e'_L$. That is, $e_2 = e'_L e_R$. By inversion of the rule $(T^{\langle\rangle}_{app})$ for e_1, we can deduce the following hypotheses:

$$(H1) \quad \Gamma \vdash e_L : \sigma_1$$
$$(H2) \quad \Gamma \vdash e_R : \sigma_2$$
$$(H3) \quad \sigma_1^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$$
$$(H4) \quad \sigma_2^{\Uparrow} \leq \widetilde{dom}_{\mathscr{S}}(\sigma_1)$$
$$(H5) \quad \tau_1 = \sigma_1 \widetilde{\circ} \sigma_2$$

  In particular, given hypothesis $(H1)$, we can apply the induction hypothesis on the reduction $e_L \mapsto e'_L$ to deduce two more hypotheses:

$$(IH1) \quad \Gamma \vdash e'_L : \sigma'_1$$
$$(IH2) \quad \sigma'_1 \sqsubseteq \sigma_1$$

  Note that the hypothesis $(IH1)$ gives us the first premise to apply the rule $(T^{\langle\rangle}_{app})$ to $e'_L e_R$. The second premise is given by $(H2)$.
  Now, according to $(IH2)$ and the definition of $\sqsubseteq$, it holds that $\sigma_1'^{\Uparrow} \leq \sigma_1^{\Uparrow}$. Therefore, by transitivity of static subtyping and hypothesis $(H3)$, we deduce that $\sigma_1'^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$. Using once again $(IH2)$ with Lemma 7, we can deduce that $\widetilde{dom}_{\mathscr{S}}(\sigma_1) \leq \widetilde{dom}_{\mathscr{S}}(\sigma'_1)$. Thus, by hypothesis $(H4)$ and by transitivity, it holds that $\sigma_2^{\Uparrow} \leq \widetilde{dom}_{\mathscr{S}}(\sigma'_1)$, which is the last premise of the rule $(T^{\langle\rangle}_{app})$.
  We now have all the required hypotheses to apply the rule $(T^{\langle\rangle}_{app})$ to $e'_L e_R$, thus we can deduce that $e_2 = e'_L e_R$ is well-typed: $\Gamma \vdash e'_L e_R : \sigma'_1 \widetilde{\circ} \sigma_2$. Applying Lemma 8 with hypothesis $(IH2)$ and $(H3)$ gives us that $\sigma'_1 \widetilde{\circ} \sigma_2 \sqsubseteq \sigma_1 \widetilde{\circ} \sigma_2$, hence the result (by hypothesis $(H5)$).
- (R_E) where $e_1 = v\ e$ and $e \mapsto e'$. That is, $e_2 = v\ e'$. By inversion of the rule $(T^{\langle\rangle}_{app})$ for e_1, we can deduce the following hypotheses:

$$(H1) \quad \Gamma \vdash v : \sigma_1$$
$$(H2) \quad \Gamma \vdash e : \sigma_2$$
$$(H3) \quad \sigma_1^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$$
$$(H4) \quad \sigma_2^{\Uparrow} \leq \widetilde{dom}_{\mathscr{S}}(\sigma_1)$$
$$(H5) \quad \tau_1 = \sigma_1 \widetilde{\circ} \sigma_2$$

  In particular, we can use the induction hypothesis on e using hypothesis $(H2)$. We deduce that $\Gamma \vdash e' : \sigma'_2$ where $\sigma'_2 \sqsubseteq \sigma_2$. Using the definition of $\sqsubseteq$, and hypothesis $(H4)$ we deduce that $\sigma_2'^{\Uparrow} \leq \sigma_2^{\Uparrow} \leq \widetilde{dom}_{\mathscr{S}}(\sigma_1)$. Applying back the rule $(T^{\langle\rangle}_{app})$ to e_2, we deduce that $\Gamma \vdash v\ e' : \sigma_1 \widetilde{\circ} \sigma'_2$. We then conclude by applying Lemma 9.
- (R_E) where $e_1 = (e \in t)?e_L : e_R$ and $e \mapsto e'$. That is, $e_2 = (e' \in t)?e_L : e_R$. By inversion of the typing rules for typecases, we deduce (independently of the typing rule used): that $\Gamma \vdash e : \tau$. Applying the induction hypothesis to e yields $\Gamma \vdash e' : \tau'$ where $\tau' \sqsubseteq \tau$.
  We then distinguish four possible cases:
  - $\tau'^{\Uparrow} \not\leq \neg t$ and $\tau'^{\Uparrow} \not\leq t$. Since $\tau'^{\Uparrow} \leq \tau^{\Uparrow}$, and since static subtyping is transitive, it necessarily holds that $\tau^{\Uparrow} \not\leq \neg t$ and $\tau^{\Uparrow} \not\leq t$. Therefore, by inversion of $(T^{\langle\rangle}_{case-both})$, it holds that $\Gamma \vdash e_L : \tau_L$

