Virtual Types *are* Statically Safe

Mads Torgersen
Computer Science Department, Århus University
Ny Munkegade, DK-8000 Århus C, Denmark
(+45) 89 42 32 57
madst@daimi.aau.dk

Abstract

Virtual types are a combined genericity and covariance mechanism first introduced in BETA. Like most other covariant language constructs, virtual types in their original form depend on dynamic checking for type safety. This paper presents full statical type checking for virtual types, while not relying on any other special language mechanisms for safety. Good expressiveness is retained, as demonstrated by a simple but effective solution to the infamous ColourPoint problem.

1 Introduction

Madsen [11] states that "it is well known that at most two of the following three properties can be obtained at the same time:

1. Static typing.
2. Subtype substitutability.
3. Covariance."

It is the intention of this paper to demonstrate that this need not be so.

Virtual types, originally introduced in BETA[14], are a very powerful and expressive mechanism, which gives rise to covariance. In their original form, virtual types gave rise to a certain amount of dynamic typechecking, and this was adopted by a recent proposal for introducing virtual types in Java [9] by Thorup [18].

In this paper we introduce fully statical type checking of virtual types, while not compromising subtype substitutability in any way. The key to retaining expressiveness under these conditions is the ability – also present in BETA – of virtual types to be *final bound*.

2 What are virtual types

Virtual types are best thought of as symbolic names for classes. These names (or aliases) are themselves attributes of classes, and their *binding* – the class they are a name for – may be overwritten in subclasses. This overwriting – called *further binding* – must be *covariant*, i.e. to a subclass of the previous binding.

A special kind of further binding called a *final binding* has the additional property of fixing the binding of the virtual type: It may not be further bound in subclasses. Thus, in this way the potential further covariance of a virtual type may be explicitly prohibited.

2.1 An example

The "Cow Example", originating from a brilliant treatment of covariance by David Shang [17], illustrates the mechanism of virtual types and final bindings as well as serving as a basis for demonstrating their typing implications.

```
Animal = {
   FoodType <= Food;
   eat(f: FoodType) { ... }
}

Herbivore = Animal {
   FoodType <= Plants;
}

Cow = Herbivore {
   FoodType = Grass;
}
```

Figure 1: The cow example

FoodType is a virtual type. In the declaration of Animal it is *declared*, and bound to Food. This means that "an Animal has a FoodType, which is *some* subclass of Food". The consequence for the eat method also declared in Animal is that "an Animal can eat *some* kind of Food".

In the declaration of Herbivore as a subclass of Animal, FoodType is *further bound* to Plants. This means that "the FoodType of an Herbivore is *some* subclass of Plants". As a consequence "an Herbivore can eat *some* kind of Plants".

In the declaration of Cow as a subclass of Herbivore, FoodType is *final bound* to Grass. This means that "the FoodType of a Cow is *precisely* Grass". As a consequence "a Cow can eat *any* kind of Grass".

Leaving out an implementation of the eat method should emphasize the fact (already suggested by their name) that virtual types are primarily a matter of *typing* rather than *functionality*. The only functionality directly associated with virtual types is the possibility of instantiating them with the new operation.

In order to see the typing implications of virtual types, let us investigate some uses of the Animal hierarchy of Figure 1.

```
Farmer = {
  FeedCow(c: Cow, g: Grass) {
    c.eat(g);
  }
}
```

Figure 2: Cow feeding: A sure success

A Farmer, as defined if Figure 2, is able to take a cow and some grass, and give the grass to the cow. This, according to the interpretation of the Cow class given above, is quite alright, since any Cow will eat any kind of Grass.

```
BadFarmer = {
  FeedCow(c: Cow, m: Meat) {
    c.eat(m);
  }
}
```

Figure 3: Cow choking: A sure failure

At the other end, since Meat is definately not Grass, trying as the BadFarmer of Figure 3 to give it to a cow can never succeed, and is therefore a certain static type error.

Trying to generalize this mixed experience we move on to define a Keeper (Figure 4) who is able to take an animal and some food, and give the food to the animal.

```
Keeper = {
  FeedAnimal(a: Animal, f:Food) {
    a.eat(f);
  }
}
```

Figure 4: A keeper having a problem

Trying this out with a cow and some grass should work just as fine as for the Farmer above. Unfortunately, giving the cow meat or giving the grass to a lion seems to be equally possible. Thus, the FeedAnimal method of the Keeper class is type safe on some inputs but not on others. The problem is that the method assumes that *any* Animal can eat *any* kind of Food, while according to the declaration of Animal it is only guaranteed to eat *some* kind of Food.

BETA handles this optimistically by allowing the method to be statically type checked, but inserting code for performing a *dynamic* type check every time the method is invoked to ensure that only sensible (i.e. typesound) executions of a.eat(f) are allowed.

