Gradual Typing and The Gradually Typed Lambda Calculus

Stacey Tay
National University of Singapore
stacey@u.nus.edu

ABSTRACT

This report presents an introduction to gradual typing and describes Siek and Taha's Gradually Typed Lambda Calculus (GTLC) [8]. In the process, we discuss the motivations for gradual typing and previous and current work that have been done to integrate dynamic and static type systems. We then look into the details of the GTLC, including its design goals and its gradual type system. In Section 3, we discuss a formal criteria for gradual typing [10] and in Section 4, we put forward a simple implementation of a gradually typed Scheme in OCaml. Before we conclude, we look into some of the progress that have been made since Siek and Taha's original paper in 2006 and the present challenges facing gradual typing.

1 INTRODUCTION

1.1 Motivation

It is not uncommon for applications to start out as scripts and slowly evolve to become large programs [13]. These scripts are frequently written in a dynamically typed language, which are often more expressive and easier to prototype with than their statically typed counterparts. However, as programs grow, they become more complex and difficult to maintain. Changing a part of a large application requires the programmer to understand what that fragment of code is doing, and figure out the types of values that are at play. This is when static typing comes in helpful. Static typing provides guarantees on components and allows early error detection. After all, "well-typed programs can't go wrong" [5]. As we will see, *gradual typing* gives programming languages the expressiveness of a dynamically typed language while providing strong safety guarantees, as needed, through type annotations.

1.2 The Benefits of Having Both

The two typing systems, dynamic and static, have their own complementary strengths.

Dynamic typing allows

1. **More expressive programming idioms**. Sometimes the most elegant expression to a problem does not typecheck.
2. **Rapid prototyping**.
3. **A lower learning curve**. For a beginner programmer, learning a type system adds a significant barrier [7].

Static typing aids

1. **Safety**. Having well-typed programs eliminates a whole class of bugs.
2. **Efficiency**. Types support a compiler in generating efficient code.
3. **Documentation**. Types serve as machine-checked documentation as it encodes design information into source code.

Gradual typing potentially allows for the best of both worlds, that of dynamic and static typing. As a result, there has been a growing number of languages—some created before the theory of gradual typing was presented [10]—that provides both a combination of dynamic and static typing. Examples include Dart, TypeScript [2], and Hack [15]. However, not all of these languages support the convenient evolution of code between the two typing disciplines, as described in Siek and Taha's formalization of gradual typing in Section 3.

```
def greeting(name):
    return 'Hello ' + name

def greeting(name: str) -> str:
    return 'Hello ' + name
```

Figure 1: An example of un-annotated and annotated code from Reticulated Python [16], a gradually typed dialect of Python with identical syntax as type hints in Python 3.

1.3 Related Work

Before diving into Siek and Taha's work on gradual typing, we note the other lines of research that have been taken to combine dynamic and static typing. We briefly discuss each of these approaches and their relative strengths and weaknesses.

1.3.1 Dynamic & Typecase. The idea here is to add a `Dynamic` type to a statically typed language so that the programmer can conveniently deal with data whose type cannot be determined at compile time, for example, input from the outside world [1]. However, this does not allow for programming in a dynamically typed style, as the programmer needs to insert coercions to and from type `Dynamic` [10].

1.3.2 Soft Typing. This approach applies static analysis to untyped programs to improve program performance. It was designed to not prevent programmers from running their programs and does not statically catch type errors [10]. Soft typing's weaknesses are that it infers confusing, large types for seemingly simple expressions and is hard for the average programmer to use effectively [14].

1.3.3 Interoperating a Dynamically and Statically Typed Language. Gray et al. presented an interoperable implementation of Java and Scheme [4]. While this allows the programmer to use both dynamically and statically typed languages in a single program, it is more similar to foreign function interfaces, and it does not allow for an easy evolution from dynamically typed to statically typed code.

