Match-O, a Dialect of Eiffel with Match-Types

Dominique Colnet   Luigi Liquori

Université Henry Poincaré and École des Mines de Nancy
LORIA-UHP-INPL, F-54506 Vandœuvre-lès-Nancy Cedex BP 239 France
`Dominique.Colnet@loria.fr`,`Luigi.Liquori@loria.fr`

Abstract

It is well known that the Eiffel language allows covariant redefinition. Regardless of system-level validity rules, Eiffel is not type safe. In this paper, we present a dialect of Eiffel called Match-O, which prohibits covariant redefinition. We introduce a new kind of types, the `match`*-types, inspired by the papers of Kim Bruce. The scope of this project is many-fold: – allowing binary methods; – keeping sound "mytype method specialization", i.e. anchored type using* `Current`*; – allowing subtyping in all other sound cases. We claim that* `match`*-types can be added in the Eiffel type system to eliminate type unsoundness without blocking many interesting Eiffel programs (e.g. the ones with "binary methods"). We have implemented a compiler for Match-O and we have experimented our dialect on a large system using the original source code of SmallEiffel itself.*

1: Introduction

None of the existing *Eiffel* compilers implement the system-level validity rules described in [21]. As a consequence, an *Eiffel* program may crash because of run-time type check errors. The main reason for this type unsafety problem are: *binary methods* [4, 10, 11], unsafe use of *subtyping*, the possibility to specialize (*i.e.* override) the body of methods in subclasses, the type `like Current` in argument position, and many more (see [15]). This paper presents a real experience to make a statically safe dialect of *Eiffel* – with extensions and restrictions – without loosing much of its expressiveness.

Eiffel is the only widely used language featuring the possibility to denote a type using the type of the receiver itself. Such an anchored type is denoted by `like Current` to express the fact that some entity is compatible with the type of the receiver.

Specialization of the results. Assume a method `foo:like Current` is defined in class A and redefined in subclass B of A with the same signature. For a variable `a` of type A, the type of the expression `a.foo` is A. For a variable `b` of type B, the type of the expression `b.foo` is B. This means that the type of the result *specializes* as the body does. Note that only static information is used here to type such expressions. In other words, the fact that variable `a` may reference an instance of B at run-time is not considered when typing *Eiffel* expressions.

Specialization of the arguments. Assume a method `doo(other:like Current)` is defined in class A and redefined in a subclass B of A with the same signature. Such a method is usually called a *binary method* because it takes as argument an object which has the same type as the receiver. Invoking `doo` with a target `a` of type A with another variable of type A as argument (*i.e.* `a.doo(another_a)`) is a correct typable call. Similarly, invoking `doo` with a target `b` of type B, with another variable of type B as argument (*i.e.* `b.doo(another_b)`) is also a valid call.

As for the specialization of the result, only static type information is used to accept or reject some calls. For example, `b.doo(a)` is statically rejected by *Eiffel* rules because the type of `a` – *i.e.* A – is not compatible with the type of `b` (A is not a subtype of B).

Like Current with subtyping and inheritance. Unfortunately, in *Eiffel*, as in most object-oriented languages, if B is a subclass of A, B is also a subtype of A. Actually, a variable of type A can be assigned with an expression of type B as in the instruction `a := b`, and an expression of type B is a valid actual parameter for a formal argument of type A as in the call `a.doo(b)`.

As a consequence, combining `like Current` (*i.e.* the *mytype method specialization*) with *subtyping* and *inheritance* (*i.e.* the *method specialization)*, we obtain the well-known problem presented below:

```
class POINT inherit ANY redefine is_equal end; feature
   x: REAL; y: REAL;
   is_equal(other: like Current): BOOLEAN is do
         Result := (x = other.x) and (y = other.y);
      end
end

class PIXEL inherit POINT redefine is_equal end; feature
   status: BOOLEAN;
   is_equal(other: like Current): BOOLEAN is do
         Result := (x = other.x) and (y = other.y) and (status = other.status);
      end;
end

class MAIN creation main feature
   p: POINT; q: POINT; r: PIXEL;
   main is do
         !!p; !!q; !!r;
         p := r; -- is accepted because of subtyping.
         if p.is_equal(q) then ... else ... end; -- CRASH! (1)
         breakit(r,q) -- is accepted because of subtyping.
      end;
   breakit(p1, p2: POINT) is do
         if p1.is_equal(p2) then ... else ... end;  -- CRASH! (2)
      end;
end
```

In this trivial program we observe that the *"subtyping routine"*, present in the *Eiffel* type checker, can be triggered in two ways: through a simple assignment `(1)` or using argument passing `(2)`. In both situations, the problem arises at run-time, while accessing the inexistent `status` attribute of an instance of class POINT, hence producing a run-time crash. This example points out the fact that a subclass is not always a subtype (for a rich and detailed discussion, see [4, 14]).

