Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 354 (2020) 171–186

www.elsevier.com/locate/entcs

Gradual Typing Using Union Typing With Records

Karla Ramírez Pulido[1] Jorge Luis Ortega-Arjona[2]
Lourdes del Carmen González Huesca[3]

Departamento de Matemáticas, Facultad de Ciencias
UNAM, Circuito Exterior S/N,
Cd. Universitaria 04510, CdMx, México

Abstract

Dynamic typed languages are characterized by their expressiveness and flexibility to develop prototypes, while static typed languages allow early detection of errors and the optimization of source code. Gradual typing languages allow programmers to make use of both approaches, static and dynamic typing, and thus, obtaining the advantages that both represent.
The objective here is to revisit the static part of the approach to a gradual interpretation of union types based on the design of Gradual Union Types through an extension with the record data-structure. This contributes to understand the abstraction and reasoning behind Gradual Typing in order to have useful future extensions for other data-structures.

Keywords: Gradual typing, Union types, Records

1 Introduction

Commonly, programming languages can be broadly categorized according to their typing as static or dynamic. Nevertheless, there are other typing classifications that allow to mix typing features between both static and dynamic extremes, such as hybrid typing and gradual typing [15]. In particular, gradual typing languages allow programmers to make use of static and dynamic typing, and thus, obtaining the advantages that both represent.

Gradual typing makes use of typed and untyped code in the same language. The concept has been proposed by Siek and Taha in 2006 [15], who expose the idea that mixing dynamic and static features in a single language may improve

[1] Email: karla@ciencias.unam.mx
[2] Email: jloa@ciencias.unam.mx
[3] Email: luglzhuesca@ciencias.unam.mx

https://doi.org/10.1016/j.entcs.2020.10.013

the intrinsic typing characteristics of such a programming language. Having explicit types can guarantee certain properties about the program, related with its soundness and security, and thus, certain kind of optimizations can be performed for executable code, since some typing information is already known. Moreover, not having explicit types allow programmers to have a better expressiveness, and to generate prototypes of programs faster to test and execute.

In their original work, Siek and Taha [15] start from a language which is statically typed to obtain a gradual system by adding the character ? to express an *unknown type*. Having this type enforces the static properties of the system as gradual types represent static information, in this way gradual typing supports static and dynamic checking of program properties [2]. Programmers can explicitly write static types, as well as leave some parts of the code without typing. A type that has not been defined as "static" by the programmer should be verified at run-time, this information is depicted by a gradual type system which should preserve type consistency for both typed and untyped code.

In dynamic typed languages (such as Racket, Scheme or Lisp), an identity function can be declared as $(lambda(x)x)$. In this declaration it is only stated that the type of x is unknown at this very moment. Later, during run-time, the type of x must be known. Notice that as it is an identity function, the type of the return value x can be only known until the type of the argument x is already known. In a gradual typing system, the above declaration should have a type that represents the notion of typing in run-time but in compilation time and hence a type consistency notion is essential.

The relation of consistency has a significant importance because it must be guaranteed that programs do not fail, that is, if they reach an unsoundness state. The relation of consistency has to be maintained, even when the unknown type ? of Gradual Typing is present [7]. A consistency relation is given to extend conservatively the type equality: $T \sim T'$ expresses that during run-time, T is structurally equal to T' with the exception of some features but mainly to do not allow any type inconsistency. This will ensure that soundness and security, or safety of the gradual language is preserved.

Robustness of usual extensions of functional languages are based on the preservation of fundamental properties such as security. A meta-theory, as presented by Toro and Tanter [19] allows to extend a language into a gradual one in a safe manner: the gradual union of types and the unknown type ? are combined through the Abstract Gradual Typing (AGT) methodology. The AGT is based on abstract interpretation due to Cousot and Cousot [4], for a sound abstraction using Galois connections. Abstract Interpretation describes the computations made in a given system but in a different universe. For example, the elements in a program and their operations can be modeled in another language, preserving their semantics.

Since Siek and Taha introduced the term Gradual Typing [15], several other studies have been performed which encompass Gradual Typing with type security [5],

type states [8], Object-Orientation [14,18], and so on. Although the interest for Gradual Typed languages has been increased over the last few years, there are still questions about how more sophisticated language features can be included into them [7].

We present an extension of a functional language with records. Such an structure allows the programmer to manipulate expressions that could have different types. The gradual extension is carried out with gradual union types, which are intended to preserve the safety property. In this paper the extension of the type system is provided in order to understand the abstraction behind the AGT methodology, as proposed by [19].

2 Gradual Union Typing

Some proposals have been made to address the inexactness introduced by the unknown type ? of Gradual Typing languages. Two are mainly used [19]:

(i) The use of disjoint (or tagged) types, such as the sum of the types $T_1 + T_2$ explicitly tagged; and

(ii) Untagged union types, denoted by $T_1 \vee T_2$, which denotes the union of values of type T_1 and of type T_2.

Toro and Tanter combine these two proposals into a Gradual Union Typing (GUT) approach, in which it is possible to have the two different types of union: disjoint or tagged union types, and untagged union types. Mixing tagged and untagged unions in a language is named as *optimistic type checking* to ensure that statically any term will have a type. In particular, optimistic type checking means that some statement is well typed without the occurrence of any explicit projection or case analysis.

An Abstract Gradual Interpretation of union types is studied, following [19], in order to understand both proposals for the union of types. Thus, it is possible to mix both: disjoint or tagged union types and their variant types, as well as untagged union types:

- Disjoint union types or tagged unions ($T_1 + T_2$) denote values of different types. They are called disjoint because some elements are explicitly tagged, so the type system recognizes the type of each element. Hence, it guarantees and ensures the type of each element during compilation and execution time.
- Untagged unions ($T_1 \vee T_2$) denote the unions of values of types T_1 and T_2 without any tagging mechanism. For example, a value v of type T_1 is included as a value of type $T_1 \vee T_2$, that is, the type of value v appears in the union of both sets of types T_1 and T_2, i.e. the *highest lower bound* between T_1 and T_2. Untagged unions can be used in the branches of conditionals inside of an *if* expression, where *then* and *else* branches can return a different type of values. Thus, untagged unions can be safely compiled, but can lead to errors during execution time.

