Typing Dynamic Inheritance
A Trade-Off between Substitutability and Extensibility

Carine Lucas, Kim Mens, Patrick Steyaert

Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, BELGIUM

email: clucas@vnet3.vub.ac.be , kimmens@is1.vub.ac.be,
prsteyae@vnet3.vub.ac.be

Abstract

Recent developments in subjectivity, composition technology and novel prototype-based languages demonstrate that dynamic object extension is an essential feature in modern object-orientation. But the total absence of static type systems for dynamic object extension is a major obstacle for its adoption. The key principle of type-safe dynamic object extension is a trade-off between possible assignments and possible extensions. We describe a static type system using specialisation interfaces to refine the notion of subtyping and to limit dynamic extension. We furthermore argue that the introduction of specialisation interfaces in the system opens up a lot of new perspectives in software engineering in general. The type system is proven to be consistent and complete.

1 Introduction

Prototype-based languages [Lieberman86] [Ungar&Smith87] are characterised by the ability to dynamically extend objects. Dynamic object extension is for example used to create different views on an object after it has been created. See e.g. [Dony&al.92] (split objects) or [Harrison& Ossher93] (subjectivity) for examples. In contrast with conventional class-based languages where each class statically knows its parent class, prototype-based languages allow the inheritance hierarchy to be created at run time. This results in a much more flexible approach.

On the other hand there is a need for static type checking. While static type checkers for class-based languages are very well known, for prototype-based languages this is not the case. In general, type systems for object-oriented languages are characterised by substitutability of subtypes by supertypes. As a consequence, it is always possible for a variable to contain an object of a subtype of this variable's formal type. In addition, not all object extensions are type correct. Therefore, type checking dynamic object extension is very complex as static type checkers must determine the type correctness of extending an object, of which the actual type is not statically known.

One possible solution is to prohibit all extensions that do not result in subtypes. This can be determined on the basis of the objects' client interface, but results in a very restrictive approach. But even in the case where objects that are not in a subtype relationship with their parents can be created, some extensions are not type-safe, as they create objects with internal inconsistencies. To determine which cases are type-safe additional information is needed. We will show that this additional information can be given by annotating objects with specialisation interfaces (as introduced in [Lamping93]). The specialisation interface of the object under extension (parent) makes its internal structure visible to the extension (inheritor) in an encapsulated way. This internal structure reveals which methods are defined in terms of what other methods. Annotating objects with specialisation interfaces results in a form of negative type information as it restricts possible inheritors.

We will show that the ability to type check dynamic object extension is based on a trade-off between possible assignments and possible extensions. Specialisation interfaces give the programmer the means to indicate whether he wants to restrict either the set of object types with which an object can be substituted or the set of extensions that can be made of it. This results in a mechanism that can be statically type checked and lies somewhere between uncontrolled dynamic object extension and fixed static inheritance. Moreover, the introduction of specialisation interfaces shows a lot of promise in software engineering in general.

2 Related Work and Terminology

This section briefly introduces some basic terminology by discussing related work.

2.1 Mixin-based Inheritance

Consider inheritance as an incremental modification mechanism [Wegner& Zdonik88], where a parent P is transformed with a modifier M to form a result R=P+M(P). The modifier M is parameterised by a parent P to model the fact that a subclass can invoke operations defined in the superclass. In classical inheritance the modifier M does not exist on its own. In *mixin-based inheritance* a mixin is exactly this modifier that exists as an abstraction apart from parent and result. Mixins were first introduced in Flavors as a means to construct inheritance hierarchies in a more flexible way [Moon86]. In *pure* mixin-based inheritance [Bracha&Cook90] [Steyaert&al.93], mixin application is the *only* way to make extensions. In an object-based language with pure mixin-based inheritance like we are going to use, inheritance can then be modelled as $O_1=O_2+M(O_2)$, where O_1 and O_2 are objects and M is a mixin.

2.2 Typing

A large range of type systems for object-oriented languages has already been proposed [Abadi&Cardelli94] [Bruce&al.93] [Palsberg&Schwartzbach94], but none of them take into account the possibility of dynamic object extension. [Nierstrasz93] describes a type system for active objects. But active objects should not be confused with dynamic object extension. The problems involved in type checking dynamic object extension are mainly due to method type incorrectness, whereas the problems involved in type checking active objects concern evolving interfaces.

