Gradual Verification of Program Properties: A short survey

Raimil Cruz *[1]

[1] *Computer Science Department (DCC), University of Chile, Chile*

Abstract

Gradual typing combines static and dynamic typing safely within a single language. Siek and Taha defined its theory in 2006 to combine the untyped lambda calculus and the simply typed lambda calculus. In the last decade many proposals have developed gradual typing for other language features. This paper surveys some of the most important work on gradual typing and its application to real language design. In particular, we are interested in exploring the extension of gradual typing to gradually checking advanced program properties.

1 Introduction

Statically and dynamically typed languages have well-known advantages and drawbacks making them more appropriate in different scenarios. On the one hand, statically-typed languages like Java, C# and Haskell provide early error detection, are more efficient, provide machine-checked documentation and have better support for tools (e.g auto-completion features, code analysis). On the other hand, dynamically-typed languages are more suitable for rapid prototyping, support more flexible programming idioms, do not report spurious errors, and are easier to learn. The strengths of ones are the weaknesses of the others. There has been a lot of effort in combining the best of both worlds in the same language using different approaches. One of the most studied recently has been *gradual typing* [31].

Gradual typing was introduced by Siek and Taha [31] to name a class of type systems that combine static and dynamic type checking in the same program. Gradual typing gives programmers the control over which discipline is used in each part of the code. Static regions can be safely optimized, providing a "pay-as-you-go" runtime cost of dynamism. It means that a fully annotated program in the gradual language runs as fast as in a statically-typed language. Another fact that makes the gradual typing approach interesting is that it allows gradual

*Funded by grant CONICYT, CONICYT-PCHA/Doctorado Nacional/2014-63140148

evolution from static to dynamic or vice versa. It means that a gradualllly-typed language is a superset of both a static and a dynamic language, and more than that partially-typed programs are supported.

The theory of gradual typing introduces the *unknown type* ? to deal with dynamism. (The unknown type is also named *dyn* or *dynamic* type.) To work with the unknown type, type equality is relaxed to *type consistency*. Type consistency expresses that the unknown type is consistent with any type. The gradual type system accepts or rejects programs based on type consistency. The presence of the unknown type delays type checking to runtime. The runtime semantics of the gradual language is not directly defined over source programs. Instead, to run a gradual program, a translation is done to an intermediate language with *casts*. During translation, casts are inserted where consistency is used. Casts perform dynamic type checking, safeguarding static assumptions.

Gradual typing has received attention in both academia and industry. Its ideas have been applied to real language design such as TypeScript [5], Dart [13], Hack [16] and an extension of C# with the dynamic type [6]. Beyond simple typing, gradual typing has been applied to other typing disciplines such as ownership types [30], type-and-effects [4, 39], security typing [15, 17], annotated type systems [38], typestates [41] and refinement types [36].

In this short survey we describe gradual typing (in Section 2), presenting the original work of Siek and Taha and subsequent work that extends gradual typing to other language features. In Section 3 we highlight some of the most important uses of gradual typing applied to the verification of program properties. Finally in Section 4 we comment on some of the challenges of gradual typing such as performance issues and the search for a unified theory that can make it easier to extend gradual typing to other typing disciplines.

2 Gradual typing

In this section we provide a gentle introduction to gradual typing by examples. After that, we summarize the original paper of Siek and Taha [31] with the theory of gradual typing, and other subsequent relevant work in this research area.

2.1 Gradual typing by examples

In a gradually-typed language, type annotations are optional. The gradual type system uses the information of the typed part of the programs to detect type inconsistency and statically reject programs. In the absence of static type information, errors are detected dynamically during the execution of the program. A gradual system is a superset of a static and dynamic type system, which means that either a fully annotated code or a code with no type annotation at all can be accepted. It also supports partially-annotated programs. This is the most interesting.

Figure 1 presents a program that does not have type annotations. We have a function `f` that adds `2` to its argument. As the argument is not annotated, the `f` function could be applied with an argument of any type. There is another function `h` which applies its argument `g` to `1`. Reading the definition of `h` we can see that there is an implicit intention of receiving a function, but as the parameter `g` is not annotated, this assumption is not statically enforced. The third line is an application of `h` with `f`. After two steps of substitutions the addition is performed, producing `3`.

```
def f(x) = x + 2
def h(g) = g(1)
h(f)
```

Figure 1: Dynamically-checked valid program

In the example of Figure 2, the function `h` applies its argument `g` to `true`. In this case the application of `h` with `f` produces a runtime type error when trying to evaluate `true + 2`. The `+` operator requires its operands to be of type `int`, and the first operand is a `boolean` value.

```
def f(x) = x + 2
def h(g) = g(true)
h(f)
```

Figure 2: Dynamically-checked invalid program (runtime error)

In these previous examples type checking is done dynamically because there is no static type information. A gradual system also supports fully annotated programs. Instead of detecting errors at runtime, the gradual type system uses the type information to report type errors statically. In the following example we have annotated the arguments of functions `f` and `h`. A static validation is done for the expressions `g(1)` and `h(f)`. No type inconsistencies are found and the program executes to `3`.

```
def f(x:int) = x + 2
def h(g:int -> int) = g(1)
h(f)
```

Figure 3: Statically-checked valid program

In the next example (Figure 4), we change the function `h`. Now it applies its parameter to `true`. In this case the parameter `g` is annotated with the type `int` $\rightarrow$ `int` indicating it is a function from `int` to `int`. This program does not

type check because `g` is applied with a `boolean` value. The program is therefore rejected statically.

```
def f(x:int) = x + 2
def h(g:int -> int) = g(true)
h(f)
```

Figure 4: Statically-checked invalid program (compile-time error)

Sound Interoperability Gradual typing goes beyond supporting fully annotated or fully dynamic programs in the same program. It is possible to combine static and dynamic checking. Let us see how gradual typing provides a sound interoperability between both checking disciplines.