2.1 A Gradual Interpretation of Union Types

Toro and Tanter, based on the AGT methodology, propose static semantics for a Gradual Typed Functional Language, which make use of gradual unions and unknown typing. The notion of gradual union types is to restrict the imprecise types to denote a finite number of static types [19]. Such a semantic is based on the Simply-Typed Functional Language (STFL), which is statically typed.

Types of a Gradual Type Functional Language ($\text{GTFL}_{\oplus}$), as defined by Toro and Tanter [19], extend the STFL syntax by adding the symbol ?, introduced as the unknown type. This type is present in the Gradual Type Functional System (GTFL) [7], which preserves the STFL structure, where static types, predicates, and partial functions are lifted to gradual types. This transformation between sets of static types and gradual types is based on a Galois connection as the AGT approach states. The consistency relationship extends static type equality and is defined by the following judgments [7]:

$$\frac{}{T \sim T} \qquad \frac{}{T \sim ?} \qquad \frac{}{? \sim T} \qquad \frac{T_{11} \sim T_{21} \qquad T_{12} \sim T_{22}}{T_{11} \to T_{12} \sim T_{21} \to T_{22}}$$

In this paper, the extension with records is carried out using the methodology given in [19]. The first transformation uses an abstraction function $\alpha_?$ to convert STFL into an GTFL, and a concretization function $\gamma_?$ from GTFL to STFL to complete the Galois connection. Second step is to obtain a $\text{GTFL}_{\oplus}$ from GTFL by adding the gradual union denoted by: $T_1 \oplus T_2$ and using $\alpha_{\oplus}$ and $\gamma_{\oplus}$ functions between these languages for the Galois connection. Figure 1 shows these connections between languages.

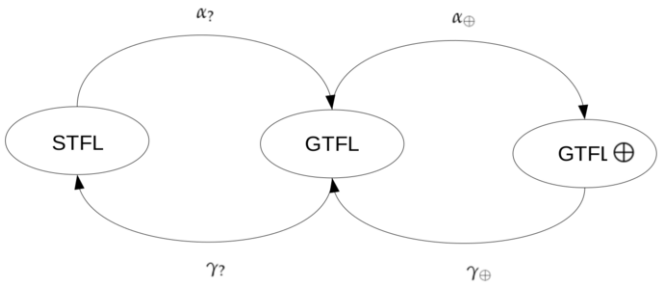

Figure 1. Methodology proposed by Toro and Tanter, using α, γ, $\alpha_{\oplus}$, $\gamma_{\oplus}$ functions to convert STFL into $\text{GTFL}_{\oplus}$.

For the purposes of this paper, the left side of Figure 2 shows the lifting of the types on STFL, which is expressed by the infinite power set $\wp(TYPE)$, to the syntactic category GTYPE using functions $\alpha_?$ and $\gamma_?$.

From these sets, it is necessary to lift the set of types into a finite one, this implies the use of the power set operator $\wp$.

Then, functions $\widehat{\alpha_?}$ and $\widehat{\gamma_?}$ are defined between $\wp_{fin}(\wp(TYPE))$ and $\wp_{fin}(GTYPE)$ [19].

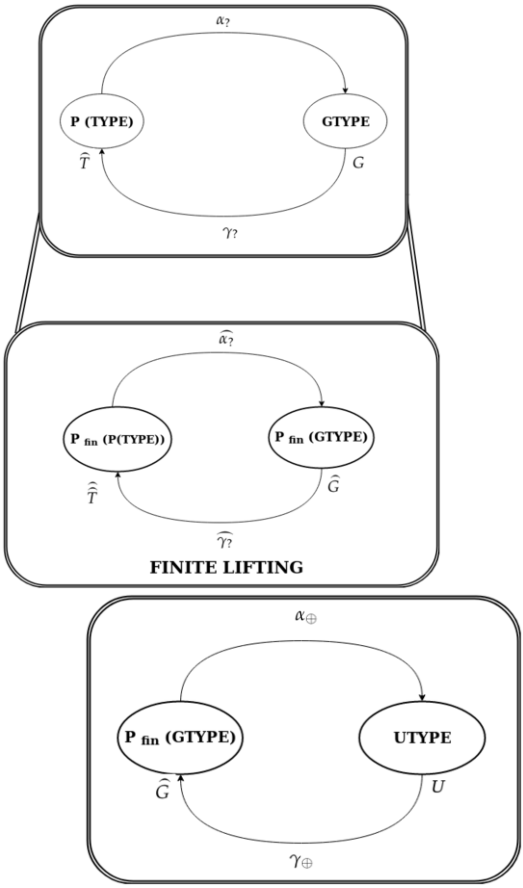

Figure 2. On the left, the finite lifting over $\wp(TYPE)$ and $GTYPE$. On the right, the finite lifting over $\wp_{fin}(GTYPE)$ and $UTYPE$.

The right side of Figure 2 shows the last transformation in order to obtain system $GTFL_{\oplus}$, through functions $\alpha_{\oplus}$ and $\gamma_{\oplus}$, in particular between $\wp(GTYPE)$ and $UTYPE$ (i.e. between $GTFL$ and $GTFL_{\oplus}$) [19].

3 Gradual Union Types with Records

This paper extends the GUT by Toro and Tanter with records. A record is a data-type that describes variables (called fields) and their values. Fields possibly have different types. Some programming languages use records to define new types, and therefore, are able to extend the language [6,12]. A record is composed by one or more labels, here denoted as ℓ. Each label has an assigned term, which also has a type.