1.3.4 Types for Dynamic Languages. Create a static type system for a dynamically typed language. Tobin-Hochstadt and Felleisen designed Typed Racket [14], which was built on top of their earlier work on interlanguage migration [13]. This approach allows migrating a program on a per-module basis, but does not allow for a more granular strategy. In particular, it is much less granular than Siek and Taha's model that we focus on in this report.

1.3.5 Quasi-static typing. Thatte's theory of Quasi-static Typing is similar to gradual typing, but uses the standard subtyping relation <: with a top type Ω to represent the dynamic type. It also includes the usual subsumption rule. In Quasi-static typing, the following program

((λ (x) x) #t) ;; works since boolean <: Ω

is allowed. It is type checked by up-casting the value #t of type boolean to be Ω. However, Quasi-static typing also includes the rule

$$\text{QApp}\ \frac{\Gamma \vdash e_1 : \sigma \to \sigma' \qquad \Gamma \vdash e_2 : \tau \qquad \sigma <: \tau}{\Gamma \vdash (e_1 e_2) : \sigma'}$$

which combined with the subsumption rule allows a program such as

((λ (x : number) x) #t)

to pass the type checker. This is because the subsumption rule allows #t to be implicitly cast to Ω and then the above rule for application implicitly casts Ω down to a number. To prevent "stupid casts" such as these, Thatte included a second phase to the type system called plausibility checking [8]. While it did catch these errors, it still failed to catch all the type errors in statically typed code, which Siek and Taha's gradual typing system does [8].

2 THE GRADUALLY TYPED LAMBDA CALCULUS

2.1 Design Goals

When considering the GLTC, Siek et al. had three goals in mind. These three goals will later form the formalized criteria for gradual typing in *Refined Criteria for Gradual Typing* [10], which we discuss in Section 3.

2.1.1 Gradual includes both fully dynamic and fully static. The GLTC should be a superset of both the STLC and the (Dynamically Typed) Lambda Calculus (DTLC). A program of the GLTC, when fully annotated (without ⋆ in it), should behave the same as in the STLC, and when un-annotated, the same as the DTLC.

2.1.2 Sound interoperability. We would like partially typed code to be safe. To do this, the runtime system in gradual typing casts values as they flow between dynamically and statically typed code to protect the static typing assumptions.

2.1.3 Gradual evolution of code. Programmers should be able to add and remove annotations in their program without any unexpected behavior in their programs. Their program, if annotated, should type check if well-typed and should behave in the same manner if its annotations are removed. The latter property is the basis of the *gradual guarantee*, that if a gradually typed program is well typed, then removing type annotations always produces a program that is still well typed [10]. This allows the gradual evolution of code from being untyped to typed, assuming correct type annotations are added.

2.2 Syntax and Type System

With the design goals above in mind, we now take a look at the Gradually Typed Lambda Calculus (GTLC). The GTLC is the Simply Typed Lambda Calculus (STLC) extended with a type ⋆ to represent dynamic types [8]. In the GTLC, the job of the type system is to reject programs that have inconsistencies in known parts of types. For example, the program

((λ (x : int) x) #t)

should be rejected since the type of #t is not consistent with the type of parameter x, boolean is not consistent with int. However, the program

((λ (x) x) #t)

should be accepted by the type system since there is no type annotation and so the expression is not statically type checked, and the type error will be caught at run-time, as in the case of a dynamically typed language.

$$\begin{aligned}
\text{Variables } & x \in \mathbb{X} \\
\text{Ground Types } & \gamma \in \mathbb{G} \\
\text{Constants } & c \in \mathbb{C} \\
\text{Types } & \tau ::= \gamma \mid \star \mid \tau \to \tau \\
\text{Expressions } & e ::= c \mid x \mid \lambda x : \tau . e \mid e e \\
& \lambda x . e \equiv \lambda x : \star . e
\end{aligned}$$

Figure 2: Syntax of the Gradually Typed Lambda Calculus.

