# Eiffel as a functional language

Bertrand Meyer

25 September 2012
Revised 25 October 2012 and 23 January 2013
Notes added, 25 January 2014
Corrections by Alexander Kogtenkov, 28 January 2014

Eiffel Software

ETH
Chair of
Software Engineering

ИТМО
САНКТ-ПЕТЕРБУРГ

# Note

This is a working document describing plans for Eiffel extensions

It represents no commitment on the part of the author or Eiffel Software

Any language extensions have to be approved by the Ecma International TC49-TG4 (the committee in charge of Eiffel language standardization)

# The issue

Eiffel can be used as a functional language, embedded in an O-O one, thanks in particular to **agents**

Many of the fancy features that people excitedly talk about in functional languages tend either to have simple counterparts in Eiffel (thanks in particular to multiple inheritance) or to have little value

- Example: "traits"

**However**:

- Expressing some common program patterns, especially functional, can be wordy, sometimes very wordy

**The goal of this proposal is to remove the wordiness**

# Plan

1. Sources of wordiness: an analysis
2. Library and language extensions
3. Examples
4. Semantic specification of the contextual type mechanism

# - 1 -

# Sources of wordiness: an analysis

# Sources of wordiness

1. Keyword-based style
2. Limited expression sublanguage, e.g. no conditional expressions
3. Wordiness of calling a function agent
4. Explicit typing
5. No generic features

The next slides show a wordy example, then examine these points in turn

# An example of wordiness

(From Nadia Polikarpova, for EiffelBase 2)

Part of a postcondition clause:

```
(map.domain / upper).for_all
    (agent
        (i: INTEGER;
        o: PREDICATE [ANY, TUPLE [G, G]]):
            BOOLEAN
    do
        Result := o.item ([map [i], map [i + 1]])
    end (?, order))
```

In fact this example uses an older Eiffel style; in current Eiffel it would use the **across** syntax, which makes it clearer and more concise, but we keep it as an extreme (if exaggerated) example of a complex functional expression

# Problem 1: keyword-based syntax

Eiffel is keyword-oriented, not symbol-oriented, e.g.

**do** ... **end**

rather than

{...}

I did play with more symbol-oriented syntax, then realized keywords are not the problem, especially carefully chosen, short keywords like **do** and **end**

Earlier attempts at more symbol-oriented syntax, e.g. the infamous !! for creation, were generally rejected by the community

In any case, keywords will be needed, and (for example) ***agent*** is shorter than **lambda**

My conclusion: the general syntactic style of Eiffel should remain what it is. The serious sources of wordiness are elsewhere

# Problem 2: limited expression sublanguage

```
abs (x: INTEGER): INTEGER
    do
        if x < 0 then Result := -x else Result := x end
    end
```

(and I am using condensed indentation!)

There are two issues here:

- Need to assign to **Result**; introduces an imperative element (assignment) for a strictly applicative need
- No conditional expressions

Of course we should not renounce **Result**, one of the great innovations of Eiffel, which avoids the awful **return** instruction and whose absence is sorely missed in contract extensions, e.g. .NET languages (see its awkward introduction in Code Contracts)

But we need to make **Result** implicit in simple cases

# Problem 3: calling function agents

This is a simple problem, easier than the others

In the example, why do we have to write

```
o.item ([map [i], map [i + 1]])
```

(where o is a predicate), instead of just calling o on the two arguments?

The good news: a trivial library extension, complemented by a simple language extension, both a few slides away, make it possible to write this more concisely and clearly

# Problem 4: explicit typing

Eiffel is statically typed, with many benefits!

Unlike some others, Eiffel programmers actually *like* declaring stuff; it makes the program readable by expressing the intent behind every entity

But explicit declarations become tedious under combination of

- Genericity, especially classes with several generic parameters
- Agents (whose classes indeed have many parameters), especially inline agents

Can we make typing explicit in such cases?

Does this require some kind of Hindley-Milner type inference? (Maybe not)

# Our example again

```
(map.domain / upper).for_all
    (agent
        (i: INTEGER;
        o: PREDICATE [ANY, TUPLE [G, G]]):
            BOOLEAN
    do
        Result := o.item ([map [i], map [i + 1]])
    end (?, order))
```

# Problem 5: generic features

Eiffel does not have generic features, only generic classes

Although generally OK, this absence can limit expressiveness

Typical example: cannot add function composition to FUNCTION* since (in FUNCTION [C, ARGS, RES] ) it would have the signature

```
compose alias "@"
        (other: FUNCTION [ANY, TUPLE [RES], ?]):
                FUNCTION [ANY, ARGS, ?]
```

but we cannot express the type marked "?"

