Type Compatibility

Introduction

Type compatibility in TypeScript is based on structural subtyping. Structural typing is a way of relating types based solely on their members. This is in contrast with nominal typing. Consider the following code:

```
interface Named {
    name: string;
}

class Person {
    name: string;
}

let p: Named;
// OK, because of structural typing
p = new Person();
```

In nominally-typed languages like C# or Java, the equivalent code would be an error because the `Person` class does not explicitly describe itself as being an implementer of the `Named` interface.

TypeScript's structural type system was designed based on how JavaScript code is typically written. Because JavaScript widely uses anonymous objects like function expressions and object literals, it's much more natural to represent the kinds of relationships found in JavaScript libraries with a structural type system instead of a nominal one.

A Note on Soundness

TypeScript's type system allows certain operations that can't be known at compile-time to be safe. When a type system has this property, it is said to not be "sound". The places where TypeScript allows unsound behavior were carefully considered, and throughout this document we'll explain where these happen and the motivating scenarios behind them.

Starting out

The basic rule for TypeScript's structural type system is that `x` is compatible with `y` if `y` has at least the same members as `x`. For example:

```
interface Named {
    name: string;
}

let x: Named;
// y's inferred type is { name: string; location: string; }
let y = { name: "Alice", location: "Seattle" };
x = y;
```

To check whether `y` can be assigned to `x`, the compiler checks each property of `x` to find a corresponding compatible property in `y`. In this case, `y` must have a member called `name` that is a string. It does, so the assignment is allowed.

The same rule for assignment is used when checking function call arguments:

```
function greet(n: Named) {
    console.log("Hello, " + n.name);
}
greet(y); // OK
```

Note that `y` has an extra `location` property, but this does not create an error. Only members of the target type (`Named` in this case) are considered when checking for compatibility.

This comparison process proceeds recursively, exploring the type of each member and sub-member.

Comparing two functions

While comparing primitive types and object types is relatively straightforward, the question of what kinds of functions should be considered compatible is a bit more involved. Let's start with a basic example of two functions that differ only in their parameter lists:

```
let x = (a: number) => 0;
let y = (b: number, s: string) => 0;

y = x; // OK
x = y; // Error
```

To check if `x` is assignable to `y`, we first look at the parameter list. Each parameter in `x` must have a corresponding parameter in `y` with a compatible type. Note that the names of the parameters are not considered, only their types. In this case, every parameter of `x` has a corresponding compatible parameter in `y`, so the assignment is allowed.

The second assignment is an error, because `y` has a required second parameter that `x` does not have, so the assignment is disallowed.

You may be wondering why we allow 'discarding' parameters like in the example `y = x`. The reason for this assignment to be allowed is that ignoring extra function parameters is actually quite common in JavaScript. For example, `Array#forEach` provides three parameters to the callback function: the array element, its index, and the containing array. Nevertheless, it's very useful to provide a callback that only uses the first parameter:

```
let items = [1, 2, 3];

// Don't force these extra parameters
items.forEach((item, index, array) => console.log(item));

// Should be OK!
items.forEach(item => console.log(item));
```

Now let's look at how return types are treated, using two functions that differ only by their return type:

```
let x = () => ({name: "Alice"});
let y = () => ({name: "Alice", location: "Seattle"});

x = y; // OK
y = x; // Error, because x() lacks a location property
```

The type system enforces that the source function's return type be a subtype of the target type's return type.

Function Parameter Bivariance

When comparing the types of function parameters, assignment succeeds if either the source parameter is assignable to the target parameter, or vice versa. This is unsound because a caller might end up being given a function that takes a more specialized type, but invokes the function with a less specialized type. In practice, this sort of error is rare, and allowing this enables many common JavaScript patterns. A brief example:

```
enum EventType { Mouse, Keyboard }

interface Event { timestamp: number; }
interface MouseEvent extends Event { x: number; y: number }
interface KeyEvent extends Event { keyCode: number }

function listenEvent(eventType: EventType, handler: (n: Event) => void) {
    /* ... */
}

// Unsound, but useful and common
listenEvent(EventType.Mouse, (e: MouseEvent) => console.log(e.x + "," + e.y));

// Undesirable alternatives in presence of soundness
listenEvent(EventType.Mouse, (e: Event) => console.log((<MouseEvent>e).x + "," + (<MouseEvent>e).
y));
listenEvent(EventType.Mouse, <(e: Event) => void>((e: MouseEvent) => console.log(e.x + "," + e.
y)));

// Still disallowed (clear error). Type safety enforced for wholly incompatible types
listenEvent(EventType.Mouse, (e: number) => console.log(e));
```

Optional Parameters and Rest Parameters

When comparing functions for compatibility, optional and required parameters are interchangeable. Extra optional parameters of the source type are not an error, and optional parameters of the target type without corresponding parameters in the source type are not an error.

