A Gradual Interpretation of Union Types

Matías Toro(✉) and Éric Tanter

PLEIAD Laboratory, Computer Science Department (DCC),
University of Chile, Santiago, Chile
{mtoro,etanter}@dcc.uchile.cl

Abstract. Union types allow to capture the possibility of a term to be of several possibly unrelated types. Traditional static approaches to union types are untagged and tagged unions, which present dual advantages in their use. Inspired by recent work on using abstract interpretation to understand gradual typing, we present a novel design for union types, called gradual union types. Gradual union types combine the advantages of tagged and untagged union types, backed by dynamic checks. Seen as a gradual typing discipline, gradual union types are restricted imprecise types that denote a finite number of static types. We apply the Abstracting Gradual Typing (AGT) methodology of Garcia et al. to derive the static and dynamic semantics of a language that supports both gradual unions and the traditional, totally-unknown type. We uncover that gradual unions interact with the unknown type in a way that mandates a stratified approach to AGT, relying on a composition of two distinct abstract interpretations in order to retain optimality. Thanks to the abstract interpretation framework, the resulting language is type safe and satisfies the refined criteria for gradual languages. We also show how to compile such a language to a threesome cast calculus, and prove that the compilation preserves the semantics and properties of the language.

1 Introduction

Gradual typing originated as an approach to smoothly combine static and dynamic type checking within the same programming language [29]. Over the years, gradual typing has been applied to languages with more advanced features, such as objects [26], polymorphism [2], and type inference [13] among others. Gradual typing has also been developed beyond the original static/dynamic typing dualistic view to accommodate the integration of static typing disciplines of different strengths, such as information-flow typing [10], effects [3], and logical refinements [21].

Recently, Garcia et al. [14] identified that the general framework of Abstract Interpretation (AI) [9] can be applied, *at the type level*, to lay down solid foundations of gradual typing in its various forms, thereby justifying several design

M. Toro—Funded by CONICYT-PCHA/Doctorado Nacional/2015-21150510.
É. Tanter—Partially funded by FONDECYT Project 1150017.

F. Ranzato (Ed.): SAS 2017, LNCS 10422, pp. 382–404, 2017.
DOI: 10.1007/978-3-319-66706-5_19

decisions and criteria that were originally discovered and refined by trial-and-error. In essence, the Abstracting Gradual Typing approach (AGT for short) conceives of gradual types as abstracting (in the AI sense) a set of possible static types. Exploiting an underlying Galois connection, one can systematically derive a gradually-typed language that is crisply connected to the original static discipline, and satisfies a number of essential criteria for such languages [30].

Stepping back, AGT reinforces an even broader interpretation of gradual typing: that of soundly dealing with *imprecision* at the type level. Indeed, one can see dynamically-typed languages as languages with highly-imprecise static type information, and the original gradually-typed languages as allowing to reason about partial type information. For instance, consider a function f of gradual type $\mathsf{Int} \to ?$; this type is imprecise in that it does not provide any information about the values returned by f, but it does specify precisely that f is a function, which furthermore expects an integer argument. Therefore, the gradual language can statically reject $f + 1$ or $f(\mathsf{true})$, accept $f(1)$, and optimistically accept $f(1) + 2$ subject to a *dynamic check* that the value of $f(1)$ is indeed an integer. Similarly, integrating a simply-typed language with gradual support for effects [3] can be viewed as dealing with imprecision of effect information.

Inspired by this focus on imprecision, we observe that standard static type systems have long been proposed to deal with a basic form of imprecision: the possibility for a value to be of several, possibly-unrelated types. In the literature, two approaches have been developed to safely, and fully statically, deal with the possibility of an expression to have possibly different types: *disjoint (or tagged) union types*, such as sum types $T_1 + T_2$ and variant types, and *untagged union types*, usually noted $T_1 \vee T_2$ [23]. Both forms of union types have complementary pros and cons when viewed from a pragmatic angle.

The understanding of both gradual types and union types as different ways to deal with imprecision at the type level suggests a novel, *gradual* interpretation of union types. Following the abstract interpretation of gradual types put forth in AGT, a *gradual union* $T_1 \oplus T_2$ is a gradual type that abstracts both T_1 and T_2. Seen in this light, a gradual union is a gradual type that is more precise than the prototypical, fully-unknown, gradual type ?, which abstracts any possible type. Starting from this insight, systematically applying the AGT methodology yields a novel point in the design space of both union types and gradual types.

Adding gradual unions to a simply-typed language relaxes the typing discipline, but does not allow full dynamic type checking. To achieve this, one needs to include both the unknown type ? and gradual unions. A second contribution of this paper is to uncover that combining these two gradual type constructors in the same language demands a stratified approach to AGT, in which the semantics of gradual types comes from the *composition* of two distinct abstract interpretations.

Contributions. This article makes the following specific contributions:

- **A novel design of union types** that combines benefits of both tagged and untagged unions, with added static flexibility backed by runtime checks. Compared to a standard gradually-typed language with only the totally-unknown

type ?, the resulting design is stricter, allowing more blatantly wrong programs to be statically rejected.
- **A first example of a stratified approach to AGT.** To derive the static semantics of a gradual language, AGT requires a Galois connection between gradual types and sets of static types, which then guides the lifting of functions and predicates on static types to their gradual counterparts [14]. We observe that applying AGT directly to introduce both the unknown type and gradual unions breaks optimality of the abstraction, thereby weakening the meaning of type information, both statically and dynamically. To address this, we develop a *stratified* approach that allows us to recover optimality. More specifically, we first apply AGT to support only the unknown type, and lift this Galois connection and derived liftings to their powerset counterpart. We then apply AGT once more with another Galois connection to introduce support for gradual unions, which allows us to define liftings based on the previously-defined powerset liftings. We prove that the composed abstraction is optimal. We conjecture that this technique might prove helpful in integrating other gradualization efforts.
- **The formalization and meta-theory of the proposed language**, including type safety and the gradual guarantees of Siek et al. [30]; these results follow directly, by construction, from relying on the AGT methodology.
- **A compilation scheme to an internal language with threesomes**, a space-efficient representation for casts [28]. We prove the correctness of the compilation with respect to the reference semantics derived by AGT using logical relations; this is the first case of formally relating the reference dynamic semantics obtained by AGT with a cast insertion translation.

Structure. Section 2 briefly reviews tagged and untagged unions, highlighting their pros and cons, and then informally introduces gradual unions, comparing them with standard gradual types and with the other kinds of unions, including those supported by several recent languages such as Flow and TypeScript, among others. Section 3 describes the static semantics of GTFL$^{\oplus}$, a language with both gradual unions and the unknown type, using the Abstracting Gradual Typing methodology. Section 4 describes the runtime semantics of the language by translation to a threesome cast calculus, and gives the formal properties of the language. Section 5 discusses related work and Sect. 6 concludes.

Complete definitions, as well as the proofs of all the results stated in the paper, can be found in the companion technical report [33]. A prototype implementation is available online, showing interactive typing and reduction derivations for arbitrary source programs: http://pleiad.cl/gradual-unions/.

2 Background and Motivation

We first briefly review standard tagged and untagged union types [23], highlighting the tradeoffs associated with each approach, and then introduce gradual union types as a novel point in the design space. We compare gradual unions to other approaches to union types, including practical languages with unions supported by runtime type tests. Finally, we compare gradual unions with the standard gradual types introduced by Siek and Taha [29].

2.1 Tagged Unions

Tagged unions, also called *disjoint* union types, denote values of possibly different types. The "disjointness" of the union comes from the fact that elements must be explicitly *tagged* so that it is clear to which type an element belongs. Tagging allows type-safe disambiguation through a case analysis construct.

The simplest form of tagged unions are binary *sum types*, noted $T_1 + T_2$, with injection forms inl and inr, and a disambiguation case expression. For instance, $\mathsf{inl}\ 10 :: \mathsf{Int} + \mathsf{Bool}$ injects the integer 10 into the sum type $\mathsf{Int} + \mathsf{Bool}$. The tag inl denotes the *left* part of the sum. Similarly, $\mathsf{inr}\ \mathsf{true} :: \mathsf{Int} + \mathsf{Bool}$ injects true to the right of the sum. Note that the ascription :: is necessary to maintain a simple syntax-directed type system; different techniques can be used to alleviate notation for programmers [23].

Given a value of type $\mathsf{Int} + \mathsf{Bool}$, one cannot use it directly. For instance, $\lambda x : \mathsf{Int} + \mathsf{Bool}.x + 1$ is not well typed. To use a tagged value, one must first disambiguate through an explicit case analysis, considering each tag explicitly, *e.g.* $\lambda x : \mathsf{Int} + \mathsf{Bool}.\mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\ x \Rightarrow x + 1 \mid \mathsf{inr}\ x \Rightarrow \mathsf{if}\ x\ \mathsf{then}\ 1\ \mathsf{else}\ 0$.