Essentially the *only* difference between the BETA approach and the one suggested in this paper is, that we do not allow such optimism. In order to buy the full static type safety that BETA doesn't have, the possibilty of writing code such as the Keeper is sacrificed. Where BETA inserts runtime type checks, we flag a static type error instead.

3 Other covariance mechanisms

This section investigates the relationship of virtual types with other language constructs giving rise to covariance. We look at the ability of virtual types to express these constructs, as well as the possibility of adding equivalents of final bindings to them, in order to try to gain the same static safety for these mechanisms as we just achieved with virtual types.

3.1 Signature covariance

In Eiffel [15], overwriting a method in a subclass allows for specialization of the method parameters, a facility one could call *signature covariance*. Since any parameter types may be specialized, only a global analysis of the program can determine whether unsafe situations arise.

It should be clear that this mechanism can be simulated with method parameters of virtual type, like f:FoodType in the eat method from Figure 1. Furthermore, more static typechecking is possible, exploiting that parameters with non-virtual or closed virtual type are guaranteed *not* to be covariant.

It is quite straightforward to imagine making languages like Eiffel safer by adding an anonymous parallel of final bindings; some kind of marker explicitly flagging a given parameter type as "not covariant" in subclasses.

3.2 Self recursive classes

[3] gives a good overview of various approaches to the problem of so-called *binary methods*; methods that are supposed to take arguments of the type of their enclosing object. One way to express this is to introduce a special keyword MyType [2] or SAME[16] which is covariant in the sense that it always denotes the class of the *actual* enclosing object - not the one where the MyType appears.

One might view (and implement) MyType as a virtual type which is automatically bound in all classes [18]. Such a solution makes static typing problematic, however, because MyType obviously cannot be final bound (except of course in a language that has final classes, such as Java). BETA lets the programmer manually specify such virtuals whenever needed [13], and final bind them at suitable places. An example of this approach is the CompareType of the ColourPoint example in Figure 12 and 13.

4 A pseudo-language

Until now virtual types have most often been described in the context of full-fledged commercial programming languages, specifically BETA and Java [14, 18]. This has the disadvantage that the typing aspects of virtual types are mangled with those of other constructs of these languages; e.g. part objects and block structure in BETA, or abstract and final classes, interfaces and constructors in Java.

For this reason we have chosen here to use an absolutely minimal pseudo-language, which contains no such features and is designed so as to let the properties of virtual types stand out for themselves.

```
Program      ::= ClassDecl*
ClassDecl    ::= ClassId = ClassId { VirtualDecl* VarDecl* MethodDecl* }
VirtualDecl  ::= VirtualId <= ClassId ;
               | VirtualId = PreType ;
VarDecl      ::= VarId : PreType ;
MethodDecl   ::= MethodId ( ParDecl* ) : PreType { Statement* }
ParDecl      ::= ParId : PreType
PreType      ::= ClassId
               | VirtualId
Statement    ::= Expression ;
               | Expression.VarId := Expression ;
               | return Expression ;
Expression   ::= this
               | ParId
               | Expression.VarId
               | Expression.MethodId ( Expression* )
               | new Type
Type         ::= ClassId
               | Expression.VirtualId
```

Figure 5: The pseudo-language

This approach carries with it the danger that its results cannot survive subsequent exposure to a more complex environment (indeed this is acknowledged as one of the motivating factors for applying formal type theory to Java in [6]). In order to try out the type checking scheme and programming techniques suggested in this paper in a somewhat more realistic setting, statically safe virtual types are currently being added to a compiler for a subset of Java; see Section 9.

The language, sketched in figure 5, is intended to be the essential core of a statically typed, statically scoped, class-based, single-inheritance object-oriented language. The only thing special is the presence of virtual types.

For clarity, and to ease the formulation of type rules, the syntax is somewhat restrictive. Several kinds of programming constructs and syntactic sugar with little impact on type checking might easily be added, and indeed many of the program examples throughout this paper belong to a more permissive superset. In the course of the following description some obvious relaxations and additions will be noted.

4.1 Declarations

A *program* is a number of class declarations. Each *class* has one superclass, which could have been optional. (Multiple inheritance would seriously complicate things and is not considered.) It furthermore has a body of attribute declarations, of which there are three kinds: Virtual types, instance variables and methods.

Virtual types are declared and further bound with "<=" and final bound with "=". A final binding prohibits further binding in subclasses. Virtual types are covariant, which means that a binding must always specialize the previous one. A virtual type V can be final bound to another one V', meaning that V will automatically have the same binding as V' in all subclasses. A virtual type is *closed* in a class if it is final bound to a class or another closed virtual type. Otherwise it is *open*.

Instance variables are mutable object references. Their declaration may not be overridden in subclasses. Immutable references could be added, but would complicate things due to the need for an initialization mechanism. As will be evident below, immutable references play a special role in the type checking of expressions, but method parameters, which *are* immutable, will suffice to demonstrate this point.

