Rationalising Eiffel's Type System

A J H Simons

Department of Computer Science, University of Sheffield,
Regent Court, 211 Portobello Street, Sheffield S1 4DP, United Kingdom.
Email: A.Simons@dcs.shef.ac.uk

Abstract

Eiffel has too many polymorphic type mechanisms: conformance, generic and anchored types, some of which are flawed and others redundant. Cook's suggested 1989 corrections to Eiffel's type rules, most notably to make inheritance obey subtyping, were not accepted by Meyer, Eiffel's designer, who found them too restrictive. F-bounded polymorphism has since been widely proposed as an alternative to subtyping for describing the inherited *self*-type. Here, we adopt F-bounds for *all* polymorphic types in Eiffel, giving the flexibility Meyer wants. We analyse the intended expressiveness of conformance, anchored and generic types and show how these can all be replaced by a single, elegant parametric mechanism.

1. Introduction

When William Cook reported, in his *Proposal for Making Eiffel Type-safe* [Cook 89b], that there were loopholes in Eiffel's type system, this came as a surprise to those who look to Eiffel as a model for strongly-typed object-oriented languages. The crux of this now well-known problem stems from Eiffel's intention to base type *conformance* on subtyping, while not managing to obey all the rules necessary to achieve this.

Cook's many suggested amendments to Eiffel's type rules were intended to enforce strict subtyping. These included linking exports with inheritance, forbidding the redefinition of attribute types, inverting the routine argument redefinition rule to observe *contravariance* (redefined arguments should have more general types), judging type compatibility between parameterised types after replacing the type parameters and introducing an explicit type attribute scheme to handle Eiffel's anchored types. Contravariance is a counter-intuitive finding for subtyping models of inheritance because it prevents the uniform specialisation of function arguments and results. It forbids the replacement of a function $f:\tau\rightarrow\tau$ closed over a type τ by a function $f:\sigma\rightarrow\sigma$ closed over a subtype $\sigma \subseteq \tau$ [Cardelli 86]. While insisting that Eiffel should obey the contravariant rule, Cook ruefully admits that:

> "[Contravariance] has the unfortunate effect of making argument type redefinition almost useless, since it is usually not very useful to allow a redefined method to accept a larger class of arguments" [Cook 89b, p62].

In his reply to Cook [Meyer 89], Meyer objected to the linking of exports with inheritance, which he called impractical, and especially to the adoption of contravariance. Rather than change Eiffel's type rules to obey strict subtyping, Meyer introduced a "global system validity check" to catch type errors retrospectively in situations where polymorphic aliasing would lead to run-time failure (see section 3 below). Technically, Meyer's solution works, although from a mathematical standpoint it is unsatisfying.

Meyer's refusal to adopt contravariance arose initially from observing the regularity captured by Eiffel's anchored types:

> "Examples such as the above, of which there are thousands in practical Eiffel applications, make it very hard to imagine how significant object-

Proc. 18th Conf. Technology of Object-Oriented Languages and Systems, eds. C Mingins, R Duke and B Meyer (Melbourne: Prentice-Hall, 1995).

oriented software can be written in a typed language without a *covariant* policy. Many of these examples use declaration by association, which is only a syntactical abbreviation, but in practice an essential one; its very availability for routine arguments is only possible because of the covariant rule" [Meyer 89, p12].

Anchored types are those declared "by association". The most common case is where an argument is said to have the type *like Current* (ie is anchored to the type of the current object). Anchored types express something intuitive about classification which we would want to preserve in an object-oriented language, namely that a function $f:\tau\rightarrow\tau$ closed over the class τ can be inherited by a class σ, in which it is automatically retyped and closed over the new class $f:\sigma\rightarrow\sigma$. Meyer assumed that this mechanism was merely a "syntactic abbreviation" for type redefinition, subject to an interpretation in a simple subtyping model of inheritance. As a result, he was forced to conclude that a covariant policy should be observed elsewhere for argument redefinition. This conclusion is wrong, not least because it is based on a false assumption: anchored types are *not* syntactic abbreviations, they are better explained using a different mechanism.

Ironically, Cook discovered the Eiffel type failure problem while researching an alternative mathematical model of class inheritance, F-bounded quantification [Cook 89a, Canning 89a, Canning 89b] which is different from subtyping [Cook 90]. F-bounds were devised chiefly to explain the evolution of the *self*-type under inheritance, but can also be used to explain other anchored type declarations [Simons 94a]. In the F-bounded approach, classes are type families characterised by a type generator and inheritance is a pointwise inclusion relationship between type generators.

In the rest of this paper, we shall develop an alternative solution to Cook's Eiffel type failure problem, based on F-bounds. In section 2 we outline the main differences between subtyping, bounded quantification and F-bounded quantification. The F-bounded approach permits *covariant* restriction on redefined argument types and the implicit retyping of inherited functions, in the style desired by Meyer; however type conformance is no longer based on subtyping. In sections 3 and 4, we review the three different type mechanisms provided by Eiffel [Meyer 88, Meyer 92] - *conformance*, *generic types* and *anchored types*. In section 3, we illustrate the inconsistencies of Eiffel's *conformance*, which is a weakened notion of subtyping. Many instances of type failure lead back to violations of the contravariant rule. In section 4, we review *generic types* and *anchored types*. We describe the intended use of these mechanisms, but go on to show how, in combination with conformance, they are multiply redundant. In section 5, we introduce a cleaner type semantics for Eiffel, making a proper distinction between monomorphic and polymorphic types. We eliminate conformance and combine anchored and generic types in a single new mechanism. In section 6, we describe a scheme for propagating static types into polymorphic parameters. The kind of type failure illustrated in section 3 is successfully trapped in our scheme.

