# Type conversions in an O-O language with inheritance

***Bertrand Meyer***

**Pullout quote:**

**A type may conform or convert to another, but not both.**

We expect, in programming languages, that certain conversions will occur silently and automatically. The most common case, permitted by almost all languages, is assigning an integer value to a real variable, which we expect to work, with the effect that it will convert the integer to the closest real equivalent. We also accept mixed-type arithmetic expressions, such as *your_real_number* + *my_integer*, with the understanding that the result will be of the "heavier" type, real. Most of the time these possibilities of mixed-type assignment and mixed-type arithmetic only apply to a few basic types (integer, real, double, possibly boolean) and cannot be generalized to others, whereas it would often be nice to do the same thing for some user-defined types; another criticism is that these rules are *ad hoc*, breaking the simplicity and regularity of language design. This is particularly true in an object-oriented language where we have another mechanism for assigning between two different types: polymorphic assignment or argument passing. We end up accepting both

*your_polygon* := *your_rectangle*

and

*my_real* := *your_integer*

but with very different semantics: in the first case, assuming the corresponding types are *RECTANGLE* and *POLYGON*, inheriting from each other, the assignment simply reattaches a reference, without any change to any object; the second case causes a change of values (formally, objects of simple types). This suffices to show that the first intuition — making *INTEGER* inherit from *REAL* —, although natural enough for someone who looks at the issue of conversions from an O-O background, is quite wrong. As this article will (I think) clearly demonstrate, we must instead accept that the two mechanisms are not just different but incompatible, and in fact take great pains to ensure that they can never be both applicable in a given situation.

I will describe the work that has recently been performed to integrate conversions into the fabric of a strongly typed object-oriented language, Eiffel, leading to a new language mechanism that addresses the issues just raised: permitting mixed-type assignments and mixed-type argument passing as well as mixed-type arithmetic, with the validity and semantic rules that correspond to the practice of mathematics and of traditional programming languages, while remaining compatible with the letter and spirit of object-oriented principles, keeping predefined types fully within the confines of the object-oriented type system, avoiding any conflict with inheritance, and making the mechanism applicable not only to predefined types but also to arbitrary programmer-defined types.

## A word of warning

In this Eiffel column we usually describe existing realizations in Eiffel and stay away from speculative discussions. Only once did I present a mechanism that had not been fully implemented; this was when my co-authors and I first discussed [1] the agent mechanism, which is now operational (and, not surprisingly, a little different in its details from that original description, although the general features are essentially as given then). The present article is our first real exception to this rule, since the proposal described here is, for the moment, a paper design.

This is of course a bit risky. In early September I was privileged to attend the Joint Modular Languages Conference in Zurich, which turned into a farewell symposium for the recently retired Niklaus Wirth and an homage to his language series (Pascal, Modula, Oberon). A book was published for the occasion [2]. We were reminded more than once, by the conference and by the book, of the Wirth school's principle that you shouldn't propose a language construct until you have a running implementation for it. In the evolution of Eiffel we have almost always followed this rule, with the ISE compiler usually serving as the testbed for new ideas. But of course the successful integration of a proposed extension into a compiler, while indeed a necessary condition of its final acceptance, is by no means a guarantee that the extension is good; history has shown over and again that a language feature may be compilable but still imperfect or even logically flawed. Language design and evolution is inevitably a back-and-forth process of validating proposed ideas through discussion and through pilot implementations. In the present case I think that it's OK to start with the discussion. Just accept that there is a big CAUTION sign at the entrance to that discussion: the ideas presented here have not been implemented, and until that happens you should look at them with the consequent dose of skepticism.

## Getting rid of the Balancing Rule

We are looking for a mechanism that will clarify and generalize the implicit conversions that exist between basic types, giving them a clear place in the class system of the language. This is largely a theoretical cleanup effort; no fundamental change will result for the practicing programmer, but we will have gone one step further towards the goal of providing the language with a simple, general set of mechanisms, whereby all types — including basic types such as *INTEGER* and the like — are defined by classes, with little or no privileges or special rules. Eiffel goes much further than other typed O-O languages in treating predefined types as normal classes (while ensuring optimal execution performance by letting the compiler know about these types); the small extension described here removes any remaining special theoretical treatment.