The choices of Match-O are rather simple and are based on a simple restriction plus a simple extension of the *Eiffel* type system; this solution is inspired by Bruce's papers and the Undergraduate Honor Thesis of Jonathan Burstein [9]. The *Match-O* dialect is obtained by applying the following Extension/Restriction to *Eiffel*:

Extension: We add a type keyword, namely "`match`", which can prefix any legal explicit non anchored type mark, and we add the new type "`match Current`".

Restriction: We eliminate the *Eiffel* type "`like Current`" and we forbid covariant specialization in subclasses, with the only exception of `match Current`.

Intuitively, a variable declared of type `match` POINT can only be assigned with an expression which is *exactly* of type POINT. As an example, an expression of type PIXEL cannot be assigned to a variable of type `match` POINT. Respectively, a variable of type `match Current` has exactly the same dynamic type as `Current`. Roughly speaking, using `match`-types is a way to block the subtyping mechanism.

Since we don't have unsound covariant specialization, the construct `like x` is still allowed, also in the case of `x` declared of type `match Current`. If a variable or a formal parameter is declared of type `like x`, then its concrete type is calculated by a *syntactic replacement* of the type of `x` (also if its type is `match Current`).

It follows that all "`match`-free" *Eiffel* programs which do (not) respect the typing *Match-O* rules are (not) accepted by the compiler of *Match-O*, and the *Match-O* language has enough expressiveness to encode binary methods (*i.e.* methods with some arguments typed with `match Current`). As such, the language *Match-O* features sound subtyping in presence of a non-trivial form of covariant specialization (*i.e.* the one related to the use of `match Current`).

Road map. Section 2 presents the theoretical background of our proposition as well as its translation into *Match-O*. Section 3 describes how `match`-types can be smoothly integrated to *Eiffel* and its libraries. Section 4 explains some problems we encountered while trying to bootstrap the *Match-O* compiler. Section 5 concludes and presents some related work, together with some future work.

2: The Match-O types

A great theoretical effort has been devoted in the last years to finding sound and expressive type systems [5, 6, 8, 3, 1, 16, 19, 18, 22, 23, 2, 24] featuring a subtyping relation compatible with mytype method specialization.

The matching relation over object-types. In Bruce's theory, an object-type (essentially a *type-interface*) has the form $\mathtt{OT}\,\langle\!\langle \mathtt{v}_i : \mathrm{A}_i, \mathtt{m}_j : \mathrm{B}_j \rangle\!\rangle_{j\in J}^{i\in I}$, where `OT` is a binder for the type-variable `Mytype` (representing the type of `Current`), $\mathtt{v}_i : \mathrm{A}_i$ are the instance variables with their types, and $\mathtt{m}_j : \mathrm{B}_j$ are the methods and their results types, respectively. The type-variable `Mytype` can freely occurs in the $\mathrm{A}_i, \mathrm{B}_j$ and represents the type of `Current`.

The matching relation is a type-relation weaker than the subtyping relation. Matching over object-types behaves as follows: two object-types matches (denoted by `<#`) *if the former has more variables and/or methods than the latter.* The very intuitive semantics of A `<#` B, is that any message sent to an object of type B can also be sent to an object of type A. Conversely, the intuition for subtyping (denoted by `<:`) is slightly different: given A `<:` B, an object of type A can be used in every context which expects an object of type B. As example, if we consider only type-interfaces, we can say that PIXEL `<#` POINT, but PIXEL $\not<:$ POINT.

2.1: Exact and hash-types

Exact-types. In [9], Burstein introduced (in the context of an extension of the *Java* language) the *exact-types* (denoted by `@`A) to represent types whose elements are *not susceptible to be subsumed.* The intuitive meaning of the exact-types is as follows: objects typed with an exact-type are type-checked with a *more restrictive* algorithm, namely without the subtyping routine. This rigidity in the type system will block many assignments and method calls.

