PropName : {
    < *TypeParamList*$_1$ > (*ParamList*$_1$) : *ReturnType*$_1$;
    < *TypeParamList*$_2$ > (*ParamList*$_2$) : *ReturnType*$_2$;
    …
    < *TypeParamList*$_n$ > (*ParamList*$_n$) : *ReturnType*$_n$; }

In the following example of an object type

```
{
    func1(x: number): number;          // Method signature
    func2: (x: number) => number;      // Function type literal
    func3: { (x: number): number };    // Object type literal
}
```

the properties 'func1', 'func2', and 'func3' are all of the same type, namely an object type with a single call signature taking a number and returning a number. Likewise, in the object type

```
{
    func4(x: number): number;
    func4(s: string): string;
    func5: {
        (x: number): number;
        (s: string): string;
    };
}
```

the properties 'func4' and 'func5' are of the same type, namely an object type with two call signatures taking and returning number and string respectively.

3.8 Type Relationships

Types in TypeScript have identity, subtype, supertype, and assignment compatibility relationships as defined in the following sections.

For purposes of determining type relationships, all object types appear to have the members of the 'Object' interface unless those members are hidden by members with the same name in the object types, and object types with one or more call or construct signatures appear to have the members of the 'Function' interface unless those members are hidden by members with the same name in the object types. Apparent types (section 3.8.1) that are object types appear to have these extra members as well.

3.8.1 Apparent Type

In certain contexts a type appears to have the characteristics of a related type called the type's ***apparent type***. Specifically, a type's apparent type is used when determining subtype, supertype, and assignment compatibility relationships, as well as in the type checking of property accesses (section 4.10), `new` operations (section 4.11), and function calls (section 4.12).