Throughout the history of object-oriented programming, the use of inheritance has evolved from simply a means to reuse code to a way of achieving reuse of behaviour [Wegner&Zdonik88]. The latter implies *substitutability*, which means that if B inherits from A, an instance of B can be used whenever an instance of A is expected. Problems concerning the typing of such languages lead to the realisation that a separation of the notions of object and interface is necessary [Canning&al.89]. Interfaces can be seen as the formalisation of the notion of an object protocol. To check whether two objects are holding a subtype-supertype relationship one should consider the containment of their interfaces. In [Canning&al.89] *interface containment* is described as follows: 'An interface `Big` *contains* an interface `Small` if it contains all the methods of `Small` , and for each of their common methods, each parameter interface for `Small` *contains* the corresponding parameter interface for `Big` while the result interface for `Big` *contains* the result for `Small`. The important thing to notice here is that the condition on the parameter interfaces is reversed.' This last remark concerns the *contravariance rule* [Cardelli&Wegner85].

Although the covariance rule might seem more intuitive, it has repeatedly been shown not to be type safe [Cook89][Pierce92]. The contravariance rule guarantees type safety, but reduces expressiveness. As an illustration consider an interface `Point` and an interface `ColourPoint` containing all the messages of `Point`. While these interfaces intuitively appear to be in a containment relationship, they are not because the equal message on `ColourPoint` does not respect the contravariance rule. We will opt for a less strict variant of the contravariant rule.

```
Point         =  Interface: [  … ; equal (Point) Result: Boolean ]
ColourPoint   =  Interface: [  … ; equal (ColourPoint) Result: Boolean ]
```

2.3 Specialisation Interfaces

[Mitchell90] presents a typed calculus of objects and classes, featuring method specialisation when methods are added or redefined. As a bookkeeping mechanism for keeping the types of the methods straight, methods are annotated with a list of all methods that were defined before, and thus could be used in the implementation of the new method. Instead of keeping track of all methods that are known at definition time of a method, one could record only those methods that are actually used in the new method. This would come very close to the definition of specialisation interfaces as introduced by [Lamping93].

Specialisation interfaces are a means to extend type systems in order to describe information about the organisation of a class; that is which methods are called through self sends by what other methods. This information is important to inheritors since it provides knowledge about the effect of overriding a method. Furthermore, inheritors need to know which parent operations they can invoke. All this information is captured in the parent's specialisation interface. Lamping approaches this from a library specification angle and uses the interfaces primarily as documentation. We will use specialisation interfaces for type checking dynamic object extension.

As mentioned in the introduction, annotating objects with specialisation interfaces results in a form of negative type information. Indeed, the specialisation interface describes details of the internal structure of an object and excludes inheritors not respecting this structure. The idea of negative type information is not new. [Cardelli&Mitchell89], amongst others, used positive and negative assumptions to associate types to objects, but not for typing dynamic object extension.

An approach similar to specialisation interfaces is suggested in [Hauck93], where typed interfaces are introduced to check the type-safe exchange of base classes (for the maintenance of class libraries) and to allow determining the concrete base class on a per object basis.

3 Programming Model

We will explain our approach by means of a type system for an object-based language with pure mixin-based inheritance. The toy language derived from *Agora* [Codenie&al.94]. To keep things simple, we do not describe the entire language. The full syntax is given in Appendix A.1. Here we will briefly discuss the core features.

3.1 Our language

Every program is formed by a list of mixin declarations, followed by a list of expressions. Every mixin defines a number of object and method declarations.

```
MixinDeclaration ::=   MixinName   Mixin: [ MixinExpression ]
                                   Super: SpecInterface

MixinExpression   ::=  MethodDeclaration
                     | ObjectDeclaration
                     | MixinExpression ; MixinExpression
```

To avoid cluttering up the syntax, we restrict ourselves to methods with exactly one argument and one result. Both the argument and result are typed. In the examples we will often use methods with no argument or result, this is expressed in the type system by introducing an empty type. The same goes for the clauses **`Super:`** (in mixin declarations) and **`Self:`** (in method declarations). The meaning of, and necessity for, these clauses (which represent the specialisation interface) will become clear in the following sections.

```
MethodDeclaration ::=  MethodName ( Variable:Variable )
                          Method:[ MethodExpressionList ]
                          Result: Variable
                          Self: SpecInterface
```

An example of a mixin declaration (where the self- and super-clauses have been omitted) is given in Listing 1.

```
MakeMammal  Mixin:   [  habitat define: place;
                        giveHabitat () Method: [ … ] Result: place];
                        eats (someFood:food) Method: [ … ] ];
```