When a program only uses variables declared of type "`@`" it contains only exact-expressions, and hence it conforms to the matching-based theory of Bruce [6]. In fact, exact-types *are* the Bruce object-types.

Hash-types. In [3] Bruce introduced another kind of type, the *hash-types* (denoted by `#`A). The intuitive meaning of the hash-types is as follows: objects typed with a hash-type are type-checked with a *less restrictive* algorithm, namely with the subtyping routine. An unfortunate feature of hash-types is that *binary methods* (*i.e.* methods which take formal parameters of the same type of the receiver, and specialize its signature in subclasses) cannot be sent to hash-typeable objects.

Is MyType `@` or `#` ? Given `Mytype`, the type of `Current` and the above two shapes of types, one natural question that may arise is: "is `Mytype` an exact-type or a hash-type?" The difference is substantial since the type rules for message sending we want to apply are not the same at all. Both views are correct; for instance, if the receiver `a` is typed with type `@`A (resp. `#`A), and it contains in its interface a method `foo` which returns an object of type `Mytype`, then `a.foo()` will be of type `@`A (resp. `#`A).

2.2: The Match-O translation of `@` and `#`

Let A be any concrete type. The interpretation in *Match-O* of an exact-type `@`A is `match` A, whereas that of a hash-type `#`A is simply A. The interpretation of `Mytype` is `match Current`, *i.e.* the dynamic type of `Current`. Every time we declare a class A in *Match-O*, we (implicitly) declare another class `match` A which "specializes" A.

Following the symmetry between exact and hash-types, in *Match-O*, we have two kinds of objects: objects assigned to concrete types (type-checked with the subtyping routine), and objects assigned to `match`-types (type-checked without the subtyping routine).

The subtyping relation between concrete and `match`-types says that the type `match` A is a subtype of the concrete type A. This is sound with respect to our interpretation of hash and exact-types into `match` and concrete types. Following this rule, an entity declared of type `match` A can be also considered as an object of type A.

Another usual rule for objects activates the subtyping routine in the *Match-O* type system: an object type-checked of type A can masquerade as an object of type B, being A a subtype of B.

2.3: Assignments, passing parameters, and sending messages in *Match-O*

Assignment in Match-O. The type rules for assignment in *Match-O* are rather simple. Let A be any concrete type. The rationale is as follows: variables declared of type `match` A can be assigned only to expressions of the same exact type (*i.e.* `match` A). Counterwise, variables declared of type A can be both assigned with expressions of type `match` A, or A, and any subtype of A.

Let B be a subclass of A; the following table gives the type checking rules for the assignment instruction: "`destination := source;`". The types of the `destination` variable are listed in the column, while the types of the `source` expression are listed in the rows:

`destination\source`	A	`match` A	B	`match` B
A	Accepted	Accepted	Accepted	Accepted
`match` A	Rejected	Accepted	Rejected	Rejected
B	Rejected	Rejected	Accepted	Accepted
`match` B	Rejected	Rejected	Rejected	Accepted

For example, the assignment of a `source` expression of type B into a `destination` variable of type `match` A is rejected since B is not a subtype of `match` A; counterwise, the assignment of a `source` expression of type `match` B to a `destination` variable of type B is is accepted since `match` B is a subtype of B.

Passing parameters in Match-O. The type rules for passing parameters in *Match-O* are the same as the ones for assignment (with the only exception of declaring a formal parameter of type `match Current`). In the simplest case, you can consider the above table for assignments, where `destination\source` is substituted by `formal_arg\actual_arg`. For instance, let A and C be concrete classes, and let `foo(x:A):C` be a method declared in class A (and inherited in a subclass B of A). Let `a`:A and `b:match` B. The call `a.foo(b)` is accepted, since the actual parameter `b` is of type `match` B, `match` B being a subtype of B. Note that the type of the receiver is used only to guarantee that `foo` is in the interface of A.

A different issue is raised when we declare a parameter of type `match Current` *i.e.* `binary(x:match Current):C`. In this case the rationale is as follows: the receiver must be typed with a `match`-type and the argument must be typed with exactly the same `match`-type, *i.e.* if `a:match` A and `b:match` B, then the call `a.binary(b)` is rejected. We will see in Section 3 that this restriction still allows for expressivity and that it only has a minor practical impact from the software engineering point of view.