Next, we consider first class procedures. Below, we have a map procedure that applies a function to each element in a list.

map : (int ⟶ int) ∗ int list ⟶ int list
(map (λ (x) x) (list 1 2 3))

The map procedure expects to receive a first argument that has a type int ⟶ int, but the argument (λ (x) x) has the type ⋆ ⟶ ⋆. We would like the type system to accept this since we know that expression would be well typed in STLC. To do so, Siek and Taha uses the intuition that known portions of two types should be equal and unknown portions are ignored [8]. This is similar to the mathematics of partial functions, where two partial functions are *consistent* when every element that is in the domain of both functions is mapped to the same result.

Hence, in gradual typing, instead of using type equality, Siek and Taha defines the *consistency relation* [8], as shown in Figure 3, to determine if two types are consistent with each other.

$$\text{CRefl}\ \tau \sim \tau \qquad \text{CUnR}\ \tau \sim \star \qquad \text{CUnL}\ \star \sim \tau$$

$$\text{CFun}\ \frac{\sigma_1 \sim \tau_1 \qquad \sigma_2 \sim \tau_2}{\sigma_1 \to \sigma_2 \sim \tau_1 \to \tau_2}$$

Figure 3: Type consistency axioms.

The consistency relation is reflexive and symmetric. It is similar to the subtyping relation, but unlike the latter, it is not transitive.

$$\text{GVar}\ \frac{\Gamma x = \lfloor \tau \rfloor}{\Gamma \vdash x : \tau}$$

$$\text{GConst}\ \frac{\Delta c = \tau}{\Gamma \vdash c : \tau}$$

$$\text{GLam}\ \frac{\Gamma(x \mapsto \sigma) \vdash e : \tau}{\Gamma \vdash \lambda x : \sigma . e : \sigma \to \tau}$$

$$\text{GApp1}\ \frac{\Gamma \vdash e_1 : \star \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \star}$$

$$\text{GApp2}\ \frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_2 \sim \tau}{\Gamma \vdash e_1 e_2 : \tau'}$$

Figure 4: A Gradual Type System. Note that ⋆ is used to denote the dynamic type. From *Gradual Typing for Functional Languages* [8].

The consistency relation allows the gradual type system, as shown in Figure 4, to be more liberal with the dynamic type ⋆ and helps satisfy the design goals mentioned in Section 2.1. In the gradual type system shown, the environment Γ is a function from variables to optional types ($\lfloor \tau \rfloor$ or $\bot$). The type system is parameterized on a signature Δ that assigns types to constants. The rules for variables, constants, and functions are standard. The first rule for function application GApp1 handles the case when the function type is unknown. In this case, the argument may have any type and the resulting type of the application is unknown, ⋆. The second case for function application GApp2 handles the case when the function type is known and the argument's type is consistent with the function's argument's type.

2.3 Run-time Semantics and Cast Calculus

To ensure that statically typed portions in gradually typed programs are safe, a run-time semantics with explicit casts is used. The explicit casts have the syntactic form $\langle \tau \rangle e$, where τ is the target type. When e evaluates to v, the cast will check that the type of v is consistent with τ. If not consistent, a CastError will be raised. The cast insertion judgment, defined in Figure 5 has the form $\Gamma \vdash e \Rightarrow e' : \tau$. Siek and Taha proves that the cast insertion rules combined with the evaluation rules in their paper preserves type safety [8].

In later versions of GTLC, Siek et al. incorporates a modified version of the Blame Calculus from Wadler and Findler's *Well typed programs can't be blamed* [17] to track and assign blame to relevant portions of the code that triggers a CastError during run-time. This also allows programmers to detect which parts of a gradually typed program might have unsafe casts during compile time [10].

