the best common type (section 3.10) of 'string' and 'number' is the empty object type, the call to 'choose' is equivalent to

```
var x = choose<{}>("Five", 5);
```

and the resulting type of 'x' is therefore the empty object type. Note that had both arguments been of type 'string' or 'number', 'x' would have been of that type.

In the example

```
function map<T, U>(a: T[], f: (x: T) => U): U[] {
    var result: U[] = [];
    for (var i = 0; i < a.length; i++) result.push(f(a[i]));
    return result;
}

var names = ["Peter", "Paul", "Mary"];
var lengths = map(names, s => s.length);
```

inferences for 'T' and 'U' in the call to 'map' are made as follows: For the first parameter, inferences are made from the type 'string[]' (the type of 'names') to the type 'T[]', inferring 'string' for 'T'. For the second parameter, inferential typing of the arrow expression 's => s.length' causes 'T' to become fixed such that the inferred type 'string' can be used for the parameter 's'. The return type of the arrow expression can then be determined, and inferences are made from the type '(s: string) => number' to the type '(x: T) => U', inferring 'number' for 'U'. Thus the call to 'map' is equivalent to

```
var lengths = map<string, number>(names, s => s.length);
```

and the resulting type of 'lengths' is therefore 'number[]'.

In the example

```
function zip<S, T, U>(x: S[], y: T[], combine: (x: S) => (y: T) => U): U[] {
    var len = Math.max(x.length, y.length);
    var result: U[] = [];
    for (var i = 0; i < len; i++) result.push(combine(x[i])(y[i]));
    return result;
}

var names = ["Peter", "Paul", "Mary"];
var ages = [7, 9, 12];
var pairs = zip(names, ages, s => n => ({ name: s, age: n }));
```

inferences for 'S', 'T' and 'U' in the call to 'zip' are made as follows: Using the first two parameters, inferences of 'string' for 'S' and 'number' for 'T' are made. For the third parameter, inferential typing of the outer arrow expression causes 'S' to become fixed such that the inferred type 'string' can be used for the parameter 's'. When a function expression is inferentially typed, its return expression(s) are also inferentially typed. Thus, the inner arrow function is inferentially typed, causing 'T' to become fixed such