13

Anonymous classes and tuple types

13.1 OVERVIEW

Based on a bare-bones form of class —with no class names —, tuple types provide a concise and elegant solution to a number of issues:

- Writing functions with multiple results, treated in a way that is completely symmetric with multiple arguments.
- Describing sequences of values of heterogeneous types, or "tuples", such as [*some_integer*, *some_string*, *some_object*], convenient for example as arguments to printing routines.
- Achieving the effect of routines with a variable number of arguments.
- Achieving the effect of generic classes with a variable number of generic parameters.
- Using simple classes, defined by a few attributes and the corresponding **assign** procedures — similar to the "structures" (C) or "records" (Pascal, Ada) of non-O-O languages, but made compatible with O-O principles — without writing explicit class declarations.
- Making type-safe the *agent* mechanism through which you can handle routines as objects and define higher-order routines.

→ Agents are the topic of chapter 25.

Tuples are a mechanism for advanced users. If this is your first reading, you should probably skip this chapter.

13.2 ANONYMOUS CLASSES AND TUPLE TYPES IN A NUTSHELL

The object-oriented type system of Eiffel is based on classes, possibly equipped with generic parameters. Normally, you will give each class a name and write a class declaration ... **class** *CLASS_NAME* ... **end**, as seen in previous chapters. But in some simple cases it is convenient to define a class without a name, by merely listing its properties at the place where you need it. Such *anonymous classes* can only have elementary features: a few attributes and the corresponding field-setting procedures. Being anonymous, they require no specific class declaration: you just use a *tuple type* of the form

```
TUPLE [x1: T1; x2: T2; ... xn: Tn]
```

where the T_i are types. This has the same effect as if you were using a type based on an explicit — non-anonymous — class with *n* attributes of names *x1*, ..., *xn*, of the types given, and *n* corresponding procedures **assign** "*x1*", ..., **assign** "*xn*", used to set the field values, with no preconditions. But you don't need to find a name for a class, or write a class declaration. The class is implicitly defined by the tuple type.

A simple notation exists for describing instances of tuple types: just use

```
[v1, v2, ..., vn]
```

where *v1* is of type *T1* and so on. Here you don't need the labels (*x1* and so on). In fact you may omit them in the tuple type too, writing it as just *TUPLE* [*T1*; *T2*; ... *Tn*]; all you lose then is the notation *t*.*xi* to access fields of a tuple *t* (although you can still use *t*. *item* (*i*) which returns an *ANY*), and similarly for modifying fields (use *put*).

The conformance rule for tuple types also disregards the labels: *TUPLE* [*T*, *U*] conforms to *TUPLE* [*T*]; *TUPLE* [*T*, *U*, *V*] conforms to *TUPLE* [*T*, *U*]; and so on regardless of the presence of any labels.

Short as it is, the preceding description includes the essential properties of anonymous classes and tuple types. The rest of this chapter gives the details.

13.3 USING TUPLE TYPES AND TUPLES

The syntax of tuple types is straightforward:

```
        Tuple_type ≜ TUPLE [Tuple_parameters]
  Tuple_parameters ≜ "[" Entity_declaration_list "]"
```

← Tuple_type *is one of the variants of* Type, *introduced page* 234.

The notion of Entity_declaration_list was defined in the discussion of routines, since it also serves to define the formal arguments of a routine. Here is that syntax again:

```
  Entity_declaration_list ≜ {Entity_declaration_group ";" ...}
 Entity_declaration_group ≜ Identifier_list Type_mark
          Identifier_list ≜ Identifier "," ...}*
                Type_mark ≜ ":" Type
```

← *This first appeared on page* 144.

The syntax for Tuple_type permits examples such as:

```
[1]   TUPLE
[2]   TUPLE [INTEGER]
[3]   TUPLE [INTEGER; REAL; POLYGON]
[4]   TUPLE [i, j: INTEGER; r: REAL; p: POLYGON]
[5]   TUPLE [i: INTEGER; INTEGER; REAL; p: POLYGON]
[6]   TUPLE [i, j: INTEGER; lr: LIST [REAL];
          tpi: TUPLE [p: POLYGON; i: INTEGER]]
```

The type names appearing in brackets after *TUPLE*, if any, are called the **parameters** of the tuple type, by analogy with the actual generic parameters of a generically derived type *C* [*TYPE1*, *TYPE2*]. The optional names attached to individual parameters, such as *i*, *j*, *r* and *p* in example 4, are the **labels** of these parameters.