3 GRADUAL TYPING CRITERIA

Since Siek and Taha's 2006 paper on gradual typing, there has been an increased interest in the integration of dynamic and static typing. These combined systems are often termed gradual typing. However, Siek et al. claim that not all of them fulfill the original goal of enabling convenient evolution of code between the two disciplines. To

$$\text{CVar}\ \frac{\Gamma x = \lfloor \tau \rfloor}{\Gamma \vdash x \Rightarrow x : \tau}$$

$$\text{CConst}\ \frac{\Delta c = \tau}{\Gamma \vdash c \Rightarrow c : \tau}$$

$$\text{CLam}\ \frac{\Gamma(x \mapsto \sigma) \vdash e \Rightarrow e' : \tau}{\Gamma \vdash \lambda x : \sigma . e \Rightarrow \lambda x : \sigma . e' : \sigma \to \tau}$$

$$\text{CApp1}\ \frac{\Gamma \vdash e_1 \Rightarrow e_1' : \star \qquad \Gamma \vdash e_2 \Rightarrow e_2' : \tau_2}{\Gamma \vdash e_1 e_2 \Rightarrow (\langle \tau_2 \to \star \rangle e_1')\, e_2' : \star}$$

$$\text{CApp2}\ \frac{\begin{array}{ll} \Gamma \vdash e_2 \Rightarrow e_2' : \tau_2 & \tau_2 \neq \tau \\ \Gamma \vdash e_1 \Rightarrow e_1' : \tau \to \tau' & \tau_2 \sim \tau \end{array}}{\Gamma \vdash e_1 e_2 \Rightarrow e_1'(\langle \tau \rangle e_2') : \tau'}$$

$$\text{CApp3}\ \frac{\Gamma \vdash e_1 \Rightarrow e_1' : \tau \to \tau' \qquad \Gamma \vdash e_2 \Rightarrow e_2' : \tau}{\Gamma \vdash e_1 e_2 \Rightarrow e_1' e_2' : \tau'}$$

Figure 5: Cast insertion. From *Gradual Typing for Functional Languages* [8].

that end, a *gradual guarantee* criterion, which relates the behavior of programs that differ only with respect to the precision of their type annotations [10], was introduced. In this section, we introduce Siek et al.'s *gradual guarantee* and discuss how the GTLC satisfies it and two criteria for gradual typing that appear in the literature [10].

3.1 Gradual as a Superset of Static and Dynamic

A gradually typed language is intended to include both an untyped language and a typed language. The GTLC is equivalent to the STLC for fully annotated terms. Siek and Taha prove that the GTLC type system, as described in Figure 4 is equivalent to the STLC [8]. The dynamic semantics of the GTLC and STLC has also been shown to be equivalent for fully annotated terms [10]. On the other end, the relationship between the GTLC and the DTLC is a little more nuanced as there are partially annotated types in GTLC and hence there could also be ill-typed terms in the GTLC. However, all terms in the DTLC are, trivially, well-typed. But, we can simply encode the GTLC to the DTLC by casting constants to the unknown type, ⋆ [10].

3.2 Soundness for Gradually Typed Languages

The GTLC is sound in the same way that dynamically typed languages are sound, execution never encounters trapped errors [3]. Siek et al. use the Blame Theorem to characterize safe versus unsafe upcasts. For a program to be totally safe, it should be fully annotated and it should only call statically typed functions. Note that if it refers to dynamically typed variables in its body, then its partially typed. The Blame Theorem shows that statically typed regions of code are never to blame for cast errors [10].