We could use ANY but then the typing becomes dynamic, and we certainly do not want to forsake full static typing

** But wait!*

# Function composition: in fact...

... it **is** possible to declare "@" in FUNCTION, if we change FUNCTION to have one more generic parameter **X**; the signature will be

```
compose alias "@"
        (other: FUNCTION [ANY, TUPLE [RES], X]):
            FUNCTION [ANY, ARGS, X]
```

This approach works in the current language, but like the previous one it is too tedious to scale up to systematic usage

It does, however, provide a hint towards a usable solution

# Problem 6

Passing arguments to an agent call requires brackets:

- a.call ([x, y, z])
- a.call ([])
- f.item ([x, y, z])

It is possible to get rid of the brackets (suggestion by Simon Peyton-Jones at LASER, now implemented, see in later slide)

# - 3 -

# Library & language extensions

# Principles

1. Apply general Eiffel design principles (static typing, consistency, one good way to do anything etc.)
2. Do not unreasonably increase the size of the language
3. Since this is an expressiveness discussion, avoid defining new semantics in favor of providing new notations for existing mechanism, following the tradition of Eiffel's "**unfolded forms**"

Indeed there is no semantic extension in what follows: it is all about **abbreviations** of existing mechanisms

This property follows from our liminal observation that the matter is not expressiveness but concision

# Extension A (library)

Give item, in class FUNCTION (and hence PREDICATE) the bracket alias

This trivial change can be carried out immediately, without any obvious drawback (e.g. compatibility)

It solves Problem 3: in the example

```
o.item ([map [i], map [i + 1]])
```

becomes just

```
o [[map [i], map [i + 1]]]
```

The brackets remain, but the expression is simpler & clearer

January 2014 note: this extension is no longer so interesting since extension D (parenthesis alias) goes further

# Extension B : omit **Result** in queries

In a query or a query agent, as shorthand for

**do** instructions ; **Result** := expression **end**

allow writing just

**do** instructions **then** expression **end**

In the absence of instructions, the "**do**" part is not needed, so that the query body becomes just

**then** expression **end**

# Extension C: conditional expressions

Accept expressions of the form

```
if c then exp1 else exp2 end
```

and more generally

```
if c1 then exp1 elseif c1 then exp2 ... else exp0 end
```

where the **then** and **else** clauses are always required

I believe this extension causes no syntactic ambiguity

January 2014 note: this is now implemented

# Extension D: parenthesis alias

(This was a suggestion of Alexander Kogtenkov and has now been implemented)

In the same way that a class can have a feature with the bracket alias, it can also declare one of its features r with the parenthesis alias "()"

Then for a of the corresponding type, a (x, …) is a shorthand for a.r (x, …)

The most immediate application is to agents: by giving item the parenthesis alias we can write a (x, …) instead of a.item (x, …)

Makes for a very natural style, e.g. in an integration loop we just write **Result** := **Result** + step * f (x),
just the way integration is explained in a math textbook

# Extension E: implicit tuples

(This mechanism, now implemented, comes from ideas of Alexander Kogtenkov and Emmanuel Stapf, following a suggestion by Simon Peyton-Jones at LASER 2012)

The rule is simple: if the last formal argument of a routine is of a tuple type, the brackets can be omitted in the corresponding formal argument

So a call r (x, y, [u, v, w]) can be written just r (x, y, u, v, w)

There is no ambiguity and the rule gives us a clean, type-safe form of C's "varargs": to obtain the equivalent of a routine with a variable number of arguments, just declare a formal argument of a tuple type

Of course we have always been able to do this; the new element is that calls no longer need brackets for the tuple, so we get the appearance of a routine with a variable number of arguments

# Calling agents

With the implicit tuple mechanism we can write the earlier

```
o.item ([map [i], map [i + 1]])
```

as just

```
o (map [i], map [i + 1])
```

This example is typical of the spirit of the extensions: the functional mechanisms are already present in the language thanks to agents; we are making the corresponding notations lighter, allowing a functional-language-like programming style for cases in which programmers deem it appropriate. There is no semantic change, and the full object-oriented power of Eiffel is there.

# Extension F: Contextual typing*

General idea:

- Allow declarations to use a type declared as "?X", for some fresh name X, or just "?"
- This stands for an actual type defined by the **context of use** of the corresponding entities
- If these uses are all consistent with some existing type, "?X" is understood as that type; it provides a **notational simplification** allowing the programmer to skip type declarations that can be **inferred in a simple way** from the program text
- If the actual uses are consistent but do not define a known type, the mechanism is equivalent to **adding a generic parameter** to the class

* *Possible alternate name: type-by-use*

# Effect of contextual types

Contextual types define no new semantics

It would be possible to avoid any contextual type by either:

- Declaring the type explicitly
- In the applicable case, introducing a formal generic

The IDE should make this clear by showing the reconstructed ("unfolded") form, with explicit typing, on demand, e.g. with a tooltip showing the type as one moves the mouse over the corresponding entity

It should also be possible to generate equivalent code that has full explicit declarations

# Language vs IDE

We have not completely decided what part of the contextual type mechanism is on the language side and what part on the environment side:

- The type inference might be used to let the IDE (EiffelStudio) fill in the types that the user does not want to write explicitly. With this solution the new mechanism implies no change to the language proper, it is only a facility of the environment
- Or the inferred types might remain unspecified in the program text, although in the IDE it will always be possible to see them (e.g. in tooltips) and an option will be available to add them in the program text anyway (as with the first option)

In any case Eiffel remains a fully typed language & the type system does not change. We are just making programmers' life easier.