Listing 1

To allow static type checking all variables have to be *declared* through an **`define:`**-statement. The language has no classes and new objects can only be created by copying or extending existing objects. Objects can only be extended by applying mixins to them. Either another object or a mixin application can occur on the right-hand side of an **`define:`**-statement. The meaning of the (optional) **`withSelf:`**-clause will be explained in the following sections.

```
ObjectDeclaration ::=  Variable define: ProtoType withSelf: SpecInterface

ProtoType         ::=  Variable | MixinApp

MixinApp          ::=  Variable MixinName | rootObject MixinName
```

As objects are extended dynamically, it is not possible to declare all prototypes that will be used in a program beforehand and all objects are candidate for extension. It is important to note that as we work in a prototype-based system, it is possible for objects themselves to serve as types. We could include an explicit notion of types in our language, but this would only make the notation more complex without changing anything to the core of the system. For an example of an object serving as type, see how in Listing 1 the object `food` was used as type of the `eats` method in `MakeMammal`. The **`define:`**-statement thus acts both as a variable declaration and a type declaration. Listing 2 shows some examples.

```
yogi define: aBear;

food define: rootObject MakeFood;
plant define: food MakePlant;
```

Listing 2

The body of the program is a list of expressions.

```
Expression ::=    ObjectDeclaration                        (object declaration)
                | Variable MethodName ( Variable )         (method send)
                | Variable := Object                       (assignment)
```

Note that it is impossible to put **self** or **super** at the right-hand side of an assignment or to perform self or super sends at the top level of a program. Note also that we assume that we only perform super calls of methods with the same name as the method performing the call, not just of any method. The system could be adjusted to change this, but we feel it is a good programming habit only to perform such super calls. Listing 3 shows some example expressions.

```
aPlace := aMammal giveHabitat();
aMammal eats (anApple);
```

Listing 3

3.2 An Example

Throughout this text the example of an information system in a zoo will be used (example due to [Lippman93]). Listing 4 shows the corresponding code fragment. The system is installed near each cage so that visitors can ask questions about an animal's behaviour. We consider the subsystem concerning mammals. The mixin `MakeMammal` describes the general behaviour of mammals. E.g. it contains a method `eats` that displays information about the animals eating habits. A whole range of mixins describe the particular behaviour of different kinds of mammals. For example in `MakeBear`, which describes the family of bears, the method `eats` is overridden to specify a bear's eating habits. After the mixins are declared a number of objects are created through mixin application. For example, `aBear` is created by consecutively applying `MakeMammal` and `MakeBear` to `rootObject`. Then `aPanda` is defined as an endangered bear with the behaviour of a herbivore, as pandas only eat bamboo.

```
MakeMammal  Mixin: [ habitat define: place;
                     giveHabitat () Method: [ … ] Result: place;
                     eats (someFood:food) Method: [ … ];
                     displayBehaviour Method: [ … ] ];

MakeBear    Mixin: [ eats (someFood:food) Method: [ … ];
                     displayBehaviour Method:
                        [  self eats(fish);
                           super (); … ] ] ;

MakeHerbivore  Mixin: [ eats (aPlant:plant) Method: [ … ]];
MakeCarnivore  Mixin: [ … ] ;
MakeEndangered Mixin: [ … ] ;

food define: rootobject MakeFood;
plant define: food MakePlant;
```

```
aMammal define: rootObject MakeMammal;
aBear define: aMammal MakeBear;
anEndangeredBear define: aBear MakeEndangered;
aPanda define: anEndangeredBear MakeHerbivore;

aMammal := aBear;
aMammal := aHerbivoreMammal;
aPanda displayBehaviour.
```

Listing 4

4 Typing Dynamic Inheritance

4.1 Reuse of Code versus Reuse of Behaviour

The choice between covariance and contravariance is one of the main discriminators between type systems for object-oriented languages. The issues concerned with this choice are clearly discussed in [Dodani&Tsai92]. They also observe that inheritance is used to express two kinds of relationships: the substitutable is-a relationship and abstraction of common behaviour. The choice between covariance and contravariance should therefore be directly related to the specific use of inheritance. Every model that explicitly chooses strictly for either covariance or contravariance therefore has to make exceptions to model the other relationship.

The core of our type system is based on the contravariance rule. With contravariance two kinds of type systems are possible. Either one applies strict contravariance with the corresponding loss of expressiveness or one also allows inheritors to be created that are not subtypes. We opted for the latter, thus offering both the possibility to model substitutable is-a relationships (by respecting contravariance) and abstraction of common behaviour.