Methods are declared with a number of formal parameters, a return type (which in a "real" language would have been optional) and a body of statements. The body may be overridden in subclasses, but the signature (formal parameters and return type) cannot be changed. As already mentioned, formal parameters are immutable object references. Because covariance on signatures is not allowed, statically resolved method overloading as in $C++$ [7] could be added without problems. For the same reason a super construct for calling overwritten methods as in Smalltalk and Java [8, 9] would have identical type properties to this.

4.2 Statements and expressions

The syntax in figure 5 contains only statements and expressions that are important from a typing point of view. This means that there are no control structures, literal expressions or binary operators. Likewise, local variables within statements are omitted.

Only method parameters and the special constant this, denoting the current instance of the enclosing class, may be accessed directly, while all kinds of object attributes – variables, methods and virtual types – must be accessed via a *dot expression* (as in $e.a$, $e.m()$ and new $e.V$ respectively). Of course a free *VarId* a could easily be treated as syntactic sugar for this.a, as is normal practice.

4.3 Runtime behaviour

At runtime, the evaluation of an expression results in an object. Every object has a class, and is created with the primitive new. If the argument to new is a simple class, the new object gets that class. If is is of the form e.V, where V is a virtual type, then e evaluates to some runtime object, o. The binding of V in the class of o will then be the class of the new object.

The implementation of object creation requires the binding of virtual types to be represented at runtime. If the language included casting or typecase constructs, they too would need such runtime information, as would of course implicit runtime type checks as used in BETA.

5 Static type checking with virtual types

The purpose of this section is to give a detailed constructive description of how the type checking mechanism is designed to achieve static safety.

To motivate the design we first review what it takes to be type safe in terms of program *execution* and then move on to discuss in an abstract way how this can be checked statically in terms of program *inspection*.

On the basis of this, a detailed account of the proposed type checking mechanism is given. Type rules are presented only for statements and expressions, while the typing constraints on declarations are described more informally. Apart from a complete set of type rules, an actual proof of type safety would also require the specification of a formal semantics of the language.

5.1 Type safety

We assume the following definitions of type safety:

- A program execution is type safe if it no attempt is made to access an attribute of an object that the object doesn't have.
- A program is type safe if all executions of it are type safe.
- A language is type safe if any program in the language is type safe.

Additionally, type safety is *static*, if it does not depend on checks performed during program execution.

As for the runtime behaviour of a language implementation it seems fair to assume that

- The class of an object does not change.
- An object actually has the attributes declared in its class.

Thus it suffices to verify that objects are used according to their class. This in turn follows if each reference (instance variable or method parameter) has an associated *qualification* so that

- The qualification of a reference is a superclass of the class of the object it refers to.
- Only attributes declared in the qualification of a given reference are accessed on the object it refers to.

This rendering is operational in the sense that it tells us what to check for:

- When assigning an object to a reference (by variable assignment or method parameter passing), check for the appropriate qualification.
- When accessing an attribute of a reference, check the class for an appropriate declaration.

We are now ready to move to static typing. Inspecting a given statement in the program, we can establish its static safeness (the fact that it preserves static type safety) if we can conservatively assert that the above will hold *whenever* it is executed.

A natural way to do this is to let expressions and references have *static types* which summarize the possible classes they may have at runtime. This will allow us to statically check type safety in a way that nicely parallels the above:

- Assignment is checked with the use of a *subtype* relation between *static* types, which asserts that the needed subclass relation will always hold at runtime.
- Attribute access is checked using the most specific statically known common superclass of all classes denoted by the type.

If a static type is *covariant*, it generally doesn't have any guaranteed subtypes at compile time (the only exceptions being quite specific special cases, as exemplified in Section 5.3 and Figure 6). This is the situation where BETA would introduce runtime checks.

The pivotal point of the following is the fleshing out of the static type concept and subtype relation for the pseudo-language.

5.2 Basic assumptions

In the following, different names denote different kinds of entities, according to this table:

name	*is used for*
C_0, C, C'	classes
A	attributes
V	virtual types
T, T_i	pretypes
M_0, M	methods
S	statements
a,	instance variables
p	parameters
e, e_i	expressions
τ, τ', τ_i	types
i, n	numbers

Attributes is a common designation for virtual types, instance variables and methods. Types and pretypes will be further explained below.

5.3 Pretypes and types

With all that in place, we can now attack the central problem: What should a static type be, in order to facilitate full static type checking, yet be as permissive as possible, so as not to inhibit expressiveness.

A good starting point is to see what types we might assign to a new expression, since this is where objects begin their existence. To go with "new C" it seems obvious to introduce a plain class type, denoted by $\langle C \rangle$.

What should be the type of "new $e.V$"? Since the binding of V depends recursively on the type of e (say τ), the only way not to throw away potentially useful type information is to keep τ along with V in the type of the expression. Such a type is written $\langle \tau, V \rangle$.