Note that $\mathsf{Int} + \mathsf{Bool}$ is different from $\mathsf{Bool} + \mathsf{Int}$ because the injection tag is relative to the *position* in the sum type. Sums can be generalized to *variants*, which are n-ary sums with custom labels instead of the positional inl and inr tags. In the case of variants, a type-case construct similar to case forces programmers to consider all possible alternatives, thereby statically ensuring the absence of runtime type errors.

To deal with with values of statically-unknown types, several proposals add a type $\mathsf{Dynamic}$ whose values are pairs of a plain value and a type tag [1,16]. The type $\mathsf{Dynamic}$ is therefore akin to an infinite tagged union, where tags are types. Disambiguation through case analysis therefore requires a default branch to handle unconsidered alternatives generally.

This general approach also explains how several languages support union types without needing any explicit tagging operation. For instance, in safe dynamic languages, all values are readily tagged with their *class* (either in the sense of Harper [15], *e.g.* Int, Bool, $\mathsf{Function}$, or, for class-based object-oriented languages, their actual class). This allows disambiguation of unions through runtime type testing (either via a type-case analysis or casts that can fail). This approach is exploited in several retrofitted type systems such as TypeScript [8], Flow [11] and Typed Racket [31,32]. Explicit disambiguation of unions can also be supported through pattern matching, as in ℂDuce [5] and Dotty [25].

2.2 Untagged Unions

An untagged union, noted $T_1 \vee T_2$, denotes the union of the values of type T_1 and of type T_2, without any tagging mechanism to support disambiguation. In this set-theoretic interpretation [4,12,22], $\mathsf{Int} \vee \mathsf{Int}$ is the same type as Int; and a value of type T_1 is a value of type $T_1 \vee T_2$, without any injection construct.

Untagged unions can be used to allow the branches of conditionals to have unrelated types: for instance, the function $\lambda x : \mathsf{Bool}.\mathsf{if}\ x\ \mathsf{then}\ 1\ \mathsf{else}\ \mathsf{false}$ can be considered well-typed at $\mathsf{Bool} \rightarrow \mathsf{Int} \vee \mathsf{Bool}$. This is the approach followed by the ℂDuce programming language [5], for instance.

Automatically introducing imprecision through untagged unions can however lead to unwanted programs being accepted. An alternative approach is for the typing rule for conditional expressions to require both branches to be of the same type, and to expect the programmer to use an explicit type *ascription* to specify that imprecision is desired; *e.g.* $\lambda x : \mathsf{Bool}.\mathsf{if}\ x\ \mathsf{then}\ (1 :: \mathsf{Int} \vee \mathsf{Bool})\ \mathsf{else}\ \mathsf{false}$. Note that the ascription does *not* imply any runtime tagging; it is a purely static artifact. Also, because of the set-theoretic interpretation of types, it is sufficient to ascribe imprecision in one of the two branches.

Untagged unions have no projection construct either; the only safe operations on a value of type $T_1 \vee T_2$ are those that are supported by *both* T_1 and T_2. Note that this makes untagged union restrictive to *use*; for instance, nothing useful can be done with a value of type $\mathsf{Int} \vee \mathsf{Bool}$. For instance, $\lambda x : \mathsf{Int} \vee \mathsf{Bool}.x + 1$ is not well-typed, because x could be a boolean value; and there is no disambiguation expression like case to handle each alternative separately.

This does not mean that untagged unions are useless; for instance, if the language has records, then it is safe to access fields that are common to both types. As noted by Pierce [23], untagged unions have traditionally been much more frequent in program analysis than in programming languages, where they were mostly used in type systems for semi-structured data [6,18], before being generalized in ℂDuce. Finally, note that the C language supports *unsafe* untagged unions, allowing programmers to use operations that are supported by *either* T_1 or T_2, at their own risk!

2.3 Gradual Unions

Tagged and untagged unions are the only safe approaches to *statically* deal with imprecision: either explicitly tag the imprecision so as to be able to safely discriminate later on, or assume the loss of precision and restrict what can be done with imprecisely-typed values. Tagged unions have the benefit of allowing programmers to fully use values, but only after explicit case-based disambiguations. Untagged unions have the benefit of requiring neither explicit injection nor projection, but only allow restricted usage of values.

If we are willing to accept some form of dynamic checking errors, however, we can combine the benefits of both tagged and untagged unions by viewing a union type as a kind of *gradual* type: $T_1 \oplus T_2$ is a gradual type that represents both T_1 and T_2. A gradual union supports the same kind of optimistic static checking that standard gradual typing provides.

For instance, $f \triangleq \lambda x : \mathsf{Int} \oplus \mathsf{Bool}.x + 1$ is (optimistically) well typed, because x might possibly be an Int, without any explicit projection or case analysis. The expressions $f\ 1$ and $f\ \mathsf{true}$ are also well-typed because injection to a gradual union is implicit. The expression $f\ 1$ evaluates to 2, as expected. But because x might in fact be a Bool, a runtime check is implicitly introduced before applying the $+$ operator: hence the expression $f\ \mathsf{true}$ produces a runtime cast error.

As a gradual type, a gradual union allows clearly incorrect programs to be rejected statically. For instance, changing the body of f to $x\ 1$ is statically rejected, because x cannot possibly be a function. Similarly, f "hola" is statically rejected, because f only tolerates integer or boolean arguments.

Note that compared to untagged unions, the use of a value with a gradual union type $T_1 \oplus T_2$ is accepted if the operations make sense for *either* T_1 *or* T_2 (and not both). This is just like untagged unions in C, but backed by runtime checks to ensure type safety. Injecting values into a gradual union type can be done implicitly as when applying g in the example above, or using an ascription, *e.g.* $g \triangleq \lambda x : \mathsf{Bool}.\mathsf{if}\ x\ \mathsf{then}\ (1{::}\mathsf{Int} \oplus \mathsf{Bool})\ \mathsf{else}\ (\mathsf{false}{::}\mathsf{Int} \oplus \mathsf{Bool})$ has type $\mathsf{Bool} \rightarrow \mathsf{Int} \oplus \mathsf{Bool}$.[1]

2.4 Comparing Unions

We summarize the characteristics of each form of union types as follows:

	injection	projection	use
Tagged unions			
sums	explicit	explicit	full
type tests/casts	implicit	explicit	full
Untagged unions	none	none	restricted
Gradual unions	implicit	implicit	full

To illustrate the convenience of gradual unions compared to alternative approaches, consider the following simple program:

```
let x: Bool ⊕ Int ⊕ String = 10
(λx: Int ⊕ Bool. x+1) x
```

The program introduces a variable `x` that can be one of three types, and initializes it to the number 10. It then passes it as argument to a lambda that expects either an Int or a Bool, and adds 1 to it. This program is well-typed, and returns 11. If `x` is initialized with a string, the program fails at runtime before the application of the function; if it is initialized with a boolean, the runtime error occurs before the addition.

This example would not be well-typed with untagged unions. This is because the intersection of Int and Bool and String is empty. If all three types have a common method, say `toString`, then the body of the lambda can only safely invoke `x.toString()`.

Turning to tagged unions, using standard sum types, the equivalent program would be fairly cumbersome to write because all injections and projections have to be manually introduced by the programmer, and deal with exact positions:

[1] Similarly to untagged unions, one could design a language whose conditional expression implicitly introduces imprecision, without the need for any ascription (Sect. 2.2); we do not further consider this possible design and use ascriptions explicitly.

```
let x: Bool + (Int + String) = inr inl 10
let x2: Int + Bool = case x of
  | inl y => inr y
  | inr y => case y of
               | inl z => inl z
               | inr z => throw new Error("not␣an␣Int␣or␣Bool")
(λx: Int + Bool. case x of
                   | inl y => y + 1
                   | inr y => throw new Error("not␣an␣Int")) x2
```

Note the need for an explicit intermediate step (x2) to safely go from the ternary union to the binary union.

The same program can also be written using implicitly-tagged unions with type-test disambiguation. As expected, the code is more lightweight than with sums thanks to implicit injection. For instance, in Flow:

```
1 const x: boolean | number | string = 10
2 const foo = (x: number | boolean ): number => {
3     if(typeof x =="number") return x + 1
4     else throw new Error("not␣a␣number")
5 }
6 if(typeof x =="boolean"|| typeof x =="number") foo(x)
7 else throw new Error("not␣a␣boolean␣or␣number")
```

Note that projections must be realized manually via typeof (lines $3, 4, 6$ and 7).

Evolving precision. The advantage of gradual unions does not only lie in the simplicity and compactness of the program definition. It also lies in its robustness in the face of precision-related changes. For instance, suppose that as the software matures, the programmer is now convinced that x will always be initialized with a number and that the function can simply only accept numbers. With sums, the program is so fragile that it would need to be modified at every injection and projection point to account for this change in precision. The Flow version would still run as is, but would feature a lot of dead code. Further decreasing precision would require adding checks at various projection points. With gradual unions, *it is enough to adjust the type annotations*—the rest of the program is unchanged!