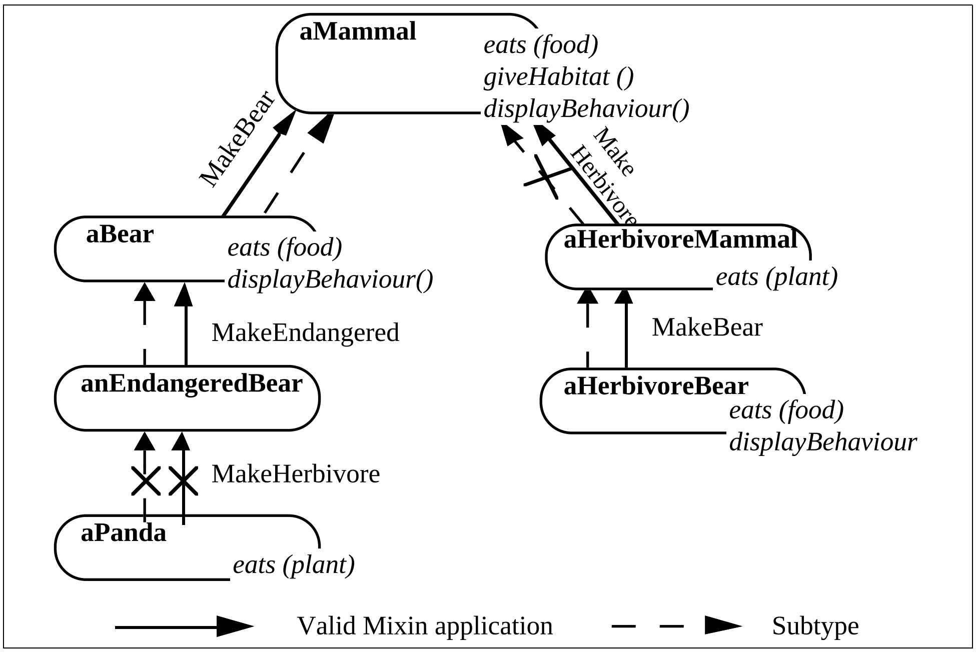

Figure 1

Figure 1 illustrates this on our example. While `aBear` is a subtype of `aMammal`, `aHerbivoreMammal` is not, because to create `aHerbivoreMammal` the mixin `MakeHerbivore` has overridden the `eats`-method with a covariant parameter type. Strictly applying contravariance would mean forbidding the creation of aHerbivoreMammal. Although this would guarantee that all inheritors are substitutable for their parent, it would imply a significant loss of expressiveness. Allowing the creation of objects like aHerbivoreMammal to obtain abstraction of behaviour is often very useful (see [Dodani&Tsai92] for examples).

4.2 Type Correct Object Extensions

But even in the case where it is allowed to create inheritors that are not in a subtype relation with their parent not all mixin applications are allowed. Consider sending the message `displayBehaviour` to `aPanda`. This method is defined in `MakeBear` and invokes `self eats(fish)` and is type correct at that level. However the version of `eats` that will actually be invoked is defined in `MakeHerbivore` and expects an argument of type `plant`, which will cause an error. Applying `MakeHerbivore` to `aBear` should not be allowed since it incorrectly overrides a method invoked through a self send in `aBear`. Of course, it should still be allowed to apply `MakeHerbivore` and `MakeBear` separately. It should even be allowed to apply `MakeHerbivore` first and then `MakeBear` (which is a possible solution to make `Panda`'s). But once `MakeBear` has been applied it should no longer be allowed to apply `MakeHerbivore`.

This example indicates that in order for the type system to restrict possible inheritors it needs to know what self sends are executed by the parent object. This is achieved by appending an extra self-clause to each method[1]. The methods in these self-clauses need to respect contravariance when they are overridden. These clauses thus impose type constraints on possible inheritors. Hence a mixin is not only typed by its client interface, but also by its internal structure. Note that the **`self:`**-clause was already included in the syntax given above.

Remark that appending the self-clause could have been achieved in two ways. By adding one single clause to each mixin declaration, or by adding separate clauses to every method declaration. We opted for the latter, as this gives us more information (i.e. *in what method* the self sends are performed) and thus makes it possible for the type system to be less restrictive in the set of mixin applications it prohibits. Consider e.g. `MakeEndangered` defining its own version of `displayBehaviour`, calling a self send of `eats(somePlant)`. As a result `MakeHerbivore` could be applied anyway, because sending `displayBehaviour` to `aPanda` would no longer result in an error. This is only so because the self send of `eats` was performed twice in the same method (`displayBehaviour`). Therefore, to be capable of type checking on such a fine-grained level, we need to know exactly in what method each self send is performed. The only possible reason left to forbid the application of `MakeHerbivore` would be that the definition of `displayBehaviour` in `MakeEndangered` also performs a super call, with as a result that the `MakeBear`-version of `displayBehaviour` would still be called after sending `displayBehaviour` to `aPanda`. Our type system takes all these possibilities into account.

