- *M* is a property and *S′* contains a property *N* where
  - *M* and *N* have the same name,
  - the type of *N* is assignable to that of *M*,
  - *M* and *N* are both public or both private, and
  - if *M* is a required property, *N* is also a required property.
- *M* is an optional property and *S′* contains no property of the same name as *M*.
- *M* is a non-specialized call or construct signature and *S′* contains a call or construct signature *N* where,
  - the signatures are of the same kind (call or construct),
  - the number of non-optional parameters in *N* is less than or equal to that of *M*,
  - *N* can be successfully instantiated in the context of *M* (section 3.8.5),
  - each parameter type in the instantiation of *N* is assignable to or from the corresponding parameter type in *M* for parameter positions that are present in both signatures, and
  - the result type of *M* is Void, or the result type of the instantiation of *N* is assignable to that of *M*.
- *M* is a string index signature of type *U* and *S′* contains a string index signature of a type that is assignable to *U*.
- *M* is a numeric index signature of type *U* and *S′* contains a string or numeric index signature of a type that is assignable to *U*.

When comparing call or construct signatures, parameter names are ignored and rest parameters correspond to an unbounded expansion of optional parameters of the rest parameter element type.

Note that specialized call and construct signatures (section 3.7.2.4) are not significant when determining assignment compatibility.

The assignment compatibility and subtyping rules differ only in that

- the Any type is assignable to, but not a subtype of, all types, and
- the primitive type Number is assignable to, but not a subtype of, all enum types.

The assignment compatibility rules imply that, when assigning values or passing parameters, optional properties must either be present and of a compatible type, or not be present at all. For example:

```
function foo(x: { id: number; name?: string; }) { }

foo({ id: 1234 });                // Ok
foo({ id: 1234, name: "hello" }); // Ok
foo({ id: 1234, name: false });   // Error, name of wrong type
foo({ name: "hello" });           // Error, id required but missing
```

### 3.8.5 Contextual Signature Instantiation

In sections 3.8.3 and 3.8.4, to determine whether a call or construct signature *A* is a subtype of or assignable to a call or construct signature *B*, *A* is ***instantiated in the context of*** *B*. If *A* is a non-generic