Safety First?

An Essay On Gradual Typing

Dan Hemgren
KTH Royal Institute of Technology
Sweden
dhemgren@kth.se

Abstract

In this essay, the concept of Gradual Typing as presented by Siek and Taha is explored. Two foundational systems to gradual typing, static and dynamic typing, are explained and key differences between the systems are detailed. The performance cost and challenges of implementing gradual typing are discussed, and several real world examples of languages that employ the system are shown. The essay concludes with a brief summary.

Keywords type-checking, dynamic typing, static typing, gradual typing, type systems, lambda calculus

1 Introduction

Gradual Typing, as presented by Siek and Taha, is a type system that aims to merge the flexibility of dynamic typing with the safety of static typing [10]. Where static typing enables error detection at compile-time, dynamic systems conversely rely on continuous error checking during run-time [3]. Thus, static typing enables earlier error detection (hopefully foregoing insidious bugs triggered by esoteric conditions in a program), but in doing so, demands that the designer of the program explicitly states the type for each declaration in the program. The broad stroke benefits of both systems are clear; one is safer, the other more flexible. The debate over which system is better is ongoing, and, as an example, Harlin et al. argues that safety nets incurred by static typing does not catch the errors developers actually make, such as mistakenly using null pointers or accessing empty arrays [3]. With no clear cut answer to the debate, the middle road of gradual typing allures.

2 λ-Calculus, Gradually Typed

The system presented by Siek and Taha is an extension of simply-typed λ-calculus, dubbed $\lambda^{?}_{\rightarrow}$, now containing a type **?** to represent dynamic types in the system (under τ in Figure 2) [10]. With this system, Siek and Taha specifies an approach meant to support types that may be partially (or entirely) unknown, but still abiding to the rules of λ-calculus. As a concrete example of a partially unknown type, Siek and Taha details a pair

number $*$?

DD2481, May 26, 2018, Stockholm, Sweden
2018.

$$\boxed{\Gamma \vdash_G e : \tau}$$

$$\text{(GVar)}\quad \frac{\Gamma\, x = \lfloor \tau \rfloor}{\Gamma \vdash_G x : \tau}$$

$$\text{(GConst)}\quad \frac{\Delta\, c = \tau}{\Gamma \vdash_G c : \tau}$$

$$\text{(GLam)}\quad \frac{\Gamma(x \mapsto \sigma) \vdash_G e : \tau}{\Gamma \vdash_G \lambda\, x{:}\sigma.\ e : \sigma \rightarrow \tau}$$

$$\text{(GApp1)}\quad \frac{\Gamma \vdash_G e_1 : ? \qquad \Gamma \vdash_G e_2 : \tau_2}{\Gamma \vdash_G e_1\ e_2 : ?}$$

$$\text{(GApp2)}\quad \frac{\Gamma \vdash_G e_1 : \tau \rightarrow \tau' \qquad \Gamma \vdash_G e_2 : \tau_2 \qquad \tau_2 \sim \tau}{\Gamma \vdash_G e_1\ e_2 : \tau'}$$

Figure 1. Gradual typing in λ-calculus as presented by Siek and Taha [2006] [10]

wherein the first type is **number**, the next is unknown. How the error detection is meant to unfold in practice is fairly straight-forward: type errors are handled as static for declared types, meaning error detection is handled at compile-time, and type errors are handled dynamically for unknown types, meaning error detection occurs at run-time [10]. Using this approach, the intent of Siek and Taha is to leave it to the programmer to decide the level of compile-time type safety, thus maintaining the flexibility of dynamic typing at will.

Variables	$x \in \mathbb{X}$		
Ground Types	$\gamma \in \mathbb{G}$		
Constants	$c \in \mathbb{C}$		
Types	τ	::=	$\gamma \mid ? \mid \tau \rightarrow \tau$
Expressions	e	::=	$c \mid x \mid \lambda x{:}\tau.\ e \mid e\ e$
			$\lambda x.\ e \equiv \lambda x{:}?.\ e$

Figure 2. The syntax of gradually typed λ-calculus as presented by Siek and Taha [2006] [10]