Besides putting constraints on what methods cannot be overridden covariantly by the mixin (through the self-clauses), constraints are also necessary on what methods should certainly be

[1] Note that this information could be generated, but for clarity reasons we add it explicitly.

implemented *by the parent*. It is allowed for mixins to do super sends, even though their possible parents are unknown. Therefore, when applying a mixin to an object it should be verified whether all messages that are called as super sends in the mixin are actually implemented in the parent object. `MakeBear` invokes the `displayBehaviour`-method defined in `MakeMammal` in its definition of `displayBehaviour`. The mixin should then be extended with a super-clause to make this clear. Note that we add the super-clause at the mixin level and not at the method level. As super calls are only performed in methods with the same name, introducing super-clauses on the method level wouldn't provide us with any extra information (only perhaps a shorter notation). The super-clause was already stated in the syntax. Listing 5 redeclares a mixin from our example extended with its specialisation interface. The type checker verifies whether all methods called through self and super sends are actually stated at the intended clauses.

```
MakeBear   Mixin: [  eats (someFood:food) Method: [...];
                     displayBehaviour ()
                        Method: [  self eats(fish);
                                   super (); … ]
                        Self: [ eats (food) ] ]
           Super: [  displayBehaviour () ].
```

Listing 5

By adding self- and super-clauses we create a structure similar to Lamping's specialisation interfaces. The extra knowledge provided by the self-clauses enables us to impose a less strict form of contravariance.

4.3 Typing Dynamic Object Extension

Now let us take a look at the problems specifically connected to *dynamic* object extension. Substitutability of supertypes by subtypes is a general characteristic of object-oriented type systems. As a consequence, an object's type is known only up to an approximation, since the actual type of an object can be a subtype of its formal type. The key problem of typing dynamic object extension is that a static type checker does not know the exact type of the object that is being extended. This is illustrated in listing 6, which features an example of dynamic inheritance in the method `displayAsHerbivore`. As far as a static type checker can determine, an object with type `aMammal` is extended with herbivore behaviour in the method `displayAsHerbivore`. The actual type of this mammal object, however, can also be `aBear`, due to the assignment of the `aBear` actual argument to the `aMammal` formal parameter. Remember that the extension of `aBear` with herbivore behaviour is not type correct. The program in listing 6 should therefore be rejected.

```
MakeHerbivoreDisplayable
   Mixin:
      [  displayAsHerbivore(someMammal:aMammal)
            Method: [ (someMammal MakeHerbivore) displayBehaviour; … ]
      ];

aMammal define: rootObject MakeMammal;
aBear define: aMammal MakeBear;

herbie define: rootObject MakeHerbivoreDisplayable;
herbie displayAsHerbivore(aMammal);
herbie displayAsHerbivore(aBear)
```

Listing 6

4.4 Substitutability versus Dynamic Extensibility

The conflict between the above assignment of the `aBear` actual argument to the `aMammal` formal parameter and the extension with herbivore behaviour is a conflict between substitutability of client interfaces and substitutability of specialisation interfaces. Whereas, with respect to the client interface `aBear` compatible with `aMammal`, with respect to the specialisation interface `aBear` is *not* compatible with `aMammal` (i.e. `aMammal` allows more extensions than `aBear`). It is therefore clear that the above typing problem can be solved by enriching the type system with type rules on substitutability that also take specialisation interfaces into account.

Applying this to the example we see that `aMammal` does not specify a type restriction on the specialisation interface of inheritors, while `aBear` does have a type restriction on inheritors, concerning the self send of the `eats`-method. On this basis an assignment `aMammal := aBear` can be prohibited by the type checker since an object with more type restrictions on the specialisation interface is not substitutable for an object with less type restrictions. When the assignment is prohibited due to the absence of type restrictions on the specialisation interface, the extension of `aMammal` with herbivore behaviour is allowed.