The remaining special treatment in current Eiffel [3] is a set of rules that govern conversions between basic types, providing exceptions to the general conformance requirements. Conformance of *b* to *a*, which governs the validity of the assignment *a* := *b*, or of argument passing if *a* is the formal argument of a routine, basically states that the type of *b* must be a descendant of the type of *a* in the inheritance hierarchy, with appropriate adaptations for generic classes. For the basic types, there are special rules implying that *INTEGER* conforms to *REAL* and *DOUBLE*, and *REAL* conforms to *DOUBLE*, with the understanding that a corresponding assignment or argument passing will cause conversions.

In addition the language needs a special “Balancing Rule” to address mixed-type arithmetic. This is because, in the strict object-oriented world of Eiffel, we view an expression of the form *a* + *b*, theoretically at least, as an abbreviation for *a*.*infix_plus* (*b*), where *infix_plus* would be a normal, non-operator function taking an argument. In fact, such a function does exist; it is called **infix** "+". If *a* is of type *REAL* and *b* of type *INTEGER*, things work as expected since *b* is formally the argument to **infix** "+" and its type, *INTEGER*, conforms to the expected type for the argument of function **infix** "+" in class *REAL*: the type *REAL*. But if we reverse the roles this doesn’t work any more, since the function **infix** "+" of *INTEGER* would expect an integer argument, so a real doesn’t conform. Viewed strictly from the perspective of object-oriented principles this situation would be justifiable, but it conflicts with the usual conventions of mathematical notation. Although + on ordinary arithmetic types is commutative — that is to say, *a* + *b* = *b* + *a* — we don’t want to force programmers to rewrite *a* + *b* as *b* + *a* in some cases, depending on the types of the operands. For that reason, the Balancing Rule states that in the evaluation of such a mixed-type arithmetic expression the first step is to convert all operands to the heaviest one, where *DOUBLE* is heavier than *REAL* and *REAL* than *INTEGER*. This works, but at the price of an ad hoc rule, applicable only to the basic arithmetic types. The framework described in this article will remove the need for the Balancing Rule.

## The inheritance-conversion exclusion principle

An important guideline in devising a conversion mechanism is that it should not conflict with inheritance. As noted at the very beginning, polymorphic assignment, controlled by inheritance in a typed O-O language, is a simple reference reattachment that doesn’t involve any change of the referenced values; in contrast, conversions transforms a value of a type into a value of another type. The mechanisms are completely disjoint and we must be careful to avoid any confusion (lest we run into the trouble caused, for example, by the conflict between overloading and dynamic binding in languages supporting both).

Hence the following rule, which provides the basic constraint on our language design for convertibility:

**Conformance-conversion exclusion principle**

A type may not both conform and convert to another.

Conformance, as used in this rule, is the property that holds between two types when values of one may be polymorphically attached to another. As noted, this essentially means that the conforming type is a descendant of the other in the inheritance structure, with some provisions for generically derived types (for example *LINNKED_LIST* [*EMPLOYEE*] conforms to *LIST* [*PERSON*] if *EMPLOYEE* is a descendant of *PERSON* and *LINKED_LIST* of *LIST*).

## Conversion basics

Based on the preceding observations we are now ready to devise the conversion mechanism. Here is a simple example:

```
expanded class DOUBLE inherit ¼ create
        from_integer convert {INTEGER},
        from_real convert {REAL},
        … Other creation procedures …
feature -- Initialization
        from_integer (n: INTEGER) is
                        -- Initialize by converting from n.
                do
                        … Conversion algorithm …
                end
        … Similarly for from_real …
        … Rest of class omitted …
end -- class DOUBLE
```

The only new possibility here is, when you list a creation procedure (constructor), to specify that it also serves to **convert** from other types (it will then be called a **conversion procedure**). Indeed that's what a conversion is from an O-O viewpoint: creating an object that you initialize from an object of some other type. We have to specify the creation mechanism; this is done simply through a procedure. A creation procedure that is equipped

The effect of declaring such a conversion procedure, with an argument of type *D*, in a class *C*, is to permit attachments (assignments or argument passing) from *D* to *C*. Such an attachment will apply the corresponding creation procedure, such as *from_integer*. In cases such as *DOUBLE*, the target class is expanded, so that no object will be created; for a reference class, the operation might create a new object. (Expanded types, similar to what is known in C# as value types, are used for variables which directly denote objects, rather than references to objects; this is useful in particular for basic types. Two different expanded types never conform to each other, even if one inherits from the other.)