2. Relationship Between Subtyping and F-Bounds

In the simple subtyping approach [Cardelli 84, Cardelli 88a], object classes are identified with types and inheritance is considered to be the same thing as subtyping. In simple subtyping, we say that an object of type X has a set of methods in the type $\sigma(X)$ since in general they may refer to self and are therefore based on the type of X. The expectation is that other objects of some type $Y \subseteq X$ can be passed legally to these methods. In a statically-bound system, this may usually be done safely, albeit with a loss of type information since methods in the type $\sigma(X)$ can only return an object of type X, even when passed an object of type Y. However, it is a common practice in object-oriented programming to replace some of a class X's methods by redefined versions for a subclass Y, in the type $\sigma(Y)$ with the intention that these methods will be invoked dynamically where the original methods in the type $\sigma(X)$ were expected. For this reason, it is important to ensure that the type signature of every redefined method in $\sigma(Y)$ is a subtype of its counterpart in $\sigma(X)$. In particular, redefined results must be subtypes (*covariance*) and redefined arguments must be supertypes (*contravariance*). This ensures that objects of type Y can safely be passed to variables of type X.

Due to the loss of type information when a class Y inherits methods in the parent type $\sigma(X)$, later work [Cardelli 85, Cardelli 88b, Ghelli 90] explored the

addition of bounded universal quantification to express the type of polymorphic methods. In bounded quantification, an object's methods are given the type $\forall(t \subseteq X).\sigma(t)$ to express the idea that they may acquire more precise type signatures for each subtype object of a given type. This work was founded on a simple generalisation of the second-order λ-calculus [Cardelli 85, Danforth 88]. The expectation was that a class Y, inheriting a method from X with the type $\forall(t \subseteq X).\sigma(t)$, would obtain a version typed in $\sigma(Y)$ by the replacement of the type parameter t. Unfortunately, this expectation was only fulfilled in the context of non-recursive types. In λ-calculus explanations of type recursion, the recursive type of t has to be fixed at t = X, making bounded quantification no more expressive than simple subtyping for methods inherited by the subtype Y. Apart from the addition of polymorphism to the type model, a class Y was still expected to conform to a class X according to simple subtyping. Cardelli has since moved away from λ-calculus models in favour of his own object-calculus [Cardelli 90, Cardelli 91, Cardelli 92a, Cardelli 92b].

At around the same time, work by Cook, Canning, Hill, Olthoff and Mitchell [Cook 89a, Canning 89a, Canning 89b, Cook 89c, Cook 90] showed that bounded quantification did not deliver useful type signatures for polymorphic functions in the context of recursive types. Their F-bounded quantification, a higher-order subtyping theory, delivers more appropriate types for the same polymorphic functions. In F-bounded polymorphism, an object's methods are given the type $\forall(t \subseteq F[t]).\sigma(t)$ to express the fact that the type of *self* in the inherited part of an object must change before being combined with new methods. A class is not a type X, but a family of types whose upper bound is constrained by a type generator $\forall(t \subseteq F_X[t])$, a polymorphic construct from which the simple types of objects can be recovered when they are created. The simple type X of an object created from a class $\forall(t \subseteq F_X[t])$ is given by $X = F_X[X]$, which is explained in the λ-calculus in terms of the fixed point theory of recursion. The result of applying $F_X[t]$ to X delivers a set of methods for the object in the type $\sigma(X)$. An object of type Y inheriting methods in the type $\forall(t \subseteq F_X[t]).\sigma(t)$ obtains versions which are retyped in $\sigma(Y)$, exactly as desired. As a result, inheritance is not the same thing as simple subtyping, since it is nearly always the case that methods $\sigma(Y)$ are not subtypes of methods $\sigma(X)$, therefore Y is not a subtype of X. Instead, different type checking rules are used, based on a point-wise comparison of types created from generators. Instead of insisting that $Y \subseteq X$, F-bounded quantification requires all possible instantiations of the two generators to be pointwise in a subtype relationship, in other words, where the class constrained by $\forall(t \subseteq G_Y[t])$ inherits from the class $\forall(t \subseteq F_X[t])$, we require that:

$$\forall(t \subseteq G_Y[t]).G_Y[t] \subseteq F_X[t].$$

Object-oriented languages reflect the above findings in different ways, depending on their adoption of simple subtyping, bounded quantification or F-bounded quantification to explain the typing of inheritance. Trellis [Schaffert 86] is one of the earliest languages treating classes uniformly as types to handle subtype relations correctly. Together with the type rules of POOL-I [America 90] and of Emerald [Black 86], Trellis observes both the covariant rule for function results and contravariant rule for function arguments. Sather [Omohundro 94] has a polymorphic type variable *SAME*, which it interprets as a syntactic abbreviation for the type of each new inheriting class [Szypersky 93]. This means that inherited functions returning results in the type *SAME* are considered subtype functions, so long as *SAME* does not also occur as an argument type.

Generally, Eiffel identifies *class* with *type* and *inheritance* with *subtyping* [Meyer 88, Meyer 92]. In its typing rules for function replacement, it follows the covariant rule for results, but rejects the contravariant requirement for arguments [Meyer 89]. Coincidentally, Eiffel obeys contravariance for axiom redefinition [Simons 94b], although this is justified in terms of Meyer's *programming by contract* metaphor, rather than from subtyping considerations [Meyer 88, p256-7]. This internal inconsistency has been discussed on the internet newsgroup *comp.lang.eiffel*.

Above, Cook and Meyer both agree that subtyping models unfairly restrict the expressiveness of object-oriented languages. In view of this, it is clear that F-bounded polymorphism has a lot to offer. F-bounded type matching rules have been introduced in the experimental languages Abel

[Harris 91], TOOPL [Bruce 94a] and PolyTOIL [Bruce 94b], exclusively to describe the *self*-type. Elsewhere, the substitution of one component for another is explained using simple subtyping. In our proposal below, we extend the use of F-bounds to describe the internal polymorphic parts of a class. So far, no language has based its polymorphic types uniformly on F-bounds. We show how this approach significantly simplifies the typing of object-oriented programs, while capturing all their traditional flexibility.