Adding records allows to extend a language maintaining complex information in the same structure, and having the possibility to access it through an identifier of each one of its fields. The extension here presented is based on the system of Toro and Tanter [19] and is given in Figure 3. The system adds records as primitive types (shown in boldface) where the labels ℓ_i belong to a predetermined set $LABEL$.

$$T \in TYPE, \quad x \in VAR, \quad t \in TERM, \quad \Gamma \in VAR \xrightarrow{\text{fin}} TYPE, \quad \boldsymbol{\ell \in LABEL}$$

$$T ::= Int \mid Bool \mid T \to T \mid [\boldsymbol{\ell_i : T_i}^{\,i \in 1..n}] \qquad (types)$$

$$v ::= n \mid b \mid \lambda x : T.t \mid [\boldsymbol{\ell_i = v_i}^{\,i \in 1..n}] \qquad (values)$$

$$t ::= v \mid x \mid t + t \mid if\ t\ then\ t\ else\ t \mid t :: T \mid\ t\ t \mid [\boldsymbol{\ell_i = t_i}^{\,i \in 1..n}] \mid \boldsymbol{t.\ell_j} \qquad (terms)$$

$$\frac{}{\Gamma \vdash n : Int}\ (T_n) \qquad \frac{}{\Gamma \vdash b : Bool}\ (T_b) \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ (T_x)$$

$$\frac{\Gamma \vdash t_1 : T_1 \quad T_1 = Int \quad \Gamma \vdash t_2 : T_2 \quad T_2 = Int}{\Gamma \vdash t_1 + t_2 : Int}\ (T_+)$$

$$\frac{\Gamma \vdash t_1 : T_1 \quad T_1 = Bool \quad \Gamma \vdash t_2 : T_2 \quad \Gamma \vdash t_3 : T_3}{\Gamma \vdash if\ t_1\ then\ t_2\ else\ t_3 : equate(T_2,\ T_3)}\ (T_{if})$$

$$\frac{\Gamma \vdash t : T \quad T = T_1}{\Gamma \vdash (t :: T_1) : T_1}\ (T_{::})$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash (\lambda x : T_1.t) : T_1 \to T_2}\ (T_\lambda)$$

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \quad T_2 = dom(T_1)}{\Gamma \vdash t_1\ t_2 : cod(T_1)}\ (T_{app})$$

$$\frac{\boldsymbol{\Gamma \vdash t_i : T_i}^{\,i \in 1..n} \quad \boldsymbol{for\ each\ i}}{\boldsymbol{\Gamma \vdash [\ell_i = t_i}^{\,i \in 1..n}\boldsymbol{] : [\ell_i : T_i}^{\,i \in 1..n}\boldsymbol{]}}\ \boldsymbol{(T_{rec})}$$

$$\frac{\boldsymbol{\Gamma \vdash t : [\ell_i : T_i}^{\,i \in 1..n}\boldsymbol{]}}{\boldsymbol{\Gamma \vdash t.\ell_j : T_j \quad for\ some\ j \in 1..n}}\ \boldsymbol{(T_{proj})}$$

dom: $TYPE \rightharpoonup TYPE$	*cod:* $TYPE \rightharpoonup TYPE$	*equate:* $TYPE \times TYPE \rightharpoonup TYPE$
$dom(T)$ undefined in other case	$cod(T)$ undefined in other case	$equate(T_1, T_2)$ undefined in other case

Figure 3. Syntax and type system of Simply Typed Functional Language, STFL [7] extended with records (shown in boldface).

From Figure 3, a record type is denoted by $[\ell_i : T_i{}^{\,i \in 1,\ldots,n}]$, where ℓ_i denote a label. A record term is expressed as $[\ell_i = t_i{}^{\,i \in 1,\ldots,n}]$, where ℓ_i denotes a field for term t_i and a record value is defined as $[\ell_i = v_i{}^{\,i \in 1,\ldots,n}]$ where each v_i is a value. Each record has a fixed size n. The typing rules for records are T_{rec} and T_{proj}, following the typing rules given in [12]:

T_{rec}: in order to assign a type to a record term, if each term t_i has type T_i under Γ, then term $[\ell_i = t_i{}^{\,i \in 1..n}]$ has type $[\ell_i : T_i{}^{\,i \in 1..n}]$, in the same context Γ.

T_{proj}: in order to assign a type to a projection j, if a term t has a record type $t : [\ell_i : T_i{}^{\,i \in 1..n}]$ in context Γ, then term t_j of type T_j can be obtained through label ℓ_j.

System STFL has been proved safe [12], we only give some properties in order to prove the gradual extensions.

Record substitution:

$$[\ell_1 = t_1, \ell_2 = t_2, ..., \ell_n = t_n][v/x] = [\ell_1 = t_1[v/x], \ell_2 = t_2[v/x], ..., \ell_n = t_n[v/x]]$$

Projection of record substitution:

$$(t.\ell_j)[v/x] = (t[v/x].\ell_j)$$

Lemma 3.1 (Inversion of typing relation)

(i) *If* $\Gamma \vdash n : R$, *then* $n = Int$.
(ii) *If* $\Gamma \vdash true : R$, *then* $R = Bool$.
(iii) *If* $\Gamma \vdash false : R$, *then* $R = Bool$.
(iv) *If* $\Gamma \vdash x : R$, *then* $x : R \in \Gamma$.
(v) *If* $\Gamma \vdash t_1 + t_2 : R$, *then* $R = Int$.
(vi) *If* $\Gamma \vdash t :: T_1 : R$, *then* $\Gamma \vdash t : R$ *and* $R = T_1$.
(vii) *If* $\Gamma \vdash if\ t_1\ then\ t_2\ else\ t_3 : R$, *then* $\Gamma \vdash t_1 : Bool$, $\Gamma \vdash t_2 : R$ *and* $\Gamma \vdash t_3 : R$.
(viii) *If* $\Gamma \vdash \lambda x : T_1.t_2 : R$, *then* $R = T_1 \implies R_2$ *for some* R_2 *with* $\Gamma, x : T_1 \vdash t_2 : R_2$.
(ix) *If* $\Gamma \vdash t_1 t_2 : R$, *then there is some type* T_{11} *such that* $\Gamma \vdash t_1 : T_{11} \implies R$ *and* $\Gamma \vdash t_2 : T_{11}$.
(x) $\boldsymbol{If\ \Gamma \vdash [\ell_i = t_i\ ^{i \in 1..n}] : [\ell_i : T_i\ ^{i \in 1..n}]\ then\ \Gamma \vdash t_i : T_i\ for\ each\ i \in 1..n}$.
(xi) $\boldsymbol{If\ \Gamma \vdash t.\ell_j : T_j\ for\ some\ j \in 1..n\ then\ \Gamma \vdash t : [\ell_i : T_i\ ^{i \in 1..n}]}$.