Sending messages which return `match Current`. The rules for sending a message which returns an object of type `match Current` are simple. Given an expression `receiver.message(arg_list)`, the type of this expression just follows the type of the `receiver`:

`receiver`	A	`match` A	B	`match` B
`receiver.message(arg_list)`	A	`match` A	B	`match` B

Observe that the type of `arg_list` is not needed for calculating the type of the result. For instance, let `twin` be a method defined in class A (and inherited in a subclass B of A) which returns a copy of the object itself, *i.e.* "`twin: match Current is ...`", and let `a`: A, `ma:match` A, `b`: B, and `mb:match` B. Then, `a.twin` (resp. `ma.twin`) will return an object of type A (resp. `match` A), while `b.twin` (resp. `mb.twin`) will return an object of type B (resp. `match` B). This is sound since the type `match Current` represents the type of the receiver.

3: From Eiffel to Match-O

In order to explain the behavior of `match`-types, we present some more *Match-O* examples in this section, as well as some decisions we made to merge this new kind of type with the existing *Eiffel* language and libraries. This section should also help to clarify the usage of the new `match`-types from the programmer's point of view.

In the following examples, we assume that we have defined in *Match-O* both classes POINT and PIXEL, where PIXEL is a subclass of POINT. To avoid repetition and for clarity, we assume in all the remainder of this paper that the variable `match_point` is declared of type `match` POINT, that the variable `match_pixel` is declared of type `match` PIXEL, the variable `match_current` is declared of type `match Current`, etc. We also assume that the variable `point` is declared of type POINT, that the variable `pixel` is declared of type PIXEL, etc. This naming scheme also applies to all standard *Eiffel* type names; for example, a variable whose name is `match_string` is declared with type `match` STRING.

3.1: Expanded and Match-types, manifest constants, and the type of Current

When a class C is declared as an expanded class, entities of type C are values which are instances of class C and only of class C [21]. Despite the fact that the initial purpose of expanded types in *Eiffel* is to avoid the need for a reference to the corresponding object, the relationship between expanded types and `match`-types clearly appears. If C is an expanded type, it is also a `match`-type, because, by definition, entities of type C always denotes an object of class C only. As an example, because the class INTEGER is expanded, the type INTEGER is completely equivalent to the type `match` INTEGER. Thus, the notation of INTEGER constants, such as `1` or `2`, also denote objects of type `match` INTEGER. As a consequence, the following assignments are allowed in *Match-O*:

```
match_integer := 1;  -- is accepted.        match_character := 'a'; -- is accepted.
```

Following the same reasoning, the type of manifest strings, such as `"foo"`, is no longer STRING but, more precisely, `match` STRING; hence the assignment

```
match_string := "foo";  -- is accepted.
```

Because the type `match` STRING is a subtype of STRING, the ordinary assignment of some manifest string into a variable of type STRING is also allowed, making old existing *Eiffel* code compatible with the *Match-O* dialect.

Typing of manifest arrays is also reconsidered the same way in *Match-O*. For example, the manifest array `<<"foo","bar">>`, which is of type ARRAY[STRING] in *Eiffel*, has in *Match-O* the type `match` ARRAY[`match` STRING]. Consequently, the following assignments are both accepted:

```
match_string := (<<"foo", "bar">>).item(1); -- is accepted.
string       := (<<"zoo", "doo">>).item(1); -- is accepted.
```

Typing Current. The problem of the type of the pseudo-variable `Current` is a bit less obvious. For example, what is the type of `Current` inside some method defined in the class POINT (and inherited in a subclass PIXEL)? Is it POINT, `match` POINT, PIXEL, or `match` PIXEL? The correct answer is `match Current`, since, as showed in the previous section, the semantic meaning of `Current` (hence of `match Current`) is *context dependent* (*i.e.* it depends on the dynamic type of the target).

Taking as example some assignments, when the target is of dynamic type POINT (resp. PIXEL), the following instructions are safe and accepted. We assume here that match_current, match_point, and point are variables declared inside a method of class POINT:

```
Target of dynamic type Point                Target of dynamic type Pixel
match_current := Current; -- is accepted.   match_current := Current; -- is accepted.
match_point   := Current; -- is accepted.   match_point   := Current; -- is rejected.
point         := Current; -- is accepted.   point         := Current; -- is accepted.
```

In other words, each method (even when there is no redefinition) is considered for each possible dynamic type and checked accordingly. This is perfectly in agreement with the whole system analysis algorithm of SmallEiffel which already performs *code customization* for each possible concrete type (see for more details [12, 27]).