and $\Gamma \vdash e_R : \tau_R$. Moreover, we have $\tau_1 \equiv \tau_L \vee \tau_R$.
Applying the rule $(T^{\langle\rangle}_{case-both})$ back to e_2, we deduce that $\Gamma \vdash e_2 : \tau_L \vee \tau_R$, hence the result.
- $\tau'^{\Uparrow} \not\leq \neg t$ and $\tau'^{\Uparrow} \leq t$. As before, it necessarily holds that $\tau^{\Uparrow} \not\leq \neg t$. Thus, we distinguish between two more cases.
  Either $\tau^{\Uparrow} \not\leq t$, in which case we can inverse the rule $(T^{\langle\rangle}_{case-both})$ on e_1, which gives us that $\Gamma \vdash e_L : \tau_L$ and $\Gamma \vdash e_R : \tau_R$, as well as $\tau_1 = \tau_L \vee \tau_R$. We can then apply the rule $(T^{\langle\rangle}_{case-L})$ on e_2, yielding $\Gamma \vdash e_2 : \tau_L$. We conclude using Corollary 4 to prove $\tau_L \sqsubseteq \tau_L \vee \tau_R$.
  The second case is $\tau^{\Uparrow} \leq t$. In this case we can inverse the rule $(T^{\langle\rangle}_{case-L})$ on e_1 yielding $\Gamma \vdash e_L : \tau_L$. Applying back the rule $(T^{\langle\rangle}_{case-L})$ on e_2 gives us that $\Gamma \vdash e_2 : \tau_L$, hence the result.
- $\tau'^{\Uparrow} \leq \neg t$ and $\tau'^{\Uparrow} \not\leq t$. This case is proved in the same way as the previous one.
- $\tau'^{\Uparrow} \leq \mathbb{0}$. This case is immediate by applying the rule $(T^{\langle\rangle}_{case-none})$ to $e2$ and Lemma 5.

- (R_E) where $e_1 = \langle\tau\rangle\ e$ and $e \mapsto e'$. This case is immediate since, by induction hypothesis, e' is well-typed. Applying the rule $(T^{\langle\rangle}_{cast})$ then proves that $\langle\tau\rangle\ e'$ has type τ.

□

Lemma 12. *(Progress) — For every term $e \in \textbf{\textit{Terms}}^{\langle\rangle}$, if $\emptyset \vdash e : \tau$ then $e \in \textbf{\textit{Values}}^{\langle\rangle}$ or $\exists e' \in \textbf{\textit{Terms}}^{\langle\rangle}, e \mapsto e'$ or $e \mapsto \texttt{CastError}$.*

Proof. Let $e \in \textbf{Terms}^{\langle\rangle}$ such that $\emptyset \vdash e : \tau$. The proof is done by induction on e and by cases over the last typing rule used in the proof $\emptyset \vdash e : \tau$.

- $(T^{\langle\rangle}_x)$. This case cannot happen since no variable is well-typed in the empty context.
- $(T^{\langle\rangle}_c)$. In this case, $e = c$, therefore $e \in \textbf{Values}^{\langle\rangle}$.
- $(T^{\langle\rangle}_\lambda)$. In this case, $e = \lambda^{\mathbb{I}}_{\langle\tau\rangle}\ x.\ e'$. Moreover, it holds by hypothesis that $\texttt{TypeOf}(\mathbb{I})^{\Downarrow} \leq \tau^{\Uparrow}$. Since $\texttt{TypeOf}(\mathbb{I})^{\Downarrow}$ is an intersection of arrows and cannot be empty, it holds that $\tau^{\Uparrow} \not\leq \mathbb{0}$. Therefore, $e \in \textbf{Values}^{\langle\rangle}$.
- $(T^{\langle\rangle}_{cast})$. In this case, $e = \langle\tau\rangle\ e'$, and by hypothesis $\emptyset \vdash e' : \sigma$. Therefore, by induction, several cases can occur:
  - $e' \in \textbf{Values}^{\langle\rangle}$, where $e' = \lambda^{\mathbb{I}}_{\langle\sigma\rangle}\ x.\ e_\lambda$. In this case, if $\texttt{TypeOf}(\mathbb{I})^{\Downarrow} \leq \tau^{\Uparrow}$ then $e \mapsto \lambda^{\mathbb{I}}_{\langle\tau\rangle}\ x.\ e_\lambda$ by rule $(R_{cast-\lambda})$. Otherwise, if $\texttt{TypeOf}(\mathbb{I})^{\Downarrow} \not\leq \tau^{\Uparrow}$, then $e \mapsto \texttt{CastError}$ by rule $(R_{fail-\lambda})$.
  - $e' \in \textbf{Values}^{\langle\rangle}$, where $e' = c$. In this case, $\emptyset \vdash c : B(c)$ by rule $(T^{\langle\rangle}_c)$. If $B(c) \leq \tau^{\Uparrow}$ then $e \mapsto c$ by rule (R_{cast-c}). Otherwise, if $B(c) \not\leq \tau^{\Uparrow}$ then $e \mapsto \texttt{CastError}$ by rule (R_{fail-c}).
  - $\exists e'' \in \textbf{Terms}^{\langle\rangle}$ such that $e' \mapsto e''$. Since $\langle\tau\rangle\ \square$ is a valid reduction context, $e \mapsto \langle\tau\rangle\ e''$ by rule (R_E).
  - $e' \mapsto \texttt{CastError}$. Since $\langle\tau\rangle\ \square$ is a valid reduction context, $e \mapsto \texttt{CastError}$ by rule (R_{fail-E}). This last subcase concludes the case for casts.
- $(T^{\langle\rangle}_{app})$. In this case, $e = e_1 e_2$ and $\tau = \tau_1 \widetilde{\circ} \tau_2$ where $\emptyset \vdash e_1 : \tau_1$ and $\emptyset \vdash e_2 : \tau_2$. Moreover, we have $\tau_1^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$ and $\tau_2^{\Uparrow} \leq \widetilde{dom}(\tau_1)$.
  Applying the induction hypothesis to e_1 yields the following cases.
  - $e_1 \in \textbf{Values}^{\langle\rangle}$, where $e_1 = c$. This case cannot occur since $\tau_1 = B(c) \not\leq \mathbb{0} \to \mathbb{1}$, which contradicts the hypothesis.
  - $e_1 \in \textbf{Values}^{\langle\rangle}$, where $e_1 = \lambda^{\mathbb{I}}_{\langle\tau_1\rangle}\ x.e'_1$. We can apply the induction hypothesis to e_2, yielding the following subcases.
    - $e_2 \in \textbf{Values}^{\langle\rangle}$, where $e_2 = c$. The application $e_1 e_2$ can be reduced either by (R_{app-c}) or by $(R_{app-fail-c})$, since both premises cannot hold at the same time.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\,(T_x) \qquad \frac{}{\Gamma \vdash c : B(c)}\,(T_c) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1^{\Downarrow} \leq \mathbb{0} \to \mathbb{1} \quad \tau_2^{\Downarrow} \leq \widetilde{dom}(\tau_1)}{\Gamma \vdash e_1\ e_2 : \tau_1 \widetilde{\circ} \tau_2}\,(T_{app})$$