3.1 A Gradual Typing Language

The following step is to add records for a Gradual Typing Language (GTFL). This means adding the unknown type ? to the STFL. In the following, we present the extension with records based on the work of [19], the new cases are in boldface:

$$G \in GTYPE \qquad G ::= Int \mid Bool \mid G \to G \mid ? \mid \boldsymbol{[\ell_i : G_i\ ^{i \in 1..n}]}$$

The concretization function is defined as follows:

$$\gamma_?(Int) = \{Int\} \qquad \gamma_?(Bool) = \{Bool\} \qquad \gamma_?(?) = TYPE$$

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

$$\boldsymbol{\gamma_?([\ell_i : G_i^{\,i \in 1..n}]) = \{[\ell_i : T_i^{\,i \in 1..n}] \mid T_i \in \gamma_?(G_i)\ ^{i \in 1..n}]\}}$$

Notice that the above function helps to preserve (i.e. to know) the static information of gradual types [19,7]. The case for $\gamma_?(?) = TYPE$ states that the unknown

type could be any of the types in the category $TYPE$ defined in STFL.

The abstraction function, following [19], is defined using $\widehat{T}$ which denotes a set of types induced by T:

$$\widehat{T_1 \to T_2} = \{T' \to T'' \mid T' \in \widehat{T_1} \ \wedge \ T'' \in \widehat{T_2}\}$$

$$\widehat{[\ell_i : T_i \ ^{i\in 1..n}]} = \{[\ell_i : T_{i,j} \ ^{i\in 1..n}] \mid T_{i,j} \in \widehat{G_i} \ \wedge \ i \in 1..n \ \wedge \ j \in 1..|\widehat{G_i}|\}$$

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

$$\boldsymbol{\alpha_?(\widehat{[\ell_i : T_i \ ^{i\in 1..n}]}) = [\ell_i : \alpha_?(\widehat{T_i}) \ ^{i\in 1..n}]} \qquad \alpha_?(\widehat{T}) = ? \ \ otherwise$$

The abstraction function $\alpha_?$ helps the language to obtain or retain more precision, while the unknown type is decreased, that is, the $\alpha_?$ function can produce a gradual type that abstracts a given set [7]. Toro and Tanter define optimal as the best approximation of $\alpha_?$, that means the most precise gradual type that is possible. Both functions $\gamma_?$ and $\alpha_?$ define the Galois connection [19].

Definition 3.2 [$GTYPE$ Precision] G_1 is less imprecise than G_2, $G_1 \sqsubseteq G_2$, if and only if $\gamma_?(G_1) \subseteq \gamma_?(G_2)$.

The next proposition defines when $\alpha_?$ is sound and optimal.

Proposition 3.3 ($\alpha_?$ is sound and optimal) *If $\widehat{T}$ is non empty, then:*

(i) $\widehat{T} \subseteq \gamma_?(\alpha_?(\widehat{T}))$

(ii) $\widehat{T} \subseteq \gamma_?(G) \implies \alpha_?(\widehat{T}) \sqsubseteq G$

Proof Proof by induction on the structure of G, we show the case for records.

Case 1 for records ($\widehat{[\ell_i : T_i^{\,i\in 1..n}]}$)

If $\widehat{T} = \widehat{[\ell_i : T_i \ ^{i\in 1..n}]} = \{[\ell_i : T_{i,j} \ ^{i\in 1..n}] \mid T_{i,j} \in \widehat{T_i} \ \wedge \ i \in 1..n \ \wedge \ j \in 1..|\widehat{T_i}|\}$ then by definition of $\alpha_?$: $\alpha_?(\widehat{[\ell_i : T_i \ ^{i\in 1..n}]}) = [\ell_i : \alpha_?(\widehat{T_i}) \ ^{i\in 1..n}]$

and by definition of $\gamma_?$ for records we have $\gamma_?(\alpha_?(\widehat{T})) = \gamma_?([\ell_i : \alpha_?(\widehat{T_i}) \ ^{i\in 1..n}]) = \gamma_?([\ell_i : G_i \ ^{i\in 1..n}]) = \{[\ell_i : T_i{}^{i\in 1..n}] \mid T_i \in \gamma_?(G_i) \ ^{i\in 1..n}\} = \{[\ell_i : T_i{}^{i\in 1..n}] \mid T_i \in \gamma_?(\alpha_?(\widehat{T_i})) \ ^{i\in 1..n}\}$.

By induction hypothesis $T_{i,j} \in \widehat{T_i} \subseteq \gamma_?(\alpha_?(\widehat{T_i}))$ for each $T_{i,j}$ then the result holds.

Case 2 for records ($\widehat{[\ell_i : T_i{}^{i\in 1..n}]}$)

Let $\widehat{T} = \widehat{[\ell_i : T_i{}^{\ i\in 1..n}]}$, and suppose $\widehat{T} = \widehat{[\ell_i : T_i{}^{\ i\in 1..n}]} \subseteq \gamma_?([\ell_i : G_i{}^{\ i\in 1..n}])$

and by definition of $\gamma_?$: $\{[\ell_i : T_i{}^{\,i\in 1..n}] \mid T_i \in \gamma_?(G_i)^{\ i\in 1..n}\}$

By definition of $\alpha_?$ for records: $\alpha_?(\widehat{[\ell_i : T_i{}^{\ i\in 1..n}]}) = [\ell_i : \alpha_?(\widehat{T_i})^{\,i\in 1..n}]$

By the induction hypothesis, for each $\widehat{T_i}$ if $\widehat{T_i} \subseteq \gamma_?(G_i)$ then $\alpha_?(\widehat{T_i}) \sqsubseteq G_i$.