This scheme matches well with the fact that the so-called *pretype* of e.g. variable declarations (see Figure 5) can be either a class or a virtual type. When a variable with class pretype (say $a : C$) is used within a method body (as in "this.a"), its type should of course be $\langle C \rangle$.

But what should be the type of a variable with virtual pretype (say $a : V$) within a method body? The type is not completely determined by the pretype, but depends on the type of the object that a resides in – its *location*. This incompleteness is why the type part of a reference declaration is called a *pre*-type.

The type scheme $\{\langle C \rangle, \langle \tau, V \rangle\}$ generally works well, but there is one nuisance still left to cope with. In general, we have no option but to forbid assignment to references of open virtual type. Reverting to the animal example for a moment, however, take a look at the class declaration in Figure 6: Given an animal, the keeper feeds it its own kind of food! There would be nothing wrong with accepting this, since the parameter type of eat – although definately virtual – is obviously the same as the one produced by new a.FoodType.

```
Keeper = {
  Feed(a: Animal) {
    a.eat(new a.FoodType);
  }
}
```

Figure 6: A keeper at a zoo

The reason is that we are refering to the same virtual type in the same object – the latter "same" guaranteed by the immutability of method parameters. To allow for this special case, we introduce an extra kind of type $\langle c \rangle$ for constant references c; in our case parameters or this. Their type is simply themselves.

5.4 Environments

The *program environment* $\mathcal{P}$ contains the class declarations of the program, and the subclass relation between them. It defines the following relations:

relation	*means*
$C \in \mathcal{P}$	C is declared in $\mathcal{P}$
$\mathcal{P} \vdash C \leftarrow C'$	C is a subclass of C'

A *class environment* contains the attributes of the class. It defines the following relations:

relation	*means*
$\mathcal{P}, C_0 \vdash V \mathtt{<=} C$	V is open, bound to C
$\mathcal{P}, C_0 \vdash V \mathtt{=} T$	V is final bound to T
$\mathcal{P}, C_0 \vdash a : T$	a is declared to T
$\mathcal{P}, C_0 \vdash M : T_1 \times ... \times T_n \rightarrow T$	signature of M

A *method environment* contains the parameters and body of the method. It defines the following relations:

relation	*means*
$\mathcal{P}, C_0, M_0 \vdash e : \tau$	e has type τ
$\mathcal{P}, C_0, M_0 \vdash \tau <: \tau'$	τ is a subtype of τ'
$\mathcal{P}, C_0, M_0 \vdash S$	S typechecks

5.5 Checking declarations

Type rules for the declaration of classes and attributes have been left out. Thus a description lacks of how the various environments are constructed and their well-formedness maintained (e.g. according to Section 4.1). While this is certainly nontrivial, it is generally not profoundly influenced by the presence of virtual types. One exception to this is final binding of one virtual type to another. This must be non-circular in order for the *Type* function of Figure 7 to terminate.

5.6 Type inference rules

Finally we can give precise meaning to the type checking requirements treated throughout this section.

As explained above (Section 5.3), the type of a reference in the context of a given method body must be computed from its pretype and the type of its location. This computation is handled by the *Type* function, which is defined by means of inference rules in Figure 7. Notice that the computation for a finally bound virtual pretype is simply propagated to its binding - regardless of whether this binding is itself virtual.

$$\mathit{Type}_{class} \quad \frac{\vdash C \in \mathcal{P}}{\mathcal{P}, C_0, M_0 \vdash \mathit{Type}(C, \tau) = \langle C \rangle}$$

$$\mathit{Type}_{virtual} \quad \frac{\mathcal{P}, C_0, M_0 \vdash C' = \mathit{Class}(\tau) \qquad \mathcal{P}, C' \vdash V \mathtt{<=} C}{\mathcal{P}, C_0, M_0 \vdash \mathit{Type}(V, \tau) = \langle \tau, V \rangle}$$

$$\mathit{Type}_{final} \quad \frac{\mathcal{P}, C_0, M_0 \vdash C' = \mathit{Class}(\tau) \qquad \mathcal{P}, C' \vdash V \mathtt{=} T}{\mathcal{P}, C_0, M_0 \vdash \mathit{Type}(V, \tau) = \mathit{Type}(T, \tau)}$$

Figure 7: The *Type* function

As concluded in Section 5.1, in order to type check attribute access of an expression e, we need to find the most specific superclass of the type of e. This is done by the *Class* function, which is defined in Figure 8.