It is hinted at in the final case under the definition of expressions, e, in Figure 2 that a form of syntactic sugar has been introduced; all terms without explicit type declarations adopt the ?-type. For example, in $t = \lambda x.\ \lambda y{:}T.\ x\ y$, it is given

that $y : T \in \Gamma$, while the type of x remains unknown, and subsequently $t : (? \to T \to \tau)$. Any type errors caused by setting the parameter x will now be caught at run-time. In this particular example, the abstraction body of t implicitly governs the possible types for x. By definition (Figure 2), x can not be a constant as $x\ y$ implies that the reduced term of x must either be an application (or an abstraction) of type $T \to \tau$. On the other hand, explicitly stating the type $x{:}T_0$ such that $t = \lambda x{:}T_0.\ \lambda y{:}T.\ x\ y$ yields $t : (T_0 \to T \to \tau)$ and enables further checking of t for type errors at compile-time.

2.1 Error-handling

Siek and Taha highlights the difference in run-time type errors between *weakly*- versus *strongly*-typed languages. In weakly-typed languages, the consequences of type errors are undefined (and most often unpredictable). One such example could be the segmentation fault caused by writing a 64-bit float to memory allocated for a 32-bit integer. On the other hand, strongly-typed languages catch these errors at run-time. In doing this, the run-time environment effectively blocks any undefined behavior caused by type errors and instead throws an exception. In the gradual type system of Siek and Taha, type errors are not allowed to run amok (read: cause undefined behavior) and instead produces an exception governed by the EKILL and ECSTE rules [10].

$$\text{(ECSTE)}\quad \frac{e \mid \mu \hookrightarrow_n v \mid \mu' \qquad \emptyset \mid \Sigma \vdash \mathit{unbox}\ v : \sigma \qquad (\sigma, \tau) \notin \mathit{op} \sim}{\langle \tau \rangle\ e \mid \mu \hookrightarrow_{n+1} \mathit{CastError} \mid \mu'}$$

$$\text{(EKILL)}\quad e \mid \mu \hookrightarrow_0 \mathit{KillError} \mid \mu$$

Figure 3. Type cast error rules as presented by Siek and Taha [2006] [10]

3 Gradual Typing In Use

There are several extensions to programming languages that shift the inherent type system towards gradual typing. One such example is presented by Vitousek et al.. Dubbed *Reticulated Python* (Figure 4), it is a source-to-source-translator based on first-order object calculus that extends Python 3 to support development with gradual typing [12]. In the report, Vitousek et al. points to several benefits of the gradual type system. Most notably, the early-bird error detection enabled by declaring parameter types help the debugger zoom in on the actual source of the error as opposed to potentially returning a stack trace from deep within a library (accidentally obfuscating the cause of the error) [12]. Much like the approach described by Siek and Taha, under Reticulated Python, any parameter that has not had a type declared explicitly will be considered unknown (read: dynamically typed).

```
def flat(l:List(List(Dyn)))->List(Dyn):
  newl = []
  for l1 in l:
    for j in l1:
      newl.append(j)
  return newl

:>> flat([[2,5,3], [6,3,2]])
[2,5,3,6,3,2]
:>> flat([['a', 'b', 'c'], [1,2,3]])
['a', 'b', 'c', 1, 2, 3]
:>> flat([[[]], [[]]])
[[], []]
:>> flat([1,2,3])
====STATIC TYPE ERROR=====
```

Figure 4. Example code supplied by the creator of Reticulated Python [6]

Another prime example, one that goes beyond being a mere extension, is the programming language of C# (Figure 5), which as of version 4.0 support gradual typing [2]. Bierman et al. cites the challenges of interoperability between statically and dynamically typed languages as an argument for introducing a gradual type system; interfacing between, for example, Java and Javascript can infer awkward or suboptimal code. The result of this new (to the language) gradual typing is the new *dynamic* type, which is declared much like other type declarations in the language.

Notably, TypeScript (Figure6) is a static type-checker extension for JavaScript - an inherently dynamically typed language. It has recieved criticism for foregoing soundness for ease-of-use [7], and Rastogi et al. presented Safe TypeScript in an attempt to enforce soundness while retaining the benefits. Safe TypeScript implements a plethora of changes to cater to the new typing system, such as removing the use of *null* in favor of only *undefined*.