In the example of Figure 5 we have a statically-typed function `f`. Function `g` is dynamically typed. The interaction between both worlds takes place when the function `h` is applied with function `g`. After one reduction step we have the expression `f(1)`. Instead of doing another step of reduction directly, for this example, a dynamic check is performed before calling `f`, which protects assumptions made in the static code. This program is well-typed, and evaluates to `3` successfully.

```
def f(x:int) = x + 2
def h(g) = g(1)
h(f)
```

Figure 5: Gradual program, valid

In the next example we changed the definition of the function `h`. Now it applies its argument `g` to `true`. After a reduction step we get the expression `f(true)`. A dynamic check is done and a type inconsistency is detected: we are trying to apply a function that receives an `int` with a `boolean`. For this program, we get a dynamic cast error. Unlike the example in Figure 2, where

```
def f(x:int) = x + 2
def h(g) = g(true)
h(f)
```

Figure 6: Gradual program, invalid (runtime cast error)

the dynamic type error is detected in the body of the function `f` (just before the expression `x + 2` is executed), in this example the type error is detected earlier, specifically when the function `f` is applied in the expression `f(true)`.

The difference is that the type annotation of the argument of function `f` make its body safe. That mean no dynamic error occurs in the body of `f` and that therefore `f` can be compiled efficiently (using an unchecked `+` primitive).

These examples show that is possible to move from a fully dynamically-typed program to a fully statically-checked program. We can get early error detection by adding type annotations. The gradual system detects static type errors if there is enough type information to prove that there exists a static type violation. Otherwise, it defers type checking to runtime.

2.2 Theory

We now present the formal work of Siek and Taha [31], which defines how static and dynamic type checking are combined in a gradually-typed language.

2.2.1 Gradual Typing for functional languages

Siek and Taha published the original gradual typing paper in 2006 [31]. Before that, several programming languages provide gradual typing to some degree such as Cecil [9] or extensions proposed to C#, Java, ML and Scheme. Siek and Taha provide a theoretical foundation for these languages. Other important approaches for combining static and dynamic checking are Quasi-static typing [37], soft typing [8], optional types [7] and hybrid types [24]. Siek and Taha compare gradual typing to those approaches in their paper. The key feature of gradual typing is the fine-grained, sound interoperability it provides.

They present a calculus named $\lambda^{?}_{\rightarrow}$ which extends the simply typed lambda calculus $\lambda^{\rightarrow}$ with the unknown type (?). This gives the desired dynamic flexibility in the type checking. In their calculus the unknown type appears in the absence of type annotations. For instance, the expression $\lambda x.x + 1$, where the argument x does not have an annotated type, corresponds to the expression $\lambda x :?.x + 1$. Siek and Taha introduce the notion of *consistency* to express compatibility between types in the presence of the unknown type.

Type Consistency

$$\frac{}{\tau \sim \tau}\ [\text{CRefl}] \qquad \frac{\tau_1 \rightarrow \tau_1' \quad \tau_2 \rightarrow \tau_2'}{\tau_1 \rightarrow \tau_2 \sim \tau_1' \rightarrow \tau_2'}\ [\text{CFun}]$$

$$\frac{}{\tau \sim ?}\ [\text{CUnR}] \qquad \frac{}{? \sim \tau}\ [\text{CUnL}]$$

Type consistency captures the notion that it is plausible for the unknown type to be any type. The consistency relation is reflexive and symmetric, but not transitive. With transitivity we could prove that all types are consistent, for example `bool` and `int`, using that `bool` $\sim$ `?` and `?` $\sim$ `int`, but a gradual type system rejects statically the use of an `int` for a `bool`. The consistency relation affects how type checking is done. For example in the simply typed lambda

calculus $\lambda^{\rightarrow}$ the application rule requires that the function formal argument type must be equal to the actual argument type:

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ [\text{TApp}]$$

Siek and Taha relax in $\lambda^{?}_{\rightarrow}$ the equality relation between types, changing the above rule for two rules appealing to the consistency relation.

$$\frac{\Gamma \vdash_G e_1 :? \quad \Gamma \vdash_G e_2 : \tau_2}{\Gamma \vdash_G e_1\ e_2 :?}\ [\text{GApp1}]$$

$$\frac{\Gamma \vdash_G e_1 : \tau_{11} \rightarrow \tau_2 \quad \Gamma \vdash_G e_2 : \tau_{12} \quad \tau_{11} \sim \tau_{12}}{\Gamma \vdash_G e_1\ e_2 : \tau_2}\ [\text{GApp2}]$$

These rules allow the gradual type checker to accept programs like `(λx:?. x+1) true`, though this concrete program is going to fail at runtime. The introduced flexibility in the static type rules is checked at runtime. Instead of providing a runtime semantics for the gradual language, Siek and Taha compile gradual source programs to a cast calculus where dynamic type checking is performed via casts. The intermediate language extends $\lambda^{?}_{\rightarrow}$ with a cast expression and a `CastError` term to represent runtime type errors. Values are extended with a tagged value $\langle ? \Leftarrow \tau \rangle\, v$.

$$\begin{array}{llll} \text{(terms)} & e & ::= & \ldots \mid \langle \tau_2 \Leftarrow \tau_1 \rangle\, e \mid \texttt{CastError} \\ \text{(values)} & v & ::= & \ldots \mid \langle ? \Leftarrow \tau \rangle\, v\ (\tau \neq ?) \end{array}$$

In a cast expression $\langle \tau_2 \Leftarrow \tau_1 \rangle\, e$, τ_2 expresses the desired type of the expression e and τ_1 the known type of the expression. The following small-step evaluation rules are the important ones to understand how dynamic type checking is done with casts. The evaluation cast rules here are written with a different syntax than the ones of Siek and Taha but are equivalent. Rule [EC1] compacts two casts with ? in the middle, [EC2] raises an error if both types are not consistent and [EC3] handles successful cast. Rule [EC4] applies for higher-order casts. For such a cast, a wrapper function is created, in which the argument and the result are cast to the expected types.