$$\mathit{Class}_{class} \quad \frac{\vdash C \in \mathcal{P}}{\mathcal{P}, C_0, M_0 \vdash \mathit{Class}\langle C \rangle = C}$$

$$\mathit{Class}_{virtual} \quad \frac{\mathcal{P}, C_0, M_0 \vdash C' = \mathit{Class}(\tau) \qquad \mathcal{P}, C' \vdash V \mathtt{<=} C}{\mathcal{P}, C_0, M_0 \vdash \mathit{Class}\langle \tau, V \rangle = C}$$

$$\mathit{Class}_{final} \quad \frac{\mathcal{P}, C_0, M_0 \vdash C' = \mathit{Class}(\tau) \qquad \mathcal{P}, C' \vdash V \mathtt{=} T \qquad \mathcal{P}, C_0, M_0 \vdash C = \mathit{Class}(\mathit{Type}(T, \tau))}{\mathcal{P}, C_0, M_0 \vdash \mathit{Class}\langle \tau, V \rangle = C}$$

$$\mathit{Class}_{this} \quad \mathcal{P}, C_0, M_0 \vdash \mathit{Class}\langle \mathsf{this} \rangle = C_0$$

$$\mathit{Class}_{par} \quad \frac{\mathcal{P}, C_0 \vdash (p : T) \in M_0 \qquad \mathcal{P}, C_0, M_0 \vdash C = \mathit{Class}(\mathit{Type}(T, \langle C_0 \rangle))}{\mathcal{P}, C_0, M_0 \vdash \mathit{Class}\langle p \rangle = C}$$

Figure 8: The *Class* function

The subtyping relation <: is defined in Figure 9. It generally allows only assignments to references with class type, and even that of course only if the type of the assigned value fits. Notice however, the special treatment of virtual types within constant references, which was motivated by the Keeper example above (Figure 6).

$$Subtype_{class} \quad \frac{\begin{array}{l}\mathcal{P}, C_0, M_0 \vdash C' = Class(\tau) \\ \mathcal{P} \vdash C' \leftarrow C\end{array}}{\mathcal{P}, C_0, M_0 \vdash \tau <: \langle C \rangle}$$

$$Subtype_{this.V} \quad \mathcal{P}, C_0, M_0 \vdash \langle \mathsf{this}, V \rangle <: \langle \mathsf{this}, V \rangle$$

$$Subtype_{par.V} \quad \frac{\mathcal{P}, C_0 \vdash p \in M_0}{\mathcal{P}, C_0, M_0 \vdash \langle p, V \rangle <: \langle p, V \rangle}$$

Figure 9: The subtype relation <:

The type rules proper (Figures 10 and 11) are pretty much standard, since most of the virtual type specific "magic" appears in the above helper functions and the subtype relation.

$$Exp_{this} \quad \mathcal{P}, C_0, M_0 \vdash \mathsf{this} : \langle \mathsf{this} \rangle$$

$$Exp_{par} \quad \frac{\mathcal{P}, C_0 \vdash p \in M_0}{\mathcal{P}, C_0, M_0 \vdash p : \langle p \rangle}$$

$$Exp_{var} \quad \frac{\begin{array}{l}\mathcal{P}, C_0, M_0 \vdash e : \tau' \\ \mathcal{P}, C_0, M_0 \vdash C = Class(\tau') \\ \mathcal{P}, C \vdash a : T \\ \mathcal{P}, C_0, M_0 \vdash \tau = Type(T, \tau')\end{array}}{\mathcal{P}, C_0, M_0 \vdash e.a : \tau}$$

$$Exp_{app} \quad \frac{\begin{array}{l}\mathcal{P}, C_0, M_0 \vdash e : \tau' \\ \mathcal{P}, C_0, M_0 \vdash C = Class(\tau') \\ \mathcal{P}, C \vdash M : T_1 \times ... \times T_n \rightarrow T \\ \mathcal{P}, C_0, M_0 \vdash \tau = Type(T, \tau') \\ \mathcal{P}, C_0, M_0 \vdash e_i : \tau_i \quad 1 \leq i \leq n \\ \mathcal{P}, C_0, M_0 \vdash \tau_i <: Type(T_i, \tau') \quad 1 \leq i \leq n\end{array}}{\mathcal{P}, C_0, M_0 \vdash e.M(e_1, ..., e_n) : \tau}$$

$$Exp_{new_C} \quad \frac{\vdash C \in \mathcal{P}}{\mathcal{P}, C_0, M_0 \vdash \mathsf{new}\ C : \langle C \rangle}$$

$$Exp_{new_V} \quad \frac{\begin{array}{l}\mathcal{P}, C_0, M_0 \vdash e : \tau' \\ \mathcal{P}, C_0, M_0 \vdash \tau = Type(V, \tau')\end{array}}{\mathcal{P}, C_0, M_0 \vdash \mathsf{new}\ e.V : \tau}$$

Figure 10: Expression types

6 Expressiveness

To demonstrate that full static typing of virtual types does not sacrifice too much expressiveness, we mainly focus on the covariance problem as exemplified by the now classical ColourPoint example.

6.1 Points and ColourPoints

A standard example in the literature on covariance (see e.g. [11, 3, 5]) is the so called ColourPoint problem, which is shown in Figure 12. It is usually stated using signature covariance or self recursive types (see Section 3), but is here refrased with virtual types.

The idea is that both classes are concrete[1]. The reason for ColourPoint to inherit Point supposedly is twofold: To reuse its implementation, and to make references to Point able to polymorphically also refer to ColourPoints.