3.2: Adapting some Eiffel constructs

In order to fit with the new match-types, we also have to adapt the behavior of two important constructs of the *Eiffel* language: the creation instruction and the assignment attempt.

The creation instruction. The *Eiffel* creation instruction allows the user to select the concrete class to instantiate, which may be different from the static type of the destination variable. When we declare a variable of a match-type, the revisited *Match-O* creation instruction avoids the possibility to assign an instance of class PIXEL into some match_point variable:

```
!!match_point.set(1.0,2.0);       -- create a POINT.
!POINT!match_point.set(1.0,2.0); -- is accepted.
!PIXEL!match_point.set(1.0,2.0); -- is rejected.
```

Once more, this decision enforces the fact that a match-typed variable can only hold one dynamic type, *i.e.* the matched one.

The assignment attempt. The *Eiffel* assignment attempt operator ?= provides a safe way to go down in type hierarchy [21]. Because the variable point may reference a PIXEL at run-time, the assignment of point into pixel makes sense but is not always applicable.

As for the creation instruction, the customization of the assignment attempt for match-types is guided by type safety. When the right-hand-side of the assignment attempt is a match-type, only the corresponding exact dynamic type will make the assignment effective.

```
match_pixel ?= point;
if match_pixel /= Void then ... -- match_pixel is now an alias for point.
                       else ... -- The dynamic type of point is not a PIXEL.
end;
```

From the implementation point of view, the generated code for an assignment attempt with a right-hand side variable typed with a match-type is a simple efficient conditional expression with only one hard-coded possibility. When the right-hand side is not a match-type, instead, the number of accepted possibilities can be greater than 1 and thus less efficient (see [13] for a detailed description of code customization for the ?= assignment).

3.3: Typing some interesting methods

An accurate signature for the twin method. The twin method defined in the GENERAL class is the basic object's duplication primitive of the *SmallEiffel* library. This method is originally defined in *Eiffel* with the following signature:

```
twin: like Current is ...          -- The old Eiffel signature.
```

Because of the usage of the `like Current` type mark, the previous signature is not valid in *Match-O*. Furthermore, because the purpose of the `twin` method is to create a new object with the exact dynamic type of the target, the new *Match-O* signature we have chosen is more accurate now:

```
twin: match Current is ...         -- The new Match-O signature.
```

According to the rules explained before, here is the *Match-O* behavior for clients of method `twin`:

```
match_string := ("foo").twin;      -- is accepted.
string       := ("bar").twin;      -- is accepted.
match_string := string.twin;       -- is rejected.
```

Indeed, the type of the expression `point.twin` is POINT (not `match` POINT) avoiding the possibility to assign a PIXEL into a `match_point` variable:

```
point       := match_pixel;         -- is accepted.
match_point := point.twin;          -- is rejected.
```

This new definition of method `twin` using `match Current` as a result type is more accurate and is backward compatible with existing *Eiffel* code.

A safe definition of is_equal. The *Eiffel* definition of `is_equal`, rejected in *Match-O*, must be modified in order to remove the `like Current` type mark used in argument position:

```
is_equal(other: like Current): BOOLEAN is
   require
      other_not_void: other /= Void
   ensure
      consistent: standard_is_equal(other) implies Result;
      symmetric: Result implies other.is_equal(Current);
   end;
```

A close look at the `symmetric:` part of the ensure clause reveals that the `is_equal` feature itself is recursively called with `Current` and `other` swapped (*i.e.* `other.is_equal(Current)`). This information from the ensure assertion reveals the very nature of `is_equal`: only two objects with exactly the same dynamic type can be considered to be equal. Knowing this, the very first idea is to change the signature of `is_equal` as follows:

```
is_equal(other: match Current): BOOLEAN is do ... end;
```

The obvious advantage would be to make `is_equal` type safe. Unfortunately, this modification breaks a lot of existing code, just because you can use `is_equal` only with `match`-types both for the target and the argument. Another possibility is to introduce a new method we called `match_is_equal` and to redefine a safe version of `is_equal` without covariance, to wrap this method:

```
match_is_equal(other: match Current): BOOLEAN is do ... end;
is_equal(other: ANY): BOOLEAN is
   local match_current: match Current;
   do
      match_current ?= other;
      if match_current /= Void then
         Result := match_is_equal(match_current);
      end;
   end;
```

The assignment attempt `?=` is used to know if the dynamic type of `other` is the one of `Current`. Inside the `then` part, the `match_is_equal` method is called to finish the comparison process. When the `then` part is not executed (because the `other` argument has a different dynamic type), the `Result` variable keeps its default `false` value.