If one *does* want to allow an assignment `aMammal := aBear`, then one needs to add a type restriction that makes the specialisation interface of `aMammal` compatible with the specialisation interface of `aBear`; i.e. one has to add a type restriction on the possible extensions of `aMammal`. Obviously this extra type information should be such that `MakeHerbivore` is not applicable to the annotated `aMammal`. We therefore provide the possibility (already mentioned in the syntax) to extend the specialisation interface of an object with additional restrictions, i.e. an extra self clause (listing 7). It is obviously superfluous to relate the methods in this extra self clause to a certain method, as that would not have the same meaning as in the other cases[2].

```
aMammal  define: rootObject MakeMammal withSelf: [ eats (food) ] ;
```

Listing 7

The choice whether or not to extend an object type with additional constraints depends on the programmer’s intentions for further use of the object. Neither option is as radical as might seem at first. Even when the programmer does want to allow the `MakeHerbivore` extension, it should, for example, still be possible to assign an endangered mammal to `aMammal` as this does not conflict with this extension. For the same reason it should still be possible to apply the `MakeEndangered` mixin to `aMammal`, when the assignment of `aBear` to `aMammal` is allowed.

In any way, the trade-off should be made by the programmer and not by the type checker. The programmer is therefore given the ability to indicate whether he wants to restrict the set of object types with which an object can be substituted or whether he wants to restrict the set of mixins that can be applied to this variable. He can do this by (not) extending the specialisation interface of the object.

[2] In our type system we will introduce a method named ‘ghost’ to which this extra information will be attached. We therefore impose the restriction that the user cannot define a method named ‘ghost’.

5 The Type Rules

The first step in formalising the trade-off between substitutability and extensibility is defining the syntax of type expressions and a number of selector functions and restrictions on it. The following three subsections respectively give separate type rules on substitutability, extensibility and object declaration. The final subsection gives proofs of the consistency and completeness of the type system.

Note that we will use a notation somewhat closer to natural language, than the standard notation used for describing type systems. The main argument for doing so is that our primary goal is to explain the innovative principles and ideas on which the type system is based to a public larger than just type system specialists.

5.1 Type Syntax

5.1.1 Type Syntax and Syntactic Domains

In the following abstract grammar *'ghost'*, *bottom* , *top* and all *methodname*s are terminals.

```
methodint   ::=   methodname objecttype1 objecttype2
ghostint    ::=   'ghost' bottom top

clientint   ::=   methodint
                | ghostint
specint     ::=   ε
                | methodint specint

methodtype  ::=   clientint specint

constructedobjecttype   ::=   ε
                            | methodtype constructedobjecttype

objecttype  ::=   constructedobjecttype
                | top
                | bottom

mixintype   ::=   constructedobjecttype specint
```

On the one hand, the type of a method consists of a method name, a type for the argument of the method and a result type (*methodint*). As already mentioned in an earlier footnote, we also introduce a special "ghost"-method type (*ghostint*). This first part is referred to as the method client interface (*clientint*). When type checking an object we also need to know which self calls are being executed in each of its methods. The part of the method type describing this information is called its specialisation interface (*specint*). A complete method type (*methodtype*) then consists of a method client interface and a method specialisation interface. Since an object is a collection of methods, the type of an object is simply an enumeration of the types of its methods (*constructedobjecttype*). Apart from these constructed object types, we also have two predefined object types denoted by the terminal symbols `top` and `bottom` denoting the greatest, respectively the least object type according to the subtype relation (*objecttype*). The first part of a mixin type (*mixintype*) is a collection of methods, which is the same as an object type, except that it cannot be `top` or `bottom`, as it is impossible to actually construct these types. The second part denotes all super calls that can be invoked by the mixin. Recall that this type information is necessary because a mixin can only be applied to objects that handle these super calls correctly. As for self calls, this information is captured in a specialisation interface.

The following syntactic domains are defined.

$\mathfrak{I}^m_{client}$	=	set of all type expressions of the form *methodint*
$\mathfrak{I}^g_{client}$	=	singleton containing only the type expression *ghostint*
$\mathfrak{I}_{client}$	=	set of all type expressions of the form *clientint*
	=	$\mathfrak{I}^m_{client} \cup \mathfrak{I}^g_{client}$
$\mathfrak{N}_{method}$	=	set of all method names (except the name of the ghost method)
$\mathfrak{T}^c_{object}$	=	set of all type expressions of form *constructedobjecttype*
$\mathfrak{T}^g_{object}$	=	set containing only the type expressions *top* and *bottom*
$\mathfrak{T}_{object}$	=	set of all type expressions of the form *objecttype*
	=	$\mathfrak{T}^c_{object} \cup \mathfrak{T}^g_{object}$
$\mathfrak{I}_{spec}$	=	set of all type expressions of the form *specint*
$\mathfrak{T}_{method}$	=	set of all type expressions of the form *methodtype*
$\mathfrak{T}_{mixin}$	=	set of all type expressions of the form *mixintype*

5.1.2 Some Useful Operators on Type Expressions

The following selector functions on these syntactic domains are useful. Note that the notation $\mathfrak{P}$(X) denotes the powerset of a set X, i.e. the set of all subsets of X.