$$\frac{\forall (\sigma \to \tau) \in \mathbb{I}, \quad \Gamma, x : \sigma \vdash e : \tau' \quad \tau'^{\Downarrow} \leq \tau^{\Uparrow}}{\lambda^{\mathbb{I}} x.\ e : \mathtt{TypeOf}(\mathbb{I})}\,(T_\lambda) \qquad \frac{\Gamma \vdash e : \tau \quad \begin{cases} \tau^{\Uparrow} \not\leq \neg t & \Gamma \vdash e_1 : \sigma_1 \\ \tau^{\Uparrow} \not\leq t & \Gamma \vdash e_2 : \sigma_2 \end{cases}}{\Gamma \vdash ((e \in t)?e_1 : e_2) : \sigma_1 \vee \sigma_2}\,(T_{case-both})$$

$$\frac{\Gamma \vdash e : \tau \quad \begin{cases} \tau^{\Uparrow} \not\leq \neg t \\ \tau^{\Uparrow} \leq t \end{cases} \quad \Gamma \vdash e_1 : \sigma_1}{\Gamma \vdash ((e \in t)?e_1 : e_2) : \sigma_1}\,(T_{case-L}) \qquad \frac{\Gamma \vdash e : \tau \quad \begin{cases} \tau^{\Uparrow} \leq \neg t \\ \tau^{\Uparrow} \not\leq t \end{cases} \quad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash ((e \in t)?e_1 : e_2) : \sigma_2}\,(T_{case-R})$$

$$\frac{\Gamma \vdash e : \tau \quad \tau^{\Uparrow} \leq \mathbb{0}}{\Gamma \vdash ((e \in t)?e_1 : e_2) : \tau}\,(T_{case-none})$$

Fig. 7. Full typing rules for the gradually typed language

  - $e_2 \in \mathbf{Values}^{\langle\rangle}$, where $e_2 = \lambda^{\mathbb{I}'}_{\langle\tau_2\rangle}\, x.e_2'$. As before, the application $e_1 e_2$ can be reduced either by $(R_{app-\lambda})$ or $(R_{app-fail-\lambda})$ since both premises cannot hold at the same time.
  - $\exists e_2' \in \mathbf{Terms}^{\langle\rangle}$ such that $e_2 \mapsto e_2'$. In this case, since e_1 is a value, the reduction context $e_1\ \Box$ is valid, and thus $e_1 e_2 \mapsto e_1 e_2'$ by rule (R_E).
  - $e_2 \mapsto \mathtt{CastError}$. Once again, since e_1 is a value, the reduction context $e_1\ \Box$ is valid, thus $e_1 e_2 \mapsto \mathtt{CastError}$ by rule (R_{fail-E}).
- $\exists e_1' \in \mathbf{Terms}^{\langle\rangle}$ such that $e_1 \mapsto e_1'$. The reduction context $\Box\ e_2$ is valid, thus $e_1 e_2 \mapsto e_1' e_2$ by rule (R_E).
- $e_1 \mapsto \mathtt{CastError}$. The reduction context $\Box\ e_2$ is valid, thus $e_1 e_2 \mapsto \mathtt{CastError}$ by rule (R_{fail-E}). This last subcase concludes the case for applications.

- $(T^{\langle\rangle}_{case-x})$. All these cases are treated in the same way. We have $e = (e' \in t)?e_1 : e_2$ and $\emptyset \vdash e' : \tau'$. We can therefore apply the induction hypothesis to e' which yields the following cases.
  - $e' \in \mathbf{Values}^{\langle\rangle}$. Since either $B(c) \leq t$ or $B(c) \not\leq t$ hold, e reduces to e_1 by rule (R_{case-L}) or to e_2 by rule (R_{case-R}).
  - $\exists e'' \in \mathbf{Terms}^{\langle\rangle}$ such that $e' \mapsto e''$. Since $(\Box \in t)?e_1 : e_2$ is a valid reduction context, $e \mapsto (e'' \in t)?e_1 : e_2$ by rule (R_E).
  - $e' \mapsto \mathtt{CastError}$. Since $(\Box \in t)?e_1 : e_2$ is a valid reduction context, $e \mapsto \mathtt{CastError}$ by rule (R_{fail-E}). This last subcase concludes the case for typecases and the proof.