$$\frac{\tau_1 \neq ?}{\langle \tau_2 \Leftarrow ? \rangle\, \langle ? \Leftarrow \tau_1 \rangle\, v \longmapsto \langle \tau_2 \Leftarrow \tau_1 \rangle}\ [\text{EC1}]$$

$$\frac{\tau_1 \nsim \tau_2}{\langle \tau_2 \Leftarrow \tau_1 \rangle\, v \longmapsto \texttt{CastError}}\ [\text{EC2}]$$

$$\frac{\tau_1 = \tau_2}{\langle \tau_2 \Leftarrow \tau_1 \rangle\, v \longmapsto v}\ [\text{EC3}]$$

$$\frac{\tau_1 \neq \tau_2 \quad \tau_1 = \tau_{11} \rightarrow \tau_{12} \quad \tau_2 = \tau_{21} \rightarrow \tau_{22} \quad \tau_1 \sim \tau_2}{\langle \tau_2 \Leftarrow \tau_1 \rangle\, v \rightarrow \lambda x : \tau_{21}.\, \langle \tau_{22} \Leftarrow \tau_{12} \rangle\, (v\, \langle \tau_{11} \Leftarrow \tau_{21} \rangle\, x)}\ [\text{EC4}]$$

The translation from $\lambda^{?}_{\rightarrow}$ to the intermediate calculus introduces cast when the unknown type appears. For programs that are fully annotated no casts are inserted. This fact is expressed in the rule [CIApp1] and ensures a *"pay as you go"* cost of dynamism.

$$\frac{\Gamma \vdash e_1 \Rightarrow e_1' : \tau \to \tau' \quad \Gamma \vdash e_2 \Rightarrow e_2' : \tau}{\Gamma \vdash e_1\ e_2 \Rightarrow e_1'\ e_2' : \tau'}\ [\text{CIApp1}]$$

$$\frac{\Gamma \vdash e_1 \Rightarrow e_1' : \tau \to \tau' \quad \Gamma \vdash e_2 \Rightarrow e_2' : \tau_2 \quad \tau_2 \neq \tau \quad \tau_2 \sim \tau}{\Gamma \vdash e_1\ e_2 \Rightarrow e_1'\ \langle \tau \Leftarrow \tau_2 \rangle\ e_2' : \tau}\ [\text{CIApp2}]$$

$$\frac{\Gamma \vdash e_1 \Rightarrow e_1' : ? \quad \Gamma \vdash e_2 \Rightarrow e_2' : \tau}{\Gamma \vdash e_1\ e_2 \Rightarrow (\langle \tau \to ? \Leftarrow ? \rangle\ e_1')\ e_2' : ?}\ [\text{CIApp3}]$$

Let us see how cast insertion works, inserting casts on the expressions of the example of Figure 6. The argument `g` of function `h` has an unknown type. In the body of `h`, `g` is cast from unknown (`?`) to `bool → ?` using rule [CIApp3]. In the line 3, the translation process inserts a cast from `int → int` to `?` for argument `f` (rule [CIApp2]):

```
def f(x:int) = x + 2
def h(g:?) = (<bool→? ⇐ ?>g)(true)
h(<? ⇐ int→int>f)
```

The expression $\langle ? \Leftarrow \texttt{int} \to \texttt{int} \rangle$ `f` is already a value so the first evaluation step is a substitution getting the following expression:

```
(<bool→? ⇐ ?><? ⇐ int→int>f)(true)
```

We apply rule [EC1] to combine cast and we get:

```
(<bool→? ⇐ int→int>f)(true)
```

Evaluation of this cast produces `CastError` (rule [EC2]), because `bool →?` is not consistent with `int→int`.

The key property for a typed language is *type safety*, which informally says that if a program p is well typed, then the evaluation of p does not reach an invalid ending state. For example for $\lambda^{\rightarrow}$ the only valid ending state is a value. In gradually-typed languages a new valid ending state is added: `CastError`. Siek and Taha prove indirectly type safety for the gradual language, proving type safety for the intermediate language and proving that the translation from the gradual source language to the intermediate language preserves typing [31].

Siek and Taha also explicitly express the relation with fully annotated type programs and fully unannotated programs. The next propositions make clear that the gradually typed language supersedes both the dynamically and the statically typed languages:

1. *Equivalence to the Simply Typed Lambda Calculus (STLC) for fully annotated terms.* This theorem says that for a fully annotated expression e the

gradual type system gives the same static type to e that the STLC type system gives, and the evaluation of e in both system produces the same value.

2. *Embedding of Dynamically Typed Lambda Calculus (DLTC).* For a term e of the DTLC, the gradual system also accepts its lifted version $\lceil e \rceil$ and gives it the unknown type. Besides, if e reduces to a value v in DTLC, $\lceil e \rceil$ reduces to v in $\lambda^{?}_{\rightarrow}$.

The $\lceil e \rceil$ embeds a term of DTLC into the $\lambda^{?}_{\rightarrow}$. This encoding casts all values to the unknown type.

2.2.2 Gradual typing for objects

After their first gradual typing paper, Siek and Taha published Gradual Typing for Objects [32] in 2007. In this work they extend the statically typed object calculus of Abadi and Cardelli $\text{Ob}^{<:}$ [1] with gradual typing. The $\text{Ob}^{<:}$ calculus has structural subtyping and the main contribution of that paper is to combine subtyping with gradual typing safely.

The consistency relation has to deal with structural object types. Siek and Taha follow the intuition that two types are consistent if their known parts are equal. The following example shows the idea. The notation $[m_1 : s_1, ..., m_n : s_m]$ represents a type for an object where $m : s$ is the name m of the attribute and s its signature.

$$\tau \sim \tau \quad \tau \sim ? \quad ? \sim \tau$$
$$[x : int \rightarrow ?, y : ? \rightarrow bool] \sim [y : ? \rightarrow bool, x : int \rightarrow ?]$$

However for the following types the consistency relation does not hold. In the first case there is an inconsistency in the type of attribute x. In the second case types have different sizes.

$$[x : int \rightarrow int, y : ? \rightarrow bool] \nsim [x : \ bool \rightarrow int, y :? \rightarrow bool]$$
$$[x : int \rightarrow int, y : ? \rightarrow ?] \nsim [y : int \rightarrow int]$$

This consistency relation is formalized in the paper, but is unsurprising.