The fact that the static-to-dynamic spectrum is navigated solely through the precision of type annotations, without requiring further modification of the program, is a key asset of gradual typing in general. The gradual guarantee of Siek et al. [30] further characterizes the relation between the static and dynamic semantics of programs that only differ in the precision of their type annotations, and will be discussed further when addressing the meta-theory of GTFL$^{\oplus}$.

Higher-order types. Finally, a major limitation of projections from unions using explicit type tests is that they do not support higher-order types. For instance, because one cannot decide whether an arbitrary function (*e.g.* of tag/class Function) always behave as a function of a particular type, programmers have to manually wrap functions with pre-post type checks.

Consider, in Flow or TypeScript, two functions of the following types:

```
f: (number | boolean) => (number | boolean)
g: ((number | string) => (boolean | string)) => string
```

To safely support the application `g(f)`, one needs to explicitly wrap `f` as follows:

```
const wrapper = (x: number | string): string | boolean =>{
    if (typeof x ==="number") {
        const result = f(x)
        if (typeof result ==="boolean") return result
        else throw new Error("not␣a␣boolean")
    } else throw new Error("not␣a␣number")
}
```

and then pass the wrapped function as argument: `g(wrapper)`.

Conversely, with gradual unions, one can simply write:

```
let f: (Int ⊕ Bool) -> (Int ⊕ Bool) = ...
let g: ((Int ⊕ String) -> (Bool ⊕ String)) -> String = ...
g(f)
```

for the exact same behavior; all the necessary checks and wrappers are handled under the hood.

2.5 Gradual Unions vs. Standard Gradual Types

Gradual typing has always been formulated in terms of an unknown type, frequently written ?, which denotes any possible type [29]. Furthermore, when structural types are supported, gradual types can be more precise than the fully-unknown type: for instance $\mathsf{Int} \rightarrow ?$ denotes all function types from Int to possibly any type.

To illustrate the key difference between gradual unions and standard gradual types, consider a function h that always returns either an Int or a Bool. Starting from a simple typing discipline, with standard gradual types, the most precise type one can give to h is $\mathsf{Bool} \rightarrow ?$. However, this type allows for too much flexibility that was not intended: because h true has type ?, it can subsequently be used in any context, even $(h\ \mathsf{true})\ 1$, which is clearly always going to fail since h never returns a function. The problem comes form the fact that the gradual type used for the codomain of h, ?, is too imprecise—yet it is the only available type to denote both Int and Bool. Hence, the programmer cannot express a more restricted form of flexibility. Gradual unions address this need. For instance, recalling function g from Sect. 2.3 above, $(g\ \mathsf{true})\ 1$ is statically rejected.

Gradual union types are a novel way to relax a static typing discipline in a restricted manner. While the discussion above insists on the advantages of this restricted flexibility, it necessarily presents drawbacks as well. In particular, a language with only gradual unions cannot fully embed the untyped lambda calculus. In order to get the best of both worlds, one needs a language that supports both the fully-unknown gradual type ? in addition to gradual unions. This way, programmers can navigate the full static-to-dynamic spectrum, with more interesting intermediate points offered by gradual unions. In the rest of this paper, we design and formalize such a language.

3 GTFL$^{\oplus}$: Static Semantics

We now formalize GTFL$^{\oplus}$, a gradual language with both gradual unions and the unknown type. As hinted previously, we follow the Abstracting Gradual Typing (AGT) methodology [14] to derive the static semantics of GTFL$^{\oplus}$:

1. We start from a language with a fully static typing discipline, STFL.
2. We define the syntax of gradual types, and give them meaning via a concretization function and its corresponding most precise abstraction, forming a Galois connection. Crucially, in this step we realize that the two forms of gradual types must be handled in a stratified manner in order to ensure optimality.
3. We derive the static semantics of the gradual language by lifting type predicates and type functions used in the static type system through the Galois connection.

The most novel part of our development are steps 2 and 3, which showcase how to compose Galois connections related to different gradual type constructors. We address the dynamic semantics of GTFL$^{\oplus}$ to Sect. 4.

3.1 The Static Language: STFL

Our starting point is a simply-typed functional language with booleans and integers, called STFL [14]. A term can be a lambda abstraction, a boolean, a number, a variable, an application, an addition, a conditional, or an ascription. The typing rules are standard—omitted for space reasons, available in [33]—save for the fact that their presentation follows some simple conventions, helpful for gradualization [14]: the type of each sub-expression is kept opaque, the type relations ($=$) are made explicit as side conditions, and partial type functions (dom, cod, $equate$) are used explicitly instead of relying on matching metavariables. The dynamic semantics and type safety of STFL are completely standard.

3.2 Defining Gradual Types Separately

GTFL$^{\oplus}$ supports both the unknown type ? and gradual unions with $\oplus$. In this section, we look at both gradual type constructors separately in order to precisely define their meaning. Recall that following AGT, the *meaning* of a gradual type is the set of static types that it possibly represents, defined by a concretization function γ. Given such a meaning, the AGT methodology directs us to define a sound and optimal abstraction function α, hence forming a Galois connection [14], which is then used to lift static type predicates and type functions to operate on gradual types. We study both gradual type constructors in turn.

Let us first recall from [14] the Galois connection for gradual types made up with the (nullary constructor) ?, here denoted GTYPE.

$$\begin{array}{l} G \in \text{GTYPE} \\ G ::= \boxed{?} \mid \mathsf{Bool} \mid \mathsf{Int} \mid G \to G \end{array}$$

The meaning of these gradual types is standard, and defined through concretization by Garcia et al. [14] as follows:

Definition 1 (GType Concretization). $\gamma_? : \text{GTYPE} \to \mathcal{P}(\text{TYPE})$

$$\gamma_?(\mathsf{Int}) = \{\, \mathsf{Int} \,\} \qquad \gamma_?(\mathsf{Bool}) = \{\, \mathsf{Bool} \,\} \qquad \boxed{\gamma_?(?) = \text{TYPE}}$$

$$\gamma_?(G_1 \to G_2) = \{\, T_1 \to T_2 \mid T_1 \in \gamma_?(G_1) \land T_2 \in \gamma_?(G_2) \,\}$$

Note in particular that the meaning of the fully unknown type ? is the set of all types. Similarly, the meaning of the imprecise type $\mathsf{Int} \to ?$ is the set of all function types $\mathsf{Int} \to T$, for any $T \in \text{TYPE}$.

Concretization naturally induces the notion of *precision* among gradual types, which reflects the amount of static information of a gradual type [14].

Definition 2 (GType Precision). *G_1 is less imprecise than G_2, notation $G_1 \sqsubseteq G_2$, if and only if $\gamma_?(G_1) \subseteq \gamma_?(G_2)$.*

The following abstraction $\alpha_?$ naturally forms a Galois connection with $\gamma_?$:[2]

Definition 3 (GType Abstraction). $\alpha_? : \mathcal{P}(\text{TYPE}) \rightharpoonup \text{GTYPE}$

$$\alpha_?(\{\, T \,\}) = T \qquad \alpha_?(\widehat{T_1 \to T_2}) = \alpha_?(\widehat{T_1}) \to \alpha_?(\widehat{T_2}) \qquad \alpha_?(\emptyset) = \textit{undefined}$$

$$\alpha_?(\widehat{T}) = ? \ \textit{otherwise}$$

The abstraction retains as much precision as possible (*e.g.* singletons, function type constructor) and degrades to the unknown type otherwise. Note that $\alpha_?$ is undefined for the empty set [14].

Importantly, $\gamma_?$ and $\alpha_?$ form a Galois connection:

Proposition 1 ($\alpha_?$ is Sound and Optimal). *If $\widehat{T}$ is not empty, then*

$$(a)\ \widehat{T} \subseteq \gamma_?(\alpha_?(\widehat{T})). \qquad (b)\ \widehat{T} \subseteq \gamma_?(G) \Rightarrow \alpha_?(\widehat{T}) \sqsubseteq G.$$

Soundness (a) means that α always produces a gradual type whose concretization overapproximates the information in the original set. Optimality (b) means that α is the best sound approximation function: it produces the most precise gradual type that abstracts a given set.

[2] We use the hat notation $\widehat{X}$ to refer to a set of elements X (*e.g.* $\widehat{T}$ is a set of static types, and $\widehat{\widehat{T}}$ is a set of sets of static types).