[1] An *abstract* class is a class that is not supposed to be instantiated, but only functions as a superclass. By contrast, a *concrete* class is one that may have instances.

$$Stm_{exp} \quad \frac{\mathcal{P}, C_0, M_0 \vdash e : \tau}{\mathcal{P}, C_0, M_0 \vdash e;}$$

$$Stm_{assign} \quad \frac{\begin{array}{l}\mathcal{P}, C_0, M_0 \vdash e' : \tau' \\ \mathcal{P}, C_0, M_0 \vdash C = Class(\tau') \\ \mathcal{P}, C \vdash a : T \\ \mathcal{P}, C_0, M_0 \vdash e : \tau \\ \mathcal{P}, C_0, M_0 \vdash \tau <: Type(T, \tau')\end{array}}{\mathcal{P}, C_0, M_0 \vdash e'.a := e;}$$

$$Stm_{return} \quad \frac{\begin{array}{l}\mathcal{P}, C_0, M_0 \vdash e : \tau \\ \mathcal{P}, C_0 \vdash M_0 : T_1 \times ... \times T_n \rightarrow T \\ \mathcal{P}, C_0, M_0 \vdash \tau <: Type(T, \langle C_0 \rangle)\end{array}}{\mathcal{P}, C_0, M_0 \vdash \mathsf{return}\ e;}$$

Figure 11: Statements

```
Point = {
  x,y: Integer;
  CompareType <= Point;
  equal(p: CompareType): Bool {
    return (this.x=p.x) and (this.y=p.y);
  }
}

ColourPoint = Point {
  c: Color;
  CompareType = ColourPoint;
  equal(p: CompareType): Bool {
    return super.equal(p) and (this.c = p.c);
  }
}
```

Figure 12: The usual colour point problem

Covariance comes into play with the method equal, which expects its argument to be of the virtual type CompareType. In ColourPoint, CompareType has to be further (or in this case final) bound to ColourPoint so that the equal method can take its specific attributes into consideration.

Of course, with this structure, calling the equal method on a reference to a Point is generally not statically type safe precisely because the reference might contain instances of subclasses (indeed of ColourPoint), which further bind CompareType uncontrollably. Alas, we cannot safely compare two Points for equality!

This situation is usually regarded as a shortage of the covariance mechanism. we prefer to view it as a modelling mistake: If a Point is something that can be compared with another Point for equality, then a ColourPoint is not a subtype of Point – since it cannot. Hence, it should not be a subclass either.

6.2 A solution

Instead, the close relationship between the two should be captured by a common abstract superclass as in Figure 13.

All desirable properties of the original approach are retained: Point and ColourPoint still share most of their implementation, References capable of polymorphically referring both can still be declared, now using AbstractPoint.

```
AbstractPoint = {
   x,y: Integer;
   CompareType <= Point;
   equal(p: CompareType): Bool {
      return (this.x=p.x) and (this.y=p.y);
   }
}

Point = AbstractPoint {
   CompareType = Point;
}

ColourPoint = AbstractPoint {
   c: Color;
   CompareType = ColourPoint;
   equal(p: CompareType): Bool {
      return super.equal(p) and (this.c = p.c);
   }
}
```

Figure 13: A slight change fixes the problem

On top of this comes the fact that Point may now be used in a safe way, and that references to Points that can *not* contain ColourPoints are now possible.

```
AbstractPoint = {
   x,y: Integer;
   CompareType <= Point;
   equal(p: CompareType): Bool {
      return (this.x=p.x) and (this.y=p.y);
   }
}

Point = AbstractPoint {
   CompareType = AbstractPoint;
}

ColourPoint = AbstractPoint {
   c: Color;
   CompareType = AbstractPoint;
   }
```

Figure 14: A different choice of semantics

Furthermore the programmer now has the full choice of whether or not to allow Points and ColourPoints to be compared with each other. If this should be possible, these classes should just final bind CompareType to AbstractPoint instead of to themselves, as shown in Figure 14. The two classes could even have different semantics, so that a Point will accept a ColourPoint, but not vice versa – the choice belongs entirely to the programmer.

This restructuring is always possible: If you need a concrete version of a class that has open virtual types, make a concrete subclass of it, where you final bind all the virtuals. If the creation of such trivial subclasses proves too tedious, one might consider simply adding a language construct for doing it automatically.

In our opinion, this cuts the Gordian knot.

7 Related work

The covariance problem as exemplified through the ColourPoint example, has been treated in numerous ways over the past decade or so. This section relates some of the proposed solutions to the one presented here.

7.1 Separating subclassing and subtyping

Due to the covariance problem, it was proposed in [4] to separate the subtype and subclass relations. This has the appeal that a variable declared by a given type does not need to be able to hold all subclasses of an associated class, but only those that are known to be substitutable for the declared type.