The syntactical analogy between tuple types and generically derived types is intentional, but note that the two categories are different; *TUPLE* is not a class name but a reserved word of the language.

Example 1 has no parameters. Examples 2 and 3 have parameters but no labels. In the remaining examples the parameters are labeled: *i*, *r* and so on. As shown by example 5, it is possible to mix labeled and unlabeled parameters, although this is not frequently useful. This example also illustrates that two or more parameters may be the same type.

Example 6 shows that the parameters can be arbitrary types, including (as in the example's last parameter) tuple types themselves.

A syntax rule brings more flexibility to the writing of tuple types:

Comma rule for tuple types

It is permitted to use commas instead of semicolons to separate the parameters of a tuple type.

This rule alllows you to write a type without labels as *TUPLE* [*X*, *Y*, *Z*], mostly for compatibility with earlier forms, and because it is not in the spirit of Eiffel to force people to remember small syntactic nuances. **The form with semicolons**, *TUPLE* [*X*; *Y*; *Z*] **remains the preferred style**, especially when labels are present, making the comma form look not too elegant (although it is legal from the above rule).

There is no specific validity constraint on tuple types, but the Entity Declaration Rule requires all the parameter labels to be different. This doesn't of course prevent a label for reappearing in the definition of a parameter that is itself a tuple type, the way *i* appears in the last parameter of example 6.

← *The Entity Declaration rule appeared on page 145.*

The intuitive semantics of tuple types is that they denote sequences of values, corresponding to the sequence of types given by the parameters. So the above examples have **direct instances** such as, respectively:

```
[ ]                     -- Empty tuple
[25]                    -- Tuple with one integer element
[25, –-8.75, pol]       -- With pol of type POLYGON
[25, 32, –-8.75, p1]
[25, 32, –-8.75, p1]    -- Same as previous one
[25, –-8.75, lr, [p1, 100]]-- With lr of type LIST [REAL]
```

These examples introduce the self-explanatory syntax for tuple expressions, listing between brackets a sequence of values, separated by commas (or, indifferently, semicolons). As the last example shows, tuple expressions can be nested.

→ On tuple expressions see “MANIFEST TU-PLES”, 13.6, page 281 below.

The first of our tuple type examples, *TUPLE* with no parameters, may not seem very useful since its only direct instance is the empty tuple [ ]. But throw in conformance and the picture changes. The conformance rule for tuples, previewed below, will state that *TUPLE* [*T1*, ..., T_n] always conforms to *TUPLE* [*T1*, ..., T_m] for $n >= m$ — labels, if present, playing no part here. So even though the *direct* instances of *TUPLE* [*T1*, ..., T_m] are sequences of exactly *m* values of the respective types given, the *instances* of this type include any sequence of *m* or more values, of which the first *m* have to conform to these types, the following ones being arbitrary. So the tuple [*25*, –-*8.75*, *pol*], given as a direct instance of *TUPLE* [*INTEGER*; *REAL*; *POLYGON*], is also an instance of *TUPLE* [*INTEGER*; *REAL*], of *TUPLE* [*INTEGER*], and of just *TUPLE*. More generally, **every tuple type** conforms to *TUPLE*, and every tuple expression, such as any of the examples above, is an instance of *TUPLE*, although not a direct instance.

→ “CONFORM-ANCE”, 13.5, page 280 below for the preview, “TUPLE TYPE CON-FORMANCE”, 14.10, page 297 for the full rule.