To combine gradual typing with subtyping is the real challenge. Siek and Taha make a separation between the unknown type (?) with the top of the subtyping relation (e.g. `Object` in Java). The challenge is that subtyping is transitive, while consistency should not be (2.2.1). Therefore, they treat `?` neutral to subtyping, which is expressed with the new rule $? <: ?$:

$$\tau <: \tau \quad ? <: ?$$
$$[m_i : s_i \ ^{i \in 1..n+m}] <: [m_i : s_i \ ^{i \in 1..n}]$$

In the above definition we omit the subtyping rule that allows covariance and contravariance for object types, because it is not relevant to get the intuition

about the role of the unknown type with respect to subtyping. Once the separation between consistency and subtyping is done, they define *consistent subtyping* $\lesssim$. This relation is characterized alternatively as:

1. $\sigma \lesssim \tau$ iff $\sigma <: \tau'$ and $\tau' \sim \tau$ for some τ'
2. $\sigma \lesssim \tau$ iff $\sigma \sim \tau'$ and $\tau' <: \tau$ for some τ'

For example

$$[x : int \to ?, y : \ bool \to bool] \lesssim [x : ? \to int]$$

because there exists the type $[x :? \to int, y : \ bool \to bool]$ for which :

$$[x : int \to ?, y : \ bool \to bool] \sim [x : ? \to int, y : \ bool \to bool]$$
$$[x :? \to int, y : \ bool \to bool] <: [x : ? \to int]$$

The gradual type system is straightforward to derive with the consistent-subtyping relation. The runtime semantics of the gradual calculus is also defined through a translation to an intermediate language with casts named $\mathtt{Ob}^{\langle \cdot \rangle}_{<:}$. A key aspect in the dynamic semantics is the computation of type τ' that makes the consistent-subtyping relation holds between two types. They define a *merge operator* written $\sigma \leftharpoonup \tau$ to calculate τ'. Proofs of type safety are done similar to those for gradual typing for functional languages.

2.2.3 Gradual typing for generics

Generics, or parametric polymorphism, is another language feature for which gradual typing was applied. Ina and Igarashi [23] presented gradual typing for generics. They develop a gradual calculus for generics over a class-based language (with nominal subtyping instead of structural subtyping), similar to Java.

One of the contributions of their work is to provide a flexible consistency relation for parametric types in the presence of the unknown type (noted `dyn`). In a language like Java, the subtying relation for generic types is invariant with respect to type parameters. For instance for the types `List<T1>` and `List<T2>` the subtyping relation only holds if `T1 = T2`. Their consistency relation allows covariance and contravariance with `dyn` as a type parameter, for example, `List<Rectangle>` $\lesssim$ `List<dyn>` and `List<dyn>` $\lesssim$ `List<Rectangle>`. The following assignments are valid.

```
List<dyn> l1 = new List<Rectangle>();
List<Rectangle> l2 = l1;
```

This consistency relation is used in the gradual type system, but it is too permissive for the intermediate language. Let us see the following code:

```
List<dyn> l1 = new List<Rectangle>();
l1.add(new Object());
List<Rectangle> l2 = l1;
l2[0].area();
```

In the above program the third line introduces the type `List<Rectangle>` which is fully known. Note that the method invocation on the fourth line is statically valid (`l2[0]` has type `Rectangle`), so it should not fail at runtime. However, because of the assignment on line 3, `l2[0]` is not a `Rectangle` object, so the static assumption is violated.

In order to address this issue, Ina and Igarashi introduce a more restrictive consistency relation $\prec:$ for casts, which removes `List<dyn>` $\prec:$ `List<Rectangle>`, but still supports `List<Rectangle>` $\prec:$ `List<dyn>`. In that case the expected cast `List<Rectangle>` $\Leftarrow$ `List<dyn>` on the third line is going to fail in the intermediate language, protecting the statically typed regions from performing an unsafe operation.

2.2.4 Refined criteria for gradual typing

Despite the fact that much work has been done in last decade to integrate dynamically and statically typed programs associated to gradual typing, not all satisfy the original intention of gradual typing. In 2015, Siek et al [34] present a stronger criterion to characterize gradual typing precisely.

The *gradual guarantee* is related to a notion of precision on types and terms. For instance, the type $int \to ?$ is more precise that $?$, but less precise than $int \to int$. The gradual guarantee informally says three things:

1. if a gradually typed program is well typed, the same program with less precise type annotations is also well typed.
2. if a gradually typed program evaluates to a value, the same program with less precise type annotations also evaluates to this value.
3. if a gradually typed program evaluates to a value, the same program with more precise type annotations also evaluates to this value or produces a runtime error.

A corollary to the first statement is that if a program is not well typed, adding more type annotations will not make the program well typed. In a gradually-typed language, the gradual guarantee gives to programmers a notion of what they can expect when adding or removing type annotation in a well-typed program.

2.3 Gradual typing vs type inference

Gradual typing and type inference have one thing in common, ie. that programmers do not need to annotate types. The main difference is that type inference requires static type checking. Some work has been done in integrating gradual typing and type inference [33, 18, 28].

In $\lambda^{?}_{\to}$ the expression $\lambda x.x\ x$ is equivalent to $\lambda x :?.x\ x$, because the absence of a type annotation is interpreted as the unknown type. This expression in $\lambda^{?}_{\to}$ has type $? \to ?$. However in the STLC with type inference, the expression $\lambda x.x\ x$ is rejected.

Siek and Vachharajani combine gradual typing with type inference [33]. They define a new calculus $\lambda^{?\alpha}_{\rightarrow}$ which is the result of extending $\lambda^{?}_{\rightarrow}$with type variables. The type inference algorithm is responsible for choosing a solution that does not introduce unnecessary casts. This means that if there are many options for a type, they choose the less informative (that is the one that contains more unknown types).

In this new system a function parameter that is not annotated is implicitly annotated with the dynamic type, however for local variables (defined with a `let` expression) static types are inferred. An expression `let x = e1 in e2`, where the type of `x` is not annotated, is converted to $(\lambda x : \alpha.\, e_2)$ `e2` instead of the classical encoding for this `let` expression : $(\lambda x : ?.\, e_2)$ `e2`. The type variable α in the argument of the function denotes that its type must be inferred. In their system the expression $\lambda x.x\,x$ is well typed, but the expression $\lambda x : \alpha.\, x\,x$ is not well typed because of this distinction between the unknown type and type variables.