3. Difficulties with Conformance and Subtyping

In Eiffel, a class *conforms* to another if it inherits from it [Meyer 92, p219], whether directly or transitively. Eiffel's conformance deviates from subtyping by allowing the uniform specialisation of function arguments and results in descendant classes. Programs may therefore be passed as type-correct, but hide run-time type failure [Cook 89b]. We illustrate this with a practical signal-processing example in Eiffel [Simons 94a]. Two kinds of SAMPLE are devised to encapsulate INTEGER data values:

```
class SAMPLE
feature     -- Creation omitted for brevity
    data : INTEGER;
    magnitude : INTEGER is
    do      -- absolute value of sample
            if data < 0
                then Result := - data
                else Result := data
            end
    end
end -- SAMPLE

class POWER_SAMPLE inherit SAMPLE
feature     -- Creation omitted for brevity
    power : INTEGER is
    do      -- square of sample
            Result := data * data
    end
end -- POWER_SAMPLE
```

such that a POWER_SAMPLE object has in total the features *data*, *magnitude* and *power*. Next, we devise two kinds of SIGNAL which process samples of each respective variety. Our intuition is that SIGNALs rectify their SAMPLEs by calculating their magnitude, whereas POWER_SIGNALs do this by calculating the power of their POWER_SAMPLEs:

```
class SIGNAL
feature     -- Creation omitted for brevity
    rectify(arg : SAMPLE) : INTEGER is
    do      -- simple strategy
            Result := arg.magnitude
    end
end -- SIGNAL

class POWER_SIGNAL inherit SIGNAL
redefine rectify
feature     -- Creation omitted for brevity
    rectify(arg : POWER_SAMPLE) :
            INTEGER is
    do      -- more sophisticated strategy
            Result := arg.power
    end
end -- POWER_SIGNAL
```

The salient fact is that *rectify()* in SIGNAL has been replaced in POWER_SIGNAL by a function expecting a subtype argument, in violation of contravariance. This seems reasonable until you have a variable elsewhere in the program which expects a SIGNAL and is passed an argument which is a POWER_SIGNAL:

```
some_routine is
local
    sam : SAMPLE;
    sig : SIGNAL;
    pow : POWER_SIGNAL;
do
    !!sam.make(...);
            -- creates a SAMPLE
    !!pow.make(...);
            -- creates a POWER_SIGNAL
    sig := pow;
            -- statically correct,
            -- since pow ⊆ sig
    sig.rectify(sam);
            -- statically correct; but
            -- hides runtime type failure!
end -- some_routine
```

Here, since *sig* now legally contains an instance of POWER_SIGNAL, *sig.rectify(sam)* invokes the replaced function defined in POWER_SIGNAL. This call was checked statically with respect to SIGNAL, yet the replaced function invokes in turn a call to *sam.power*, which fails since its argument

is of type SAMPLE, for which *power* is not defined.

Meyer's patch to fix this [Meyer 89, p14-17] monitors aliasing of the kind *sig := pow*; and in this context retypes the features of SIGNAL with the most restricted types of any object it aliases, anywhere in the system. For this reason it is called a "global" check. At assembly time, *sig.rectify(sam)* would therefore raise a type error, where the retyped *rectify* is passed too general an argument.

Violating contravariance is also responsible for other kinds of formal type failure involving generic classes in Eiffel. If we parameterise our SIGNAL and POWER_SIGNAL classes:

```
class SIGNAL [S -> SAMPLE]
feature     -- Creation omitted for brevity
    rectify(arg : S) : INTEGER is
    do      -- simple strategy
            Result := arg.magnitude
    end
end -- SIGNAL

class POWER_SIGNAL
    [T -> POWER_SAMPLE]
    inherit SIGNAL [T] redefine rectify
feature     -- Creation omitted for brevity
    rectify(arg : T) : INTEGER is
    do      -- more sophisticated strategy
            Result := arg.power
    end
end -- POWER_SIGNAL
```

we can construct types POWER_SIGNAL [POWER_SAMPLE] and SIGNAL [SAMPLE] which obey the parametric form of the conformance rule in [Meyer 88, p262] and [Meyer 92, p221-224]; yet clearly the same kind of type failure as the one above results from aliasing an object of type POWER_SIGNAL [POWER_SAMPLE] with a variable of type SIGNAL [SAMPLE]. Cook points out that a subtyping relationship between two generic classes (here, between POWER_SIGNAL [T] and SIGNAL [S]) is not assured by respectively substituting conformant actual types (here, POWER_SAMPLE which conforms to SAMPLE) into their formal generic parameters (T and S). Instead, the issue is whether the resulting types, with all parameters replaced, obey subtyping. Where a generic parameter occurs as an argument type in some routine, contravariance requires that a *more general* substitution be made.

It is clear (notwithstanding the "global system validity check") that Eiffel's conformance rules are formally too lax. But it is equally clear that forcing Eiffel to obey subtyping would severely limit the language. In the worst case, a type appearing both as an argument and as a result to a function could never be redefined, since it would have to obey contravariance and covariance simultaneously. For this reason, we reject simple subtyping as a model for type substitution during polymorphic inheritance.

4. Difficulties with Generic and Anchored Types

Eiffel has three distinct type mechanisms, which can interact in unusual ways. Hereafter, we shall use *conformance* in a restricted sense, to distinguish Eiffel's inheritance-based type compatibility from its other mechanisms. The second mechanism is *anchored types*. These are declared using a syntax: *<variable> : like <anchor>* and have the intention of linking the type of the variable to the type of some other attribute or argument, named as the anchor. If the type of the anchor is subsequently redefined, then the variable type changes to match this. Mathematically, this is a kind of type substitution mechanism [Palsberg 94], or an implicit variant of parametric polymorphism with a default instantiation for the parameters [Cook 89b]. The third mechanism is called *genericity*, or *generic types*. This is an explicit parametric polymorphism, generalised in [Meyer 92] to allow constraints placed on the actual types which may replace the parameters. In our example from section 3 above, SIGNAL [S -> SAMPLE] is the type of a SIGNAL which has a parameterised sample type S. S may be replaced by SAMPLE or indeed by any type conforming to SAMPLE, such as POWER_SAMPLE.