Such sequences of values, deriving from a tuple type, can be defined more precisely:

Value sequences associated with a tuple type

The **value sequences** associated with a tuple type *T* are sequences of values, each of the type appearing at the corresponding position in *T*’s type sequence.

based on an auxiliary notion of type sequence

Type sequence of a tuple type

The **type sequence** of a tuple type is the sequence of types obtained by listing its parameters, if any, in the order in which they appear, every labeled parameter being listed as many times as it has labels.

The type sequence for *TUPLE* is empty; the type sequence for *TUPLE* [*INTEGER*; *REAL*; *POLYGON*] is *INTEGER*, *REAL*, *POLYGON*; the type sequence for *TUPLE* [*i*, *j*: *INTEGER*; *r*: *REAL*; *p*: *POLYGON*] t is *INTEGER*, *INTEGER*, *REAL*, *POLYGON*, where *INTEGER* appears twice because of the two labels *i*, *j*.

As you will have noted, parameter labels play no role in the tuple type properties seen so far, in particular conformance. They also never intervene in tuple expressions (such as [*25*, *–-8.75*, *pol*]). Their only use, as discussed below, will be to enable us to access by name the fields of tuple objects.

13.4 ANONYMOUS CLASSES

To capture the precise semantics of tuple types we must realize that the run-time values they describe, tuples, are not just sequences of values — which wouldn't have a clear place in among the objects of a system's execution — but perfectly ordinary objects.

Like all other objects, these tuples are instances of classes; the only difference is that you don't have to write these classes. Instead, any tuple type implicitly defines a class, said to be *anonymous*. An anonymous class has the following features:

- *count*: *INTEGER*, the tuple size (number of values in a direct instance of the tuple type).
- *item* (*i*: *INTEGER*): *ANY*, returning the value of the *i*-th item of a tuple, for *i* between 1 and *count*.
- *put* (*x*: *ANY*; *i*: *INTEGER*), to change the value of the *i*-th item, applicable only if *x* denotes a value whose type conforms to the type of the corresponding parameter.
- For any label *l* associated with a parameter *T*, an attribute *l* of type *T*, returning the corresponding tuple item.
- For any such label, an assignment procedure **assign** "*l*", to assign values to the corresponding tuple item.

So with the declarations

```
tuple4: TUPLE [i: INTEGER; INTEGER; REAL; p: POLYGON]
c, n: INTEGER
poly1, poly2, poly3, poly4, poly5: POLYGON
x: SOME_TYPE
```

you can perform the following instructions:

```
tuple4 := [25, –2, r, poly1]
c := tuple4.count          -- Assigns value 4 to c
poly2 ?= tuple4.item (4)   -- Assigns value of poly1 to poly2, but
                           -- note need for assignment attempt
                           -- since item returns an ANY
tuple4.put (poly3, 4)      -- Succeeds in replacing last item by poly3
                           -- since poly3 is of type POLYGON
tuple4.put (x, 4)          -- If SOME_TYPE does not conform to
                           -- POLYGON, makes last item void
n := tuple4.i              -- Assigns value 25 to n
poly4 := tuple4.p          -- Assigns last item's value, poly3, to
                           -- poly4; no need for assignment attempt
tuple4.p |= poly5          -- Replaces last item by poly5
```

Feature *item* and *put* both use an integer argument *i* indicating the position of the tuple element to be accessed or replaced. Because *i* is variable, these features can use no type more precise than *ANY* as the type of *item*'s result and *put*'s first argument:

```
item (i: INTEGER): ANY
        -- i-th item of tuple
put (v: ANY; i: INTEGER)
        -- Replace i-th element of tuple by v.
```

As a result, any practical use of *ANY* will need to perform an assignment attempt on the result, like the assignment to *poly2* in the third example instruction above.

For an unlabeled parameter, you cannot do any better. But this is where the labels, if present, come in handy: they are treated as attributes with the associated assignment procedures. The last three examples illustrate this. Our example type *TUPLE* [*i*: *INTEGER*; *INTEGER*; *REAL*; *p*: *POLYGON*] is equivalent to a class with four attributes, of which the first and last have been given names *i* and *p*, of respective types *INTEGER* and *POLYGON*. This is how we can access the corresponding tuple items *tuple4*.*i* and *tuple4*.*p*, with the exact types — not just *ANY*, and without assignment attempt. Similarly we can perform procedure assignments, such as the last example instruction *tuple4*.*p* |= *poly5*, with exact type checking.