Let us now consider a Galois connection for the novel gradual type constructor introduced in this work, gradual unions. We use STYPE to denote gradual types made up only of gradual unions, *i.e.* without ?.

$$\begin{array}{l} S \in \textsc{SType} \\ S ::= S \oplus S \;\mid\; \mathsf{Bool} \;\mid\; \mathsf{Int} \;\mid\; S \rightarrow S \end{array}$$

Note that this syntax admits n-ary unions recursively through $S \oplus S$. We consider gradual unions to be *syntactically* equivalent up to associativity of $\oplus$, *i.e.* $S_1 \oplus (S_2 \oplus S_3) \equiv (S_1 \oplus S_2) \oplus S_3$. Gradual unions represent the *finite* set of types represented (recursively) by each constituent:

Definition 4 (SType Concretization). $\gamma_\oplus : \textsc{SType} \rightarrow \mathcal{P}_{\mathit{fin}}(\textsc{Type})$

$$\gamma_\oplus(\mathsf{Int}) = \{\,\mathsf{Int}\,\} \qquad \gamma_\oplus(\mathsf{Bool}) = \{\,\mathsf{Bool}\,\} \qquad \gamma_\oplus(S_1 \oplus S_2) = \gamma_\oplus(S_1) \cup \gamma_\oplus(S_2)$$

$$\gamma_\oplus(S_1 \rightarrow S_2) = \{\, T_1 \rightarrow T_2 \mid T_1 \in \gamma_\oplus(S_1) \wedge T_2 \in \gamma_\oplus(S_2)\,\}$$

For instance $\gamma_\oplus(\mathsf{Int} \oplus \mathsf{Bool} \oplus (\mathsf{Int} \rightarrow \mathsf{Bool})) = \{\,\mathsf{Int}, \mathsf{Bool}, \mathsf{Int} \rightarrow \mathsf{Bool}\,\}$. Because gradual unions only produce finite sets of static types, the corresponding abstraction also only needs to be defined on finite sets, and therefore can produce the gradual union with all the elements, noted $\oplus T$:

Definition 5 (SType Abstraction). $\alpha_\oplus : \mathcal{P}_{\mathit{fin}}(\textsc{Type}) \rightharpoonup \textsc{SType}$

$$\alpha_\oplus(T) = \oplus T \qquad \mathit{if}\, T \neq \emptyset$$

Here again, $\langle \gamma_\oplus, \alpha_\oplus \rangle$ forms a Galois connection.

3.3 Combining Gradual Types: Take 1

Now that we have defined the meaning of gradual types formed with the unknown type ?, as well as the meaning of gradual types formed with gradual unions $\oplus$, we turn to defining the meaning of gradual types in GTFL$^\oplus$, which combine both constructors, denoted UTYPE:

$$\begin{array}{ll} U \in \textsc{UType} & \\ U ::= \; ? \;\mid\; U \oplus U \;\mid\; \mathsf{Bool} \;\mid\; \mathsf{Int} \;\mid\; U \rightarrow U & \text{(gradual types)} \end{array}$$

A first seemingly natural approach is to define the concretization function for UTYPE by combining both concretization functions for GTYPE and STYPE:

Definition 6 (UType Concretization, Take 1). $\gamma : \textsc{UType} \rightarrow \mathcal{P}(\textsc{Type})$

$$\gamma(\mathsf{Int}) = \{\,\mathsf{Int}\,\} \qquad \gamma(\mathsf{Bool}) = \{\,\mathsf{Bool}\,\} \qquad \gamma(U_1 \oplus U_2) = \gamma(U_1) \cup \gamma(U_2)$$

$$\gamma(?) = \textsc{Type} \qquad \gamma(U_1 \rightarrow U_2) = \{\, T_1 \rightarrow T_2 \mid T_1 \in \gamma(U_1) \wedge T_2 \in \gamma(U_2)\,\}$$

While this definition seems sensible, it does not accommodate a corresponding optimal abstraction. Indeed, the abstraction functions for GType and SType conflict with each other: how should we abstract a set of different types?

For a set of base types, say $\{\,\mathsf{Int}, \mathsf{Bool}\,\}$, we can either abstract to $?$ or to $\mathsf{Int} \oplus \mathsf{Bool}$; the latter being optimal, while the former is not. In fact, to preserve optimality, we ought to defer to the unknown type only for heterogeneous *infinite* sets. Even if we would adjust the definition of the combined abstraction to make such a distinction, it would not be optimal. To see why, consider the type $(? \to \mathsf{Int}) \oplus (\mathsf{Int} \to ?)$, whose concretization is:

$$\begin{aligned}\gamma((? \to \mathsf{Int}) \oplus (\mathsf{Int} \to ?)) &= \gamma(? \to \mathsf{Int}) \cup \gamma(\mathsf{Int} \to ?)\\ &= \{\, T \to \mathsf{Int} \mid T \in \textsc{Type} \,\} \cup \{\, \mathsf{Int} \to T \mid T \in \textsc{Type} \,\}\\ &= \{\, \mathsf{Int} \to \mathsf{Int}, \mathsf{Bool} \to \mathsf{Int}, \mathsf{Int} \to \mathsf{Bool}, \ldots \,\}\\ &\triangleq T\end{aligned}$$

By taking the union of both sets, we "forget" a specificity of the original gradual type—namely that it only represents functions that necessarily have Int either as domain or as codomain. For instance, $\mathsf{Bool} \to \mathsf{Bool}$ is *not* present in the resulting set T. However, the abstraction function that we obtain by directly combining the two abstractions we have seen above is unable to recover an optimal gradual type: because T is infinite and only contains arrow types, the best the abstraction can do is to keep the arrow constructor, and then separately abstracts the domain and codomain types (just like $\alpha_?$). As a result:

$$\alpha(T) = ? \to ?$$

While this abstraction is sound, it is not optimal: there exists a more precise gradual type that represents T, the type $(? \to \mathsf{Int}) \oplus (\mathsf{Int} \to ?)$ we started with.

Losing optimality directly affects the programmer's experience. For instance, in the type system, this means that the gain of precision that gradual unions are supposed to provide (recall Sect. 2.5) is lost; similarly, type annotations would not be strictly enforced at runtime.

3.4 Combining Gradual Types: Take Two

In order to define a proper Galois connection to give meaning to the gradual types of GTFL$^{\oplus}$, we introduce a *stratified*, sketched in Fig. 1:

- **Step 1.** We start from the Galois connection between GType and $\mathcal{P}(\textsc{Type})$, named *classic interpretation* hereafter, which interprets the unknown type. We already described this Galois connection in Sect. 3.2.
- **Step 2.** We lift this connection to operate on finite sets of gradual types, with the standard collecting semantics, forming a new Galois connection between $\mathcal{P}_{\mathit{fin}}(\textsc{GType})$ and $\mathcal{P}_{\mathit{fin}}(\mathcal{P}(\textsc{Type}))$, named the *classic set interpretation*.

$$\begin{tikzcd}
 & & & \text{GType} \arrow[r, bend left, "\gamma_?"] \arrow[r, phantom, "\textcircled{1}"] \arrow[dl, "\mathcal{P}_{\mathit{fin}}(\cdot)"'] \arrow[dl, phantom, "\textcircled{2}"] & \mathcal{P}(\text{Type}) \arrow[l, bend left, "\alpha_?"] \\
\text{UType} \arrow[r, bend left, "\gamma_\oplus"] \arrow[r, phantom, "\textcircled{3}"] \arrow[rr, dashed, bend left=40, "\gamma", "\textcircled{4}"'] & \mathcal{P}_{\mathit{fin}}(\text{GType}) \arrow[l, bend left, "\alpha_\oplus"] \arrow[r, bend left, "\widehat{\gamma_?}"] & \mathcal{P}_{\mathit{fin}}(\mathcal{P}(\text{Type})) \arrow[l, bend left, "\widehat{\alpha_?}"] \arrow[ll, dashed, bend left=40, "\alpha"] & &
\end{tikzcd}$$

Fig. 1. Stratified interpretation of UType.

- **Step 3.** We introduce a Galois connection between UType and $\mathcal{P}_{\mathit{fin}}$(GType), named *union interpretation*, which adds support for gradual unions among gradual types that include the unknown type.
- **Step 4.** We combine the classic set interpretation and the union interpretation. This combination gives a *stratified interpretation* of GTFL$^\oplus$ gradual types, UType, in terms of finite sets of (possibly-infinite) sets of static types.

As we show, the stratified interpretation is itself a proper Galois connection, and we can subsequently use it to lift the static (and dynamic) semantics of STFL in order to define the semantics of GTFL$^\oplus$.