□

Theorem 6. *(Soundness of Cast Language) — For every term $e \in \boldsymbol{Terms}^{\langle\rangle}$, if $\emptyset \vdash e : \tau$ then either e diverges or $\exists v \in \boldsymbol{Values}^{\langle\rangle}$ such that $e \mapsto^* v$ or $e \mapsto^* \mathtt{CastError}$.*

Proof. Direct consequence of Lemmas 11 and 12. □

A.3 Compilation

A.3.1 Soundness of compilation. In this first part, we prove the soundness of the compilation and the resulting safety property for the gradually-typed language.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x \rightsquigarrow x : \tau}\ (C_x) \qquad \frac{}{\Gamma \vdash c \rightsquigarrow c : B(c)}\ (C_c)$$

$$\frac{\Gamma \vdash e \rightsquigarrow e' : \tau \quad \begin{cases} \tau^{\Uparrow} \not\leq \neg t & \Gamma \vdash e_1 \rightsquigarrow e_1' : \sigma_1 \\ \tau^{\Uparrow} \not\leq t & \Gamma \vdash e_2 \rightsquigarrow e_2' : \sigma_2 \end{cases}}{\Gamma \vdash ((e \in t)?e_1 : e_2) \rightsquigarrow ((e' \in t)?e_1' : e_2') : \sigma_1 \vee \sigma_2}\ (C_{case-both})$$

$$\frac{\Gamma \vdash e \rightsquigarrow e' : \tau \quad \begin{cases} \tau^{\Uparrow} \not\leq \neg t \\ \tau^{\Uparrow} \leq t \end{cases} \quad \Gamma \vdash e_1 \rightsquigarrow e_1' : \sigma_1}{\Gamma \vdash ((e \in t)?e_1 : e_2) \rightsquigarrow e_1' : \sigma_1}\ (C_{case-L})$$

$$\frac{\Gamma \vdash e \rightsquigarrow e' : \tau \quad \begin{cases} \tau^{\Uparrow} \leq \neg t \\ \tau^{\Uparrow} \not\leq t \end{cases} \quad \Gamma \vdash e_2 \rightsquigarrow e_2' : \sigma_2}{\Gamma \vdash ((e \in t)?e_1 : e_2) \rightsquigarrow e_2' : \sigma_2}\ (C_{case-R})$$

$$\frac{\Gamma \vdash e \rightsquigarrow e' : \tau \quad \tau^{\Uparrow} \leq \mathbb{0}}{\Gamma \vdash ((e \in t)?e_1 : e_2) \rightsquigarrow e' : \tau}\ (C_{case-none})$$

$$\frac{\begin{array}{ll} \Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1 & \tau_2^{\Uparrow} \leq \widetilde{dom}_{\mathscr{J}}(\tau_1) \\ \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_2 & \tau_1^{\Uparrow} \leq \mathbb{0} \to \mathbb{1} \end{array}}{\Gamma \vdash e_1\ e_2 \rightsquigarrow e_1'\ e_2' : \tau_1 \widetilde{\circ} \tau_2}\ (C_{app-1})$$

$$\frac{\begin{array}{lll} \Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1 & \tau_2^{\Uparrow} \not\leq \widetilde{dom}_{\mathscr{J}}(\tau_1) & \\ \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_2 & \tau_2^{\Downarrow} \leq \widetilde{dom}_{\mathscr{J}}(\tau_1) & \tau_1^{\Uparrow} \leq \mathbb{0} \to \mathbb{1} \end{array}}{\Gamma \vdash e_1\ e_2 \rightsquigarrow \langle \tau_1 \widetilde{\circ} \tau_2 \rangle\ (e_1'\ \langle \widetilde{dom}_{\mathscr{J}}(\tau_1) \rangle\ e_2') : \tau_1 \widetilde{\circ} \tau_2}\ (C_{app-2})$$

$$\frac{\begin{array}{ll} \Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1 & \\ \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_2 & \tau_1^{\Uparrow} \not\leq \mathbb{0} \to \mathbb{1} \quad \text{or} \quad \tau_2^{\Downarrow} \not\leq \widetilde{dom}_{\mathscr{J}}(\tau_1) \end{array}}{\Gamma \vdash e_1\ e_2 \rightsquigarrow (\langle \tau_2 \to (\tau_1 \widetilde{\circ} \tau_2) \rangle\ e_1')\ e_2' : \tau_1 \widetilde{\circ} \tau_2}\ (C_{app-3})$$

$$\frac{\begin{array}{ll} \forall \sigma_i \to \tau_i \in \mathbb{I}, & \\ \Gamma, x : \sigma_i \vdash e \rightsquigarrow e_i : \tau_i' & e_i' = \begin{cases} e_i & \text{if } \tau_i' \sqsubseteq \tau_i \\ \langle \tau_i \rangle\ e_i & \text{otherwise} \end{cases} \end{array}}{\Gamma \vdash \lambda^{\mathbb{I}} x.\ e \rightsquigarrow (\lambda^{\mathbb{I}} x.(x \in \sigma_1^{\Uparrow})?\ e_1' : \cdots : (x \in \sigma_{i-1}^{\Uparrow})?\ e_{i-1}' : e_i') : \mathtt{TypeOf}(\mathbb{I})}\ (C_\lambda)$$

Fig. 8. Full compilation rules for the gradually-typed language

Lemma 13. *(Exhaustiveness of Compilation) — For every term $e \in \textbf{\textit{Terms}}$ and every typing context Γ, if $\Gamma \vdash e : \tau$ then $\Gamma \vdash e \rightsquigarrow e' : \tau$ where $e' \in \textbf{\textit{Terms}}^{\langle\rangle}$.*

Proof. Let Γ be any typing context and $e \in \textbf{Terms}$, such that $\Gamma \vdash e : \tau$. The proof is done by induction and case disjunction on the last typing rule used in the derivation $\Gamma \vdash e : \tau$.