Here is the precise definition of the anonymous classes associated with tuple types. (You may skip the rest of this section on first reading.) The only one of these classes that your software can explicitly use is *ANONYMOUS*. Eiffel compilers and tools are not required to build the other classes explicitly; but the way they handle tuple types must be the same as if the classes were actually present.

The definition proceeds by induction on the number of parameters.

Type *TUPLE*, with no parameters, is considered as an abbreviation for *ANONYMOUS*, the name of a Kernel Library class whose features have been listed above: *count*, *item* and so on. The precise specification of the class appears in the Kernel library chapter.

→“ANONYMOUS”, A.6.17 CLASS, page 819.

Then type *TUPLE* [l_1: *X1*; ...; l_n: X_n], with $n \geq 1$, is defined by induction as an abbreviation for a class

```
class ANONYMOUS_n inherit
        TUPLE [l_1: X1; ...; l_n-1: X_n-1]
feature -- Access
        l_n: X_n assign
invariant
        large_enough: count >= n
end
```

without the **feature** clause if there is no label for the last parameter *Xn*. In that case, as noted, there is no way to access and modify the *n*-th component of a corresponding tuple other than through *item* and *put*, which treat the item as being of type *ANY*.

13.5 CONFORMANCE

TUPLE [l_1: X_1; ...; l_n: X_n], with or without labels, conforms to the following types (the precise wording of the rule will appear in the chapter on conformance):

→ "TUPLE TYPE CONFORMANCE", 14.10, page 297 .

1 • Any other tuple type with the same parameters X_1, ..., X_n. (In other words: the labels don't matter at all for conformance.)

2 • Any other tuple type *TUPLE* [t_1: X_1; ...; t_m: X_m], again with some or all of the labels possibly absent, for $m \leq n$. (So *TUPLE* [*X*] conforms to *TUPLE*; *TUPLE* [*X*; *Y*] conforms to *TUPLE* [*X*], and so on.)

3 • Any array type *ARRAY* [*T*] such that every one of the X_i conforms to *T*. (This allows us to treat tuples as arrays if we want to.)

A mathematical note will be of interest to the curious reader (non-mathematical readers should skip to the following section). The rule in case 2 may seem to run counter to mathematical intuition. We are used to consider that, if *U* conforms to *T*, the instances of *U* form a subset of the instances of *T*. Case 2 then doesn't seem right if you think of *TUPLE* [*X*; *Y*] as representing the mathematical set $X \times Y$ — the cartesian product of the sets represented by *X* and *Y*. The rule states that *TUPLE* [*X*; *Y*] conforms to *TUPLE* [*X*], but we do not normally think of $X \times Y$ as a subset of *X* (or $X \times Y \times Z$ as a subset of *X x Y*, and so on). The trivial relation is the other way around: *X* is in one-to-one correspondence with a subset of *X x Y* (the subset made of pairs of the form [*x*, *y0*] for some arbitrary element *y0* of *Y*).

To remove this apparent paradox, it suffices to use another model than cartesian product. Consider *TUPLE* [X_1; ...; X_n] as modeling a set T_n of partial functions from **N** (the set of natural integers) to the set *X* of all possible objects. T_n is the set of all such functions *f* whose domain includes the interval *1*, *2*, ..., *n*. Any tuple may be viewed as such a function; for example, the tuple [*a*, *b*, *c*] is the function *f* whose domain only includes the integers *1*, *2* and *3*, such that $f(1) = a$, $f(2) = b$ and $f(3) = c$.

With this interpretation the subsetting relation and the subtyping (conformance) relation do coincide: *TUPLE* [X_1; ...; X_{n+1}] represents the set of functions whose domain includes *1*, *2*, ..., $n+1$; this is a superset of *TUPLE* [X_1; ...; X_n], containing the functions whose domain includes *1*, *2*, ..., *n*.