Therefore, by definition 3.2 $\gamma_?(\alpha_?(\widehat{[\ell_i : T_i{}^{\,i\in 1..n}]})) = \gamma_?([\ell_i : \alpha_?(\widehat{T_i})^{\,i\in 1..n}]) \subseteq$

$\gamma_?([\ell_i : G_i{}^{\,i\in 1..n}])$ and the result holds. □

The next step to give a Gradual Union Typing is to lift functions $\alpha_?$ and $\gamma_?$ to sets, that is from $\wp_{fin}(\wp(TYPE))$ to $\wp_{fin}(GTYPE)$ and vice-versa. Observe that this lifting is over finite sets of types. We follow again Toro and Tanter [19].

Definition 3.4 [$\wp_{fin}(GTYPE)$ Concretization]

$\widehat{\gamma_?} : \wp_{fin}(GTYPE) \to \wp_{fin}(\wp(TYPE))$ is defined as: $\widehat{\gamma_?}(\widehat{G}) = \{\gamma_?(G) \mid G \in \widehat{G}\}$

Definition 3.5 [$\wp_{fin}(GTYPE)$ Abstraction]

$\widehat{\alpha_?} : \wp_{fin}(\wp(TYPE)) \rightharpoonup \wp_{fin}(GTYPE)$ is defined as:

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

Proposition 3.6 ($\widehat{\alpha_?}$ is sound and optimal) *If $\widehat{\widehat{T}}$ is non empty, then:*

(i) $\widehat{\widehat{T}} \subseteq \widehat{\gamma_?}(\widehat{\alpha_?}(\widehat{\widehat{T}}))$

(ii) $\widehat{\widehat{T}} \subseteq \widehat{\gamma_?}(\widehat{G}) \implies \widehat{\alpha_?}(\widehat{\widehat{T}}) \sqsubseteq \widehat{G}$

Proof The proof is the same as in [19] as we are working with sets. An important observation is that $\widehat{\widehat{T}}$ is indeed a $\widehat{G}$. □

3.2 A Gradual Union Typing Language

In order to construct a Gradual Union Typing, it is proposed in [19] a $STYPE$ grammar for the static types. This grammar is made up only of gradual unions (without the unknown type ?).

$$S \in STYPE$$

$$S ::= Int \mid Bool \mid S \to S \mid S \oplus S \mid [\boldsymbol{\ell_i : S_i}^{\,i \in 1..n}]$$

The static types are integers (Int), booleans ($Bool$), functions ($S \to S$), the gradual unions denoted by $S \oplus S$ and the record type. Recall that gradual unions represent a finite set of types [19], where the concretization and abstraction functions are defined between static types and finite sets of types generating a Galois connection.

For a gradual interpretation of union types with records, the next step is to design gradual union types, using the AGT to construct the semantics for gradual types in terms of pre-existing sets of static types.

The new type system is composed of Gradual Unions $\oplus$, named GTFL$_\oplus$, combining both $STYPE$ and $UTYPE$. The last syntactic category, $UTYPE$, is the extension including the unknown type, the gradual unions and records. Figure 4 shows the grammar of $GTFL_\oplus$, extended with records, where types are extended from GTFL with union types, x denotes the set of VARiables, $\widetilde{t}$ denotes the set of gradual union terms, and contexts are partial function $\Gamma \in VAR \xrightarrow{\text{fin}} UTYPE$.

We define a Galois connection between $UTYPE$ and $\wp_{fin}(GTYPE)$ as the next concretization function in $UTYPE$ [19]. The extension with records is again expressed in boldface:

Definition 3.7 [$UTYPE$ concretization] $\gamma_\oplus : UTYPE \to \wp_{fin}(GTYPE)$

$$\gamma_\oplus(Int) = \{Int\} \quad \gamma_\oplus(Bool) = \{Bool\} \quad \gamma_\oplus(?) = \{?\}$$

$$\gamma_\oplus(U_1 \to U_2) = \{G_1 \to G_2 \mid G_1 \in \gamma_\oplus(U_1) \wedge G_2 \in \gamma_\oplus(U_2)\}$$

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

$$\boldsymbol{\gamma_\oplus([\ell_i : U_i}^{\,i \in 1..n}\boldsymbol{]) = \{[\ell_i : G_i}^{\,i \in 1..n}\boldsymbol{] \mid G_i \in \gamma_\oplus(U_i)}^{\,i \in 1..n}\boldsymbol{\}}$$

The definition of abstraction function $\alpha_\oplus$ ($UTYPE$ abstraction) is the same defined by Toro and Tanter [19] and produces finite sets, that is the gradual union of all elements in $\widehat{G}$:

Definition 3.8 [$UTYPE$ abstraction] $\alpha_\oplus : \wp_{fin}(GTYPE) \rightharpoonup UTYPE$

$$\begin{aligned} \alpha_\oplus(\widehat{G}) &= \bigoplus \widehat{G} && if\ G \neq \emptyset \\ &= \alpha_\oplus(G_1) \oplus \alpha_\oplus(G_2) \oplus \ldots \oplus \alpha_\oplus(G_m) && where\ G_i \in \widehat{G} \end{aligned}$$

The next proposition states that $\alpha_\oplus$ is sound and optimal for $UTYPE$. The notion of precision for $UTYPE$ is the same as the one given above, that is $U_1 \sqsubseteq U_2$ if and only if $\gamma_\oplus(U_1) \subseteq \gamma_\oplus(U_2)$

$$U \in UTYPE, \quad x \in VAR, \quad \widetilde{t} \in UTERM, \quad \Gamma \in VAR \overset{\text{fin}}{\rightharpoonup} UTYPE, \quad \boldsymbol{\ell \in ULABEL}$$

$$U ::= Int \mid Bool \mid U \to U \mid U \oplus U \mid ? \mid [\boldsymbol{\ell_i : U_i}^{\,i\in 1..n}] \qquad (types)$$

$$v ::= n \mid true \mid false \mid (\lambda x : U.\widetilde{t}) \mid [\boldsymbol{\ell_i = v_i}^{\,i\in 1..n}] \qquad (values)$$