The new `match_is_equal` function is implemented with a compiler built-in field by field comparison, as in the original *Eiffel* `is_equal` definition. Because `match_is_equal` is type safe, the built-in code, which performs the field by field comparison, no longer needs to check that both objects have the same dynamic type, just because this is statically enforced.

Finally, the new signature of `is_equal` uses the `ANY` type, making the new definition applicable to any kind of object. This is more tolerant than the original definition in *Eiffel* and allows backward compatibility.

4: Towards a bootstrap of the Match-O compiler

Starting from the original source code of *SmallEiffel*, a large program of about 300 classes for about 80K lines of source code, we have progressively worked to obtain our first compiler for *Match-O*. Before trying to bootstrap our compiler itself, we have validated our first release of the *Match-O* compiler, written in *Eiffel*, with a set of examples including correct *Match-O* programs as well as rejected *Match-O* programs (those examples are part of the *Match-O* distribution). We then started to modify the *Match-O* compiler written in *Eiffel*, trying to obtain a compiler written in pure *Match-O*. During this last process, except for one covariant redefinition we will see below, the transformation from *Eiffel* to *Match-O* appears to be quite straightforward.

The remainder of this section points out the problems we encountered to migrate our compiler written in *Eiffel* to the current one which is *nearly* written in pure *Match-O*. The result of this experiment should also help other people to switch their unsafe *Eiffel* code to *Match-O* safe code.

Tracking unsafe generic dispatches. The most common problem encountered during bootstrap attempts was related to dynamic dispatch when using a generic container with an indirect unsafe covariant derivation. Here is a very simple example of some accepted *Eiffel* code which points out a safety problem:

```
local   point: POINT; pixel: PIXEL;
        array_of_points: ARRAY[POINT]; array_of_pixels: ARRAY[PIXEL];
do      !!point; !!array_of_pixels.make(1,1); !!array_of_points.make(1,1);
        array_of_points := array_of_pixels;       -- (1)
        array_of_points.put(point,1);             -- (2)
        pixel := array_of_pixel.item(1);          -- (3)
```

Because the type ARRAY[PIXEL] is considered to be a subclass of ARRAY[POINT], the previous assignment `(1)` is valid in *Eiffel*. The instruction `(2)` allows storing an instance of class POINT into an ARRAY[PIXEL], which is obviously wrong and dangerous. Finally, because the same array is referenced by `array_of_pixel`, it is possible to assign a POINT into the `pixel` variable as done with instruction `(3)`. Unfortunately, this erroneous program is accepted by all *Eiffel* compilers, because none of them implements the system-level validity rules. Running such a wrong program leads to a run-time type exception error.

The previous example is *statically* rejected by our *Match-O* compiler, which reports a compile time covariance error for the instruction `(2)`. Indeed, the `put` procedure of class

ARRAY is covariantly redefined from ARRAY[POINT] to ARRAY[PIXEL]: the first argument cannot be changed from POINT to PIXEL in *Match-O*. To fix this problem, one must change the type of `array_of_point` to `match` ARRAY[POINT]. With this simple modification the assignment `(1)`, which is actually at the origin of the problem is now statically rejected and the `put` call `(2)` is now accepted in *Match-O*. Furthermore, because the target of the call `(2)` has only one possible dynamic type (*i.e.* ARRAY[POINT]), the *Match-O* generated code is a direct fast call.

From like Current to match Current. Many other detected problems come from the `like Current` type which is often used just because `match Current` does not exist in *Eiffel*. The most important example is the definition of `twin`. The goal of `twin` is to return an object of the same dynamic type as its target. Furthermore, the replacement from `like Current` to `match Current` as a result type does not break the client code. Unfortunately, such a replacement in argument type position often implies a lot of modifications, because all calls to the modified method must be a target of the corresponding match type, as explained in Subsection 2.3.

Since `like Current` does not belong to the *Match-O* types, it is mandatory to remove all occurrences of `like Current` in argument position. As we have previously seen in the new definition of `is_equal`, it is sometimes better to replace a `like Current` type with another concrete type, *i.e.* no more anchored type. At the time being, the *Match-O* rule which prohibits `like`-notations (*i.e.* `like Current`, `like feature`, and `like argument`) in argument position is not systematically enforced in the compiler code. Actually, as we will see below, a `like`-notation does not always imply covariant redefinition.