# Contextual typing: syntax (1) and validity

A type appearing in the declaration of a feature is of the form "?t" for some identifier t

It is then known as a **contextual type**

There is no validity constraint!

(More explanations on this follow)

**Note**: we do not allow contextual types in other uses of types, such as Parent clause (for inheritance), or agent outside of a feature declaration (e.g. in an assertion)

(An agent appearing in a feature declaration can use a contextual type, whose scope is the entire feature)

This convention avoids issues of scope; it may be possible to relax it in the future if it turns out to be too restrictive

# Names of contextual types

Since a contextual type is always preceded by a "?", there is no need to constrain the possible names

Recommended convention: use ?X, ?Y, ?Z by default

For clarity, we might want to add a rule requiring the name not to clash with any class name, but this proposal includes no such restriction

The scope of such a name is the enclosing class text

The form "?" is an abbreviation for "?t" where t is a fresh name, the same one throughout the class text

- (At some point I removed this possibility for fear or a confusion with "?" for agents, but the context is different – types vs values – and I think it's OK, but it could be removed again)

# Contextual types: syntax (2)

The following abbreviations are supported

1. In an entity declaration

    x, y, …: T

the final part ": T" can be omitted; T is then understood to be "?t" where t is a fresh identifier

2. In a generic derivation

    C [A, B, …]

the *whole* parameter list [A, B, …] can be omitted, so that the declaration only retains the class name C; the actual parameters are then understood to be ?ta, ?tb, … where ta, tb, … are fresh identifiers

# Semantics of contextual types (informal)

If uses of ?X in the feature's declaration are all compatible with an existing type T, then ?X denotes T. In that case the contextual type is just an abbreviation mechanism

If these uses are inconsistent, the inferred type is NONE, with the result that the class will not compile

If they are consistent but do not include enough information to match an existing type, then ?X is understood as denoting a new implicit formal generic (unconstrained) of the enclosing class

Part 5 defines this semantics rigorously

# - 4 -

# Examples

# Conditional expressions and agents

```
abs (x: INTEGER): INTEGER
        then
                if x < 0 then -x else x end
        end

previous_salary (p: PERSON): INTEGER
        local
                p: PERSON
        do
                retrieve (filename)
                p := retrieved
        then
                p.salary
        end
```

# Contextual typing

An example that is possible but not necessary:

```
a: INTEGER

r (n: INTEGER)
        local
                i, j, k, m
        do
                a := i
                j := a
                k := 3
                print (i + m)
        end
```

The IDE will show that the inferred type for all these variables is INTEGER

January 2014 note: for such simple cases the mechanism now works

# Agents made simple

Nadia's agent example

```
agent
    (i: INTEGER;
     o: PREDICATE [ANY, TUPLE [G, G]]):
        BOOLEAN
    do
        Result := o.item ([map [i], map [i + 1]])
    end
```

now becomes:

```
agent (i; o: PREDICATE) then o (map [i], map [i + 1]) end
```

Note that we have to specify PREDICATE but may omit the actual generic parameters

# Function composition made simple

```
In FUNCTION [C, ARGS -> TUPLE, RES]):
    compose alias "@"
        (other: FUNCTION [ANY, TUPLE [RES], ?X]):
            FUNCTION [ANY, ARGS, ?X]
            -- Function that applies current function then `other'.
        then
            agent (x: ARGS): ?X then other (item (x)) end
        end
```

This example could use just ? instead of ?X

Note that in the final expression:

- The parenthesis alias allows us to write other (…) as a shorthand for item.other (…)
- The implicit tuple rule enables us to write item (x) as a shorthand for [item (x)]

# - 5 -

# Semantic specification of contextual types

# Reminder: the informal semantics

If uses of ?X in the feature's declaration are all compatible with an existing type T, then ?X denotes T. In that case the contextual type is just an abbreviation mechanism

If these uses are inconsistent, the inferred type is NONE, with the result that the class will not compile

If they are consistent but do not match an existing type, then ?X is understood as denoting a new implicit formal generic (unconstrained) of the enclosing class

We now define this semantics rigorously

# Background: conformance, minima, maxima

Conformance is a partial order relation; we write $u \leq t$ to express that u conforms to t