García and Cimini [18] recently presented an approach to combine gradual typing and type inference in a gradual implicitly typed language (ITGL). A key aspect in ITGL is that gradual types are not inferred. Implicit types are static types. This means the expression $\lambda x.x\,x$ is not well typed in their system. In order for this expression to be well typed we have to add an unknown type annotation for the function argument: $\lambda x :?.x\,x$. Gradual types in this system only appear with explicit type annotations. García and Cimini prove that their type system accepts the same programs as the one of Siek and Vachharajani, despite the fact that they are conceptually distinct. They claim that their approach is modular and extensible and they extend ITGL to solve an open problem in the work of Siek and Vachharajani, namely to support *let polymorphism* [26].

2.4 Applications in concrete languages

Many industrial languages have adopted the idea of combining static and dynamic type checking. These combinations usually differ from gradual typing proper. Also, some research projects have applied gradual typing to existing languages.

A common fact in industrial gradual typed languages is that designers sacrifice type soundness of the gradual languages for pragmatics reason of performance. Casts, in particular higher-order casts, are costly. Type safety is preserved because the execution environments of these languages are type safe.

- TypeScript is an extension to JavaScript created to be a better choice for building large-scale JavaScript applications [6]. TypeScript is a superset of JavaScript, so every JavaScript program is a TypeScript program. The TypeScript compiler emits JavaScript code. TypeScript implements a form of gradual typing, but in order to support common JavaScript idioms and codebases, the designers of TypeScript sacrificed type soundness. This means that well typed TypeScript program can produce errors in a

JavaScript execution environment. Rastogi et al. [29] present a proposal for achieving safe and efficient gradual typing for TypeScript.

- Dart is programming language of Google that was initially created for programming web applications [13]. Type annotations in Dart are optional. Dart does not report static type errors, but warning. This means it is possible to run a Dart program even though it is not well typed. One interesting design aspect of Dart is that it can be run in two modes: *checked* and *production*. In checked mode, which is intended to be used during development, type annotations are used to do dynamic type checking during the execution just like gradual typing. However in production mode no type checking is performed and one can get `MessageNotUndertood` errors of the Dart Virtual Machine just like in TypeScript. This makes Dart unsound too, but type safety is ensured by the Dart Virtual Machine. Checked mode follows the theory of gradual typing except that static error are reported as warning. Production mode, where type annotations do not affect the semantics of the language, follows the ideas of optional typing [7] of Bracha.

- Dynamic in C# : A dynamic type was added to C# to support interaction with dynamic languages on the .NET platform. Bierman et al formalize this extension [6]. C# programs are executed over the Dynamic Language Runtime, which enables efficient implementation of dynamic programming languages. In this extension the flexibility of the dynamic type is reduced to achieve an efficient implementation. For example generic type parameters are invariant with respect to the dynamic type.

- Hack is a Facebook's programming language that extends PHP with type annotations, generics, first-class functions and other modern programming language features [16]. Hack supports most of PHP features but there are some features that are not supported such as mixing HTML with Hack code. Hack programs are executed on the Hip Hop Virtual Machine (HHVM). Hack has its own type checker, which is useful as a tool for the programmer to get static type errors before running a Hack program. The type checker has different modes, but the most interesting is the *partial mode*. In partial mode the type checker reports errors based on type information, but it does not demand the code to be fully annotated. However, in any case, it is possible to run a Hack program with type errors and in this case we can get an error in the HHVM, similarly to Dart in checked mode.

- Gradual Typing for Python. Reticulated Python is a system for experimenting with gradually-typed dialects of Python [40]. Reticulated Python is a source-to-source translator that implements gradual typing on top of Python 3. The authors discuss different dynamic semantics to deal with mutable objects (references) and the application of gradual typing to third-party Python programs.

- Gradual Typing for Smalltalk. Allende et al report on the design, implementation and application of Gradualtalk, a gradually-typed Smalltalk meant to enable incremental typing of existing programs [2]. Any Smalltalk program is a valid Gradualtalk program and type annotations can be added selectively per expression. For a Gradualtalk program, the programmer has the option of using a gradual typing approach or an optional typing approach.

3 Gradual Verification

Gradual typing ideas have been applied to advanced typing disciplines such as effects [4, 39], ownership types [10], and security typing [15, 17] among others. The original motivation of gradual typing was to combine dynamic and static checking. In most systems we present in this section, the motivation is the combination of a system that is unaware of the advanced typing discipline with another that is aware of the typing discipline.

3.1 Gradual effects

Effect systems allow to track side effects that happen in a program. Different kinds of effect systems have been developed for different effect domains, such as I/O and exceptions. To abstract from specific effect disciplines, Marino and Millstein developed a generic framework [25]. Their framework allows to express effect systems seeing a specific effect as a *privilege* that can be *granted* and *checked*. For example, for Java checked exceptions, the *privilege* is "to throw an exception", a `try` block is the one that *grants* the exception privilege and the `throw` statement is the one that *checks* if it is valid to use the privilege of throwing an exception in a certain context. In Marino and Millstein's framework, function types are annotated with the effect they may produce when applied. For example the following function has type $\texttt{int} \xrightarrow{IO} \texttt{int}$ indicating that it produces the `IO` effect.

```
int add1(int x){
 print("hello!");
 return x+1;
}
```

In Marino and Millstein's framework each expression is checked under a privilege set that represents the allowed privileges the expression can produce during its execution in this context.

Bañados et al. [4] develop a gradual effect calculus based on Marino and Millstein's framework. They introduce ¿ to denote statically unknown effects. One of the most important contributions of their work is to give a meaning to unknown effect and consistent privilege sets using ideas of abstract interpretation [12]. The unknown effect ¿ stands for any number of effects: zero or many. A consistent privilege set is a set of privileges that can include the unknown

effect. For example for a mutable state effect discipline, privilege sets Φ and consistent sets Ξ are defined as follow:

$$\Phi \in \texttt{PrivSet} = \mathcal{P}(\{\texttt{read, write, allow}\})$$
$$\Xi \in \texttt{CPrivSet} = \mathcal{P}(\{\texttt{read, write, allow, ¿}\})$$

For instance $\Xi = \{\texttt{read, ¿}\}$ represents all privileges sets that include at least the `read` privilege. Their work provides some insight on how to derive the gradual static semantics in more sophisticated typing disciplines, based on interpreting the gradual concept X as sets of possible static X. (We come back to this point in Section 4.3.) They provide the runtime semantics for the gradual calculus though a translation to an intermediate cast calculus. The cast calculus is unusual in that it captures dynamic effect checking and runtime adjustment of privilege sets.