- (T_x). We have, by hypothesis, $e = x$ and $x : \tau \in \Gamma$. Therefore, the rule (C_x) can be applied, yielding $\Gamma \vdash x \rightsquigarrow x : \tau$.
- (T_c). In this case, $e = c$ and $\Gamma \vdash c : B(c)$. The rule (C_c) gives immediately the result: $\Gamma \vdash c \rightsquigarrow c : B(c)$.
- (T_{app}). By hypothesis, $e = e_1 e_2$ and $\Gamma \vdash e_1 e_2 : \tau_1 \widetilde{\circ} \tau_2$, where $\Gamma \vdash e_1 : \tau_1$ and $\Gamma \vdash e_2 : \tau_2$. Applying the induction hypothesis to e_1 and e_2, we deduce:

$$(IH1) \quad \Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1$$
$$(IH2) \quad \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_2$$

  We then distinguish the following cases on τ_1:
  - $\tau_1^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$. We distinguish the following subcases on τ_2:
    * $\tau_2^{\Uparrow} \leq \widetilde{dom}_{\mathscr{S}}(\tau_1)$. We can apply the rule (C_{app-1}) using both $(IH1)$ and $(IH2)$ yielding $\Gamma \vdash e_1 e_2 \rightsquigarrow e_1' e_2' : \tau_1 \widetilde{\circ} \tau_2$, which is the result.
    * $\tau_2^{\Uparrow} \not\leq \widetilde{dom}_{\mathscr{S}}(\tau_1)$ but $\tau_2^{\Downarrow} \leq \widetilde{dom}_{\mathscr{S}}(\tau_1)$. In this case, we can apply the rule (C_{app-2}) to deduce $\Gamma \vdash e_1 e_2 \rightsquigarrow \langle \tau_1 \widetilde{\circ} \tau_2 \rangle \ (e_1' \langle \widetilde{dom}_{\mathscr{S}}(\tau_1) \rangle \ e_2') : \tau_1 \widetilde{\circ} \tau_2$, which is the result.
    * $\tau_2^{\Downarrow} \not\leq \widetilde{dom}_{\mathscr{S}}(\tau_1)$. We can apply the rule (C_{app-3}) which yields $\Gamma \vdash e_1 e_2 \rightsquigarrow (\langle \tau_2 \to (\tau_1 \widetilde{\circ} \tau_2) \rangle \ e_1') e_2' : \tau_1 \widetilde{\circ} \tau_2$, hence the result.
  - $\tau_1^{\Uparrow} \not\leq \mathbb{0} \to \mathbb{1}$. In this case, the rule (C_{app-3}) can be applied using both $(IH1)$ and $(IH2)$ yielding $\Gamma \vdash e_1 e_2 \rightsquigarrow (\langle \tau_2 \to (\tau_1 \widetilde{\circ} \tau_2) \rangle \ e_1') e_2' : \tau_1 \widetilde{\circ} \tau_2$ which is the result.
- (T_λ). By hypothesis, $e = \lambda^{\mathbb{I}} x.e'$ and $\Gamma \vdash e : \mathtt{TypeOf}(\mathbb{I})$. Moreover, for every $(\sigma_i \to \tau_i) \in \mathbb{I}$, it holds that $\Gamma, x : \sigma_i \vdash e' : \tau_i'$ where $\tau_i'^{\Downarrow} \leq \tau_i^{\Uparrow}$.
  Applying the induction hypothesis to e' for every typing context $\Gamma, x : \sigma_i$, we deduce the following hypotheses:

$$(H) \quad \forall (\sigma_i \to \tau_i) \in \mathbb{I},\ \Gamma, x : \sigma_i \vdash e' \rightsquigarrow e'' : \tau_i' \text{ where } \tau_i'^{\Downarrow} \leq \tau_i^{\Uparrow}$$

  Thus, we can apply the rule (C_λ) to deduce that

$$\Gamma \vdash \lambda^{\mathbb{I}} x.\ e \rightsquigarrow (\lambda^{\mathbb{I}} x.(x \in \sigma_1^{\Uparrow})?\ e_1' : \cdots : (x \in \sigma_{i-1}^{\Uparrow})?\ e_{i-1}' : e_i') : \mathtt{TypeOf}(\mathbb{I})$$

  Hence the result.
- $(T_{case-both})$. By hypothesis, $e = (e_t \in t)?e_1 : e_2$ and $\Gamma \vdash e : \sigma_1 \vee \sigma_2$ where $\Gamma \vdash e_t : \sigma, \Gamma \vdash e_1 : \sigma_1$ and $\Gamma \vdash e_2 : \sigma_2$.
  Applying the induction hypothesis to e_t, e_1 and e_2, we deduce:

$$(H1) \quad \Gamma \vdash e_t \rightsquigarrow e_t' : \sigma$$
$$(H2) \quad \Gamma \vdash e_1 \rightsquigarrow e_1' : \sigma_1$$
$$(H3) \quad \Gamma \vdash e_2 \rightsquigarrow e_2' : \sigma_2$$

  Moreover, it holds, by hypothesis, that $\sigma^{\Uparrow} \not\leq t$ and $\sigma^{\Uparrow} \not\leq \neg t$. Therefore, we can apply the rule $(C_{case-both})$ to deduce $\Gamma \vdash e \rightsquigarrow (e_t' \in t)?e_1' : e_2' : \sigma_i \vee \sigma_2$, which is the result.
- (T_{case-L}). By hypothesis, $e = (e_t \in t)?e_1 : e_2$ and $\Gamma \vdash e : \sigma_1$ where $\Gamma \vdash e_t : \sigma$ and $\Gamma \vdash e_1 : \sigma_1$.
  Applying the induction hypothesis to e_1, we deduce that $\Gamma \vdash e_1 \rightsquigarrow e_1' : \sigma_1$. Moreover, it holds by hypothesis that $\sigma^{\Uparrow} \not\leq \neg t$ and $\sigma^{\Uparrow} \leq t$. Therefore, we can apply the rule (C_{case-L}) to deduce that $\Gamma \vdash e \rightsquigarrow e_1' : \sigma_1$, which is the result.
- (T_{case-R}) this case is proved identically to the previous one.