In Emerald [10] and many theoretical languages (such as $\mathcal{LOOM}$ [2]) this lead to a complete separation of classes (that describe implementation) and types (that describe interface), which in turn leads to quite a specification overhead when types and classes coincide.

To avoid this, Sather [16] reunifies types and classes, and only separate the two *relations*, subtyping and subclassing.

Clearly, stepping down on the demand that subclassing yields subtyping, gives new possiblities for static type checking. It is not without problems however – keeping track of two different but closely related relations can be confusing in practice.

A more profound objection seen from our point of view aims at the way this solution views inheritance: Primarily as a code reuse mechanism. Instead of modelling a conceptually clear specialization relation, subclassing is an *ad hoc* relationship between classes. This is why the need is felt for a seperate, "cleaner" relation.

From the perspective of this article, subtyping and the conceptual specialization relation coincide, and inheritance should only be allowed to be used in this way. As seen in Section 6, this does not prohibit the reuse of code, it just requires it to occur in a slightly more structured way: as inheritance of a common ancestor rather that as conceptually unsound inheritance from "cows to horses".

7.2 Monomorphic types

Covariance is only a problem because references to classes with covariant parameter types may contain instances of subtypes. Some solutions therefore introduce various constructs to avoid the unfortunate consequences of subtype substitutability in certain cases – in other words to have monomorphically as well as polymorphically typed variables.

Building on a proposal by Dodani and Tsai [5], Sather [16] divides classes into abstract and concrete, of which only the latter can be instantiated. While concrete classes can have subclasses, they do not have any subtypes. For this reason a variable qualifed by a concrete type can only refer objects of *exactly* that class.

This is reminiscent of Java's final classes [9], which prohibit further specialization, and thus make references qualified by them monomorphic.

A different way to do this is to let the references themselves be declared type exact. This is the case with part objects in BETA [12] and with exact types in $\mathcal{LOOM}$, where subtyping has been completely dropped in favour of matching; see below.

For describing instances of concrete classes, which in a natural way often are (or should be) leaf classes, monomorphism seems like a sensible idea, although it appears to be somewhat vulnarable to later redesigns of the kind where one concrete class is turned into an abstract one to introduce different variants as subclasses.

The reason why it is not considered in our langauage is that it is not necessary. Since all virtual types can be final bound, a concrete subclass without covariant parameters can aways be declared. As opposed to a final class, such a "finalized" class (like Cow in Section 2.1) can have subclasses (like DairyCow, Hereford etc.), and references qualifed by it can refer to instances of these subclasses.

7.3 Matching

Another approach is to say: well, if inheritance does not give us subtyping, what *does* it give us then? The answer is *matching* [1, 2]. In $\mathcal{LOOM}$ references can be qualifed with so called "hash types", say $\#C$, which cover any classes that *match* C, that is, all subclasses of C. A reference to a hash type is restricted so that only methods without covariant parameters can be called on it. Therefore the covariance problem does not occur.

In $\mathcal{LOOM}$, If you want to refer only to Points you use the type Point (which is monomorphic), while if you want to refer also to ColourPoints you use the type #Point, but are then not allowed to call a covariant equal method on it.

Notice how closely this corresponds to Point and AbstractPoint respectively in our solution. The only difference seems to be that we have to declare an extra class, but on the other hand get polymorphism on references to Point, plus the ability to call equal on AbstractPoint in special situations, as described in Section 5.3.

8 Genericity

This paper has mainly focused on the covariance issues of virtual types.

Virtual types as an alternative *genericity mechanism* to parameterized types (as in Eiffel [15], C++ [7] and many others) is usually criticized because of it's alleged dependence on dynamic typechecking. With this proposal that argument loses validity – as for static type safety, virtual types must now be seen as an equally valid choice.

Yet, rather than looking at virtual and parameterized types as contenders, it is compelling to ponder the possibilities of combining the strengths of the two approaches. While it is certainly outside the scope of this paper to give a proper proposal for such a fusion, it should be noted in passing, that the statical safety of virtual types brings it a lot closer.

9 Riding the Java Train

Java currently supports neither genericity nor covariance. Since these are issues that many people have opinions about, a lot of different proposals are currently being made for adding such facilities one way or the other.

And yes, a proposal for adding virtual types to Java has also come forth [18]. While blending very elegantly into the syntactic framework of Java, and – as it seems – neatly fitting the virtual types in with particular Java constructs such as interfaces and abstract classes, this otherwise convincing proposal has one serious drawback: It requires implicit dynamic type checking.

In BETA all so called reverse assignments[2] are statically allowed, but subjected to a runtime type check. The runtime checking introduced by virtual types fits nicely into this scheme, and can be seen as a special case.

Java, however, is a fully statically type checked language. With the one unfortunate exception of arrays, which are covariant and require dynamic checks, runtime checking of types can only come about as a result of *explicit* type casts in the code. The introduction of *implicit* runtime checks would profoundly violate this very deliberate principle of Java.