Name : $\mathfrak{I}_{client} \rightarrow \mathfrak{N}_{method}$:
    Name (*ghostint*) = *'ghost'*
    Name (*methodname objecttype*$_1$ *objecttype*$_2$) = *methodname*
ArgType : $\mathfrak{I}_{client} \rightarrow \mathfrak{T}_{object}$:
    ArgType (*ghostint*) = *bottom*
    ArgType (*methodname objecttype*$_1$ *objecttype*$_2$) = *objecttype*$_1$
ResType : $\mathfrak{I}_{client} \rightarrow \mathfrak{T}_{object}$:
    ResType (*ghostint*) = *top*
    ResType (*methodname objecttype*$_1$ *objecttype*$_2$) = *objecttype*$_2$
Client : $\mathfrak{T}_{method} \rightarrow \mathfrak{I}_{client}$:
    Client (*clientint specint*) = *clientint*
Self : $\mathfrak{T}_{method} \rightarrow \mathfrak{I}^m_{client}$:
    Self (*clientint specint*) = *specint*
MixinInt : $\mathfrak{T}_{mixin} \rightarrow \mathfrak{T}^c_{object}$:
    MixinInt (*constructedobjecttype specint*) = *constructedobjecttype*
Super : $\mathfrak{T}_{mixin} \rightarrow \mathfrak{I}_{spec}$:
    Super (*constructedobjecttype specint*) = *specint*

Apart from these selector functions we have two extra functions merely for translating lists in sets which are easier to deal with mathematically. The function `Set` on specialisation interfaces generates a set of all method interfaces of which a specialisation interface consists and the function `Interface` on general object types generates a set of all method types of which an object type consists.

Set : $\mathfrak{I}_{spec} \rightarrow \mathfrak{P}(\mathfrak{I}^m_{client})$:
    Set (*methodint*$_1$ *methodint*$_2$ ... *methodint*$_n$)
        = { *methodint*$_1$, *methodint*$_2$, ... , *methodint*$_n$ }

Interface : $\mathfrak{T}^c_{object} \rightarrow \mathfrak{P}(\mathfrak{T}_{method})$:
    Interface (*methodtype*$_1$ *methodtype*$_2$ ... *methodtype*$_n$)
        = { *methodtype*$_1$, *methodtype*$_2$, ... , *methodtype*$_n$ }

Using the latter function, the client interface of a constructed object type can be computed as the union of the client interfaces of all of its methods, by means of the following function:

$\text{ClientInterface} = \mathfrak{T}^c_{object} \rightarrow \mathfrak{P}(\mathfrak{I}_{client})$:

$\text{ClientInterface}(T) = \{ \text{Client}(t) \mid t \in \text{Interface}(T) \}$

The self interface of a constructed object type corresponds to its set of restrictions and contains all possible self calls that can be made. Indeed, the more self calls, the more difficult to find an extension that does not conflict with these self calls, hence the more restrictive the object type. This self interface is the union of the self interfaces of all of its methods, and can be computed by means of the following function :

$\text{SelfInterface} = \mathfrak{T}^c_{object} \rightarrow \mathfrak{P}(\mathfrak{I}^m_{client})$:

$\text{SelfInterface}(T) = \cup_{t \in \text{Interface}(T)} \text{Set}(\text{Self}(t))$

Notice that in this definition, the self interface is defined as a union of sets, rather than as a set of elements (as in the previous definition). This is because the self interface of a method describes the set of all methods that are invoked through self calls within that method, which can be more than one method.

5.1.3 Syntactic Constraints

Some restrictions were not explicitly mentioned in the syntax (for reasons of brevity), but should nevertheless be respected.

• $\forall T \in \mathfrak{T}^c_{object}$: (1) `Client` is injective on `Interface(T)`

(2) `Name` is injective on `Client(Interface(T))`

The first restriction is needed to assure that a method is uniquely determined by its client interface. The second constraint states that it is impossible for an object to have two methods with the same name (but different types) in its client interface, since we do not allow overloading.

• *`'ghost'`* $\notin \mathfrak{N}_{method}$

To make a distinction between ordinary methods and ghost methods we require that the name `'ghost'` of a ghost method be distinct from the names of ordinary methods.