- $(T_{case-none})$. In this case, $e = (e_t \in t)?e_1 : e_2$ and $\Gamma \vdash e : \sigma$ where $\Gamma \vdash e_t : \sigma$. Moreover, σ verifies $\sigma^{\Uparrow} \leq \mathbb{0}$. Applying the induction hypothesis to e_t yields $\Gamma \vdash e_t \rightsquigarrow e'_t : \sigma$. Thus, we can apply the rule $(C_{case-none})$ to deduce that $\Gamma \vdash e \rightsquigarrow e' : \sigma$, which is the result. This case concludes the proof.

□

Lemma 14. *Let Γ be any typing context and let $\sigma_1, \ldots, \sigma_n$ be any gradual types verifying:*

$$\forall (i,j) \in \{1;n\}^2, i \neq j \implies \sigma_i^{\Uparrow} \wedge \sigma_j^{\Uparrow} \leq \mathbb{0}$$

$$\forall i \in \{1;n\}, \sigma_i^{\Uparrow} \not\leq \mathbb{0}$$

Let $e_1, \ldots, e_n \in \boldsymbol{Terms}^{\langle\rangle}$ verifying:

$$\forall i \in \{1;n\}, \quad \Gamma, x : \sigma_i \vdash e_i : \tau_i$$

Then the following holds:

$$\forall i \in \{1;n\}, \quad \Gamma, x : \sigma_i \vdash (x \in \sigma_1^{\Uparrow})?e_1 : \ldots : (x \in \sigma_{n-1}^{\Uparrow})?e_{n-1} : e_n : \tau_i$$

Proof. Let Γ be any typing context. The proof is done by induction on n.

- $n = 1$. Let σ_1 be any gradual type and $e_1 \in \mathbf{Terms}^{\langle\rangle}$ such that $\Gamma, x : \sigma_1 \vdash e_1 : \tau_1$. This case is immediate since this hypothesis is exactly the result.
- $n + 1$. Let $\sigma_1, \ldots, \sigma_{n+1}$ be any gradual types and $e_1, \ldots, e_{n+1} \in \mathbf{Terms}^{\langle\rangle}$ verifying the aforementioned criteria. We pose $e = (x \in \sigma_1^{\Uparrow})?e_1 : \ldots : (x \in \sigma_{n-1}^{\Uparrow})?e_{n-1} : e_n$.
  Let $i \in \{1;n\}$. We distinguish the following cases on i:
  - $i = 1$. By reflexivity of subtyping, it holds that $\sigma_1^{\Uparrow} \leq \sigma_1^{\Uparrow}$. Moreover, by hypothesis, $\sigma_1^{\Uparrow} \not\leq \mathbb{0}$. Therefore, $\sigma_1^{\Uparrow} \not\leq \neg\sigma_1^{\Uparrow}$. Since we also know, by hypothesis, that $\Gamma, x : \sigma_1 \vdash e_1 : \tau_1$, we can apply the rule $(T^{\langle\rangle}_{case-L})$ to deduce $\Gamma, x : \sigma_1 \vdash e : \tau_1$, which is the result.
  - $i > 1$. By hypothesis, we have $\sigma_1^{\Uparrow} \wedge \sigma_i^{\Uparrow} \leq \mathbb{0}$. This is, for non-gradual set-theoretic types, equivalent to $\sigma_i^{\Uparrow} \leq \neg\sigma_1^{\Uparrow}$. Moreover, by hypothesis it holds that $\sigma_i^{\Uparrow} \not\leq \mathbb{0}$. Therefore, it necessarily holds that $\sigma_i^{\Uparrow} \not\leq \sigma_1^{\Uparrow}$. Applying the induction hypothesis to $e' = (x \in \sigma_2^{\Uparrow})?e_2 : \ldots : (x \in \sigma_{n-1}^{\Uparrow})?e_{n-1} : e_n$, we deduce that $\Gamma, x : \sigma_i \vdash e' : \tau_i$. Thus, we can apply the rule $(T^{\langle\rangle}_{case-R})$ to e, which yields $\Gamma, x : \sigma_i \vdash e : \tau_i$, hence the result.

□

Lemma 15. *(Type Preservation by Compilation) — For every term $e \in \boldsymbol{Terms}$ and every typing context Γ, if $\Gamma \vdash e \rightsquigarrow e' : \tau$ then $\Gamma \vdash e' : \tau$.*

Proof. Let Γ be any typing context and $e \in \mathbf{Terms}$, such that $\Gamma \vdash e \rightsquigarrow e' : \tau$. We show that $\Gamma \vdash e' : \tau$ by induction on e and case disjunction on the rule used to compiled e.

- (C_x). That is, $\Gamma \vdash x \rightsquigarrow x : \tau$. By hypothesis, $x : \tau \in \Gamma$, thus we can apply the rule $(T_x^{\langle\rangle})$ to deduce $\Gamma \vdash x : \tau$.
- (C_x). In this case, $\Gamma \vdash c \rightsquigarrow c : B(c)$. We can immediately apply the rule $(T_c^{\langle\rangle})$ to deduce that $\Gamma \vdash c : B(c)$.

