Why You Want Sound Gradual Typing, and Why You Won't See it for a While

Introduction

Sound gradual typing is something very useful that may one day make its way into the mainstream. Currently, it's only in niche and research languages like Racket. There are still many problems that need to be worked out before most people would go anywhere near it. But one day, we may wonder how we lived without it. Anyway, let's find out what sound gradual typing is and why it's so useful.

Gradual Typing

Gradual typing is a mixture of static and dynamic typing in a single language. In a gradually typed language, some parts of code can be typed and others can be left untyped. TypeScript and Python with type hints are examples of this. You can have totally untyped javascript files and typed TypeScript files in the same project and it works fine. Being able to do this is very helpful.

Gradual typing gives us the benefits of static and dynamic typing in one language. Dynamic typing is good for early prototyping and exploration where a static type checker isn't necessary and just gets in the way. Certain parts of code like glue code don't benefit much from static typing anyway. In contrast, systems level programming and large, complex projects can be very difficult to maintain without the safety and guarantees of static typing. It helps you keep everything together. It also can help compilers make optimizations. With gradual typing, you don't have to choose between one or the other. This also means it is easy to migrate from untyped to typed code.

You might want to start out a project dynamically typed so you can easily prototype and explore. Then, when things get complicated, you might want to migrate to static typing. Without gradual typing, you'd have to migrate your whole project at once between two completely different languages. This is error prone because it's not as simple as adding type annotations. Different languages have different idioms and ways of expressing programs. Type systems also restrict you to write code in a type-checkable way. It's very easy to accidentally introduce a bug in this migration. For example, let's consider this snippet of JavaScript code:

```
function sayHello(name) {
    if (typeof name === 'string') {
        return `hello ${name}!`
    } else {
        return `hello ${name.first} ${name.last}!`
    }
}
```

Imagine trying to migrate this to Java. You'd probably have to make some sort of `Name` interface with a `sayHello` method that returns a `String`. For the string case, you'd also need to make a `StringName` class that wraps a string. Each case of this function would live in a separate class. And you'd need to make sure you wrap and unwrap `StringName`s properly all over your codebase where names are passed around. It'd be very easy to mess up somewhere. In contrast, let's see how we'd translate this to TypeScript:

```
type Name = {first: string, last: string} | string
function sayHello(name: Name): string {
    if (typeof name === 'string') {
        return `hello ${name}!`
    } else {
        return `hello ${name.first} ${name.last}!`
    }
}
```

All we had to do was add annotations. We didn't have to change the logic at all. This is because TypeScript is designed to accommodate JavaScript idioms and make migration non-intrusive. But enough about TypeScript which we all know and love. What is soundness?

Safe and Sound

Have you ever had to work with untyped JavaScript code from TypeScript? If you have, you probably either migrated the JavaScript to TypeScript or added a declaration file. If you're working with an untyped library, you'd probably have to go with the latter. Figuring out what types to declare for a library's functions is non-trivial. What if you make a mistake or the library has a function that may return the wrong type? Let's consider an example:

```
// in indexOf.js
function indexOf(str, char) {
    const i = str.indexOf(char)
    if (i === -1) {
        return undefined
    } else {
        return i
    }
}
```

Let's say that's your untyped library. You make a mistake in your declarations by thinking `idexOf` works like the usual `Array.indexOf` and returns -1 on failure:

```
 // in indexOf.d.ts
declare module "indexOf" {
    export function indexOf(str: string, char: string): number
}
```

```
 // in main.ts
/// <reference path="indexOf.d.ts">
import {indexOf} from "indexOf"

// finds the letter "p" in `str` and prints its 1-indexed location or "not found"
function findP(str: string): void {
    const i = indexOf(str, "p")
    if (i === -1) {
        console.log("not found")
    } else {
        console.log(i + 1)
    }
}
```

If `"p"` is not in `str`, the code will print `NaN` from evaluating `undefined + 1`. If you're lucky, this kind of mistake will result in a runtime type error somewhere close to where this function was imported. If you're less lucky, this might result in a very confusing type error deep in your type-checked code, which should be impossible. Or worse, there are no errors and a correct-looking answer is returned that is, in fact, nonsense. TypeScript doesn't actually check to make sure the function always returns numbers as the declaration states. It just believes the declaration. A sound gradual type system would insert runtime checks to make sure the untyped function is of the declared type. In this case, it would check the return value every time the function is called. Let's see what this issue looks like in a language with sound gradual typing like Racket:

```
 ; index-of.rkt
#lang racket

(provide index-of)

(define (index-of str char) #f)
```

```
 ; main.rkt
#lang typed/racket

(require/typed "./index-of.rkt"
               [index-of (-> String Char Integer)])

(index-of "hello" #\p)
```