3.3 The Gradual Guarantee

Programmers should be able to add or remove type annotations without having any unexpected impacts on their program, such

$$\begin{tikzcd}
 & (\lambda y{:}\star\,.\,y)\ ((\lambda x{:}\star\,.\,x)\ 42) \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
(\lambda y{:}\texttt{int}.\,y)\ ((\lambda x{:}\star\,.\,x)\ 42) \arrow[d, dash] & (\lambda y{:}\star\,.\,y)\ ((\lambda x{:}\texttt{int}.\,x)\ 42) \arrow[dl, dash] & (\lambda y{:}\texttt{bool}.\,y)\ ((\lambda x{:}\star\,.\,x)\ 42) \arrow[d, dash] \\
(\lambda y{:}\texttt{int}.\,y)\ ((\lambda x{:}\texttt{int}.\,x)\ 42) & & (\lambda y{:}\texttt{bool}.\,y)\ ((\lambda x{:}\texttt{bool}.\,x)\ 42)
\end{tikzcd}$$

Figure 6: A lattice of differently annotated versions of a gradually typed program. From *Refined Criteria for Gradual Typing* [10].

as whether it still type checks and whether its runtime behavior remains the same [10]. With that in mind, the *gradual guarantee* guarantees that changes to a program's type annotations will not affect its static or dynamic behavior.

Consider the lattice in Figure 6. If we start at the top with the unannotated program, one would hope that adding more annotations would still result in the program evaluating to 42. This of course assumes that the programmer inserts the correct annotation, i.e. `int` for parameter `x`. If a wrong type, say `bool`, was annotated instead, this would trigger a static type error. Hence, we cannot claim "contextual equivalence" when going down the lattice, but, we can make the claim that when going up the lattice, the less precise expression will behave the same and give the same results as a more precise one [10].

The partial order in Figure 6 uses the *precision relation* on types and terms, defined in Figure 7 and Figure 8. Type precision is also known as naive subtyping [10]. We write $T \sqsubseteq T'$ when T is more precise than T' and $e \sqsubseteq e'$ when term e is more precisely annotated than e'.

$$\frac{}{\tau \sqsubseteq \star}$$

$$\frac{}{\gamma \sqsubseteq \gamma}$$

$$\frac{\tau_1 \sqsubseteq \tau_3 \qquad \tau_2 \sqsubseteq \tau_4}{\tau_1 \to \tau_2 \sqsubseteq \tau_3 \to \tau_4}$$

Figure 7: Type Precision definitions.

$$\frac{}{c \sqsubseteq c}$$

$$\frac{}{x \sqsubseteq x}$$

$$\frac{\tau_1 \sqsubseteq \tau_2 \qquad e_1 \sqsubseteq e_2}{\lambda x : \tau_1.e_1 \sqsubseteq \lambda x : \tau_2.e_2}$$

$$\frac{e_1 \sqsubseteq e_2 \qquad e_1' \sqsubseteq e_2'}{(e_1 e_1') \sqsubseteq (e_2 e_2')}$$

Figure 8: Term Precision definitions.

Formally stated, the gradual guarantee gives us the following theorem [10]

THEOREM 3.1. *Suppose $e \sqsubseteq e'$ and $\vdash e : \tau$.*

1. $\vdash e' : \tau'$ *and* $\tau \sqsubseteq \tau'$.
2. *If* $e \Downarrow v$, *then* $e' \Downarrow v'$ *and* $v \sqsubseteq v'$. *If* $e \Uparrow$ *then* $e' \Uparrow$.

Note: $e \Downarrow v$ *indicates that* e *evaluates to* v *and* $e \Uparrow$ *indicates that* e *diverges.*

The importance of the gradual guarantee is that it assures the programmer that when removing type annotations, a well-typed program will continue to be well-typed and a correctly running program will continue to do so. When adding type annotations, if the program is well-typed, the only possible change in behavior is a trapped error due to a mistaken annotation. Having this guarantee also allows for tools that support adding type annotations without risking programs misbehaving in unpredictable ways. The gradual guarantee for GTLC has been mechanically proofed by Siek et al. [10].

Although, having the gradual guarantee gives the programmer granular control over migrating code from dynamic to static typing and vice versa, at present, it is not sure if full blown languages with modern features, such as polymorphism and recursive types, can maintain this guarantee [10].