$$\widetilde{t} ::= v \mid x \mid \widetilde{t} + \widetilde{t} \mid if\ \widetilde{t}\ then\ \widetilde{t}\ else\ \widetilde{t} \mid \widetilde{t} :: U \mid \widetilde{t}\ \widetilde{t} \mid [\boldsymbol{\ell_i = \widetilde{t}_i}^{\,i\in 1..n}] \mid \boldsymbol{\widetilde{t}.\ell_i}^{\,i\in 1..n} \qquad (terms)$$

$$\frac{}{\Gamma \vdash n : Int}\ (U_n) \qquad \frac{}{\Gamma \vdash b : Bool}\ (U_b) \qquad \frac{x : U \in \Gamma}{\Gamma \vdash x : U}\ (U_x)$$

$$\frac{\Gamma \vdash \widetilde{t_1} : U_1 \quad U_1 \sim Int \quad \Gamma \vdash \widetilde{t_2} : U_2 \quad U_2 \sim Int}{\Gamma \vdash \widetilde{t_1} + \widetilde{t_2} : Int}\ (U_+)$$

$$\frac{\Gamma \vdash \widetilde{t_1} : U_1 \quad U_1 \sim Bool \quad \Gamma \vdash \widetilde{t_2} : U_2 \quad \Gamma \vdash \widetilde{t_3} : U_3}{\Gamma \vdash if\ \widetilde{t_1}\ \ then\ \ \widetilde{t_2}\ \ else\ \ \widetilde{t_3}\ :\ \ U_2 \sqcap U_3}\ (U_{if})$$

$$\frac{\Gamma \vdash \widetilde{t} : U \quad U \sim U_1}{\Gamma \vdash (\widetilde{t} :: U_1) : U_1}\ (U_{::})$$

$$\frac{\Gamma,\ x : U_1 \vdash\ \widetilde{t} : U_2}{\Gamma \vdash (\lambda x : U_1.\widetilde{t}) : U_1 \to U_2}\ (U_\lambda)$$

$$\frac{\Gamma \vdash \widetilde{t_1} : U_1 \quad \Gamma \vdash \widetilde{t_2} : U_2 \quad U_2 \sim \widetilde{dom}(U_1)}{\Gamma \vdash \widetilde{t_1}\ \widetilde{t_2} : \widetilde{cod}(U_1)}\ (U_{app})$$

$$\frac{\boldsymbol{for - each\ i\ \Gamma \vdash \widetilde{t_i} : U_i}^{\,i\in 1..n}}{\boldsymbol{\Gamma \vdash [\ell_i = \widetilde{t_i}^{\,i\in 1..n}] : [\ell_i : U_i}^{\,i\in 1..n}]}\ \boldsymbol{(U_{rec})}$$

$$\frac{\boldsymbol{\Gamma \vdash \widetilde{t} : [\ell_i : U_i}^{\,i\in 1..n}]}{\boldsymbol{\Gamma \vdash \widetilde{t}.\ell_j = U_j}^{\,j\in 1..n}}\ \boldsymbol{(U_{proj})}$$

$$\widetilde{dom} : UTYPE \rightharpoonup UTYPE \qquad \widetilde{cod} : UTYPE \rightharpoonup UTYPE$$

$$\widetilde{dom}(U) = \alpha(\widehat{dom}(\gamma(U)) \qquad \widetilde{cod}(U) = \alpha(\widehat{cod}(\gamma(U))$$

Figure 4. Syntaxs and Type System of Gradual Type Functional Language$_\oplus$, $GTFL_\oplus$ [19] extended with records in boldface.

Proposition 3.9 *($\alpha_\oplus$ is sound and optimal). If $\widehat{G}$ is non empty, then:*

$$1.\ \widehat{G} \subseteq \gamma_\oplus(\alpha_\oplus(\widehat{G})) \qquad 2.\ \widehat{G} \subseteq \gamma_\oplus(U) \implies \alpha_\oplus(\widehat{G}) \sqsubseteq U$$

Proof The proof is carried out by induction on the structure of U.

Case 1 for records.

Suppose $G = [\ell_i : G_i{}^{i\in 1..n}]$ then $\widehat{G} = \{[\ell_i : T_{i,j}{}^{i\in 1..n}] \mid T_{i,j} \in \widehat{G_i} \ \wedge\ i \in 1..n \ \wedge\ j \in$

$1..|\widehat{G_i}|\}$. Without loss of generality, take $j = 1$ for each $\widehat{G_i}$ with $i \in 1..(n-1)$ and $|\widehat{G_n}| = m$.

To prove that $\widehat{G} \subseteq \gamma_{\oplus}(\alpha_{\oplus}(\widehat{G}))$ we proceed by applying the definition of $\alpha_{\oplus}(\widehat{G})$:

$\alpha_{\oplus}(\{[\ell_i : T_i, \ell_n : T'] \mid T' \in \widehat{G_n} \ \wedge \ i \in 1..n-1\}) \ =$

$\bigoplus \{[\ell_i : T_i, \ell_n : T'] \mid T' \in \widehat{G_n} \ \wedge \ i \in 1..n-1\} \ =$

$[\ell_i : \alpha_{\oplus}(T_i), \ell_n : \alpha_{\oplus}(G_{n1})] \ \oplus \ \ldots \ \oplus \ [\ell_i : \alpha_{\oplus}(T_i), \ell_n : \alpha_{\oplus}(G_{nm})]$

Using definition of $\gamma_{\oplus}$:

$\gamma_{\oplus}([\ell_i : \alpha_{\oplus}(T_i), \ell_n : \alpha_{\oplus}(G_{n1})] \oplus \cdots \oplus [\ell_i : \alpha_{\oplus}(T_i), \ell_n : \alpha_{\oplus}(G_{nm})]) \ =$

$\gamma_{\oplus}([\ell_i : \alpha_{\oplus}(T_i), \ell_n : \alpha_{\oplus}(G_{n1})]) \cup \cdots \cup \gamma_{\oplus}([\ell_i : \alpha_{\oplus}(T_i), \ell_n : \alpha_{\oplus}(G_{nm})]) \ =$

$\{[\ell_i : \gamma_{\oplus}(\alpha_{\oplus}(T_i)), \ell_n : \gamma_{\oplus}(\alpha_{\oplus}(G_{n1}))]\} \cup \cdots \cup \{[\ell_i : \gamma_{\oplus}(\alpha_{\oplus}(T_i)), \ell_n : \gamma_{\oplus}(\alpha_{\oplus}(G_{nm}))]\} \ =$

$\{[\ell_i : \gamma_{\oplus}(\alpha_{\oplus}(T_i)), \ell_n : G_{nj}] \mid G_{nj} \in \gamma_{\oplus}(\alpha_{\oplus}(\widehat{G_n})) \ \wedge \ j \in 1..m\}$.