Step 2. Lifting the Classic Interpretation. Recall that $\langle\gamma_?, \alpha_?\rangle$ from Definitions 1 and 3 form a Galois connection between GType and $\mathcal{P}$(Type) [14]. Our first step is to lift this connection to operate on sets of gradual types with the unknown type, *i.e.* to relate $\mathcal{P}_{\mathit{fin}}$(GType) and $\mathcal{P}_{\mathit{fin}}$($\mathcal{P}$(Type)). The *powerset lifting* of $\gamma_?$, denoted $\gamma_?$, is simply the piecewise application of $\gamma_?$:

Definition 7 ($\mathcal{P}_{\mathit{fin}}$(GType) Concretization). $\gamma_? : \mathcal{P}_{\mathit{fin}}(\text{GType}) \rightarrow \mathcal{P}_{\mathit{fin}}(\mathcal{P}$ $(\text{Type}))$

$$\gamma_?(G) = \{\, \gamma_?(G) \mid G \in G \,\}$$

Similarly, the powerset lifting of the abstraction function $\alpha_?$, denoted $\alpha_?$, is the union of the piecewise application of $\alpha_?$:

Definition 8 ($\mathcal{P}_{\mathit{fin}}$(GType) Abstraction). $\alpha_? : \mathcal{P}_{\mathit{fin}}(\mathcal{P}(\text{Type})) \rightharpoonup \mathcal{P}_{\mathit{fin}}$ (GType)

$$\alpha_?(\emptyset) = \mathit{undefined} \qquad\qquad \alpha_?(T) = \bigcup_{T \in T} \alpha_?(T)$$

As expected, $\langle\gamma_?, \alpha_?\rangle$ is a proper Galois connection.

Proposition 2 ($\alpha_?$ is Sound and Optimal). *If T is not empty, then*

$$(a)\ T \subseteq \gamma_?(\alpha_?(T)). \qquad\qquad (b)\ T \subseteq \gamma_?(G) \Rightarrow \alpha_?(T) \sqsubseteq G.$$

Step 3. Introducing the union interpretation. We define a Galois connection between UTYPE and $\mathcal{P}_{fin}$(GTYPE) by naturally extending the definition of the connection between STYPE and $\mathcal{P}_{fin}$(TYPE) from Definitions 4 and 5, so that it now operates over types in UTYPE instead of only types in STYPE.

Definition 9 (UType Concretization). $\gamma_\oplus : \text{UTYPE} \to \mathcal{P}_{fin}(\text{GTYPE})$

$$\gamma_\oplus(\mathsf{Int}) = \{\,\mathsf{Int}\,\} \qquad \gamma_\oplus(\mathsf{Bool}) = \{\,\mathsf{Bool}\,\} \qquad \gamma_\oplus(?) = \{\,?\,\}$$

$$\gamma_\oplus(U_1 \to U_2) = \{\, T_1 \to T_2 \mid T_1 \in \gamma_\oplus(U_1) \wedge T_2 \in \gamma_\oplus(U_2)\,\}$$

$$\gamma_\oplus(U_1 \oplus U_2) = \gamma_\oplus(U_1) \cup \gamma_\oplus(U_2)$$

Compared to Definition 4, the only additional case to consider is that the unknown type ? can now occur: it is handled like other nullary type constructors, by concretizing to a singleton.

The abstraction is direct from Definition 5.

Definition 10 (UType Abstraction). $\alpha_\oplus : \mathcal{P}_{fin}(\text{GTYPE}) \rightharpoonup \text{UTYPE}$

$$\alpha_\oplus(G) = \oplus G \qquad \mathit{if}\ G \neq \emptyset$$

where $\oplus G$ denotes the gradual union of all the types in the set T.

Again, $\langle \gamma_\oplus, \alpha_\oplus \rangle$ is a Galois connection.

Proposition 3 ($\boldsymbol{\alpha_\oplus}$ is Sound and Optimal). *If G is not empty, then*

$$(a)\ G \subseteq \gamma_\oplus(\alpha_\oplus(G)). \qquad (b)\ G \subseteq \gamma_\oplus(U) \Rightarrow \alpha_\oplus(G) \sqsubseteq U.$$

Step 4. Composing the Connections. We can now compose the two Galois connections in order to define a stratified interpretation for UTYPE in terms of sets of sets of static types.

Definition 11 (Concretization). $\gamma : \text{UTYPE} \to \mathcal{P}_{fin}(\mathcal{P}(\text{TYPE}))$, $\gamma = \gamma_? \circ \gamma_\oplus$

Definition 12 (Abstraction). $\alpha : \mathcal{P}_{fin}(\mathcal{P}(\text{TYPE})) \rightharpoonup \text{UTYPE}$, $\alpha = \alpha_\oplus \circ \alpha_?$

Because the composition of two Galois connection is a Galois connection, the stratified interpretation $\langle \gamma, \alpha \rangle$ is a Galois connection.

Proposition 4 ($\boldsymbol{\alpha}$ is Sound and Optimal). *If T is not empty, then*

$$(a)\ T \subseteq \gamma(\alpha(T)). \qquad (b)\ T \subseteq \gamma(U) \Rightarrow \alpha(T) \sqsubseteq U.$$

The notion of precision for gradual types used above is similarly induced by concretization, *i.e.* $U_1 \sqsubseteq U_2 \iff \gamma(U_1) \subseteq \gamma(U_2)$. Note that these definitions use containment over sets of sets, defined as $T_1 \subseteq T_2 \iff \forall T_1 \in T_1, \exists T_2 \in T_2, T_1 \subseteq T_2$. Precision can equivalently be defined in terms of the lifted classic abstraction, *i.e.* $U_1 \sqsubseteq U_2 \iff \gamma_\oplus(U_1) \sqsubseteq_? \gamma_\oplus(U_2)$, where $G_1 \sqsubseteq_? G_2 \iff \gamma_?(G_1) \subseteq \gamma_?(G_2)$.

$$(U\mathrm{app})\,\frac{\Gamma \vdash \widetilde{t}_1 : U_1 \quad \Gamma \vdash \widetilde{t}_2 : U_2 \quad U_2 \sim \widetilde{dom}(U_1)}{\Gamma \vdash \widetilde{t}_1\ \widetilde{t}_2 : \widetilde{cod}(U_1)} \qquad (U+)\,\frac{\Gamma \vdash \widetilde{t}_1 : U_1 \quad \Gamma \vdash \widetilde{t}_2 : U_2 \quad U_1 \sim \mathsf{Int} \quad U_2 \sim \mathsf{Int}}{\Gamma \vdash \widetilde{t}_1 + \widetilde{t}_2 : \mathsf{Int}}$$

$$(U{::})\,\frac{\Gamma \vdash \widetilde{t} : U \quad U \sim U_1}{\Gamma \vdash (\widetilde{t} :: U_1) : U_1} \qquad (U\mathrm{if})\,\frac{\Gamma \vdash \widetilde{t}_1 : U_1 \quad \Gamma \vdash \widetilde{t}_2 : U_2 \quad U_1 \sim \mathsf{Bool} \quad \Gamma \vdash \widetilde{t}_3 : U_3}{\Gamma \vdash \mathsf{if}\ \widetilde{t}_1\ \mathsf{then}\ \widetilde{t}_2\ \mathsf{else}\ \widetilde{t}_3 : U_2 \sqcap U_3}$$

Fig. 2. GTFL$^{\oplus}$: selected typing rules

Illustration. Let us come back to the example of Sect. 3.3 that motivated the need for a stratified interpretation of UType.

$$\begin{aligned}\gamma((? \to \mathsf{Int}) \oplus (\mathsf{Int} \to ?)) &= \{\gamma_?(? \to \mathsf{Int}), \gamma_?(\mathsf{Int} \to ?)\}\\ &= \{\{\, T \to \mathsf{Int} \mid T \in \textsc{Type}\,\}, \{\, \mathsf{Int} \to T \mid T \in \textsc{Type}\,\}\}\end{aligned}$$

we can now recover exactly the same gradual type

$$\begin{aligned}&\alpha(\{\{\, T \to \mathsf{Int} \mid T \in \textsc{Type}\,\}, \{\, \mathsf{Int} \to T \mid T \in \textsc{Type}\,\}\})\\ &= \alpha_{\oplus}(\alpha_?(\{\, T \to \mathsf{Int} \mid T \in \textsc{Type}\,\}) \cup \alpha_?(\{\, \mathsf{Int} \to T \mid T \in \textsc{Type}\,\}))\\ &= \alpha_{\oplus}(? \to \mathsf{Int}, \mathsf{Int} \to ?) = \ (? \to \mathsf{Int}) \oplus (\mathsf{Int} \to ?)\end{aligned}$$

3.5 Static Semantics of GTFL$^{\oplus}$

The syntax of GTFL$^{\oplus}$ is the same as that of STFL, save for the introduction of gradual types U. Consequently, terms t are lifted to gradual terms $\widetilde{t} \in$ UTerm, *i.e.* terms with gradual type annotations.

The type system of GTFL$^{\oplus}$ is presented in Fig. 2. The typing rules present no surprise with respect to the gradual language with ? presented by Garcia et al. [14]. This is because the novelty of gradual unions is encapsulated in gradual type predicates and functions, such as $\sim$, $\sqcap$, $\widetilde{dom}$, etc.