We consider that NONE $\leq$ t and t $\leq$ ANY for any type t (this simplifies the discussion, assuming that the conformance rule has a special clause for expanded types)

For two sets of types E and F, $E \leq F$ means that $t \leq u$ for every element t of E and u of F (true if either set is empty); similarly, we may use $u \leq E$ and $E \leq u$ for an individual type u

If $E \leq u$, we say that u is an **upper bound** of E

A **least upper bound (lub)** of E is an upper bound u such that $t \leq u$ for any lower bound t of E; if it exists it is unique, and if it is itself in E it is the **minimum** of E

A **minimal element** of E is an element t of E such that no other element x of E satisfies $x \leq t$ ; a minimum is a minimal element, but E may have one or more minimal elements and no minimum

*Dual notions*: greatest lower bound (glb), maximum, maximal element

# Access and update sets

Underlying intuition: the *access set* of a type t includes all the types whose values may have to be interpreted as t, and its *update set* all the types able to interpret values of type t

The **access set** of a type t includes all of the following:

- The type of the source of any attachment with a target of type t

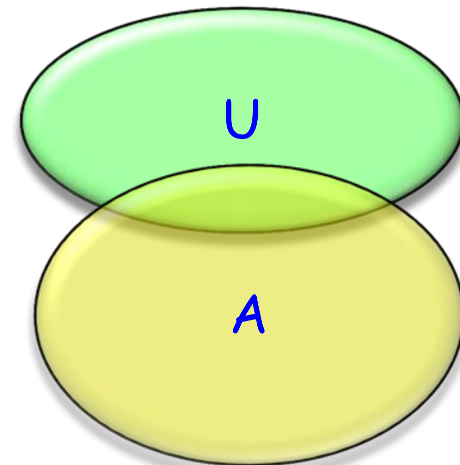
The **update set** of t includes all the following:

- The type of the target of any attachment with a source of type t
- For any use of t as actual parameter to a generic derivation, the corresponding generic constraint (ANY for unconstrained)

# The definition

The semantics of a contextual type x, whose access and update sets (deprived of x) are A and U, is as follows:

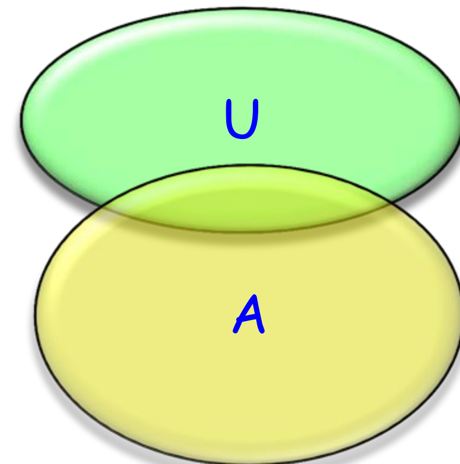
- 1. If there exists a type T that is a maximum of A and a glb for U, or a minimum of U and a lub for A: T
- 2. Otherwise, if $A \leq U$: a fictitious new formal generic type of the enclosing class, constrained by all minimal elements of U if any
- 3. Otherwise: NONE



# The definition

The semantics of a contextual type x, whose access and update sets (deprived of x) are $A$ and $U$, is as follows:

- 1. If $A \leq U$ and $A \cap U \neq \emptyset$: an (arbitrary) element of $A \cap U$
- 2. Otherwise, if $A \leq U$: a fictitious new formal generic type of the enclosing class, constrained by all minimal elements of $U$ if any
- 3. Otherwise: NONE



# Using a fictitious generic parameter

For a contextual type ?X, case 3 of the definition adds to the enclosing class C a fictitious formal generic parameter X; this happens for composition in FUNCTION

In any use of C in a class D, the corresponding actual is determined as follows: treat it as a contextual type of D and resolve it as in the previous definition, except that case 2 yields ANY (rather than a formal generic for D)

# Notes on the semantic definition

1. The idea is that to determine the type of a contextually-declared entity (and more generally a contextual type) we look at all its uses, and find the type that would fit them all, if there is one
2. There is no validity rule. We just have a mechanism that tries to find for us the type that we meant (and were too lazy to write), then moves on silently if everything is OK, and otherwise forces us to clean things up, possibly by being more explicit
3. The inference may succeed by either:
   - Finding an existing type that does the job: in this case the contextual type was just an abbreviation
   - Introducing an implicit formal generic parameter (as in the case of function composition)

# Further notes

4. If there are both greater and lower types, the lower ones must all be less than (conform to) the greater ones, otherwise no choice of type will work
5. The mechanism is modular: the type inference mechanism works at the level of a single feature (for the type a local variable) or class (for the type of a query), as long as we have the signature specifications of all used features & classes