4 A GRADUAL SCHEME IMPLEMENTATION

We present an implementation of the GTLC in OCaml. We refer to this language as Gradual Scheme and use a Scheme-like syntax for our language as desribed in Figure 9. The type system is the same as that described in Figure 4, minus the syntactical differences.

$$\begin{aligned}
\text{Variables } x &\in \mathbb{X} \\
\text{Ground Types } \gamma &::= \texttt{int} \mid \texttt{bool} \\
\text{Constants } c &::= \texttt{\#t} \mid \texttt{\#f} \\
\text{Types } \tau &::= \gamma \mid \star \mid \tau \to \tau \\
\text{Expressions } e &::= c \mid x \mid (\texttt{lambda} : \tau\ (x : \tau)\ e) \mid (ee) \\
(\texttt{lambda}\ (x)\ e) &\equiv (\texttt{lambda} : \star\ (x : \star)\ e)
\end{aligned}$$

Figure 9: Syntax of Gradual Scheme. Note that the return type of a `lambda` is annotated separately, unlike in the GTLC's syntax in Figure 2.

Since Gradual Scheme is a minimal language modelled after the GTLC, it satisfies the same criteria for gradually typed languages as described in Section 3. We also implement a type checker that uses the consistency relation, as seen in Figure 10.

```
let rec is_consistent (t1 : t) (t2 : t) : bool =
  match t1, t2 with
  | BoolT, BoolT
  | IntT, IntT
  | StarT, _ | _, StarT -> true
  | ArrowT (t1, t2), ArrowT (t3, t4) ->
     (is_consistent t1 t3) && (is_consistent t2 t4)
  | _, _ -> false

let rec type_check (env : (string * t) list) (e : exp) :
     texp =
  match e with
  | BoolL _ -> e, BoolT
  | IntL _ -> e, IntT
  | Var v -> e, StarT
  | Lambda (arg, eb, t) ->
     (match t with
      | ArrowT (argt, rett) ->
         let _, tb = type_check ((arg, argt)::env) eb
         in if is_consistent rett tb
            then e, t
            else raise (Misannotation (eb, rett))
      | _ -> assert false)
  | App (e1, e2) ->
     let _, t1 = type_check env e1
     in let _, t2 = type_check env e2
        in match t1 with
           | ArrowT (argt, rett) ->
              if is_consistent argt t2
              then e, t2
              else raise (Misapplication ((e1, argt), (e2
                   , t2)))
           | _ -> e, StarT
```

Figure 10: Our type checking implementation based on the gradual type system in Figure 4. Note that the parser auto-assigns the $\star$ type to an expression, except for constant expressions, if it is un-annotated.

As expected of a gradually typed language, our implementation accepts all un-annotated expression, regardless of whether they are well-typed, since un-annotated expressions should behave like in a dynamic language, with type errors being checked at run-time. Below is an example of checking a well-typed un-annotated expression, followed by an ill-typed un-annotated expression in our Read-Check-Print-Loop[1] (RCPL).

```
> ((lambda (x) x) #t)
[INPUT] ((lambda (x) x) #t)
[OK] ((lambda (x) x) #t)

> ((lambda (f) (f #t)) 42)
[INPUT] ((lambda (f) (f #t)) 42)
[OK] ((lambda (f) (f #t)) 42)
```

However, once we annotate the expressions with the correct annotations, the program rejects the second expression since it is ill-typed.

```
> ((lambda (x : bool) x) #t)
[INPUT] ((lambda (x : bool) x) #t)
[OK] ((lambda (x : bool) x) #t)

> ((lambda (f : bool -> ?) (f #t)) 42)
[INPUT] ((lambda (f : bool -> ?) (f #t)) 42)
[ERROR] bool -> ? is not consistent with int
```

[1]We term it a RCPL, instead of a REPL since we do not evaluate the expression, but only type check it.