13.6 MANIFEST TUPLES

An expression of tuple type can be a **manifest tuple**, of the form

```
[a, b, ...]
```

where *a*, *b*, ... are arbitrary expressions. This is of type *TUPLE* [*TA*, *TB*, ...] where *TA* is the type of *a*, *TB* the type of *b* and so on.

One application of manifest tuples was seen in the discussion of routines: providing a routine with what amounts to a variable number of arguments. In such a case — for example to write a printing routine that can print any number of values — just give the routine an argument of type *TUPLE* [*T*] where *T* covers all the possible types of acceptable values. If the values are of arbitrary types, use *TUPLE* [*ANY*]; then the routine might need to perform assignment attempts on the tuple items to treat them in the proper way according to their exact types.

"USING A VARIABLE NUMBER OF ARGUMENTS", 8.4, page 145.

Another application comes from case 3 of the conformance properties of the previous section, which specifies that *TUPLE* [*TA*, *TB*, ...] conforms to *ARRAY* [*T*] if everyone of the types *TA*, *TB*, ... conforms to *T*. This means that you can also treat a manifest tuple as a **manifest array**, enabling you to initialize an array by giving the list of its elements, as in

```
first_primes := [1, 2, 3, 5, 7, 11, 13, 17]
```

with *first_primes* of type *ARRAY* [*INTEGER*]. Similarly, if *some_routine* takes a formal argument of type *ARRAY* [*T*], you can call it as

```
some_routine ([t1, t2, t3])
```

where every *ti* is of a type conforming to *T*.

The syntax of manifest tuples is straightforward:

Manifest_tuple ≜ "**[**" Expression_list "**]**"

Expression_list ≜ {Expression "," …}*

For consistency with tuple type declarations, you can use semicolons rather than commas:

Semicolon rule for manifest tuples

It is permitted to use semicolons instead of commas to separate the items of a manifest tuple.

For manifest tuples the **form with commas remains the preferred style**, for compatibility with usual practices in writing routine calls *rout* (*a*, *b*, ...).

13.7 WRITING FUNCTIONS WITH MULTIPLE RESULTS

The mention of routine calls in the last observation is not accidental. The presence of tuples allows us to handle routine arguments and routine results in a consistent way. Most programming languages treat these two categories non-symmetrically: a routine has zero or more arguments, but it has either no result if it is a procedure, or, if it is a function, exactly one result. To achieve the equivalent of a routine with multiple results you must write a function that returns a — single — result that happens to be a complex object. This is sometimes inconvenient.

The rule in Eiffel is simpler. Conceptually, *every routine has a single argument and a single result, both of which are tuples*. This covers all the special cases:

- Procedure: the result tuple is empty.
- Routine with no argument: the argument tuple is empty.
- Function with multiple results: the result tuple has more than one element.

Without the notion of tuple, we would need to address the last case by introducing a class to represent the result type, with attributes representing the components of the result, and procedures to set these attributes. This technique works (as illustrated by the practice of Eiffel prior to Eiffel 5), and it is justified if the new class covers a valuable abstraction. But if not — if the class is just an artefact with no other usefulness — it is heavy. In such a case you will want the class to be *anonymous*, as provided by tuples.

A typical example is a division routine returning both the quotient and the remainder of a number (or other divisible object) by another. You may write its header (for a suitable *NUMBER* type) as

```
infix "/" (other: NUMBER):
        TUPLE [quotient: NUMBER; remainder: NUMBER] is
```

and, in the body of the routine, have assignments of the form

```
Result.quotient |= ...
Result.remainder |= ...
```

Then you can use the function with

```
div := number1 / number2
```

and access the two components of the result through *div.quotient* and *div.remainder*.

Beyond being conceptually satisfying, this symmetric way of handling arguments and results permits a significantly simpler and more concise writing style.

Note that a manifest tuple is an expression, not a Writable; so an assignment of the form [*a*, *b*, *c*] := [1, 2, 3], or here [*a*, *b*, *c*] := *number1* / *number2*, would be syntactically invalid . The target of an assignment must be a single entity, which of course may be of a tuple type such as *Result* and *div* in the above example.