In the typed module, we require (import) the function and declare that it returns only Integers. We get this error message:

```
 index-of: broke its own contract
  promised: exact-integer?
  produced: #f
  in: (-> any/c any/c exact-integer?)
  contract from: (interface for index-of)
  blaming: (interface for index-of)
   (assuming the contract is correct)
  at: G:\GitHub\quasarbright.github.io\blog\soundness\examples\main.rkt:5:16
  context...:
   G:\Racket\collects\racket\contract\private\blame.rkt:346:0: raise-blame-error
   G:\Racket\share\pkgs\typed-racket-lib\typed-racket\utils\simple-result-arrow.rkt:39:12
   body of "G:\GitHub\quasarbright.github.io\blog\soundness\examples\main.rkt"
```

This error is more informative, points directly to the mistake in code (the `require/typed` form), and occurs immediately when the function is called. You can also use this to perform a runtime-validated cast for any type:

```
 > (cast 3 String)
broke its own contract

  promised: string?

  produced: 3

  in: string?

  contract from: cast

  blaming: cast

   (assuming the contract is correct)

  at: eval:55:0
```

This mechanism could be used to validate and type JSON:

```
 const obj : any = JSON.parse(jsonStr)
interface Person {
    name: string
    age: number
}
const person : Person = obj as Person
```

In normal TypeScript, this "cast" on the last line wouldn't actually do anything at runtime. It would just tell the type checker to "trust you" at compile-time and carry on type checking as if `person` has the correct fields and types of fields. In a sound gradually typed language like Racket, the fields would actually be checked.

There is a caveat: For functions, these runtime checks can validate that values of an unexpected type are not returned. However, they don't help you with arguments. If you declare that a function can take in any type of number, but it can only take in integers, passing in a float will result in the same kind of runtime error as if you called it from untyped code.

These runtime checks make using untyped code from typed code much safer. There are also runtime checks that make the other direction safe:

Let's say you're writing a library in TypeScript. JavaScript users can use your library just fine, but the type information is lost at runtime. If you write a TypeScript function that takes in a number, there is nothing stopping a JavaScript program from passing in a string. Again, you're lucky if you even get a runtime type error. If you want to avoid this, you might check inputs of functions you export. But what if your library is used by TypeScript and these checks are unnecessary? And what if your library exports functions which take in other functions as arguments? Soundness solves these problems by inserting runtime checks when values flow from typed code to untyped code, but not when they flow from typed code to other typed code. For functions, a wrapper is inserted which checks arguments on the way in.

Let's consider an example:

Here is our typed library:

```
 ; lib.rkt
#lang typed/racket

(provide increment)

(: increment : Number -> Number)
(define (increment n)
  (+ n 1))
```

Here is the untyped user of this library making a type error:

```
 ; lib-user.rkt
#lang racket

(require "./lib.rkt")

(increment "one")
```

This is what happens when we run the library user program:

```
increment: contract violation
  expected: number?
  given: "one"
  in: the 1st argument of
      (-> number? any)
  contract from:
      G:\GitHub\quasarbright.github.io\blog\soundness\examples\lib.rkt
  blaming: G:\GitHub\quasarbright.github.io\blog\soundness\examples\lib-user.rkt
   (assuming the contract is correct)
  at: G:\GitHub\quasarbright.github.io\blog\soundness\examples\lib.rkt:7:9
  context...:
   G:\Racket\collects\racket\contract\private\blame.rkt:346:0: raise-blame-error
   G:\Racket\collects\racket\contract\private\arrow-higher-order.rkt:375:33
   body of "G:\GitHub\quasarbright.github.io\blog\soundness\examples\lib-user.rkt"
```

When `lib-user.rkt` imported the function from the typed library, it was wrapped in a `(-> number? any)` contract. Before the body of `increment` even runs, the contract raises an error from the argument having the wrong type. If `lib-user.rkt` was typed, we'd get a similar error before any of the code runs and there would be no contract checking arguments at runtime. It's exactly what you want.

In TypeScript, however, the function wouldn't result in an error at all. You'd actually get `"one1"` and think everything is fine even though your data is nonsense.

One other advantage of soundness is that it allows compilers to make optimizations on typed code. In a dynamically typed language, when you access a field of an object, at runtime, the language makes sure the value is actually an object and has that field before doing any de-referencing of pointers. In a statically typed language, this is unnecessary because you know that the value will be the correct type ahead of time. In an unsound gradually typed language, the compiler couldn't be fully confident that values really will have the type they're supposed to at runtime, so typed code can't safely be optimized. Soundness guarantees that values are actually the type they're supposed to be, so optimizations which rely on static typing would be safe.

Great! We should all use sound gradual typing! Right? Well, unfortunately, there are many problems with soundness that make it practically unusable.