Our first difficulty with Eiffel is that it makes no clear distinction between monomorphic and polymorphic types. In non-object-oriented languages, a variable with the type SAMPLE contains an object of exactly this type. In Eiffel, it may also contain objects of any conforming type, such as POWER_SAMPLE. In effect, any variable with a simple type is also inherently polymorphic and this weakens the notion of simple type in Eiffel.

Given better mechanisms to handle polymorphism, Eiffel should retain a straightforward static type (its non-reference *expanded* types nearly have this quality). In any case, the polymorphism based on conformance is not especially useful in the context of inheritance, since it leads to the kind of type-loss described in section 2.

To overcome this, the *like* <*anchor*> mechanism permits inherited functions to evolve in type, such that no type-loss occurs when they are attached to a new type. Anchored types, especially *like Current*, permit the automatic retyping of inherited functions. As we shall see later, anchored types form the basis for an F-bounded polymorphic interpretation of classes. However, in [Meyer 89] they are merely supposed to be syntactic abbreviations for the types which replace them. Against this, it is clear [Meyer 92, p224-6] that anchored types do not enter into further conformance relations quite like normal types. In the following:

```
sam1 : SAMPLE;
sam2 : like sam1;
sam3 : POWER_SAMPLE;
sam1 := sam2;
        -- OK because like sam1
        -- conforms to SAMPLE
sam2 := sam1;
        -- disallowed, even though
        -- sam1 and sam2 have equal type
sam2 := sam3;
        -- disallowed, even though
        -- sam3 conforms to sam1
```

only the first assignment is allowed. The others are disallowed on the grounds that anchored types *like X* may be subsequently redefined in descendant classes and are therefore not safe targets for assignment, except given another object of type *like X*. Anchored types behave less like standard Eiffel types and more like a classic parametric polymorphic mechanism in which, once the parameters are replaced, the type of the expression is static. We are in favour of such a clear distinction between monomorphism and polymorphism; however, it seems strange that Eiffel's quasi-static types are more flexible than its polymorphic types!

Generic types in Eiffel serve the same purpose as the classic parametric polymorphic mechanisms in languages like Standard ML [Milner 90] or Ada [Ichbiah 79]. In constructions such as LIST [T] and ARRAY [T], the type parameter T abstracts over the unknown element type. It is useful to delay instantiating the parameter until the point of use, since this allows client programs to make static assumptions about the element-type. A client program using a LIST [SAMPLE] can depend on its elements having the static type SAMPLE. A further flexibility is introduced by constrained genericity [Meyer 92, p202-3]. The supplier of the class SIGNAL [S -> SAMPLE] can also make assumptions about the type instantiating S. He can write routines inside SIGNAL that depend on its register containing something of the static type SAMPLE. This generalisation of parametric polymorphism was present earlier in the experimental language Russell [Demers 78].

However, it is not the case in Eiffel that instantiating all type parameters results in a static type. A LIST [SAMPLE] is still inherently a polymorphic type. Typically, this property is put to good use when constructing heterogenous collections, in which a LIST [SAMPLE] object contains elements of mixed types, such as SAMPLEs and POWER_SAMPLEs. But this flexibility also opens up the type system to allow a variable of the type LIST [SAMPLE] to receive objects of the types TREE [SAMPLE], TREE [POWER_SAMPLE] and so on (assuming that TREE [T] conforms to LIST [T]). We believe that this significantly reduces the useful type constraints provided by a parametric polymorphic mechanism. Again, it seems ironic that the anchored type mechanism in Eiffel currently provides a stronger constraint than this. Nonetheless, we shall want to retain the ability to build heterogenous collections.

In combination, it seems that Eiffel has too many type mechanisms. Its quasi-static type is actually polymorphic and mixes with two other polymorphic type mechanisms. Because of this, it is possible to specify essentially the same polymorphic type in different ways. Below, we define the SIGNAL class slightly differently from the definitions given in section 3. Here, SIGNAL has a component attribute *register* to hold a single polymorphic SAMPLE. The *rectify()* function now operates upon the value stored in this *register*. We may obtain very similar styles of SAMPLE polymorphism using conformance and genericity in the definition of SIGNAL:

```
class SIGNAL
        -- to hold polymorphic samples
feature
        -- polymorphic register based
        -- on conformance
        register : SAMPLE;
        rectify : INTEGER is
        do Result := register.magnitude end
end -- SIGNAL

class SIGNAL [S -> SAMPLE]
        -- to hold polymorphic samples
feature
        -- polymorphic register based
        -- on genericity
        register : S;
        rectify : INTEGER is
        do Result := register.magnitude end
end -- SIGNAL
```

A marginal increase in flexibility is gained through parameterising the *register*, since although a SIGNAL [POWER_SAMPLE] will *rectify* its samples in exactly the same way as a SIGNAL [SAMPLE], its *register* will return an element of type POWER_SAMPLE; whereas the conformance-based *register* can only return the static type SAMPLE, even when it contains a POWER_SAMPLE object. It is this insight which allows us to see that constrained genericity fulfils all the obligations of conformance-based polymorphism; and provides more by capturing the evolving type of the *register*, much in the spirit of *like <anchor>*. For this reason, we propose to replace Eiffel's conformance polymorphism by an explicit parametric mechanism with constraints.