4 Sound Gradual Typing And Performance

The notion of a *sound* gradually typed language is one of untyped and typed code coexisting in harmony, letting the typed code be optimized as if no untyped code was present [5]. Muehlboeck and Tate points to the fact that this separation unavoidably introduces some form of overhead, since the compiler and run-time environment must be able to differ between typed and untyped code. Takikawa et al. shows that there even exists cases in Typed Racket, a gradually typed language, where the incurring of overhead caused by gradual typing significantly hampers performance [11]. In general,

```
static void Main(string[] args)
{
    ExampleClass ec = new ExampleClass();
    // The following call causes
    // a compiler error if
    // the method only takes
    // ONE parameter.
    //ec.exampleMethod1(10, 4);

    dynamic dynamic_ec = new ExampleClass();
    // The following line is not
    // identified as an error by
    // the compiler, but it causes
    // a run-time exception.
    dynamic_ec.exampleMethod1(10, 4);

    // The following calls also do
    // not cause compiler errors,
    // whether appropriate
    // methods exist or not.
    dynamic_ec.someMethod("some argument");
    dynamic_ec.nonexistentMethod();
}
```

Figure 5. C# Dynamic Type example code supplied by Microsoft [13]

```
//JAVASCRIPT EXAMPLE
var addFunction = function (n1, n2, n3) {
var sum = n1 + n2 + n3;
return sum;
};
var sum = addFunction(10, 20);

//EQUIVALENT TYPESCRIPT EXAMPLE
var addFunction = function (n1: number,
    n2: number, n3?: number) : number {
var sum = n1 + n2 + n3;
return sum;
};
var sum: number = addFunction(10, 20);
```

Figure 6. TypeScript example code [8]

the biggest impact on performance by gradual typing is casts, and how they are used within the program largely dictates the performace of the type system in itself [5]. Muehlboeck and Tate cite three main casting strategies: *Guarded*, *Transient* and *Monotonic*. Guarded casting involves wrapping a function, upcasting its input parameter to an unknown type. After the function has been called, the output type is checked to be consistent with the type declaration. Transient casting involves both the caller and target function casting its parameters (resulting in a potentially large amount of casts for functions called frequently). Monotonic casting, formalized by Siek and Taha, means each value stores records of its type identity. These records are then checked with each assignment to detect type errors.

With extending a language to allow for gradual typing comes the challenges of a performance efficient implementation. Often times, the type system of the language is so deeply nested in its fabric that making ad hoc additions to the system run the risk of introducing adverse performance overhead [5]. For many of the modern imperative languages, a gradual type extension also needs to cater to objects and inheritance, further increasing the complexity. Kuhlenschmidt et al. notes that existing implementations have not yet managed to reach the performance of strictly dynamically or statically typed systems, but shows that it is within reach by using monotonic references and space efficient *coercions* [4], a method of handling run-time type casts.

5 Object-Oriented Gradual Typing

As an extension to the the $\mathbf{Ob}_{<:}$-calculus (object-oriented calculus) of Abadi and Cardelli [1], Siek and Taha introduces $\mathbf{Ob}^{?}_{<:}$-calculus; object-oriented calculus with gradual typing [9]. In it, they employ the notion of type *consistency* as opposed to type *equality* using the ~-operator (see Figure 7). Their presented definition of type consistency is that two known or unknown types are consindered consistent if, and only if, they are equal when both are known. To circumvent the loophole of downcasting ($R <: ?$) and upcasting ($? <: S$) in tandem, allowing any type R to be cast to S and by extension allowing *all* programs, Siek and Taha introduce a restriction operator, $\sigma|_\tau$, so that two types are consistent if and only if $\sigma|_\tau = \tau|_\sigma$. Much like the original $\lambda^{?}_{\rightarrow}$-calculus presented by Siek and Taha, types that are unannotated are typed as the $?$-type; unknown and to be determined.