Accepting invariable like-notations. During the bootstrap, a significant number of `like`-notations proved actually be a useful short-hand notation for some unique invariable concrete type name. The following table gives the type notations used in the live code of the compiler during the bootstrap:

	Attribute	Result	Argument
match Current	0	120	5
match type	40	14	25
like Current	0	132	112
like feature	0	46	820
like argument	0	5	0
Concrete types	3702	3784	3988

The column Attribute shows that the *Match-O* rules are strictly enforced for the type of attributes: the `like`-notation is never used and the most common type notation (line Concrete types) is the simple explicit notation. The result type of functions given in column Result shows that `match Current` is quite often used. This is mainly because of the result type of the `twin` function.

By looking at the table, one may observe that our *Match-O* compiler does not prohibit all `like`-notations in argument position: for instance, in the Argument column, 112 `like Current` and 820 `like feature` are used and accepted. This is clearly a violation of the strict *Match-O* rules explained in Section 1. Because our *Match-O* compiler performs *whole system analysis* [27] it is possible to safely accept such `like`-notations. Actually, those `like`-notations are equivalent to *explicit type notations*, *i.e.* each accepted `like`-notation appears to only have one possible concrete substitution with one unique concrete type. For all those accepted `like`-notations, the knowledge of the whole system allows us to check that there is no covariance at all.

A hard covariance removal. During the bootstrap process of the *Match-O* compiler, only one covariant problem was not easy to fix using, for example, a simple declaration type modification. This is the only use of covariance without generic interaction we found in the whole original compiler source code.

The class LOCAL_VAR_LIST is used to represent local variable lists, while class FORMAL_ARG_LIST is used to represent formal argument lists. Those classes both inherit from the abstract (*i.e.* never created) DECLARATION_LIST class in order to share a few template methods. The type of elements in DECLARATION_LIST is then covariantly redefined in class LOCAL_VAR_LIST to handle local variable items and also redefined in class FORMAL_ARG_LIST to handle argument names items.

In order to make the code type safe, it was necessary to duplicate the invalid template methods originally factorized in class DECLARATION_LIST. This is the only place in the whole *Match-O* compiler where such an important modification was necessary to avoid covariance. At least for the present implementation of the *Match-O* compiler, this points out the fact that covariance is not as widely used as one may imagine.

5: Related, further work and conclusion

Related work. Among the interesting related works we would like to mention, we recall the following ones: the language `Rupiah` [9], a dialect of *Java* with `match`-types, the language *Tool* [17], that integrates matching, subtyping, and parametric-types, and the language *PolyToil* [8], probably the first language integrating matching and subtyping.

The choice of blocking covariance is not the only one: [25] added *virtual-types* (or virtual-methods) to *Java*, in the style of the *Beta* language [20]. This solution gives the possibility to covariantly specialize variables and method-types, but it requires some extra run-time type checks to ensure type safety. In [7, 26] the authors proposed, independently, two different solutions to make virtual-methods statically safe. Recently, the *LGM* language [24] proposed another solution to the problem of covariance using Bruce's *generalized matching* [6].

The experience we have with *Match-O* shows that the cohabitation of matching and subtyping is fruitful and quite natural.

Further work. The question of whether *Match-O* is really type safe or not is a work in progress. We plan to formalize the static and the dynamic semantics of *Match-O*, and to explore the possibility to generalize the `match` construct to the case of formal generic arguments. Another interesting possible further work is the efficient implementation of *Eiffel* system-level validity rules combined with `match`-types.

Conclusions. We have experimented `match`-types in the *Eiffel* language. The *Match-O* compiler is available at `http://www.loria.fr/~colnet/Match-O/macho.tgz`. The distribution also includes a large set of examples showing that many unsafe programs can be statically rejected without the need for system-level validity rules.

From the designer's point of view, `match`-types appear to add expressiveness and accuracy in many situations. The possibility to use `match Current` in place of `like Current` also seems to make acceptable the decision which consists in forbidding covariant redefinition.

Acknowledgments. We would like to thank Karl Tombre for reviewing of this paper.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer Verlag, 1996.

[2] V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L Liquori. Subtyping for Extensible, Incomplete Objects. *Fundamenta Informaticae*, 38(4):325–364, 1999.