When a function has a rest parameter, it is treated as if it were an infinite series of optional parameters.

This is unsound from a type system perspective, but from a runtime point of view the idea of an optional parameter is generally not well-enforced since passing `undefined` in that position is equivalent for most functions.

The motivating example is the common pattern of a function that takes a callback and invokes it with some predictable (to the programmer) but unknown (to the type system) number of arguments:

```
function invokeLater(args: any[], callback: (...args: any[]) => void) {
    /* ... Invoke callback with 'args' ... */
}

// Unsound - invokeLater "might" provide any number of arguments
invokeLater([1, 2], (x, y) => console.log(x + ", " + y));

// Confusing (x and y are actually required) and undiscoverable
invokeLater([1, 2], (x?, y?) => console.log(x + ", " + y));
```

Functions with overloads

When a function has overloads, each overload in the source type must be matched by a compatible signature on the target type. This ensures that the target function can be called in all the same situations as the source function.

Enums

Enums are compatible with numbers, and numbers are compatible with enums. Enum values from different enum types are considered incompatible. For example,

```
enum Status { Ready, Waiting };
enum Color { Red, Blue, Green };

let status = Status.Ready;
status = Color.Green;  // Error
```

Classes

Classes work similarly to object literal types and interfaces with one exception: they have both a static and an instance type. When comparing two objects of a class type, only members of the instance are compared. Static members and constructors do not affect compatibility.

```
class Animal {
    feet: number;
    constructor(name: string, numFeet: number) { }
}

class Size {
    feet: number;
    constructor(numFeet: number) { }
}

let a: Animal;
let s: Size;

a = s;  // OK
s = a;  // OK
```

Private and protected members in classes

Private and protected members in a class affect their compatibility. When an instance of a class is checked for compatibility, if the target type contains a private member, then the source type must also contain a private member that originated from the same class. Likewise, the same applies for an instance with a protected member. This allows a class to be assignment compatible with its super class, but *not* with classes from a different inheritance hierarchy which otherwise have the same shape.

Generics

Because TypeScript is a structural type system, type parameters only affect the resulting type when consumed as part of the type of a member. For example,

```
interface Empty<T> {
}
let x: Empty<number>;
let y: Empty<string>;

x = y;  // OK, because y matches structure of x
```

In the above, `x` and `y` are compatible because their structures do not use the type argument in a differentiating way. Changing this example by adding a member to `Empty<T>` shows how this works:

```
interface NotEmpty<T> {
    data: T;
}
let x: NotEmpty<number>;
let y: NotEmpty<string>;

x = y;  // Error, because x and y are not compatible
```

In this way, a generic type that has its type arguments specified acts just like a non-generic type.

For generic types that do not have their type arguments specified, compatibility is checked by specifying `any` in place of all unspecified type arguments. The resulting types are then checked for compatibility, just as in the non-generic case.

For example,

```
let identity = function<T>(x: T): T {
    // ...
}

let reverse = function<U>(y: U): U {
    // ...
}

identity = reverse;  // OK, because (x: any) => any matches (y: any) => any
```

Advanced Topics

Subtype vs Assignment

So far, we've used "compatible", which is not a term defined in the language spec. In TypeScript, there are two kinds of compatibility: subtype and assignment. These differ only in that assignment extends subtype compatibility with rules to allow assignment to and from `any`, and to and from `enum` with corresponding numeric values.

Different places in the language use one of the two compatibility mechanisms, depending on the situation. For practical purposes, type compatibility is dictated by assignment compatibility, even in the cases of the `implements` and `extends` clauses.

For more information, see the TypeScript spec (https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md).