The essential idea of using abstract interpretation to define a gradual language is that the Galois connection that defines gradual types specifies how to lift both type predicates and functions to obtain their consistent counterpart.[3] For instance, the consistent lifting of a predicate over static types is the *existential* lifting of the predicate through the Galois connection. In other words, for a given binary predicate $P \in \textsc{Type}^2$, its consistent lifting $\widetilde{P} \in \textsc{GType}^2$ is defined as: $\widetilde{P}(U_1, U_2) \iff \exists T_1 \in \gamma(U_1), \exists T_2 \in \gamma(U_2), P(T_1, T_2)$. Similarly for functions: a lifted function is the abstraction of the application of the static function to all the possible static types denoted by the involved gradual types. Formally, $\widetilde{f} = \alpha \circ f \circ \gamma$, where f is the pointwise application of f to all elements.

[3] The AI framework provides us with *definitions* for consistent predicates and functions; we will provide some equivalent algorithmic *characterizations*.

Lifting for Stratified Interpretation. We need to adapt these definitions from AGT to our stratified setting; indeed, our Galois connection relates gradual types with sets *of sets* of static types, rather than just sets of static types.

We can base our liftings of predicates and types on inclusion and pointwise application that are extended to sets of sets.

Definition 13 (Predicate Lifting). $\widetilde{P}(U_1, U_2) \iff \exists T_1 \in \gamma(U_1), T_2 \in \gamma(U_2), P(T_1, T_2)$ *where* $\in$ *is the existential lifting of* $\in$ *to powersets:* $T \in T \iff \exists T \in T, T \in T$

Equivalently: $\widetilde{P}(U_1, U_2) \iff \exists T_1 \in \gamma(U_1), \exists T_2 \in \gamma(U_2), \exists T_1 \in T_1, \exists T_2 \in T_2, P(T_1, T_2)$

The lifting of a predicate can also be defined in terms of each of the composed interpretations:

Proposition 5. $\widetilde{P}(U_1, U_2) \iff \exists G_1 \in \gamma_{\oplus}(U_1), \exists G_2 \in \gamma_{\oplus}(U_2), \widetilde{P}_?(G_1, G_2)$ *where* $\widetilde{P}_?$ *is the predicate* P *lifted with* $\gamma_?$.

The lifting of a type function f uses the pointwise application of f to all elements of each subset of a powerset, which we note f.

Definition 14 (Function Lifting). $\widetilde{f} = \alpha \circ f \circ \gamma$

Again, we can define the lifting using the separate abstractions: $\widetilde{f} = \alpha_{\oplus} \circ \widetilde{f}_? \circ \gamma_{\oplus}$

Example Liftings. Let us look at the lifting of a type predicate and a type function. We start with *consistency*, $\sim$, which corresponds to the lifting of type equality: two gradual types are consistent if some static types in their concretization are equal.

Definition 15 (Consistency). $U_1 \sim U_2$ *if and only if* $\exists T_1 \in \gamma(U_1), \exists T_1 \in T_1, \exists T_2 \in \gamma(U_2), \exists T_2 \in T_2, T_1 = T_2$.

This definition is equivalent to the following inductive definition:

Proposition 6.

$$\frac{U \sim U_1}{U \sim U_1 \oplus U_2} \qquad \frac{U \sim U_2}{U \sim U_1 \oplus U_2} \qquad \frac{U_1 \sim U}{U_1 \oplus U_2 \sim U} \qquad \frac{U_2 \sim U}{U_1 \oplus U_2 \sim U}$$

$$\frac{}{U \sim U} \qquad \frac{}{? \sim U} \qquad \frac{}{U \sim ?} \qquad \frac{U_{21} \sim U_{11} \quad U_{12} \sim U_{22}}{U_{11} \to U_{12} \sim U_{21} \to U_{22}}$$

Let us now consider the (precision) meet of gradual types, which corresponds to the lifting of the *equate* function used in the typing rule for conditionals. Its algorithmic definition is:

Definition 16 (Gradual Meet). *Let* $\sqcap : \text{UType} \rightharpoonup \text{UType}$ *be defined as:*

1. $U \sqcap U = U$
2. $? \sqcap U = U \sqcap ? = U$
3. $U \sqcap (U_1 \oplus U_2) = (U_1 \oplus U_2) \sqcap U = \begin{cases} U \sqcap U_1 & \mathit{if}\, U \sqcap U_2 \; \mathit{is\, undefined} \\ U \sqcap U_2 & \mathit{if}\, U \sqcap U_1 \, \mathit{is\, undefined} \\ (U \sqcap U_1) \oplus (U \sqcap U_2) & \mathit{otherwise} \end{cases}.$
4. $(U_{11} \rightarrow U_{12}) \sqcap (U_{21} \rightarrow U_{22}) = (U_{11} \sqcap U_{21}) \rightarrow (U_{12} \sqcap U_{22})$
5. $U_1 \sqcap U_2$ *is undefined otherwise.*

This algorithmic definition coincides with the lifting of equate:

Proposition 7. $\sqcap = \alpha \circ \mathit{equate} \circ \gamma$

4 GTFL$^{\oplus}$: Dynamic Semantics and Properties

Following the tradition [29], we now give the dynamic semantics of GTFL$^{\oplus}$ programs by a cast insertion translation to an internal language with explicit casts. We first describe the internal language GTFL$^{\oplus}_{\Rightarrow}$, which is adapted from the (blameless) threesome calculus of Siek et al. [28], and then present a cast insertion translation from GTFL$^{\oplus}$ to GTFL$^{\oplus}_{\Rightarrow}$.

Intermediate Language. GTFL$^{\oplus}_{\Rightarrow}$ is an adaptation of the original threesome calculus without blame [28]. A threesome $\langle T_2 \overset{T_3}{\Longleftarrow} T_1 \rangle$ is a cast composed of three types: the source type T_1, the target type T_2, and the middle type T_3. Initially, the middle type of a threesome is the greatest lower bound (in terms of precision), or meet, of the source and target types. The key benefit of threesomes is that two threesomes can be merged into a single threesome by taking the meet of their middle types, hence avoiding space issues [17].

The syntax of GTFL$^{\oplus}_{\Rightarrow}$ is a simple extension of that of GTFL$^{\oplus}$, with cast expressions $\langle U_2 \overset{U_3}{\Longleftarrow} U_1 \rangle t$ and casted values $\langle U_2 \overset{U_3}{\Longleftarrow} U_1 \rangle u$, where u denotes the simple values of GTFL$^{\oplus}$. For space reasons we only present a representative selection of typing rules in Fig. 3. In the typing rule for cast expressions (IT$\langle\rangle$), the consistency premises are required for a threesome to be well-formed (this is always the case by construction). The other typing rules are basically those of GTFL$^{\oplus}$, except that type consistency is replaced with type equality; this is because uses of consistency will be guarded by the insertion of casts. For instance, Fig. 3 shows rule (ITapp): while the GTFL$^{\oplus}$ typing rule (Uapp, Fig. 2) uses the premise $U_2 \sim \widetilde{dom}(U_1)$, the new rule requires the type of t_2 to exactly be $\widetilde{dom}(U_1)$.

Figure 4 presents the dynamic semantics of GTFL$^{\oplus}_{\Rightarrow}$, which are similar to [28]. Two threesomes that coincide on their source/target types are combined by meeting their middle types. If the meet is undefined then the term steps to **error**. Otherwise both casts are merged to a new cast where the middle type is now the meet between the middle types. Note that casts are introduced using the following metafunction, which avoids producing useless threesomes:

$$\langle\!\langle U_2 \overset{U_3}{\Longleftarrow} U_1 \rangle\!\rangle t = t \text{ if } U_1 = U_2 = U_3\,;\text{ and } \langle U_2 \overset{U_3}{\Longleftarrow} U_1 \rangle t \;\text{ otherwise}$$

$$(\mathrm{IT}\langle\rangle)\frac{\Gamma \vdash_{\!\!\!\!\!\!\!\!\sim} t : U_1 \quad U_1 \sim U_2 \quad U_1 \sim U_3 \quad U_3 \sim U_2}{\Gamma \vdash_{\!\!\!\!\!\!\!\!\sim} \langle U_2 \overset{U_3}{\Leftarrow} U_1\rangle t : U_2} \qquad (\mathrm{ITapp})\frac{\Gamma \vdash_{\!\!\!\!\!\!\!\!\sim} t_1 : U_1 \quad \Gamma \vdash_{\!\!\!\!\!\!\!\!\sim} t_2 : \widetilde{dom}(U_1)}{\Gamma \vdash_{\!\!\!\!\!\!\!\!\sim} t_1\ t_2 : \widetilde{cod}(U_1)}$$