For basic types such as *DOUBLE*, we will of course continue, as today, to let the compiler cheat and apply conversions directly through built-in knowledge; we do not want to incur the overhead of a procedure call in an assignment *my_double* := *your_integer*. But the class text will look as above, legitimizing such assignments, and you may apply a similar scheme to any of your own programmer-defined classes. The special conformance rules (*INTEGER* to *REAL* etc.) go away; in fact, the conformance-conversion exclusion principle explicitly implies that a type that converts to another may not conform to it.

Here indeed are the validity rules that govern the introduction of conversion procedures into a class, as in this example:

- Every conversion procedure must have a single argument.
- The type of that argument must not conform (in the sense of inheritance, as defined above) to the current type (*DOUB* LE in the example).
- The argument types associated with two different conversion procedures must be different.

The second rule achieves the conformance-conversion exclusion principle; the third rule avoids any ambiguity between different convertible types.

## Other examples

As a non-arithmetic example, a class *DATE* could start with

```
class DATE create
        from_tuple convert {TUPLE [INTEGER, INTEGER, INTEGER]}
…
```

allowing assignments of the form *my_date* := [*day*, *month*, *year*] where the source is a tuple. (*TUPLE* introduces tuple types with a variable number of fields.)

An example that combines inheritance with conversion involves the recently introduced (and implemented!) specific-size integer *INTEGER_8* and so on. The inheritance hierarchy is shown in figure 1.

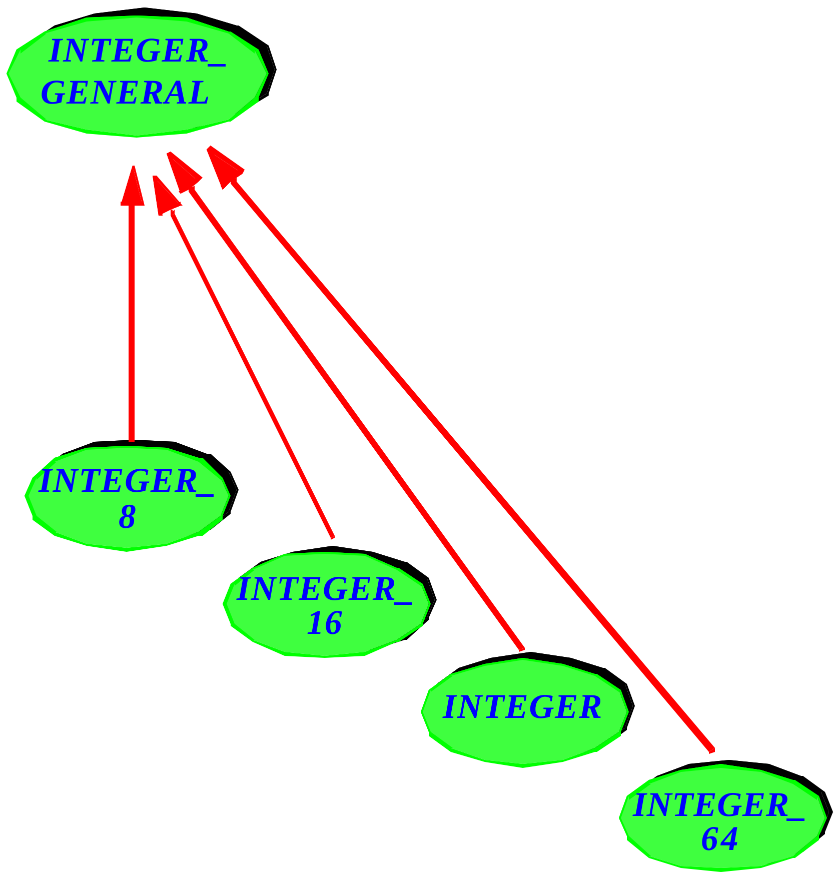

Figure 1: Integer class hierarchy

Class *INTEGER_GENERAL* will allow conversions:

```
create
        from_integer convert
                {INTEGER_8, INTEGER_16,
                INTEGER, INTEGER_64}
        … Other creation procedures …
feature -- Initialization
        from_integer (n: INTEGER_GENERAL) is
                        -- Set from n
                do … Conversion algorithm … end
        ¼ Rest of class omitted …
```