Bad Performance

A significant issue with soundness is its performance. When values flow between typed and untyped code, they are runtime checks inserted to make sure the interaction is type-safe. Making sure a value is an integer is cheap, but functions and compound data like arrays are more complicated. Functions have to be wrapped by contracts which check arguments or return values depending on the direction of the boundary-crossing. These run every time the function is called, they take up space, and if the function goes back and forth between typed and untyped code, the wrappers duplicate, multiplying these costs. For compound values like arrays, either every value is checked, even if it is never accessed, or the data structure is wrapped with something that checks the type upon accessing. These wrappers can also duplicate like functions' wrappers. These runtime checks have a significant impact on performance

The costs of these wrappings and delayed checks adds up to cause extreme slowdowns. According to a study on the performance of Typed Racket, which is the state of the art for sound gradual typing in an industrial-strength language, the cost of soundness is "overwhelming". In this study, several programs were run in various configurations of typed and untyped code. Certain configurations led to slowdowns around 100x, and adding annotations only made performance worse. It was only when all modules were typed that performance became acceptable (0.7x). This performance hit is extreme.

For a potential 100x slowdown, nobody writing a production application would touch it with a hundred-foot pole. Garbage collection is in pretty much all the mainstream languages today, but it took a very long time, hardware advancement, and research innovation before it was considered practical and worth the performance impact. Soundness has a long way to go before it is worth using. The benefits aren't enough to justify these costs, and even if they were, soundness is incomplete.

Dude, Where's my Polymorphism?

Soundness' runtime checks don't work for all types. A notable exception is parametric polymorphism. For example, consider mapping a function over each element of an array:

```
function map<A,B>(items: Array<A>, callbackFn: (value: A) => B): Array<B> {
    items.map(callbackFn);
}
```

How would you wrap this in a contract that ensured it was used properly in an untyped setting? You'd have to make sure all the elements of the array are of the same type, but how do you figure out the type of a value at runtime? What if it's a function? What if it's a value that can be one of many subtypes? You'd also have to make sure `callbackFn` takes in values of type `A` and returns values of type `B`, but you can't know what `B` is until you call the function with an input. Even then, you still have to figure out the type of a value at runtime. It's a tricky problem.

One attempt to solve this problem is dynamic sealing. This involves opaquely wrapping values as they go in and unwrapping them as they go out of functions. For example, let's consider `map`, but make it so `B` is just `A`. We'd take in an `Array<A>` and an `A => A` function and return an `Array<A>`. Every element of the input array would be wrapped in a structure that you can't look inside, like it's sealed with the letter "A". You can't seal or unseal values yourself. The `callbackFn` would be wrapped with a contract that only accepts arguments sealed with the letter "A" and makes sure everything it returns is sealed with the letter "A". Then, finally, every element would be unwrapped, ensuring it is sealed with the letter "A" before unwrapping. Since you can't seal or unseal values yourself, it is guaranteed that what was taken in and sealed with an "A" is still an "A" when it is checked and unsealed on the way out. This ensures that polymorphic functions just "shuffle around" values and don't actually look at them, combine them, or create new values from scratch. This means you can't do those things even if you want to, which causes problems.

Dynamic sealing can break your function. If you tried to map `(n) => n + 1` over an array of numbers, you'd end up adding a number to a sealed value and get an array with elements like `"3[object A]"` instead of `4`. This is nonsense. The contract changes the behavior of the function, which is unacceptable. You can only use dynamic sealing in situations where you don't need to look at the values at all. This would be fine for something like appending two arrays where you don't care about the elements directly, but doesn't work with something like map where you are passing each value to a function which may need to look at the actual values. Think about the original signature where `A` and `B` are not necessarily the same. How would `callbackFn` produce a value sealed with a `B` if it can't seal values and doesn't even know about your map function?

Functions like `map` are very important and they are not supported by soundness. In Typed Racket, if a value with such a polymorphic type flows between typed and untyped code, an error is thrown instead of trying to use dynamic sealing. This incompleteness makes soundness much less usable. Soundness isn't looking too good, especially considering that plenty of gradually typed languages are doing just fine without it.

No Soundness, no Problem

TypeScript is the most popular implementation of gradual typing. According to the 2021 Stack Overflow developer survey, TypeScript was the fifth most used programming language by professional developers with a reported usage of 36.42%. This can be explained by the popularity of JavaScript, which has been the most popular language for nine years in a row, with a reported usage of 68.62%. It is difficult to maintain large, complex projects in a dynamically typed language like JavaScript, so it is unsurprising that JavaScript has benefited greatly from a very expressive gradual type system. Python also has unsound gradual typing. Soundness is clearly not necessary for a gradual type system to be usable.

One Day

Sound gradual typing is a very good idea and makes typed-untyped interaction much safer. One day, it might make its way into the mainstream. Research will make progress. More efficient implementations will be discovered and unsolved problems will be solved. People used to think garbage collection would never be practical and look where we are today. Hopefully soundness becomes popular and one day, writing ad-hoc runtime type checks becomes a ghost story that CS students tell around a campfire at night like `free` and `malloc`.

Sources

- Migratory Typing: Ten Years Later: Information about the implementation of Typed Racket.
- Typed-Untyped Interaction: Documentation about typed-untyped interaction in Typed Racket.
- Is sound gradual typing dead?: A study of Typed Racket's performance.
- Blame for all: A paper describing a gradual type system with dynamic sealing.
- Stack overflow developer survey 2021: TypeScript and JavaScript usage stats.