Toro and Tanter [39] present an implementation of gradual effects for Scala. They extend gradual effects with effect polymorphism and develop a theory of gradual effect polymorphism. They introduce the notion of customizable effect disciplines through a domain specific language called EffScript. In EffScript, programmers define effects and can specify which Scala functions produce which effects. For example it is possible to express that `system.out.print*` Scala functions produce I/O effects. Toro and Tanter's work allows to evolve from a Scala program that does not track effects to a program where a defined effect discipline is gradually checked.

3.2 Gradual ownership types

Type systems for ownership in object-oriented languages provide a declarative way to statically enforce a notion of object encapsulation [11, 10]. Object ownership ensures that objects cannot escape from the scope of the object or collection of objects that own them, and prevents unauthorized references to object fields. In ownership types, classes have ownership parameters. All classes have at least the `owner` parameter that represents the owner of the current instance. It is possible to specify more ownership parameters, to cover advanced ownership scenarios, but they are not necessary to understand the intuition of ownership types. For example the class `Square` owns an instances of `Point p`, which is expressed by passing the owner `this` to the owner parameter of class `Point`. The `world` owner in `Color<world> c` expresses that `c` is shared. The use of the `owner` parameter in the instance field `size` indicates that the field is owned by the owner of the `Square` instance:

```
class Square<owner>{
  Point<this> p;
  Size<owner> size;
  Color<world> c;
  Square(){
    p1 = new Point<this>(0,0);
    size = new Size<owner>(1,1);
```

```
    c  = new Color<world>(0,255,0);
  }
}
```

In the following program, the assignment `s.c = new Color<world>(255,0,0)` is accepted because `c` is not owned by the `Square` instance `s`. However `s.p.getX()` fails because it is accessing the `p` field, which is owned by `s`.

```
class Program{
  void main(){
    Square<world> s = new Square<world>();
    s.c = new Color<world>(255,0,0);//accepted
       because field c is not owned by Square, it is
       shared
    int x = s.p.getX(); //fails, direct access of p
       (outside Square) and p is owned by Square.
    Size<world> size = new Square<world>();
    s.size = size; //accepted because world is the
       owner of field ``size'' (since it is the owner
       of ``s'')
  }
}
```

One of the obstacles for adopting ownership types is the verbosity and rigidity of the typing discipline. Sergey and Clarke applied gradual typing to ownership as a way of migrating from ownership-unaware to ownership-annotated code [30]. They introduce the concept of an *unknown owner*. The absence of an owner annotation is interpreted as an unknown owner. For example `Point p1` is equivalent to `Point<?> p1`. Two ownership types are consistent if their known parts are equals. For example `C<owner,?,o2>` $\sim$ `C<?,o1,o2>`. They adapt the consistent-subtyping relation of Siek and Taha [32] to ownership types. They also provide the runtime semantics of the gradual language via a translation to a cast calculus.

3.3 Gradual security typing

The idea of providing gradual verification of program properties has been explored in security typing too. Security typing allows programmers to reason about the flow of sensitive data in their applications [42]. Security type systems usually ensure *non-interference*: if we run two programs that only differ in private data, they will not produce different public results. In other words, non-interference captures confidentiality, i.e. that private data is not accidentally leaked. Types are annotated with security labels to indicate the degree of security of the data. For example we can think of a scenario where there are public and private data and we have a security lattice with two labels `L` and `H` to indicate low and high confidentiality respectively. Security labels form a partial order to indicate the degree of security: `L` $\preceq$ `H`. We can express, for

example, that a credit card number is private data by assigning it the security type $\texttt{int}^{\texttt{H}}$. For instance, in the following program a private credit card number is passed to a function that expect a public data. The function application `sendToInternet(creditCard)` is rejected because it could expose the credit card number.

```
int^H creditCard = ...;
sendToInternet(int^L v){...}
sendToInternet(creditCard); //rejected
```

However, it is possible to pass public data to private channels, because it does not affect the confidentiality of the data.

Disney and Flanagan [15] present a calculus with explicit casts, which allows the migration of code that does not have security types. They do not use the notion of unknown security labels, and hence no notion of consistency between security types is defined. Instead, data that does not have security label annotation is considered as public (`top` in the security lattice) and programmers write explicit down casts from types without security label to types with security labels.

Following this direction of gradual verification of security, Fennell and Thiemman extend the work of Disney and Flanagan taking into account references [17]. The addition of references introduces significant complications, because the heap can serve as an indirect way of violating confidentiality. They use the notion of unknown security label to get dynamism in some scenarios where the security typing discipline is too rigid. The unknown security label is allowed in type annotation, but not in security values annotation. This fact reflects that a value at runtime has a known security label. In the lattice the unknown security label is also top: $\mathrm{H} \preceq ?$. They define a compatibility relation $\sim$ for gradual security types that holds when both types have the same structure but differ in the annotated security label. For example $int^H \sim int^L$. Their source language has cast expressions like the one of Disney and Flanagan [15]. A cast $int^L \Leftarrow int^H$ is statically accepted but dynamically rejected.

3.4 Gradual annotated type systems

Thiemann and Fennell present a generic gradual calculus for annotated type systems [38]. An example of annotated types are numbers annotated with dimension (e.g velocity, time). The following function type checks in a language with annotated dimension.

```
fun eta(dist:float[m], vel:float[m/s]):float[s] =
  dist / vel
```

However the function `eta_bad` does not type check because the `-` operator requires the same dimension on both arguments.

```
fun eta_bad(dist:float[m],vel:float[m/s]):float[s] =
  dist - vel
```

They show that annotated type systems, where the annotation is restricted to base types, can be gradualized by applying a simple procedure. They start with a simply-typed lambda calculus extended with primitive operations, generic base type annotations and value annotations. The calculus is also parametrized by a set of annotation operations, such as `m/(m/s) = s`. To get the gradual calculus, they extend type annotations and value annotations with `?` and they lift operations on annotations of static types to operations on gradual annotations of static types. All operations on annotations with any ? argument gives as result ?. For example `m/? = ?` and `?/m = ?`. They also extend expressions with explicit casts between types. The typing rule for cast has a notion of compatibility between annotations to deal with the unknown annotation.