5. A Uniform F-Bounded Polymorphism

Our proposal for a *new Eiffel* depends on distinguishing the notions of *class* and *type*. A *class* defines a polymorphic family of objects having different but related types. A *type* defines a monomorphic family of objects having identical type. The polymorphic type of a class is constrained by an F-bound [Canning 89a] expressed using a type generator. This generator will always have at least one unreplaced type parameter, usually standing for the type of *self*, equivalent to *like Current*. A monomorphic type is defined as the least fixed point of such a type generator [Cook 90].

We suggest that a *new Eiffel* class SAMPLE[] might be introduced in the following way, using square brackets to enclose a type parameter T, standing for the *self*-type:

```
class SAMPLE [T]
feature     -- Creation omitted for brevity
    data : INTEGER;
    magnitude : INTEGER is
    do      -- absolute value of sample
            if data < 0
                    then Result := - data
                    else Result := data
            end
    end
end -- SAMPLE
```

This style supercedes Eiffel's current usage of square brackets to introduce generic sub-parts of a class (reconsidered below), since the parameter stands here for the whole class. The special construction *like Current* is no longer needed, since T may be used elsewhere in the class to denote the *self*-type. A class definition is understood to have the meaning of a generator for a family of objects with the polymorphic type $\forall(T \subseteq \Phi SAMPLE\ [T]).T$, where:

$$\Phi SAMPLE = \lambda T.\{data: INTEGER,\ magnitude: INTEGER\}$$

In *new Eiffel*, we suggest that a monomorphic type SAMPLE might be introduced implicitly, being understood as the least fixed point of the SAMPLE[] class. In λ-calculus terms, this relates our notions of class and type in the following way:

```
SAMPLE = Y ΦSAMPLE
        -- type is least fixed point of class
SAMPLE = ΦSAMPLE [SAMPLE]
        -- type satisfies its class's F-bound
```

Generally we contrast the notation SAMPLE[], used to denote a polymorphic *class*, with the notation SAMPLE, used to denote a monomorphic *type*. In what follows, class features will either be given static types or class types.

Next, we incorporate old-style generic parameters. A *new Eiffel* class may introduce several parameters constrained by F-bounds, abstracting over further polymorphic sub-parts. This has the syntactic feel of Eiffel's existing constrained generic

parameter mechanism, but also replaces conformance-based polymorphism. In order to avoid the proliferation of type parameters up front, we simply introduce them internally at the point where they are needed:

```
class SIGNAL [S]
feature      -- Creation omitted for brevity
    register : SAMPLE [T];
    rectify(arg : SAMPLE [T]) : INTEGER is
    do       -- simple strategy
             Result := arg.magnitude
    end
end -- SIGNAL
```

Formally, the *self*-type S of class SIGNAL[] depends on the particular type substituted into the parameter T in *register* and *rectify*, which must satisfy the F-bound of SAMPLE []. We understand such a parameterised class definition to mean a generator for a family of objects having the polymorphic type:

$$\forall(T \subseteq \Phi SAMPLE\ [T]).\ \forall(S \subseteq \Phi SIGNAL\ [T, S]).S$$

where:

$$\Phi SIGNAL = \lambda T.\lambda S.\{register: T,\ rectify: T \rightarrow INTEGER\}$$

This kind of uniform F-bounded typing for polymorphic sub-parts has not been widely advocated before. PolyTOIL's *matching* rule [Bruce 94b] explains the typing of *self* using F-bounds and otherwise requires redefined functions to obey subtyping. The Johns Hopkins group [Eifrig 94] propose (F-bounded) *open* and (monomorphic) *closed* types for *self*, with monomorphic types elsewhere. The Abel group [Harris 91] went as far as considering subtype-bounded parameters for polymorphic sub-parts in their unfinished final report.

The introduction of polymorphic sub-parts in a class results formally in the stacking up of type parameters, which later must be bound to actual types. Rather than burden the programmer with the full λ-calculus machinery for the flexible rebinding and fixing of types, we suggest an explicit parameter-substitution style which uses nesting to respect dependency among self-types. We give some example types with their equivalent λ-calculus meanings:

```
sig_of_pow_sam : SIGNAL [S = SIGNAL
      [T = POWER_SAMPLE]];

= Y (ΦSIGNAL [Y ΦPOWER_SAMPLE])

polysig_of_sam : SIGNAL [[T = SAMPLE]];

= ∀(S ⊆ ΦSIGNAL [(Y ΦSAMPLE), S]).S

sig_of_polysam : SIGNAL [S = SIGNAL];

= ∀(T ⊆ ΦSAMPLE [T]).(Y (ΦSIGNAL [T]))
```

This illustrates how parameters may be replaced in any order. In keeping with our earlier notational conventions on *class* and *type*, we adopt the usual sugared syntax to write the monomorphic type SIGNAL, meaning the (frequently used) double least fixed point:

```
signal_of_sample : SIGNAL;

= Y (ΦSIGNAL [Y ΦSAMPLE])
```

A similar binding style can be used to retype polymorphic features during inheritance:

```
class POWER_SAMPLE [U]
inherit SAMPLE [T = U]
feature      -- Creation omitted for brevity
    power : INTEGER is
    do       -- square of sample
             Result := data * data
    end
end -- POWER_SAMPLE
```

T = U captures the retyping of *self*, in the same manner as using *like Current* in old Eiffel. Formally, the *self*-type U of the new class is distributed to the parent's type generator, according to Cook's model of inheritance [Cook 89c, Cook 90]. We understand inheritance to be a shorthand for defining a child class ∀(U ⊆ ΦPOWER_SAMPLE [U]).U, where:

$$\Phi POWER\_SAMPLE = \lambda U.\Phi SAMPLE\ [U] \oplus \{power: INTEGER\}$$

$$= \lambda U.\{data: INTEGER,\ magnitude: INTEGER,\ power: INTEGER\}$$

using Cook's operator ⊕ to perform record type concatenation with override.