- $(C_{case-both})$. That is, $e = ((e_t \in t)?e_1 : e_2)$ and we have the following hypotheses:

$$
\begin{array}{ll}
(H0) & \Gamma \vdash e \rightsquigarrow ((e'_t \in t)?e'_1 : e'_2) : \sigma_1 \vee \sigma_2 \\
(H1) & \Gamma \vdash e_t \rightsquigarrow e'_t : \tau_t \\
(H2) & \Gamma \vdash e_1 \rightsquigarrow e'_1 : \sigma_1 \\
(H3) & \Gamma \vdash e_2 \rightsquigarrow e'_2 : \sigma_2 \\
(H4) & \tau_t^{\Uparrow} \not\leq \neg t \\
(H5) & \tau_t^{\Uparrow} \not\leq t
\end{array}
$$

  By induction hypothesis on $(H1)$, $(H2)$ and $(H3)$, we deduce that $\Gamma \vdash e'_t : \tau_t$, $\Gamma \vdash e'_1 : \sigma_1$ and $\Gamma \vdash e'_2 : \sigma_2$. Using the rule $(T^{\langle\rangle}_{case-both})$ with hypotheses $(H4)$ and $(H5)$, we then deduce that $\Gamma \vdash ((e'_t \in t)?e'_1 : e'_2) : \sigma_1 \vee \sigma_2$, which is the result.
- (C_{case-L}). That is, $e = ((e_t \in t)?e_1 : e_2)$ and $\Gamma \vdash e \rightsquigarrow e'_1 : \sigma_1$ where $\Gamma \vdash e_1 \rightsquigarrow e'_1 : \sigma_1$. By induction hypothesis on this derivation, it holds that $\Gamma \vdash e'_1 : \sigma_1$, hence the result.
- (C_{case-R}). We have $e = ((e_t \in t)?e_1 : e_2)$ and $\Gamma \vdash e \rightsquigarrow e'_2 : \sigma_2$ where $\Gamma \vdash e_2 \rightsquigarrow e'_2 : \sigma_2$. By induction hypothesis on this derivation, it holds that $\Gamma \vdash e'_2 : \sigma_2$, hence the result.
- $(C_{case-none})$. By hypothesis, $e = ((e_t \in t)?e_1 : e_2)$ and $\Gamma \vdash e \rightsquigarrow e'_t : \tau$ where $\Gamma \vdash e_t \rightsquigarrow e'_t : \tau$. By induction hypothesis, it holds that $\Gamma \vdash e'_t : \tau$, hence the result.
- (C_{app-1}). In this case, $\Gamma \vdash e_1\ e_2 \rightsquigarrow e'_1\ e'_2 : \tau_1 \widetilde{\circ} \tau_2$ where $\Gamma \vdash e_1 \rightsquigarrow e'_1 : \tau_1$ and $\Gamma \vdash e_2 \rightsquigarrow e'_2 : \tau_2$. By induction hypothesis, we deduce that $\Gamma \vdash e'_1 : \tau_1$ and $\Gamma \vdash e'_2 : \tau_2$. Moreover, by hypothesis of rule (C_{app-1}), it holds that $\tau_2^{\Uparrow} \leq \widetilde{dom}_{\mathscr{S}}(\tau_1)$ and $\tau_1^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$. Therefore, we can apply the rule $(T^{\langle\rangle}_{app})$ to deduce that $\Gamma \vdash e'_1 e'_2 : \tau_1 \widetilde{\circ} \tau_2$, which is the result.
- (C_{app-2}). By hypothesis, we have $\Gamma \vdash e_1\ e_2 \rightsquigarrow \langle \tau_1 \widetilde{\circ} \tau_2 \rangle\ (e'_1 \langle \widetilde{dom}_{\mathscr{S}}(\tau_1) \rangle\ e'_2) : \tau_1 \widetilde{\circ} \tau_2$, where $\Gamma \vdash e_1 \rightsquigarrow e'_1 : \tau_1$ and $\Gamma \vdash e_2 \rightsquigarrow e'_2 : \tau_2$.
  By induction hypothesis, we deduce that $\Gamma \vdash e'_2 : \tau_2$. Thus, applying the rule $(T^{\langle\rangle}_{cast})$ yields $\Gamma \vdash \langle \widetilde{dom}_{\mathscr{S}}(\tau_1) \rangle\ e'_2 : \widetilde{dom}_{\mathscr{S}}(\tau_1)$.
  Moreover, by induction hypothesis, we can also deduce that $\Gamma \vdash e'_1 : \tau_1$. Since by hypothesis of (C_{app-2}) it holds that $\tau_1^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$, we can apply the rule $(T^{\langle\rangle}_{app})$ to deduce that $\Gamma \vdash (e'_1 \langle \widetilde{dom}_{\mathscr{S}}(\tau_1) \rangle\ e'_2) : \tau_1 \widetilde{\circ} \widetilde{dom}_{\mathscr{S}}(\tau_1)$. Thus, we can apply the rule $(T^{\langle\rangle}_{cast})$ to finaly deduce that $\Gamma \vdash \langle \tau_1 \widetilde{\circ} \tau_2 \rangle\ (e'_1 \langle \widetilde{dom}_{\mathscr{S}}(\tau_1) \rangle\ e'_2) : \tau_1 \widetilde{\circ} \tau_2$, which is the result.
- (C_{app-3}). By hypothesis, we have $\Gamma \vdash e_1\ e_2 \rightsquigarrow (\langle \tau_2 \to (\tau_1 \widetilde{\circ} \tau_2) \rangle\ e'_1) e'_2 : \tau_1 \widetilde{\circ} \tau_2$, where $\Gamma \vdash e_1 \rightsquigarrow e'_1 : \tau_1$ and $\Gamma \vdash e_2 \rightsquigarrow e'_2 : \tau_2$.
  By induction hypothesis on e_1, we deduce that $\Gamma \vdash e'_1 : \tau_1$. Therefore, we can apply the rule $(T^{\langle\rangle}_{cast})$ to deduce that $\Gamma \vdash (\langle \tau_2 \to (\tau_1 \widetilde{\circ} \tau_2) \rangle\ e'_1) : \tau_2 \to (\tau_1 \widetilde{\circ} \tau_2)$.
  It holds that $(\tau_2 \to (\tau_1 \widetilde{\circ} \tau_2))^{\Uparrow} \leq \mathbb{0} \to \mathbb{1}$. Moreover, by definition of the safe domain, $\widetilde{dom}_{\mathscr{S}}(\tau_2 \to (\tau_1 \widetilde{\circ} \tau_2)) = \tau_2^{\Uparrow}$. Applying the induction hypothesis to e_2 we then deduce $\Gamma \vdash e'_2 : \tau_2$, which verifies $\tau_2^{\Uparrow} \leq \widetilde{dom}_{\mathscr{S}}(\tau_2 \to (\tau_1 \widetilde{\circ} \tau_2))$ by reflexivity of static subtyping.
  We can therefore apply the rule $(T^{\langle\rangle}_{app})$, deducing the result: $\Gamma \vdash (\langle \tau_2 \to (\tau_1 \widetilde{\circ} \tau_2) \rangle\ e'_1) e'_2 : \tau_1 \widetilde{\circ} \tau_2$.
- (C_λ). In this case, we have $\Gamma \vdash \lambda^{\mathbb{I}} x.e' \rightsquigarrow (\lambda^{\mathbb{I}} x.(x \in \sigma_1^{\Uparrow})?\ e'_1 : \cdots : (x \in \sigma_{i-1}^{\Uparrow})?\ e'_{i-1} : e'_i) : \mathtt{TypeOf}(\mathbb{I})$ under the following hypotheses:

$$\forall (\sigma_i \to \tau_i) \in \mathbb{I}, \quad \Gamma, x : \sigma_i \vdash e' \rightsquigarrow e_i : \tau'_i$$

and

$$e'_i = \begin{cases} e_i & \text{if } \tau'_i \sqsubseteq \tau_i \\ \langle \tau_i \rangle\, e_i & \text{otherwise} \end{cases}$$

Let $(\sigma_i \to \tau_i) \in \mathbb{I}$. We want to show that $\Gamma, x : \sigma_i \vdash (x \in \sigma_1^{\Uparrow})?\ e'_1 : \cdots : (x \in \sigma_{i-1}^{\Uparrow})?\ e'_{i-1} : e'_i : \rho_i$ where $\rho_i \sqsubseteq \tau_i$, to be able to apply the rule $(T_\lambda^{\langle\rangle})$.
First of all, we can apply the induction hypothesis to $\Gamma, x : \sigma_i \vdash e' \rightsquigarrow e_i : \tau'_i$, yielding $\Gamma, x : \sigma_i \vdash e_i : \tau'_i$. We then distinguish the following cases:

- If $\tau'_i \sqsubseteq \tau_i$. We have $e'_i = e_i$, and thus $\Gamma, x : \sigma_i \vdash e'_i : \tau'_i$, with $\tau'_i \sqsubseteq \tau_i$ by hypothesis. Hence the result using Lemma 14.
- Otherwise, we have $e'_i = \langle \tau_i \rangle\, e_i$. Since we know by induction hypothesis that $\Gamma, x : \sigma_i \vdash e_i : \tau'_i$, we can apply the rule $(T_{cast}^{\langle\rangle})$, yielding $\Gamma, x : \sigma_i \vdash \langle \tau_i \rangle\, e_i : \tau_i$. By reflexivity of $\sqsubseteq$, it holds that $\tau_i \sqsubseteq \tau_i$, hence the result using Lemma 14.

We can therefore apply the rule $(T_\lambda^{\langle\rangle})$ to the compiled function (which contains an implicit identity cast), deducing the result:

$$\Gamma \vdash (\lambda^{\mathbb{I}} x.(x \in \sigma_1^{\Uparrow})?\ e'_1 : \cdots : (x \in \sigma_{i-1}^{\Uparrow})?\ e'_{i-1} : e'_i) : \texttt{TypeOf}(\mathbb{I})$$

□

Theorem 7. *(Soundness of Compilation) — For every term $e \in \textbf{\textit{Terms}}$ and every typing context Γ, if $\Gamma \vdash e : \tau$ then $\Gamma \vdash e \rightsquigarrow e' : \tau$, where $e' \in \textbf{\textit{Terms}}^{\langle\rangle}$ and $\Gamma \vdash e' : \tau$.*

Proof. Direct consequence of Lemmas 13 and 15. □

Corollary 5. *(Safety of the Gradually-Typed Language) — For every term $e \in \textbf{\textit{Terms}}$, if $\emptyset \vdash e : \tau$, then $e \rightsquigarrow e' : \tau$ where $e' \in \textbf{\textit{Terms}}^{\langle\rangle}$ and either e' diverges, or $\exists v \in \textbf{\textit{Values}}^{\langle\rangle}$ such that $e' \mapsto^* v$ and $\emptyset \vdash v : \tau' \mathrel{\widetilde{\leq}} \tau$, or $e' \mapsto^* \texttt{CastError}$.*

Proof. Direct consequence of Theorems 6 and 7. □