Fig. 3. $\mathrm{GTFL}^{\oplus}_{\Rightarrow}$: selected typing rules

$$u ::= \mathsf{true} \mid \mathsf{false} \mid n \mid \lambda x.t \qquad v ::= u \mid \langle U \overset{U}{\Leftarrow} U\rangle u \quad \text{(values)}$$
$$f ::= \square + t \mid v + \square \mid \square\ t \mid v\ \square \mid \langle U \overset{U}{\Leftarrow} U\rangle\square \mid \mathsf{if}\ \square\ \mathsf{then}\ t\ \mathsf{else}\ t \quad \text{(frames)}$$

$$\boxed{t \longrightarrow t} \qquad \frac{n_3 = rval(v_1)\ [\![+]\!]\ rval(v_2)}{v_1 + v_2 \longrightarrow n_3} \qquad (\lambda x.t)\ v \longrightarrow [v/x]t$$

$$\mathsf{if}\ v\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \longrightarrow \begin{cases} t_2 \text{ if } rval(v) = \mathsf{true} \\ t_3 \text{ if } rval(v) = \mathsf{false} \end{cases}$$

$$\langle U_{21} \to U_{22} \overset{U_3}{\Leftarrow} U_{11} \to U_{12}\rangle u\ v \longrightarrow \langle\!\langle U_{22} \xleftarrow{icod(U_3)} U_{12}\rangle\!\rangle(u\ \langle\!\langle U_{11} \xleftarrow{idom(U_3)} U_{21}\rangle\!\rangle v)$$

$$\langle U_3 \overset{U_{32}}{\Leftarrow} U_2\rangle\langle U_2 \overset{U_{21}}{\Leftarrow} U_1\rangle v \longrightarrow \begin{cases} \langle\!\langle U_3 \xleftarrow{U_{32} \sqcap U_{21}} U_1\rangle\!\rangle v \\ \mathbf{error} \text{ if } U_{32} \sqcap U_{21} \text{ is undefined} \end{cases}$$

$$\boxed{t \longmapsto t} \qquad \frac{t_1 \longrightarrow t_2}{t_1 \longmapsto t_2} \qquad \frac{t_1 \longmapsto t_2}{f[t_1] \longmapsto f[t_2]} \qquad \frac{}{f[\mathbf{error}] \longmapsto \mathbf{error}}$$

Fig. 4. $\mathrm{GTFL}^{\oplus}_{\Rightarrow}$: dynamic semantics of $\mathrm{GTFL}^{\oplus}_{\Rightarrow}$

Cast Insertion. A $\mathrm{GTFL}^{\oplus}$ program is elaborated through a type-driven cast insertion translation. The key idea of the transformation is to insert casts in places where consistency is used to justify the typing derivation. For instance, if $\widetilde{t} : \mathsf{Int} \oplus \mathsf{Bool}$ is used where Int is required, the translation inserts a cast $\langle \mathsf{Int} \Leftarrow \mathsf{Int} \oplus \mathsf{Bool}\rangle t$, where t is the recursive translation of $\widetilde{t}$. This cast plays the role of the implicit projection from the gradual union type. Dually, when a term of type Int is used where a gradual union is expected, the translation adds a cast that performs the implicit injection to the gradual union, *e.g.* $\langle \mathsf{Int} \oplus \mathsf{Bool} \Leftarrow \mathsf{Int}\rangle 10$. Note that a value with a cast that loses precision is like a tagged value in tagged union type systems; the difference again is that the "tag" is inserted implicitly.

The translation judgment has the form $\Gamma \vdash \widetilde{t} \Rightarrow t : U$: under type environment Γ, $\mathrm{GTFL}^{\oplus}$ term $\widetilde{t}$ of type U, is translated to $\mathrm{GTFL}^{\oplus}_{\Rightarrow}$ term t. The translation rules given in Fig. 5 are standard. Cast insertion rules use twosomes to ease readability; a twosome $\langle U_2 \Leftarrow U_1\rangle t$ is equal to $\langle\!\langle U_2 \xleftarrow{U_1 \sqcap U_2} U_1\rangle\!\rangle t$: the initial middle type is the meet of both ends [28].

$$(\text{Capp})\frac{\Gamma \vdash \widetilde{t_1} \Rightarrow t_1' : U_1 \quad \Gamma \vdash \widetilde{t_2} \Rightarrow t_2' : U_2 \quad U_2 \sim \widetilde{dom}(U_1)}{\Gamma \vdash \widetilde{t_1}\ \widetilde{t_2} \Rightarrow \langle \widetilde{dom}(U_1) \rightarrow \widetilde{cod}(U_1) \Leftarrow U_1\rangle t_1'\ \langle \widetilde{dom}(U_1) \Leftarrow U_2\rangle t_2' : \widetilde{cod}(U_1)}$$

$$(\text{C+})\frac{\Gamma \vdash \widetilde{t_1} \Rightarrow t_1' : U_1 \quad \Gamma \vdash \widetilde{t_2} \Rightarrow t_2' : U_2 \quad U_1 \sim \mathsf{Int} \quad U_2 \sim \mathsf{Int}}{\Gamma \vdash \widetilde{t_1} + \widetilde{t_2} \Rightarrow \langle \mathsf{Int} \Leftarrow U_1\rangle t_1' + \langle \mathsf{Int} \Leftarrow U_2\rangle t_2' : \mathsf{Int}}$$

$$(\text{C::})\frac{\Gamma \vdash \widetilde{t} \Rightarrow t' : U \quad U \sim U_1}{\Gamma \vdash (\widetilde{t} :: U_1) \Rightarrow \langle U_1 \Leftarrow U\rangle t' : U_1}$$

$$(\text{Cif})\frac{\Gamma \vdash \widetilde{t_1} \Rightarrow t_1' : U_1 \quad U_1 \sim \mathsf{Bool} \quad \Gamma \vdash \widetilde{t_2} \Rightarrow t_2' : U_2 \quad \Gamma \vdash \widetilde{t_3} \Rightarrow t_3' : U_3}{\begin{array}{l}\Gamma \vdash \mathsf{if}\ \widetilde{t_1}\ \mathsf{then}\ \widetilde{t_2}\ \mathsf{else}\ \widetilde{t_3} \Rightarrow \\ \quad \mathsf{if}\ \langle \mathsf{Bool} \Leftarrow U_1\rangle t_1'\ \mathsf{then}\ \langle U_2 \sqcap U_3 \Leftarrow U_2\rangle t_2'\ \mathsf{else}\ \langle U_2 \sqcap U_3 \Leftarrow U_3\rangle t_3' : U_2 \sqcap U_3\end{array}}$$

Fig. 5. Cast insertion: from GTFL$^{\oplus}$ to GTFL$^{\oplus}_{\Rightarrow}$ (selected rules)

Properties of GTFL$^{\oplus}$. GTFL$^{\oplus}$ satisfies a number of properties. First GTFL$^{\oplus}$ satisfies a standard type safety property:

Proposition 8 (Type safety). *Suppose that* $\cdot \vdash \widetilde{t} \Rightarrow t : U$*, then either:* t *is a value* v*;* $t \longmapsto$ **error***; or* $t \longmapsto t'$ *for some* t' *such that* $\cdot \vdash_{\widetilde{i}} t' : U$

Second, the gradual type system is a conservative extension of the static type system; *i.e.* both systems coincide on fully-annotated terms.

Proposition 9 (Equivalence for fully-annotated terms). *For any* $t \in$ Term*,* $. \vdash_S t : T$ *if and only if* $. \vdash t : T$

Precision on terms, noted $\widetilde{t_1} \sqsubseteq \widetilde{t_2}$, is the natural lifting of type precision to terms. The gradual type system satisfies the static gradual guarantee of Siek et al. [30], *i.e.* losing precision preserves typeability: if a program is well-typed, then a less precise version of it also type checks, at a less precise type.

Proposition 10 (Static gradual guarantee). *If* $. \vdash \widetilde{t_1} : U_1$ *and* $\widetilde{t_1} \sqsubseteq \widetilde{t_2}$*, then* $. \vdash \widetilde{t_2} : U_2$*, for some* U_2 *such that* $U_1 \sqsubseteq U_2$*.*

Similarly, losing precision preserves reduceability: a program that runs without error continues to do so if it is annotated with less precise types.

Proposition 11 (Dynamic gradual guarantee). *Suppose* $\cdot \vdash \widetilde{t_1} \Rightarrow t_1 : U_1$*,* $\cdot \vdash \widetilde{t_2} \Rightarrow t_2 : U_2$*, and* $t_1 \sqsubseteq t_1'$*. If* $t_1 \longmapsto t_2$ *then* $t_1' \longmapsto t_2'$ *where* $t_2 \sqsubseteq t_2'$*.*

A technical novelty of our work is that we establish all the above properties following a route that differs from prior work. Usually, one establishes type safety of the gradual language by first proving type safety of the internal language and then proving that the cast insertion translation preserves typing [29]. With this approach, the gradual guarantees must then be established separately [30].