Since we must extend Cook's model to allow rebinding of polymorphic sub-parts, inheritance may now involve a multiple distribution of new type parameters to old:

```
class POWER_SIGNAL [P]
inherit SIGNAL [S = P[T = U]]
        redefine rectify
feature         -- Creation omitted for brevity
        rectify(arg : POWER_SAMPLE [U]) :
                INTEGER is
        do      -- more sophisticated strategy
                Result := arg.power
        end
end -- POWER_SIGNAL
```

Again, we use nesting to indicate that the *self*-type S has been rebound to a P in which the component type T is rebound to a U. Formally, this inheritance construction produces a generator for a more restricted class of objects. This class has the polymorphic type:

$$\forall(U \subseteq \Phi\text{POWER\_SAMPLE } [U]).\ \forall(P \subseteq \Phi\text{POWER\_SIGNAL } [U, P]).P$$

where:

$$\Phi\text{POWER\_SIGNAL} = \lambda U.\lambda P.\Phi\text{SIGNAL } [U, P] \oplus \{\text{rectify: } U \rightarrow \text{INTEGER}\}$$
$$= \lambda U.\lambda P.\{\text{register: } U, \text{rectify: } U \rightarrow \text{INTEGER}\}$$

We draw special attention to the fact that, due to the way F-bounds work, the retyping of *rectify* involves a *covariant* restriction. This is correct, since it is not possible to obtain a more general F-bound by merging a type parameter with a less restricted parameter, because the pointwise instantiation rule for F-bounds would ensure that the bound drawn over both were the more restricted of the two. Mathematically, this is the greatest lower bound F[t] in:

$$\forall(t \subseteq F[t]).\ F[t] \subseteq \Phi\text{POWER\_SAMPLE } [t] \wedge F[t] \subseteq \Phi\text{SAMPLE } [t]$$

Since we already know that:

$$\forall(t \subseteq \Phi\text{POWER\_SAMPLE } [t]).\ \Phi\text{POWER\_SAMPLE } [t] \subseteq \Phi\text{SAMPLE } [t]$$

by inheritance, then the most general solution is $F[t] = \Phi\text{POWER\_SAMPLE } [t]$.

6. Binding and Scope of Type Parameters

Everywhere in *new Eiffel*, we distinguish variables with a simple static type from those with a polymorphic type, using square brackets to indicate polymorphic types:

```
sam1 : SAMPLE;      -- monomorphic type
sam2 : SAMPLE [];   -- polymorphic class
sam3 : SAMPLE [X]; -- polymorphic class
```

The variable *sam1* has the monomorphic type SAMPLE and can only receive objects of exactly this type. The variable *sam2* has the polymorphic type $\forall(T \subseteq \Phi\text{SAMPLE } [T]).T$ and may therefore receive objects of any type satisfying this bound, for example, POWER_SAMPLE $\subseteq$ ΦSAMPLE [POWER_SAMPLE]. The variable *sam3* is also polymorphic, but links its type instantiation to other occurrences of X in the local context, so we have:

```
pow : POWER_SAMPLE;
sam1 := pow;
        -- incorrect, since pow and sam1
        -- have different types
sam2 := pow;
        -- correct, since pow observes
        -- sam2's F-bound
sam3 := pow;
        -- correct, provided that the binding
        -- X = POWER_SAMPLE is consistent
```

The most general object type which may be assigned to a polymorphic variable is the least fixed point of the F-bound. This places an upper bound on polymorphic assignment in a way analogous to the existing Eiffel conformance rule. *Linked* polymorphic types allow the propagation of strong type-constraints within a local block, like Eiffel's existing type parameters. Formally, we bind all such local type parameters upon entry to the block:

$$\lambda(X \subseteq \Phi\text{SAMPLE } [X]).(\ldots \text{sam3} := \text{pow}; \ldots)\ [\text{POWER\_SAMPLE}]$$

to prevent the multiple instantiation of X by different types. Polymorphic local variables and function arguments have the scope of their function body. Object attributes have a scope bounded by the outermost function with a handle on the same *self*. *If*-statements are problematic only when alternative type bindings occur in each branch. In this case, we compute the most specific type information possible (cf [Eifrig 94]) and insert a run-time type check when retrieving objects from polymorphic variables.

Once a polymorphic type has been instantiated, all expressions in that type are fixed for the whole of the enclosing block. We assume a generalised type binding rule of the form:

$$\frac{\Gamma \vdash v : \forall(T \subseteq F[T]).\Theta[T], \quad \Gamma \vdash e : \Theta[t \subseteq F[t]]}{\Gamma, v := e \vdash v : \Theta[t]} \text{ ASSIGN-LVAR}$$

in which T is a type parameter, t is a type and Θ[] is a type expression, possibly involving further F-bounded quantification. The rule says that in a given context Γ, if variable v has a polymorphic type and expression e replaces the first parameter in v with a suitable type, then the assignment v := e extends the context Γ in which v is partially instantiated with the type of e. The rule may be applied recursively until no more parameters can be instantiated. To illustrate the power of our parametric scheme, we shall type-check the earlier example from section 3 which was incorrectly accepted by Eiffel's current conformance rule:

```
some_routine is
local
        sam : SAMPLE;
                -- monomorphic, T fixed
        sig : SIGNAL [];
                -- polymorphic, S not fixed
        pow : POWER_SIGNAL;
                -- monomorphic, P and U fixed
do
        !!sam.make(...);
        !!pow.make(...);
        sig := pow;
                -- polymorphic assignment
        sig.rectify(sam);
                -- static type failure detected!
end -- some_routine
```

Our type system is expressive enough to permit polymorphic *sig* to alias monomorphic *pow*, but prohibit the passing of too general an argument *sam* to the aliased object's *rectify*. Initially, the type checker sets up the following context:

$\Gamma_0 \vdash$
sam : SAMPLE,
pow : POWER_SIGNAL,
sig : $\forall(T \subseteq \Phi$SAMPLE [T]).
$\forall(S \subseteq \Phi$SIGNAL [T, S]).S