If a function's return type is annotated, the RCPL also checks that the annotation is consistent with the expression returned by the function. Else, it will raise an error indicating a mis-annotation as seen below.

```
> (lambda : bool (x) 42)
[INPUT] (lambda : bool (x) 42)
[ERROR] Misannotated return type bool for (Ast.IntL
     42)
```

4.1 Assessment

The Gradual Type System presented by Siek and Taha gives the programmer fine-tuned control over choosing which parts of the code base to statically type check. It is also sound. These are clear advantages to conveniently evolve towards statically typed code and have attempted to give programmers the best of both worlds, i.e. dynamic *and* static typing in a single, unified system. However, the type system presented in *Gradual Typing for Functional Languages* does not cover modern language features such type classes, recursive types, and polymorphism. It remains to be seen if these features can be incorporated into the type system while maintaining the gradual guarantee and soundness of the type system. The paper also fails to describe embedding gradual typing into realistic systems that are suitable for writing significant applications [14].

Additionally, there is still the minor inconvenience of having to annotate each expression in order to fully benefit from having the entire codebase be type checked. While this could be potentially burdensome for a large codebase, as some expressions might need to be rewritten to type check, we think that this is a small price to pay and will be made easier with proper tooling that could infer and suggest types. At present, Siek and Vachharajani have already developed an algorithm for unification-based inference for the GTLC [9] and Rastogi et al. have developed a type inference algorithm that has been implemented for ActionScript [6]. Both are sound and complete.

Our program currently does not implement cast insertion since that would be part of run-time checks. But, we plan to allow evaluation in the future and hence will implement cast insertion with blame tracking then.

5 CHALLENGES & FUTURE WORK

Much progress has been made since Siek and Taha's original paper. There has been a growing body of research [10] and an increasing number of languages in industry that have integrated gradual typing[2]. However, gradual typing is unfortunately still not a solved problem. At present, the open questions of gradual typing, from Siek's 2017 Principles of Programming Languages (POPL) tutorial, *The State of the Art in Gradual Typing* [7], are

1. Is it possible for a gradually typed language to be efficient?
2. Have we got the blame tracking right?
3. How does gradual typing interact with advanced language features?

We focus on gradual typing's performance challenges below.

[2]Some examples of languages that support gradual typing include Dart, TypeScript, and Hack

5.1 Performance

To maintain soundness, gradual typing uses either casts or contracts to check that certain invariants, such as a higher order function's return value's type, are not violated during run-time. These checks during runtime can be expensive, but are required to maintain soundness in the overall system.

In Takikawa et al.'s paper, *Is Sound Gradual Typing Dead?* [11], the authors benchmarked Typed Racket, which offers gradual typing on a per-module basis, and compared its performances for different *configurations* of the program. A configuration is a sequence of n modules with a specific ratio of modules that are typed versus untyped. To measure the performance overheads, the paper defines the term *N-deliverable* to mean that a configuration is *N-deliverable* if its performance is worse than an Nx slowdown compared to the completely untyped configuration [11]. This allows teams to determine the amount of overhead that they are willing to accept. For example if a team is able to accept a 3x slowdown in a program, then Takikawa et al.'s charts will be able to inform them what percentage of that program will be deliverable with that slowdown.

In their experiments, Takikawa et al. measured 5 user-written libraries and programs, 5 educational programs, and 2 programs that were specially written for their paper. It turns out that the costs of enforcing soundness is expensive. Less than half of the benchmarked programs are 50% deliverable when measured at 3-deliverable proportions and the average proportion of programs that are 3-deliverable are 45.5% deliverable [11]. This potentially makes software undeliverable in a commercial setting.

They also acknowledge that there is more that can be done on the language designer's part to improve on the performance, such as improving the current conventional JIT implementation, and making the compiler contract-aware [11]. The authors have released tools to help avoid gradual typing performance pitfall, such a contract profiler to see the percentage of run-time that the contracts are taking up and which contract boundaries (between typed and untyped modules) are taking up a lot of time.