They do not provide a translation to an intermediate cast calculus, because their gradual source calculus has explicit casts. The runtime semantics is directly defined for the gradual source calculus, but programmers must introduce casts manually.

3.5 Gradual typestate

Another type discipline where gradual typing has been applied is *typestate* [35]. Typestates reflect how the legal operations on imperative objects can change at runtime as their internal state changes. For example an open file cannot be opened again. In a typestate-oriented programming language, there are static constructions to indicate how the state of an object changes during the object life. For example we can indicate that the `close` operation on a `File` object takes the object from the "open" state and leaves it in the "closed" state. A typestate checker is in charge of ensuring that objects are used in accordance with their state at any given point in a computation. Type safety for this calculus ensures that no invalid operation is performed on objects.

Wolff et al developed a gradual version of typestate [41, 20]. The gradual language allows interaction with dynamic types, so the following programa is statically accepted:

```
File f = new File();
f.open();
f.close();
Dyn d = f;
d.read();
```

For this case a runtime check ensures that `d` is in a valid state for the `read` operation. The gradual source language is also translated to a cast calculus where runtime checks take place.

4 Challenges and Conclusion

We have surveyed some of the most important pieces of work in the area of gradual typing with an inclination towards the study of gradual typing ideas

applied to verification of program properties such as: effects, ownership types, security types and typestates. Beyond the large amount of work in the last decade, gradual typing research is far from be over. It is still an active research area because there are many challenges and open problems. In this section we discuss some of them.

4.1 Resolving performance issues

Even though gradual typing is a valuable theory to integrate static and dynamic type checking, the sound execution of gradual programs presents some performance issues in practice. One of them is associated with executing higher-order casts (or casts between function types). A higher-order cast evaluates to a function wrapper that ensures that arguments and return values have the expected types. Wrapper functions affect runtime performance. Allende et al. [3] present two approaches to reduce the cost of higher-order casts and validate them on Gradualtalk [2]. The first approach reduces the expressiveness of gradual system by restricting certain interaction between typed parts and untyped parts. The second approach tries to predict when a value goes from typed code to untyped code and then back to the same original type or a subtype. In this case the wrapper function can be avoided. Rastogi et al. [28] use type inference to eliminate unnecessary casts.

Herman et al. present an implementation strategy that compact casts, and in this way reduce the space consumption of gradual programs [22]. Instead of using casts they use *coercions* [21]. Coercions are reducible. This work is presented for the gradually typed lambda calculus $\lambda^{?}_{\rightarrow}$, so it would be interesting to see what happens when it is applied to more sophisticated type systems, for example, with recursive types and polymorphic types.

4.2 Application to other typing disciplines

Despite previous experience in gradualizing typing disciplines, it is always a challenge to apply gradual typing to a new discipline. One of the key points is to determine what the unknown type information means and how static typing rules are lifted to their gradual counterpart. But what is more challenging is to provide a proper runtime semantics to evaluate a program in the presence of unknown type information. The more complex the typing discipline is, the more challenging the gradual system. Some examples of complex type disciplines are: linear types [27], dependent types [27] and session types [14].

4.3 Unifying theory and principles (AGT)

As we have described here, applying the gradual typing ideas to a new discipline requires a certain insight to get the right static and dynamic semantics of the gradual language in presence of unknown. Most proposals define an intermediate calculus to execute gradual programs. However some foundational questions remain: what is the minimal cast calculus we need to execute a specific gradual

program? Is there a way of getting this intermediate language by construction? How we can provide a runtime semantics for the gradual calculus instead of compiling it to an intermediate language?

Some advances has recently been done in that direction. One of the most promising is the work of Garcia et al [19], called *Abstracting Gradual Typing (AGT)*. They present an abstract framework to derive the gradual language from the statically typed language. Their framework is based on abstract interpretation, specifically in Galois connections [12]. This work extends the ideas first explored by Bañados et al. [4] to derive the gradual effect type system.

The AGT methodology starts with the static type system, where rules have explicit side conditions. Then, the language designer provides a meaning for the unknown type through a concretization function (γ)from gradual types to sets of static types. One must then find the abstraction function (α) from sets of static types to gradual types such as α and γ form a Galois connection. With these functions, AGT systematically lifts predicates and functions used in side condition of typing rules to their gradual versions. For example the consistency relation is the result of lifting the static type equality. The gradual type system is obtained by changing the static predicates and functions with their gradual versions.

Another interesting contribution of this work is that for the first time a direct runtime semantics for the gradual source language is provided. They use implicit *evidence* of the typing derivations for generating dynamic checks. For example in the typing derivation of $\texttt{int} \to ? \sim ? \to \texttt{bool}$ there is an implicit information about the type that makes the consistency relation holds, which is $\texttt{int} \to \texttt{bool}$. They combine evidence for each reduction step. If it is not possible to combine evidence then an error is reported. Despite this contribution, their runtime semantics does not provide "pas as you go" cost for dynamism, because all term carry evidence. To provide an efficient version for the runtime semantics is one of the challenges of future work on AGT.

Abstracting Gradual Typing opens a lot of perspectives, from validating previous gradual languages designs, to deriving new gradual verification systems.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1996.

[2] E. Allende, O. Callaú, J. Fabry, É. Tanter, and M. Denker. Gradual typing for Smalltalk. *Sci. Comput. Programming*, 96(1):52–69, Dec. 2014.

[3] E. Allende, J. Fabry, R. Garcia, and É. Tanter. Confined gradual typing. In *Proceedings of the 29th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2014)*, pages 251–270, Portland, OR, USA, Oct. 2014.

[4] F. Bañados, R. Garcia, and É. Tanter. A theory of gradual effect systems. In *Proceedings of the 19th ACM SIGPLAN Conference on Functional Programming (ICFP 2014)*, pages 283–295, Gothenburg, Sweden, Sept. 2014.