Our approach exploits the AGT methodology: we first systematically derive the *direct* runtime semantics of GTFL$^{\oplus}$ (*i.e.* which do *not* rely on a cast insertion translation). We then prove safety and the gradual guarantees, which in fact directly follow from the abstract interpretation framework [14]. Then, we prove that the compilation to threesome combined with the semantics of the internal language together are equivalent to the dynamic semantics derived with AGT. This correctness argument proceeds using logical relations.

5 Related Work

In Sect. 2, we have compared gradual unions to both tagged and untagged unions from the standard type system literature [23], highlighting their key characteristics and differences. Gradual unions are unique in admitting runtime errors, with the benefits of more flexible programming patterns. We have also compared gradual unions to retrofitted type systems for dynamic languages with support for unions. Note that in Flow, Typescript, ℂDuce and Typed Racket, a function that expects an argument of type $A + B$ can accept arguments of type A, B, or $A + B$, but neither of type $A + B + C$ nor $A + D$. In contrast, in GTFL$^{\oplus}$, as long as two gradual union types have at least one *compatible* type in their denotation, then they are compatible. So, a function that expects an argument of type $A \oplus B$ accepts arguments of types such as $A \oplus B \oplus C$ and $A \oplus D$.

Flow-sensitive typing approaches such as occurrence typing [20] support more precise type assignments based on the result of some (type) predicate check. Such techniques can avoid the insertion of unnecessary casts [24]. However, in general, the combination of gradual types with type tests raises questions regarding the dynamic gradual guarantee [30], which have not yet been answered.

Interestingly, languages with set-theoretic (untagged) unions usually also consider intersection types, with distributivity relations such as $(T_1 \vee T_2) \rightarrow T_3 \equiv (T_1 \rightarrow T_3) \wedge (T_2 \rightarrow T_3)$. This law states that if a function accepts a value that is *either* of type T_1 *or* of type T_2, then it behaves as *both* a function of type $T_1 \rightarrow T_3$ *and* a function of type $T_2 \rightarrow T_3$. Gradual unions encompass both interpretations, without having to resort to a notion of intersection types: $(T_1 \oplus T_2) \rightarrow T_3 \equiv (T_1 \rightarrow T_3) \oplus (T_2 \rightarrow T_3)$, because both types have the same interpretation, *i.e.* they represent the same concrete set of static types. This simplicity is a consequence of the optimistic interpretation with dynamic checks that is characteristic of gradual typing. (Note that it resonates with the fact that type precision is covariant in both positive and negative positions.)

Siek and Tobin-Hochstadt studied the interaction between gradual typing and union types [27]. While seemingly related, the focus of their work is very different: the addition of the unknown type to a language with *static* union types. Additionally, they only support the union of types with different type constructors, so for instance the union of two function types is not supported.

Similarly, in parallel with this work, Castagna and Lanvin developed a theory for gradual set-theoretic types, supporting union, intersection and the unknown type [7]. Their system can express constructs similar to gradual unions by using a combination of unions and intersection with the unknown type, *e.g.* $\mathsf{Int} \oplus \mathsf{Bool}$ is equivalent to $(\mathsf{Int} \mid \mathsf{Bool})\&?$. They also exploit AGT to derive the static semantics, although the more expressive setting with static unions and intersections makes the design of the Galois connection much more challenging. Our design is minimalist, providing a novel form of union types to languages that do not initially support such set-theoretic types. They mention compilation to threesomes and proving the gradual guarantees as future work.

Jafery and Dunfield [19] present a gradual language that features two types of (datasort) refinement sums, for either exhaustive or non-exhaustive matches. Non-exhaustive matches are backed by dynamic checks in case of an unsuccessful match. Elements with a sum type must be explicitly injected; the sum constructors are neither commutative nor associative. Also, they do not discuss the interaction with the fully-unknown type.

6 Conclusion

Inspired by the interpretation of gradual types as a general approach to deal with imprecision at the type level, and recognizing that unions types are a form of imprecision, we proposed the novel notion of *gradual union types*. Gradual unions are a new design for dealing with the possibility for expressions to have different, unrelated types. Accepting the possibility of runtime cast errors, gradual unions combine and extend the convenience of both tagged and untagged union types. We have presented the meta-theory of gradual union types and their interaction with the traditional unknown type, using the AGT methodology. We have described a compilation semantics to a threesome calculus, and established its desired properties through logical relations. The combination of both gradual type constructors forced us to explore a stratified approach to AGT, whereby each gradual type constructor is interpreted separately and then carefully composed in order to ensure optimality of the resulting abstraction. This compositional approach to designing a gradual language is novel. We hope that it helps understanding how to combine different gradualization efforts that have been developed independently, and may not be fully orthogonal.

Acknowledgments. We thank Gabriel Scherer, Ronald Garcia and the anonymous reviewers for their detailed comments and suggestions.

References

1. Abadi, M., Cardelli, L., Pierce, B., Plotkin, G.: Dynamic typing in a statically typed language. ACM Trans. Program. Lang. Syst. **13**(2), 237–268 (1991)
2. Ahmed, A., Findler, R.B., Siek, J., Wadler, P.: Blame for all. In: POPL 2011 (2011)
3. Bañados Schwerter, F., Garcia, R., Tanter, É.: A theory of gradual effect systems. In: ICFP 2014 (2014)

4. Barbanera, F., Dezani-Ciancaglini, M., De'Liguoro, H.: Intersection and union types: syntax and semantics. Inf. Comput. **119**, 202–230 (1995)
5. Benzaken, V., Castagna, G., Frisch, A.: CDuce: an XML-centric general purpose language. In: ICFP 2003 (2003)
6. Buneman, P., Pierce, B.: Union types for semistructured data. In: Connor, R., Mendelzon, A. (eds.) DBPL 1999. LNCS, vol. 1949, pp. 184–207. Springer, Heidelberg (2000). doi:10.1007/3-540-44543-9_12
7. Castagna, G., Lanvin, V.: Gradual typing with union and intersection types. In: ICFP 2017 (2017)
8. Microsoft Corporation: Typescript language specification. https://www.typescriptlang.org/. Accessed June 2017
9. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977 (1977)
10. Disney, T., Flanagan, C.: Gradual information flow typing. In: STOP 2011 (2011)
11. Facebook: Flow: a static type checker for JavaScript. https://flow.org/. Accessed June 2017
12. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping. In: LICS 2002 (2002)
13. Garcia, R., Cimini, M.: Principal type schemes for gradual programs. In: POPL 2015 (2015)
14. Garcia, R., Clark, A.M., Tanter, É.: Abstracting gradual typing. In: POPL 2016 (2016)
15. Harper, P.R.: Practical Foundations for Programming Languages. Cambridge University Press, Cambridge (2012)
16. Henglein, F.: Dynamic typing: syntax and proof theory. Sci. Comput. Program. **22**(3), 197–230 (1994)
17. Herman, D., Tomb, A., Flanagan, C.: Space-efficient gradual typing. In: Trends in Functional Programming (2007)
18. Hosoya, H., Vouillon, J., Pierce, B.C.: Regular expression types for XML. In: ICFP 2000 (2000)
19. Jafery, K.A., Dunfield, J.: Sums of uncertainty: refinements go gradual. In: POPL 2017 (2017)
20. Kent, A.M., Kempe, D., Tobin-Hochstadt, S.: Occurrence typing modulo theories. In: PLDI 2016 (2016)
21. Lehmann, N., Tanter, É.: Gradual refinement types. In: POPL 2017 (2017)
22. Pierce, B.C.: Programming with intersection types, union types, and polymorphism. Technical report CMU-CS-91-106 (1991)
23. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
24. Rastogi, A., Chaudhuri, A., Hosmer, B.: The ins and outs of gradual type inference. In: POPL 2012 (2012)
25. Rompf, T., Amin, N.: Type soundness for dependent object types (DOT). In: OOPSLA 2016 (2016)
26. Siek, J., Taha, W.: Gradual typing for objects. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 2–27. Springer, Heidelberg (2007). doi:10.1007/978-3-540-73589-2_2
27. Siek, J., Tobin-Hochstadt, S.: The recursive union of some gradual types. In: Wadler Festschrift (2016)
28. Siek, J., Wadler, P.: Threesomes, with and without blame. In: POPL 2010 (2010)
29. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming Workshop (2006)

30. Siek, J.G., Vitousek, M.M., Cimini, M., Boyland, J.T.: Refined criteria for gradual typing. In: SNAPL 2015 (2015)
31. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed scheme. In: POPL 2008 (2008)
32. Tobin-Hochstadt, S., Felleisen, M.: Logical types for untyped languages. In: ICFP 2010 (2010)
33. Toro, M., Tanter, É.: Gradual union types–complete definition and proofs. Technical report TR/DCC-2017-1, University of Chile, June 2017