When the checker encounters the assignment *sig := pow*, this propagates actual type information into *sig*. The assignment rule requires the unpacking of the recursive type POWER_SIGNAL since it wants to bind parameters in the order T, S. This done, the rule is applied twice to generate extended contexts:

$\Gamma_1 \vdash$
sam : SAMPLE,
pow : POWER_SIGNAL,
sig : $\forall(S \subseteq \Phi$SIGNAL[
POWER_SAMPLE, S]).S

$\Gamma_2 \vdash$
sam : SAMPLE,
pow : POWER_SIGNAL,
sig : POWER_SIGNAL

Now, the call *sig.rectify(...)* is checked. This function has acquired the static type:

POWER_SIGNAL → (POWER_SAMPLE → INTEGER)

and expects an argument of static type POWER_SAMPLE; but instead it gets one of the wrong type, SAMPLE. The error is therefore a straightforward static type mismatch.

Our type system provides a much stronger and more subtle static typing than is currently available in object-oriented languages. The call *sig.rectify(...)* in Eiffel would normally be resolved by dynamic binding. Because we propagate static type information in our system, we can bind *rectify* statically. We believe that Meyer's "global system validity check" would permit *rectify* in this context to accept an argument of static type POWER_SAMPLE [Meyer 89, p14-17] since he adopts a pessimistic policy of retroactively retyping

the features of *sig* : SIGNAL with the most restricted types of any object it aliases, anywhere in the system. Our example will clearly allow this, but will also allow the more subtle *sig.rectify(sam)* where *sam* has the declared polymorphic type $\forall(t \subseteq \text{SAMPLE}\ [t])$ and in the same block receives an object of type POWER_SAMPLE. Clearly, Eiffel could not statically detect the dynamic type of objects without an expensive global flow analysis. Our approach is not expensive and less pessimistic than Meyer's, since type constraints are propagated within local blocks, rather than across the whole system.

7. Conclusions and Further Work

We have presented a revised syntax for Eiffel which has only two kinds of typed variable: monomorphic and polymorphic. We have eliminated *conformance*-based polymorphism, on the grounds that in its current form it is mathematically unsound; and were it to be corrected, it would be practically useless as a mechanism for typing object-oriented languages. We retain polymorphic assignment and argument-passing, propagating type constraints into blocks wherever parameters receive actual types, or are merged with other type parameters (not described in detail in this paper). In the latter case, the constraint preserved is the greatest lower bound. Algorithms for performing this kind of analysis already exist, such as Robinson's unification algorithm [Robinson 65]. Our approach to polymorphism is more in harmony with other parametric approaches [Milner 78] but is new in that it is based on F-bounds. The systematic use of F-bounded parameters to type polymorphic structures has not been proposed before. Earlier work has concentrated only on the typing of *self*.

Linking the instantiation of type parameters naturally leads to a homogenous style of polymorphism. The order of quantification for types with polymorphic sub-parts usually binds the type of the sub-part first. It is therefore easy to design homogenous lists, arrays and trees using our approach. We first suggested using *unlinked* type parameters as a basis for heterogenous polymorphism in [Simons 94a]. By binding the tail of a recursive structure over a different component type parameter from the head, we obtain a structure with the same polymorphic type as *self*, but having a different parameterisation. The parameters may be instantiated by different types conforming to the same F-bound, leading to heterogenous structures. In order to do this, we must first reverse the order of quantification for the *self*-type and the component-type, binding the component inside the quantification of *self*. This is a higher-order approach, since the *self*-type parameter must now range over recursive type functions, rather than over recursive types [Harris 91].

Our parametric mechanism propagates more static information than comparable type systems [Simons 94c]. Nonetheless, heterogenous collections and branching constructs with mixed types, such as *if*-statements which return one from a collection of simple types all satisfying the same F-bound, give rise naturally to unresolved polymorphic types. A compiler can statically detect such situations [Palsberg 94] and insert exactly the required form of dynamic type check at the point where objects are transferred to variables with simple, or more restricted polymorphic types. Throughout, we still require only two kinds of typed variable: mono- and polymorphic.

Further work in this area must include a syntactic mechanism for propagating type constraints between two mutually dependent F-bounds, generating mutually recursive types. A non-parametric type substitution model [Palsberg 94] has shown that this is possible in principle. We have also recently explored the type-theoretic effect of adding class axioms into our model [Simons 94b]. This is a salient concern in any complete type system for Eiffel.

The author would like to thank William Cook for inspiration, Kim Bruce and Warren Harris for their help and influence on the development of these ideas.

References

[America 90] P America (1990), 'Designing an object-oriented language with behavioural subtyping', *Proc. Conf. Foundations of Object-Oriented Lang.*, 60-90.

[Black 86] A Black, N Hutchison, E Jul and H Levy (1986), 'Object structure in the Emerald system', *Proc. 1st ACM Conf. Object-Oriented Sys., Lang. and Appl.*, 78-86.

[Bruce 94a] K Bruce (1994), 'A paradigmatic object-oriented programming language: design, static typing and semantics', *J. of Func. Prog., 4(2)*, 127-206.

[Bruce 94b] K Bruce, A Schuett and R van Gent (1994), 'PolyTOIL: a type-safe polymorphic object-oriented language', *Technical Report, Department of Computer Science, Williams College*; also published as an extended abstract in *ECOOP '95*.

[Canning 89a] P Canning, W Cook, W Hill, W Olthoff and J Mitchell (1989), 'F-bounded polymorphism for object-oriented programming', Proc. 4th Int. Conf. Func. Prog. Lang. and Arch., Imperial College London, September, 273-280.

[Canning 89b] P Canning, W Cook, W Hill and W Olthoff (1989), 'Interfaces for strongly-typed object-oriented programming', *Proc. 4th ACM Conf. Object-Oriented Lang., Sys. and Appl.*, 457-467.