[5] G. Bierman, M. Abadi, and M. Torgersen. Understanding TypeScript. In R. Jones, editor, *Proceedings of the 28th European Conference on Object-oriented Programming (ECOOP 2014)*, volume 8586, pages 257–281, Uppsala, Sweden, July 2014.

[6] G. Bierman, E. Meijer, and M. Torgersen. Adding dynamic types to C$^{\#}$. In T. D'Hondt, editor, *Proceedings of the 24th European Conference on Object-oriented Programming (ECOOP 2010)*, number 6183, pages 76–100, Maribor, Slovenia, June 2010.

[7] G. Bracha. Pluggable type systems. In *Workshop on Revival of Dynamic Languages*, 2004.

[8] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, pages 278–292, Toronto, Ontario, Canada, 1991.

[9] C. Chambers. The cecil language, specification and rationale. Technical report, University of Washington, Department of Computer Science and Engineering, 1993.

[10] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. 37(11):292–310, Nov. 2002.

[11] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 48–64, New York, NY, USA, 1998. ACM.

[12] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages (POPL 77)*, pages 238–252, Los Angeles, CA, USA, Jan. 1977.

[13] Dart Team. Dart programming language specification, May 2013. Version 0.41.

[14] M. Dezani-Ciancaglini and U. De'Liguoro. Sessions and session types: An overview. In *Proceedings of the 6th International Conference on Web Services and Formal Methods*, WS-FM'09, pages 1–28, Berlin, Heidelberg, 2010. Springer-Verlag.

[15] T. Disney and C. Flanagan. Gradual information flow typing. In *International Workshop on Scripts to Programs*, 2011.

[16] Facebook Inc. Hack programming language specification, Feb. 2015.

[17] L. Fennell and P. Thiemann. Gradual security typing with references. In *Proceedings of the 26th Computer Security Foundations Symposium (CSF)*, pages 224–239, June 2013.

[18] R. Garcia and M. Cimini. Principal type schemes for gradual programs. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*, pages 303–315, Mumbai, India, Jan. 2015.

[19] R. Garcia, A. M. Clark, and É. Tanter. Abstracting gradual typing. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, St Petersburg, FL, USA, Jan. 2016. To appear.

[20] R. Garcia, É. Tanter, R. Wolff, and J. Aldrich. Foundations of typestate-oriented programming. *ACM Trans. Prog. Lang. Syst.*, 36(4):12:1–12:44, Oct. 2014.

[21] F. Henglein. Dynamic typing: syntax and proof theory. *Sci. Comput. Programming*, 22(3):197–230, June 1994.

[22] D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. 23(2):167–189, June 2010.

[23] L. Ina and A. Igarashi. Gradual typing for generics. In *Proceedings of the 26th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2011)*, pages 609–624, Portland, Oregon, USA, Oct. 2011.

[24] K. Knowles and C. Flanagan. Hybrid type checking. *ACM Trans. Prog. Lang. Syst.*, 32(2):Article n.6, Jan. 2010.

[25] D. Marino and T. Millstein. A generic type-and-effect system. In *Proceedings of the ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 39–50, 2009.

[26] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[27] B. C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. Cambridge, MA, USA, 2005.

[28] A. Rastogi, A. Chaudhuri, and B. Hosmer. The ins and outs of gradual type inference. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*, pages 481–494, Philadelphia, USA, Jan. 2012.

[29] A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris. Safe & efficient gradual typing for typescript. 50(1):167–180, Jan. 2015.

[30] I. Sergey and D. Clarke. Gradual ownership types. In H. Seidl, editor, *Proceedings of the 21st European Symposium on Programming Languages and Systems (ESOP 2012)*, volume 7211, pages 579–599, Tallinn, Estonia, 2012.

[31] J. Siek and W. Taha. Gradual typing for functional languages. In *Proceedings of the Scheme and Functional Programming Workshop*, pages 81–92, Sept. 2006.

[32] J. Siek and W. Taha. Gradual typing for objects. In E. Ernst, editor, *Proceedings of the 21st European Conference on Object-oriented Programming (ECOOP 2007)*, number 4609, pages 2–27, Berlin, Germany, July 2007.

[33] J. G. Siek and M. Vachharajani. Gradual typing with unification-based inference. In *Proceedings of the 4th ACM Dynamic Languages Symposium (DLS 2008)*, pages 7:1–7:12, Paphos, Cyprus, July 2008.

[34] J. G. Siek, M. M. Vitousek, M. Cimini, and J. T. Boyland. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*, pages 274–293.

[35] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. 12(1):157–171, 1986.

[36] É. Tanter and N. Tabareau. Gradual certified programming in Coq. In *Proceedings of the 11th ACM Dynamic Languages Symposium (DLS 2015)*, pages 26–40, Pittsburgh, PA, USA, Oct. 2015.

[37] S. Thatte. Quasi-static typing. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL 90)*, pages 367–381, San Francisco, CA, United States, Jan. 1990.

[38] P. Thiemann and L. Fennell. Gradual typing for annotated type systems. In Z. Shao, editor, *Proceedings of the 23rd European Symposium on Programming Languages and Systems (ESOP 2014)*, volume 8410, pages 47–66, Grenoble, France, 2014.

[39] M. Toro and É. Tanter. Customizable gradual polymorphic effects for Scala. In *Proceedings of the 30th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2015)*, pages 935–953, Pittsburgh, PA, USA, Oct. 2015.

[40] M. M. Vitousek, A. M. Kent, J. G. Siek, and J. Baker. Design and evaluation of gradual typing for python. In *Proceedings of the 10th ACM Dynamic Languages Symposium (DLS 2014)*, pages 45–56, Portland, OR, USA, Oct. 2014. , 50(2).

[41] R. Wolff, R. Garcia, É. Tanter, and J. Aldrich. Gradual typestate. In M. Mezini, editor, *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP 2011)*, volume 6813, pages 459–483, Lancaster, UK, July 2011.

[42] S. Zdancewic. *Programming Languages for Information Security.* PhD thesis, Cornell University, Aug. 2002.