[3] K. Bruce. Subtyping is not a Good Match for Object-oriented Programming Languages. In *Proc. of ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 104–127. Springer Verlag, 1997.

[4] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. Pierce. On Binary Methods. *Theory and Practice of Object Systems*, 1(3), 1996.

[5] K.B. Bruce. Safe Type Checking in a Statically-typed Object-oriented Programming Language. In *Proc. of POPL*, pages 285–298, 1993. Extended version in *Journal of Functional Programming*.

[6] K.B. Bruce. A Paradigmatic Object–Oriented Programming Language: Design, Static Typing and Semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.

[7] K.B. Bruce, M. Odersky, and P. Wadler. A Statically Safe Alternative to Virtual Types. In *Proc. of ECOOP*, volume 1445 of *Lecture Notes in Computer Science*, pages 523–549. Springer Verlag, 1998.

[8] K.B. Bruce, A. Shuett, and R. van Gent. PolyToil: a Type-safe Polymorphic Object Oriented Language. In *Proc. of ECOOP*, volume 952 of *Lecture Notes in Computer Science*, pages 27–51. Springer Verlag, 1995.

[9] J. Burstein. `Rupiah`: an Extension to Java Supporting Match-bounded Parametric Polymorphism, `ThisType`, and Exact Typing. Bachelor's thesis, Williams College, 1998.

[10] G. Castagna. Covariance and Contravariance: Conflict Without a Cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.

[11] G. Castagna. *Object-Oriented Programming: A Unified Foundation.* Progress in Theoretical Computer Science. Birkäuser, Boston, 1996.

[12] S. Collin, D. Colnet, and O. Zendra. Type Inference for Late Binding. The SmallEiffel Compiler. In *Proc. of JMLC*, volume 1204 of *Lecture Notes in Computer Science*, pages 67–81. Springer Verlag, 1997.

[13] D. Colnet and O. Zendra. Optimizations of Eiffel programs: SmallEiffel, The GNU Eiffel Compiler. In *Proc. of TOOLS*, pages 341–350. IEEE Computer Society, 1999.

[14] W. Cook, W. Hill, and P. Canning. Inheritance is not Subtyping. In *Proc. of POPL*, pages 125–135. The ACM Press, 1990.

[15] W. R.. Cook. A Proposal to Make Eiffel Type Safe. In *Proc. of ECOOP*, pages 57–70. Cambridge University Press, 1989.

[16] K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *Proc. of FCT*, volume 965 of *Lecture Notes in Computer Science*, pages 42–61. Springer Verlag, 1995.

[17] A. Gawecki and F. Matthes. Integrating Subtyping, Matching, and Type Quantification: a Practical Perspective. In *Proc. of ECOOP*, volume 1098 of *Lecture Notes in Computer Science*, pages 26–47. Springer Verlag, 1996.

[18] L. Liquori. An Extended Theory of Primitive Objects: First Order System. In *Proc. of ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 146–169. Springer Verlag, 1997.

[19] L. Liquori and G. Castagna. A Typed Lambda Calculus of Objects. In *Proc. of Asian*, volume 1179 of *Lecture Notes in Computer Science*, pages 129–141. Springer Verlag, 1996.

[20] O.L Madsen and K. Moller-Pedersen, B. Nygaard. *Object-Oriented Programming in the Beta Programming Language.* Addison-Wesley, 1993.

[21] Bertrand Meyer. *Eiffel, The Language.* Prentice Hall, 1994.

[22] D. Rémy. From Classes to Objects via Subtyping. In *Proc. of ESOP*, volume 1381 of *Lecture Notes in Computer Science*, pages 200–220. Springer Verlag, 1998.

[23] J.G. Riecke and C. Stone. Privacy via Subsumption. In *Electronic Proc. of FOOL-98*, 1998. To appear in *Theory and Practice of Object Systems.*

[24] R. Rinat. Type-safe Covariant Specialization with Generalized Matching. In *Electronic Proc. of FOOL-7*, january 1999.

[25] K. K. Thorup. Genericity in Java with Virtual Types. In *Proc. of ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 444–471. Springer Verlag, 1997.

[26] M. Togersen. Virtual Types are Statically Safe. In *Electronic Proc. of FOOL-5*, january 1998.

[27] O. Zendra, D. Colnet, and S. Collin. Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler. In *Proc. of OOPSLA*, volume 32(10), pages 125–141. The ACM Press, 1997.