that the inferred type 'number' can be used for the parameter 'n'. The return type of the inner arrow function can then be determined, which in turn determines the return type of the function returned from the outer arrow function, and inferences are made from the type '(s: string) => (n: number) => { name: string; age: number }' to the type '(x: S) => (y: T) => R', inferring '{ name: string; age: number }' for 'R'. Thus the call to 'zip' is equivalent to

```
var pairs = zip<string, number, { name: string; age: number }>(
    names, ages, s => n => ({ name: s, age: n }));
```

and the resulting type of 'pairs' is therefore '{ name: string; age: number }[]'.

4.12.3 Grammar Ambiguities

The inclusion of type arguments in the *Arguments* production (section 4.12) gives rise to certain ambiguities in the grammar for expressions. For example, the statement

```
f(g<A, B>(7));
```

could be interpreted as a call to 'f' with two arguments, 'g < A' and 'B > (7)'. Alternatively, it could be interpreted as a call to 'f' with one argument, which is a call to a generic function 'g' with two type arguments and one regular argument.

The grammar ambiguity is resolved as follows: In a context where one possible interpretation of a sequence of tokens is an *Arguments* production, if the initial sequence of tokens forms a syntactically correct *TypeArguments* production and is followed by a '(' token, then the sequence of tokens is processed an *Arguments* production, and any other possible interpretation is discarded. Otherwise, the sequence of tokens is not considered an *Arguments* production.

This rule means that the call to 'f' above is interpreted as a call with one argument, which is a call to a generic function 'g' with two type arguments and one regular argument. However, the statements

```
f(g < A, B > 7);

f(g < A, B > +(7));
```

are both interpreted as calls to 'f' with two arguments.

4.13 Type Assertions

TypeScript extends the JavaScript expression grammar with the ability to assert a type for an expression:

UnaryExpression: *(Modified)*
  …
  < *Type* > *UnaryExpression*

A type assertion expression consists of a type enclosed in < and > followed by a unary expression. Type assertion expressions are purely a compile-time construct. Type assertions are *not* checked at run-time