[Cardelli 84] L Cardelli (1984), 'A semantics of multiple inheritance', in: *Semantics of Data Types, LNCS 173*, eds. Kahn, MacQueen and Plotkin, Springer Verlag, 51-68.

[Cardelli85] L Cardelli and P Wegner (1985), 'On understanding types, data abstraction and polymorphism', *ACM Computing Surveys, 17(4)*, 471-521.

[Cardelli 86] L Cardelli (1986), 'Amber', *Combinators and Functional Programming Languages, LNCS, 242*, 21-47.

[Cardelli 88a] L Cardelli (1988), 'A semantics of multiple inheritance', *Information and Computation, 76*, 138-164.

[Cardelli 88b] L Cardelli (1988), 'Structural subtyping and the notion of power type', *Proc. 15th ACM Symp. Principles of Prog. Langs.*, 70-79.

[Cardelli 90] L Cardelli (1990), 'Notes about $F_{\leq}$', unpublished manuscript.

[Cardelli91] L Cardelli and G Longo (1991), 'A semantic basis for Quest', *J. of Func. Prog., 1(4)*, 417-458.

[Cardelli 92a] L Cardelli (1992), 'Extensible records in a pure calculus of subtyping', *Research Report 81, DEC Systems Research Center*, January. Reprinted in: *Theoretical Aspects of Object-Oriented Programming*, eds. C A Gunter and J C Mitchell (1994), MIT Press, 373-426.

[Cardelli 92b] L Cardelli and J Mitchell (1992), 'Operations on records (summary)', *Proc. 5th Int. Conf. Math. Found. Prog. Lang. Semantics*, pub. *LNCS, 442*, Springer Verlag, 22-52.

[Cook 89a] W Cook (1989), *A denotational semantics of inheritance*, PhD Thesis, Brown University.

[Cook 89b] W Cook (1989), 'A proposal for making Eiffel type-safe', *Proc. 3rd European Conf. Object-Oriented Prog.*, 57-70; reprinted in *Computer Journal 32(4)*, 305-311.

[Cook 89c] W Cook and J Palsberg (1989), 'A denotational semantics of inheritance and its correctness', Proc. 4th ACM Conf. Object-Oriented Prog., Sys., Lang. and Appl., 433-443.

[Cook 90] W Cook, W Hill and P Canning (1990), 'Inheritance is not subtyping', *Proc. 17th ACM Symp. Principles of Prog. Lang.*, 125-135.

[Demers 78] A Demers, J Donahue and G Skinner (1978), 'Data types as values: polymorphism, type-checking, encapsulation', *Proc. 5th ACM Symp. on Principles of Prog. Langs.*, 23-30.

[Danforth 88] S Danforth and C Tomlinson (1988), 'Type theories and object-oriented programming', *ACM Computing Surveys, 20(1)*, 29-72.

[Eifrig 94] J Eifrig, S Smith, V Trivonov and A Zwarico (1994), 'Application of object-oriented type theory: state, decidability and integration', *Proc. 9th ACM Conf. Object-Oriented Prog., Sys., Lang. and Appl.*, 16-30.

[Ghelli 90] G Ghelli (1990), 'Modelling features of object-oriented languages in second-order functional languages with subtypes', *LNCS, 489*, 311-337.

[Harris 91] W Harris (1991), *Typed Object-Oriented Programming: ABEL Project Posthumous Report*, Hewlett-Packard Laboratories.

[Ichbiah 79] J Ichbiah, J Barnes, J Heliard, B Krieg-Bruckner, O Roubine and B Wichmann (1979), 'Rationale and design of the programming language Ada', *ACM Sigplan Notices, 14(6)*.

[Meyer 88] B Meyer (1988), *Object-Oriented Software Construction*, Prentice-Hall.

[Meyer 89] B Meyer (1989), 'Static typing for Eiffel', *Interactive Software Engineering Technical Report TR-EI-18, January; updated in July 1989*; also supplied with: 'Eiffel 2.3 Release Notes', *Interactive Software Engineering, Inc.*, 1990.

[Meyer 92] B Meyer (1992), *Eiffel: The Language*, Prentice-Hall.

[Milner 78] R Milner (1978), 'A theory of type polymorphism in programming', *J. of Comp. and Sys. Sci., 17*, 348-375.

[Milner 90] R Milner, M Tofte and R Harper (1990), *The Definition of Standard ML*, MIT Press.

[Omohundro 94] S M Omohundro (1994), *The Sather 1.0 specification*, International Computer Science Institute, Berkley CA.

[Palsberg 94] J Palsberg and M I Schwartzbach (1994), *Object-Oriented Type Systems*, John Wiley.

[Robinson 65] J A Robinson (1965), 'A machine-oriented logic based on the resolution principle', *J. of ACM, 12(1).*

[Schaffert 86] C Schaffert, T Cooper, B Bullis, M Kilian and C Wilpolt (1986), 'An introduction to Trellis/Owl', *Proc. 1st ACM Conf. Object-Oriented Prog., Sys., Lang., and Appl.*, pub. *ACM Sigplan Notices, 21(11)*, 9-16.

[Simons 94a] A J H Simons (1994), *Exploring Object-Oriented Type Systems*, OOPSLA-94 Tutorial 28, ACM Press.

[Simons 94b] A J H Simons (1994), 'Adding axioms to Cardelli-Wegner subtyping', *Department of Computer Science Report CS-94-6*, University of Sheffield.

[Simons 94c] A J H Simons, Low E-K and Ng Y-M (1994), 'An optimising delivery system for object-oriented software', *Object-Oriented Systems 1(1)*, 21-44.

[Szypersky 93] C Szypersky, S Omohundro and S Murer (1993), 'Engineering a programming language: the type and class system of Sather', Technical Report TR-93-064, International Computer Science Institute, Berkley CA.