Note how a given conversion procedure may apply to several source types. There is no conflict between convertibility and inheritance here, since the types involved are expanded, and hence may not conform to each other.

## Mixed-type expressions

The conversion mechanism as described takes care of most of what we need, but not yet of ensuring full compatibility with mathematical tradition for mixed-type expressions. More precisely, *my_double* + *your_integer* will work, being understood as

```
my_double.infix "+" (your_integer)
```

which is OK since *DOUBLE* now has a conversion procedure for *INTEGER*, so it will convert its argument to a double. But this doesn't work for *your_integer* + *my_double*. We need a cleaner and more general replacement for the Balancing Rule.

It would be quite wrong to handle this issue by introducing overloading into the language. We must retain the basic simplicity rule of object-oriented programming that, within a class, a feature name denotes exactly one feature. What we need is not overloading (in fact rather irrelevant here) but a rule that will, in some cases, force **infix** "+" to reverse its arguments and call another routine instead.

Here is the syntax that achieves this. In the current Eiffel Kernel Library, **infix** "+" is declared inclass *INTEGER* as

```
infix "+"  (other: INTEGER): INTEGER is
                        -- Sum of current integer and other
    do
            … Implementation of integer addition …
    end
```

Similar (but unrelated) declarations appear for **infix** "+" in classes *REAL* and *DOUBLE*, and any other class that needs an infix "plus" operation.

We adapt this definition by adding a **convert** clause:

```
infix "+"  convert {REAL, DOUBLE}
            (other: INTEGER): INTEGER is
                        -- Sum of current integer and other
    do
            … Implementation of integer addition …
    end
```

This means that apart from its normal argument, an integer, **infix** "+" will also accept a *REAL* or a *DOUBLE*, but in this case it will:

- Convert the first operand (*a* in *a* + *b*, i.e. the target of the formally associated call *a*.**infix** "+" (*b*)) to the corresponding type, here *REAL* or *DOUBLE*.

- **Ignore the body of the routine** as given in the current class, here *INTEGER*..
- Instead, call the function with the same name in the corresponding type, here *REAL* or *DOUBLE*.

The validity constraints are obvious: the current type, here *INTEGER*, must be convertible to all the listed types, here *REAL* and *DOUBLE*; they must all have the appropriate function; and they must neither convert nor conform to the argument type listed for the given function, here *INTEGER* (the type of *other*), as this would cause an ambiguity.

This achieves, in a simple way, the desired goal of providing us with a completely general mechanism avoiding the need for special rules such as the Balancing Rule, and avoiding any ambiguity.

## An assessment

One may debate how broadly it is wise to use the facilities described in this article beyond the case of predefined arithmetic types. The case of dates and tuples seems appropriate; but as soon as things become more complicated you should use explicit conversions. For example, to produce a linked list form of some structure, it is just as clear and simple to write *my_list* := *your_structure*.*linear_representation*, using a function *linear_representation* that the EiffelBase library indeed provides for all applicable structures. Implicit conversion in such a case doesn't seem particularly necessary.

But even if the mechanism described here only serves to provide a clean language framework for the common operations on standard arithmetic types, it will have justified its design — at least when it's implemented and available.

## References

[1] Paul Dubois, Mark Howard, Bertrand Meyer, Michael Schweitzer and Emmanuel Stapf: *From Calls to Agents*, in *Journal of Object-Oriented Programming*, vol. 12, no. 6, September 1999.

[2] *The School of Niklaus Wirthl*, ed. László Böszörményi, Jürg Gutknecht and Gustave Pomberger, dpunkt.verlag, 2000

[3] Bertrand Meyer: *Eiffel: The Language*, Prentice Hall, 1991.