By induction hypothesis, $T_i \in \widehat{G_i}$ and also belongs to $\gamma_{\oplus}(\alpha_{\oplus}(\widehat{G_i}))$ for each $i \in 1..n-1$ and $G_{nj} \in \widehat{G_n}$ belongs also to $\gamma_{\oplus}(\alpha_{\oplus}(\widehat{G_n}))$ and therefore $\widehat{G} \subseteq \gamma_{\oplus}(\alpha_{\oplus}(\widehat{G}))$.

Case 2 for records.

Suppose that $\widehat{G} \subseteq \gamma_{\oplus}(U)$ where $\widehat{G} = \widehat{[\ell_i : G_i{}^{i \in 1..n}]}$

and

$\gamma_{\oplus}(U) = \gamma_{\oplus}([\ell_i : U_i{}^{i \in 1..n}]) = \{[\ell_i : G_i{}^{i \in 1..n}] \mid G_i \in \gamma_{\oplus}(U_i)^{\,i \in 1..n}\}$

To prove $\alpha_{\oplus}(\widehat{G}) \sqsubseteq U$ we proceed by definition of $\alpha_{\oplus}(\widehat{G})$:

$\alpha_{\oplus}(\widehat{[\ell_i : G_i{}^{i \in 1..n}]}) = \bigoplus \{[\ell_i : T_{i,j}{}^{i \in 1..n}] \mid T_{i,j} \in \widehat{G_i} \ \wedge \ i \in 1..n \ \wedge \ j \in 1..|\widehat{G_i}|\}$

Without loss of generality, take $j = 1$ for each $\widehat{G_i}$ with $i \in 1..(n-1)$ and $|\widehat{G_n}| = m$, then by definition of $\gamma_{\oplus}$

$\gamma_{\oplus}(\{[\ell_i : T_i, \ell_n : T'] \mid T' \in \alpha_{\oplus}(\widehat{G_n}) \ \wedge \ i \in 1..n-1\}) =$

$$\bigcup \gamma_{\oplus}(\{[\ell_i : T_i, \ell_n : T'] \mid T' \in \alpha_{\oplus}(\widehat{G_n}) \ \wedge \ i \in 1..n-1\}) =$$
$$\bigcup \{[\ell_i : T_i, \ell_n : T'] \mid T' \in \gamma_{\oplus}(\alpha_{\oplus}(\widehat{G_n})) \ \wedge \ i \in 1..n-1\}$$

The induction hypothesis states that $\widehat{G_i} \subseteq \gamma_{\oplus}(U_i) \implies \alpha_{\oplus}(\widehat{G_i}) \sqsubseteq U_i$ for each i which by definition of precision gives $\gamma_{\oplus}(\alpha_{\oplus}(\widehat{G_i})) \subseteq \gamma_{\oplus}(U_i)$, in particular for G_n. Therefore $\bigcup \{[\ell_i : T_i, \ell_n : T'] \mid T' \in \gamma_{\oplus}(U_i) \ \wedge \ i \in 1..n-1\}$ which is indeed $\{[\ell_i : G_i{}^{i \in 1..n}] \mid G_i \in \gamma_{\oplus}(U_i)^{\,i \in 1..n}\}$. □

Therefore, following Toro and Tanter [19], the stratified interpretation of $UTYPE$ is defined in terms of sets of sets of static types with the composition of two presented Galois conections:

Definition 3.10 [$UTYPE$ concretization]

$$\gamma : UTYPE \to \wp_{fin}(\wp(TYPE)) \qquad \gamma = \widehat{\gamma_?} \circ \gamma_{\oplus}$$

Definition 3.11 [$GTYPE$ abstraction]

$$\alpha : \wp_{fin}(\wp(TYPE)) \rightharpoonup UTYPE \qquad \alpha = \alpha_{\oplus} \circ \widehat{\alpha_?}$$

The next proposition is needed to prove that α is sound and optimal:

Proposition 3.12 *(α is sound and optimal). If $\widehat{\widehat{T}}$ is non empty, then:*

(i) $\widehat{\widehat{T}} \subseteq \gamma(\alpha(\widehat{\widehat{T}}))$

(ii) $\widehat{\widehat{T}} \subseteq \gamma(U) \implies \alpha(\widehat{\widehat{T}}) \sqsubseteq U$

Proof This proposition holds by Propositions 3.3, 3.9, and composition of sound and optimal abstractions. □

The next consistency relation is defined for $UTYPE$ as $U_1 \sim U_2$ if and only if there exists $\widehat{T_1} \in \gamma(U_1)$, $T_1 \in \widehat{T_1}$, $\widehat{T_2} \in \gamma(U_2)$ and $T_2 \in \widehat{T_2}$, such that $T_1 = T_2$:

Proposition 3.13 (Consistency Relation)

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

$$\frac{U_1 \sim U}{U_1 \oplus U_2 \sim U} \qquad \frac{U_2 \sim U}{U_1 \oplus U_2 \sim U} \qquad \frac{U_{21} \sim U_{11} \quad U_{12} \sim U_{22}}{U_{11} \to U_{12} \sim U_{21} \to U_{22}}$$

$$\frac{\boldsymbol{U_{11} \sim U_{21} \ldots U_{1n} \sim U_{2n}}}{\boldsymbol{[\ell_1 : U_{11}, \ldots, \ell_n : U_{1n}] \sim [\ell_1 : U_{21}, \ldots, \ell_n : U_{2n}]}}$$

In order to present the static semantics of Gradual Union Typing, is provided the next definition of Gradual Meet [19], which is extended for records.

