Gradual Typing

Goals (According to Jeremy Siek)

- Static and dynamic languages may be good for different tasks. Enable programmer to choose between static and dynamic typing without switching languages
- Provide a foundation for what optionally-typed languages mean (how type checking works).

Credit for content: Jeremy Siek (https://wphomes.soic.indiana.edu/jsiek/what-is-gradual-typing/)

Dynamic vs. Static Types

Static	Dynamic
Catches bugs earlier	But catch even more bugs with a thorough test suite
Guides changes when needed	Never gets in your way
Faster execution (avoid run time checks; store values more efficiently)	Address situations where type depends on runtime info
Improves modularity (if I call a function with an appropriate input, I expect no type errors later)	

Credit for content: Jeremy Siek (https://wphomes.soic.indiana.edu/jsiek/what-is-gradual-typing/)

Bad Arguments About Static vs. Dynamic Types

- Static type checking makes you think more carefully, preventing bugs.
  - But: type checkers only check fairly simple properties of your program. Tests are still critical!
- Dynamic type checking obviates the need for type annotations.
  - But: type inference works pretty well and writing annotations doesn't actually take that long.
    - Note: I'm not aware of any work that measures the annotation cost.

Credit for content: Jeremy Siek (https://wphomes.soic.indiana.edu/jsiek/what-is-gradual-typing/)

A Record and Subtyping Interlude

- (see whiteboard)

Gradual Typing

- What is the *gradual guarantee?* Siek et al.:
- The programmer should be able to conveniently evolve code from being statically typed to dynamically typed, and vice versa.
- When **removing** type annotations, a well-typed program will continue to be well-typed (with no need to insert explicit casts) and a correctly running program will continue to do so.
- When **adding** type annotations, if the program remains well typed, the only possible change in behavior is a trapped error due to a mistaken annotation.

Gradual Typing

- What i

e has a more precise type (T) than e' does (T')

e is more precise than e'

▶ **Theorem 5** (Gradual Guarantee). *Suppose* $e \sqsubseteq e'$ *and* $\vdash e : T$.

1. $\vdash e' : T'$ *and* $T \sqsubseteq T'$.
2. *If* $e \Downarrow v$, *then* $e' \Downarrow v'$ *and* $v \sqsubseteq v'$.
   If $e \Uparrow$ *then* $e' \Uparrow$.
3. *If* $e' \Downarrow v'$, *then* $e \Downarrow v$ *where* $v \sqsubseteq v'$, *or* $e \Downarrow \texttt{blame}_T\, l$.
   If $e' \Uparrow$, *then* $e \Uparrow$ *or* $e \Downarrow \texttt{blame}_T\, l$.

e evaluates to something more precise than e' does

if e runs forever, so does e'

-

if e' evaluates to v', either e evaluates to something more precise than v', or we have a bad cast at location l

TypeScript

- Can implicitly convert to and from the `any` type

```
var answer : string = "42";

var a : any = answer;

var the_answer : string = a;

document.body.innerHTML = the_answer;
```

These examples are from: http://siek.blogspot.com/2012/10/is-typescript-gradually-typed-part-1.html

TypeScript Checks Types of Non-`any` Expressions

- `var answer : number = 42;`
- `var the_answer : string = answer;` ERROR

Function Types Are Structural

```
function f(x:string):string { return x; }
var g : (any)=>string = f;
var h : any = g;
document.body.innerHTML = h("42");
```

Implicit downcast (violates contravariance)

Standard upcast

Cast from `any` to `(string)=>any`

Object Types Are Structural

```
var o : {x:number; y:any;} = {x:1, y:"42"};
var p : {x:any; y:string;} = o;
var q : {x:number;} = p;
var r : any = p;
document.body.innerHTML = r.y;
```

conversion to and from **any** are OK

access to y OK here even though it might not exist (statically)

No Implicit Widening

```
var o : {x:number; y:any; } = {x:1, y:"42"};
var q : {x:number;} = o;
var r : {x:number; y:any;} = q;
document
```

```
example.ts(3,29): Cannot convert '{ x: number; }'
    to '{ x: number; y: any; }':
Type '{ x: number; }' is missing property 'y'
    from type '{ x: number; y: any; }'
```

"Something Funny Happens at the Left of the Arrow" (John Reynolds)

Jonathan Turner:

"@Jeremy - Covariance in the argument type, while odd to those of us who work in type systems, can get in the way of how many programmers work. By being slightly more relaxed, there's less friction between expectations and the rules being enforced. I wasn't actually part of the call for this, so Anders would be a good person to go into it in more detail.

In general, the type system for TypeScript serves two purposes: early error detection and tooling. Because we allow working directly with JavaScript, there is no (and can be no) type safety guarantee. Instead, we aim for type utility - that the types you provide make sense in a way that helps you create software more easily and with fewer errors."

http://siek.blogspot.com/2012/10/is-typescript-gradually-typed-part-1.html

TS Is Unsound by Design

Non-goals

3. Apply a sound or "provably correct" type system. Instead, strike a balance between correctness and productivity.

https://github.com/Microsoft/TypeScript/wiki/TypeScript-Design-Goals

Other Sources of Unsoundness in TS

- `const xs = [0, 1, 2];  // type is number[]`
- `const x = xs[3];  // type is number, but at runtime x is "undefined"`

Note: the behavior is well-defined, unlike in C,

https://effectivetypescript.com/2021/05/06/unsoundness/

Other Sources of Unsoundness in TS

```
function alertNumber(x: number) {
  alert(x.toFixed(1));
}
const x1 = Math.random() || null;  // type is number | null
alertNumber(x1);
//          ~~ ... Type 'null' is not assignable to type 'number'.
alertNumber(x1 as number);  // type checks, but might blow up
```

Note: the behavior is well-defined, unlike in C,

https://effectivetypescript.com/2021/05/06/unsoundness/

Groups: Identify Questions About the Video

- Pick one question we should discuss as a group.

Tradeoffs

- Why might gradual typing help?
- In what cases?

The Gradual Guarantee

- Does it need to be a *guarantee*?
- Guarantee says that safety properties are enforced either statically or dynamically
- What if they're enforced statically or not at all?