$$\text{int} \sim \text{int} \qquad \text{int} \not\sim \text{bool} \qquad ? \sim \text{int} \qquad \text{int} \sim ?$$
$$[x : \text{int} \rightarrow ?, y : ? \rightarrow \text{bool}] \sim [y : \text{bool} \rightarrow ?, x : ? \rightarrow \text{int}]$$
$$[x : \text{int} \rightarrow \text{int}, y : ? \rightarrow \text{bool}] \not\sim [x : \text{bool} \rightarrow \text{int}, y : ? \rightarrow \text{bool}]$$
$$[x : \text{int} \rightarrow \text{int}, y : ? \rightarrow ?] \not\sim [x : \text{int} \rightarrow \text{int}]$$

Figure 7. Type consistency example as presented by Siek and Taha [2007] [9]

6 Conclusion

The rationale behind the conception of gradual typing is clear; both static and dynamic type systems give their own set of benefits to designers of a program, and the gradual type system aims to retain the perks of both systems. Simplified, this is done by housing both systems in tandem, having the compiler error-check all statically typed variables and the run-time environment govern the dynamic types during execution. While this might seem like an easy best-of-both-worlds solution to the debate over which system is better, there is a caveat in the form of performance cost. The overhead introduced by continually checking the nature of any given variable (is it statically typed? is the type known or to be determined? et.c.) can in cases be substantial, enough so that the gradual type system is arguably no longer a good fit for the program. However, one must remember that the performance cost is very much dependent on the nature of the implementation, and adding ad hoc type functionality to a language with a deeply ingrained type system poses its own set of challenges.

There are no shortages of gradually typed languages out in the wild. Widespread languages like C# have adopted this new approach, and there exists several extensions to other heavy-weight languages like Javascript and Python. However, the benefits to having a statically typed system have been questioned. It has been argued that the errors that statically typed systems aim to prevent are not the errors program designers are most prone to make, meaning the effort is misguided, making the loss of dynamic typing a heavy-handed solution to a lesser problem. Harlin et al. even goes as far as discussing whether static typing could potentially *worsen* the quality of code by incurring a false safety giving the designer a reason to think a program with logical flaws is *good enough* since it compiled without errors.

The research topic of gradual typing is very much alive and well, with several key papers having been presented in recent years. The system earning a permanent place in a language as widespread as C# bodes well for its future, and further research into reducing the performance overhead of type checking could maybe someday render strictly static / dynamic type systems obsolete.

References

[1] MartÃŋn Abadi, Luca Cardelli, and Ramesh Viswanathan. 1996. An interpretation of objects and object types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL '96)*. ACM, 396–409.

[2] Gavin Bierman, Erik Meijer, and Mads Torgersen. 2010. Adding Dynamic Types to C. 76–100.

[3] Ismail Rizky Harlin, Hironori Washizaki, and Yoshiaki Fukazawa. 2017. Impact of Using a Static-Type System in Computer Programming. *High Assurance Systems Engineering (HASE), 2017 IEEE 18th International Symposium on*, 116–119.

[4] Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. 2018. Efficient Gradual Typing. (February 2018).

[5] Fabian Muehlboeck and Ross Tate. 2017. Sound Gradual Typing is Nominally Alive and Well. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 56 (Oct. 2017), 30 pages. https://doi.org/10.1145/3133880

[6] Mvitousek. [n. d.]. mvitousek/reticulated. ([n. d.]). https://github.com/mvitousek/reticulated/blob/master/docs/tutorial.md

[7] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. 2015. Safe & efficient gradual typing for TypeScript. *ACM SIGPLAN Notices* 50, 1 (2015), 167–180.

[8] Sachinohri. [n. d.]. TypeScript - 101: The Basics. ([n. d.]). https://www.codeproject.com/Articles/802722/TypeScript-The-Basics

[9] Jeremy Siek and Walid Taha. 2007. Gradual Typing for Objects. In *ECOOP 2007 – Object-Oriented Programming*, Erik Ernst (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 2–27.

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

[11] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual Typing Dead? *SIGPLAN Not.* 51, 1 (Jan. 2016), 456–468. https://doi.org/10.1145/2914770.2837630

[12] M.M. Vitousek, A.M. Kent, J.G. Siek, and J. Baker. 2015. Design and evaluation of gradual typing for python. *ACM SIGPLAN Notices* 50, 2, 45–56.

[13] Bill Wagner. [n. d.]. Using Type dynamic (C Programming Guide). ([n. d.]). https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/using-type-dynamic