Definition 3.14 [Gradual Meet] Let $\sqcap : UTYPE \rightharpoonup UTYPE$ be defined as:

(i) $U \sqcap U = U$

(ii) $? \sqcap U = U \sqcap ? = U$

(iii) $U \sqcap (U_1 \oplus U_2) = (U_1 \oplus U_2) \sqcap U = \begin{cases} U \sqcap U_1 & if\ U \sqcap U_2\ is\ undefined \\ U \sqcap U_2 & if\ U \sqcap U_1\ is\ undefined \\ (U \sqcap U_1) \oplus (U \sqcap U_2) & otherwise \end{cases}$

(iv) $(U_{11} \to U_{12}) \sqcap (U_{21} \to U_{22}) = (U_{11} \sqcap U_{21}) \to (U_{12} \sqcap U_{22})$

(v) $[\ell_1 : U_{11}, \ldots, \ell_n : U_{1n}] \sqcap [\ell_1 : U_{21}, \ldots, \ell_n : U_{2n}] = [\ell_1 : (U_{11} \sqcap U_{21}), \ldots, \ell_n : (U_{1n} \sqcap U_{2n})]$

(vi) $U_1 \sqcap U_2$ is undefined otherwise.

The following definition states the translation of the *equate* from STFL to a function in $\text{GTFL}_\oplus$, in order to preserve typing:

Definition 3.15 [Equate Lifting]

$$\widetilde{equate}(U_1, U_2) = U_1 \sqcap U_2 = \alpha(\{\widehat{T_1} \cap \widehat{T_2} \mid \widehat{T_1} \in \gamma(U_1),\ \widehat{T_2} \in \gamma(U_2)\}) =$$

$$\alpha_\oplus(\{\widetilde{equate_?}(G_1, G_2) \mid G_1 \in \gamma_\oplus(U_1),\ G_2 \in \gamma_\oplus(U_2)\})$$

And the next proposition is needed to prove the gradual meet for $GTFL_\oplus$

Proposition 3.16 $\sqcap = \alpha \circ \widehat{\widetilde{equate}} \circ \gamma$

Proof Proof is made using structural induction and the definition of meet as intersection of sets of sets [19]. The case for records is straightforward as the meet is pointwise. □

4 Conclusion and Future Work

Gradual Union Typing was initially proposed by Toro and Tanter [19], showing properties such as safety and consistency using AGT and Galois connections. Nevertheless, there has been investigated a lot of settings related to gradual typing in order to promote the development of more expressive programming languages. For instance, subtyping by García, Siek and Taha [14,7], Gradual Typing systems and cast insertion procedures by Cimini and Siek [3], the interaction between union types and Gradual Typing [16,1], typestates [8], refinement types [11], information-flow security typing [5], set-theoretic types and subtyping [2], refinement sums for exhaustive and non-exhaustive matches [10], and studies focused on performance at run-time and space management [13,9].

The type setting here provides a static extension using records for the study of Gradual Union Typing, as presented by Toro and Tanter in [19]. This approach contributes to the study of the theoretical perspective for the foundations of programming languages. Adding records to a functional language allows to create new variants of types, using them as basic types predefined in the language, for a richer and more versatile language. Extending the methodology for languages with gradual typing helps to combine the advantages of using both static and dynamic typing, while preserving the properties of consistency and security in it.

A related work from Garcia, Clark and Tanter [7] proposes an extension of a type system with records using subtyping, and formalizing a relation of consistency on gradual typing language, by following the methodology proposed by Siek and Taha [15]. Our approach here differs from theirs as we rely on gradual union types. Here it is already provided a static formal system. Further, the dynamic formal part, demonstrating its static and dynamic semantics properties, is aimed as future work. The main goal is to obtain a gradual system with records using the so-called Threesome Calculus [17] for the dynamic semantics.

References

[1] Giuseppe Castagna and Victor Lanvin. Gradual typing with union and intersection types. *Proceedings of the ACM on Programming Languages*, 1(ICFP):41, 2017.

[2] Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyễn. Set-theoretic types for polymorphic variants. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 378–391, New York, NY, USA, 2016. ACM.

[3] Matteo Cimini and Jeremy G Siek. The gradualizer: a methodology and algorithm for generating gradual type systems. *ACM SIGPLAN Notices*, 51(1):443–455, 2016.

[4] Patrick Cousot and Radhia Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *FPCA*, volume 95, pages 170–181, 1995.

[5] Tim Disney and Cormac Flanagan. Gradual information flow typing. In *International workshop on scripts to programs*, 2011.

[6] Daniel P Friedman, Mitchell Wand, and Christopher Thomas Haynes. *Essentials of programming languages*. MIT press, 2001.

[7] Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 429–442, New York, NY, USA, 2016. ACM.

[8] Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of typestate-oriented programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(4):12, 2014.

[9] David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. *Higher-Order and Symbolic Computation*, 23(2):167, 2010.

[10] Khurram A Jafery and Joshua Dunfield. Sums of uncertainty: Refinements go gradual. *ACM SIGPLAN Notices*, 52(1):804–817, 2017.

[11] Nico Lehmann and Éric Tanter. Gradual refinement types. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 775–788, New York, NY, USA, 2017. ACM.

[12] Benjamin C Pierce and C Benjamin. *Types and programming languages*. MIT press, 2002.

[13] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for typescript. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 167–180, New York, NY, USA, 2015. ACM.

[14] Jeremy Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2007.

[15] Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

[16] Jeremy G Siek and Sam Tobin-Hochstadt. The recursive union of some gradual types. In *A List of Successes That Can Change the World*, pages 388–410. Springer, 2016.

[17] Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 365–376, New York, NY, USA, 2010. ACM.

[18] Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 793–810, New York, NY, USA, 2012. ACM.

[19] Matías Toro and Éric Tanter. A gradual interpretation of union types. In *International Static Analysis Symposium*, pages 382–404. Springer, 2017.