This is where the cavalry comes in. If the type checking scheme proposed here expands well to Java, this last problem is overcome. To gain further insight into this, statically safe virtual types are currently being added to an experimental compiler for the JOOS subset of Java (generously provided by Michael Schwartzbach and Laurie Hendren). This implementation project will serve as a proof of concept both of the feasibility of virtual types in Java and of the practicality of the type checking scheme of this paper.

10 Conclusion

The main conclusion to draw from this paper is that covariance, static type safety and subtype substitutability are not incompatible after all. But to make them coexist without losing out fatally on expressiveness, a mechanism must be provided to control covariance, and programmers need to be aware of its impact on type checking.

The type checking framework in Section 5 is to the best of our knowledge the first in-depth account of a type checking mechanism for a language with virtual types. Since the permission of dynamic checks is merely a matter of modifying the subtype relation $<:$, the approach might inspire a more formal description of e.g. the BETA type system, although admittedly the presence of block structure probably complicates matters somewhat.

While resting heavily on the presence of final bindings for safety, the proposed solution to the ColourPoint problem (Figure 13) in our oppinion also points to a more philosophical consideration. The inheritance relation prescribed by the original version reflects a simple minded desire for code reuse. Whether or not it is meaningful to say that ColourPoints actually *are* Points, and to what extend sensible semantics for comparing the two different kinds of objects may exist, is not at all considered.

Taking a *conceptual* view, we think that – even without the type safety – the new version is how we would have modelled the situation. This is just one example of what in our experience – but this *is* just a gut feeling – seems to be a common pattern: That conceptual modelling and type-theoretic soundness tend to go hand in hand.

References

[1] ABADI, M., AND CARDELLI, L. On subtyping and matching. In *ECOOP'95 – Object-Oriented Programming* (1995), pp. 145–167.

[2] BRUCE, K. B. Subtyping is not a good "match" for object-oriented languages. In *ECOOP'97 – Object-Oriented Programming* (1997), M. Aksit and S. Matsuoka, Eds., no. 1241 in Lecture Notes on Computer Science, Springer, pp. 104–127.

[3] BRUCE, K. B., CARDELLI, L., CASTAGNA, G., THE HOPKINS OBJECT GROUP, LEAVENS, G. T., AND PIERCE, B. On binary methods. *Theory and Practice of Object Systems*, 1 (1995), 221–242.

[2] Assignments from general to more special types

[4] Cook, W. R., Hill, W. L., and Canning, P. S. Inheritance is not subtyping. In *ACM Conference on Principles of Programming Languages (POPL'90)* (1990), ACM Press, pp. 125–135.

[5] Dodani, M., and Tsai, C.-S. Acts: A type system for object-oriented programming based on abstract and concrete classes. In *ECOOP'92 – Object-Oriented Programming* (1992), O. L. Madsen, Ed., no. 615 in Lecture Notes in Computer Science, Springer, pp. 309–328.

[6] Drossopoulou, S., and Eisenbach, S. Java is type safe - probably. In *ECOOP'97 – Object-Oriented Programming* (1997), M. Aksit and S. Matsuoka, Eds., no. 1241 in Lecture Notes on Computer Science, Springer, pp. 389–418.

[7] Ellis, M. A., and Stroustrup, B. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[8] Goldberg, A., and Robson, D. *Smalltalk-80 – The Language*. Addison-Wesley, 1989.

[9] Gosling, J., Joy, B., and Steele, G. *The Java Language Specification*. Addison-Wesley, 1996.

[10] Hutchinson, N. *Emerald: An Object-Oriented Language for Distributed Programming*. PhD thesis, Department of Computer Science, University of Washington, Seattle, WA, January 1987.

[11] Madsen, O. L. Open issues in object-oriented programming–a scandinavian perspective. *Software–Practice and Experience 25*, S4 (December 1995).

[12] Madsen, O. L., Magnusson, B., and Møller-Pedersen, B. Strong typing of object-oriented languages revisited. In *Proceedings of OOPSLA'90* (Ottawa, Canada, 1990), SIGPLAN, ACM Press.

[13] Madsen, O. L., and Møller-Pedersen, B. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings of OOPSLA'89* (1989), SIGPLAN, ACM Press.

[14] Madsen, O. L., Møller-Pedersen, B., and Nygaard, K. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[15] Meyer, B. *Object-Oriented Software Construction*. Prentice Hall International Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ, 1988.

[16] Omohundro, S. The Sather Programming Language. *Dr. Dobb's Journal 18*, 11 (October 1993).

[17] Shang, D. Are cows animals? *Object Currents 1*, 1 (January 1996). http://www.sigs.com/objectcurrents/.

[18] Thorup, K. K. Genericity in java with virtual types. In *ECOOP'97 – Object-Oriented Programming* (1997), M. Akit and S. Matsuoka, Eds., no. 1241 in Lecture Notes on Computer Science, Springer, pp. 444–471.