Typed Racket is not the only gradually typed language to suffer from performance issues. The authors of Reticulated Python recognized this and designed the language to allow the exploration of efficient cast mechanisms [11]. Vitousek et al. acknowledge that Reticulated programs perform "far worse than their unchecked Python implementations" and their `slowSHA` library test suite took 10x longer when compared to normal Python [16].

6 CONCLUSION

We have seen the motivation behind combining dynamic and static typing, and the related lines of research that have attempted to draw on the strengths of both type systems. Then, we studied the Gradually Typed Lambda Calculus (GTLC), and explored a formal criteria for gradual typing. We also discussed our implementation of the GTLC and the strengths and weakness of the gradual typing system proposed by Siek and Taha. There are still a number of open questions for gradual typing, including that of performance and having gradual typing interact with advanced language features. However, gradual typing is still an active area of research. The popularity of languages such as Dart and TypeScript, and recent efforts such as Typed Racket and Type Hints in Python, signals the appeal of integrating static type analysis into dynamic languages.

REFERENCES

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. 1989. Dynamic Typing in a Statically-typed Language. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89).* ACM, New York, NY, USA, 213–227. DOI: https://doi.org/10.1145/75277.75296

[2] Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586.* Springer-Verlag New York, Inc., New York, NY, USA, 257–281. DOI: https://doi.org/10.1007/978-3-662-44202-9_11

[3] Luca Cardelli. 1996. Type Systems. *ACM Comput. Surv.* 28, 1 (March 1996), 263–264. DOI: https://doi.org/10.1145/234313.234418

[4] Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. 2005. Fine-grained Interoperability Through Mirrors and Contracts. *SIGPLAN Not.* 40, 10 (Oct. 2005), 231–245. DOI: https://doi.org/10.1145/1103845.1094830

[5] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17 (1978), 348–375.

[6] Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. 2012. The Ins and Outs of Gradual Type Inference. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12).* ACM, New York, NY, USA, 481–494. DOI: https://doi.org/10.1145/2103656.2103714

[7] Jeremy G. Siek. 2017. State of the Art in Gradual Typing. (2017). https://wphomes.soic.indiana.edu/jsiek/ Tutorial at POPL 2017.

[8] Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*, Vol. 6. 81–92.

[9] Jeremy G. Siek and Manish Vachharajani. 2008. Gradual Typing with Unification-based Inference. In *Proceedings of the 2008 Symposium on Dynamic Languages (DLS '08).* ACM, New York, NY, USA, Article 7, 12 pages. DOI: https://doi.org/10.1145/1408681.1408688

[10] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 274–293.

[11] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual Typing Dead?. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16).* ACM, New York, NY, USA, 456–468. DOI: https://doi.org/10.1145/2837614.2837630

[12] Satish Thatte. 1990. Quasi-static Typing. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '90).* ACM, New York, NY, USA, 367–381. DOI: https://doi.org/10.1145/96709.96747

[13] Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: From Scripts to Programs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06).* ACM, New York, NY, USA, 964–974. DOI: https://doi.org/10.1145/1176617.1176755

[14] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08).* ACM, New York, NY, USA, 395–406. DOI: https://doi.org/10.1145/1328438.1328486

[15] Julien Verlaguet. 2013. Facebook: Analyzing PHP statically. In *Commercial Users of Functional Programming (CUFP).*

[16] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. In *Proceedings of the 10th ACM Symposium on Dynamic Languages (DLS '14).* ACM, New York, NY, USA, 45–56. DOI: https://doi.org/10.1145/2661088.2661101

[17] Philip Wadler and Robert Bruce Findler. 2009. Well-typed programs can't be blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009 (ESOP '09).* Springer-Verlag, Berlin, Heidelberg, 1–16. DOI: https://doi.org/10.1007/978-3-642-00590-9_1