• The order of the *`methodint`*s of which a *`specint`* is composed is immaterial.

As a consequence, `Set` is bijective (up to a permutation of *`methodint`*s) since it simply maps a sequence of *`methodint`*s on a set of *`methodint`*s, and thus Set^{-1} exists as well.

• Analogously, the order of the *`methodtype`*s in a *`constructedobjecttype`* is not important.

5.2 Type Checking Substitutability

One object is substitutable for another object if its type is a subtype of the other object's type. To define the subtyping rule, definitions of contravariance and interface containment are necessary.

The contravariance rule on methods states that a method with client interface m_1 contravariantly overrides a method with the same name but with client interface m_2, if the argument type of m_2 is a subtype of the argument type of m_1 and the result type of m_1 is a subtype of the result type of m_2. The subtype relation will be defined later on in this section.

Contravariance Rule
$\forall\, m_1, m_2 \in \mathcal{I}_{client}$ with $Name(m_1) = Name(m_2)$: m_1 ***contravariantly overrides*** m_2 $\Leftrightarrow$
$ArgType(m_2)$ ***is subtype of*** $ArgType(m_1)$ $\wedge$
$ResType(m_1)$ ***is subtype of*** $ResType(m_2)$

The notion of interface containment is defined almost exactly as in [Canning&al.89] (see section 2.1). The only difference is that our definition should take care of the fact that an interface can contain two methods with the same name. More specifically, this is the case for specialisation interfaces, since it is possible to perform several self sends of the same method with different argument types. In the client interface however it is not allowed to have two methods with the same name (because of the injectivity constraints). Nevertheless, we will use the same definition for self and client interfaces, although a more specific definition could be given for the latter.

Interface Containment
$\forall\, I_1, I_2 \in \mathcal{P}(\mathcal{I}_{client})$: I_1 ***contains*** I_2 $\Leftrightarrow$
$\forall\, m_2 \in I_2 : \exists\, m_1 \in I_1 : Name(m_1) = Name(m_2)$ $\wedge$
$\forall\, m_2 \in I_2 : \forall\, m_1 \in I_1$ with $Name(m_1) = Name(m_2)$: m_1 ***contravariantly overrides*** m_2

Intuitively, an object type T_1 is a subtype of another object type T_2 if it can be used in any context requiring an object of type T_2. More specifically, at least all messages that can be sent to T_2 can also be sent to T_1, and at least all mixins that can be applied to T_2 can also be applied to T_1. The first condition can be captured by requiring the client interface of T_1 to contain the client interface of T_2. The latter condition means demanding that there should be less restrictions on T_1 than on T_2. Note the contravariant relation between the containment relationships on both interfaces.

Subtyping Rule
$\forall\, T_1, T_2 \in \mathcal{T}^c_{object}$: T_1 ***is subtype of*** T_2 $\Leftrightarrow$
$ClientInterface(T_1)$ ***contains*** $ClientInterface(T_2)$ $\wedge$
$SelfInterface(T_2)$ ***contains*** $SelfInterface(T_1)$

Notice that this is an inductive definition because "*is subtype of*" is defined in terms of "*contains*" which is defined in terms of "*contravariantly overrides*" which is in turn defined in terms of "*is subtype of*". As in [Amadio&Cardelli93] the base case is handled by the primitive object types `top` and `bottom` which satisfy the following subtyping axioms:

Definition: Subtyping Rule Bottom
$\forall\, T \in \mathcal{T}_{object}$: T *is subtype of* `bottom` $\Leftrightarrow$ T=`bottom`
$\forall\, T \in \mathcal{T}_{object}$: `bottom` *is subtype of* T

Definition: Subtyping Rule Top
$\forall\, T \in \mathcal{T}_{object}$: `top` *is subtype of* T $\Leftrightarrow$ T=`top`
$\forall\, T \in \mathcal{T}_{object}$: T *is subtype of* `top`

In other words, `top` is the upper bound of the subtype relation, and `bottom` is the lower bound.

5.3 Type checking dynamic extensibility

A mixin cannot be applied to an object if one of the methods that is called in the object through a self send is overridden by a method of the mixin in a non-contravariant way. In this case we say that the object "*excludes*" the mixin.

Exclusion
$\forall\ T \in \mathfrak{T}^{c}_{object} : \forall\ M \in \mathfrak{T}_{mixin} :$ T ***excludes*** M $\Leftrightarrow$
$\exists\ m_1 \in$ SelfInterface(T) : $\exists\ t_2 \in$ Interface(MixinInt(M)) with Name(m_1) = Name(Client(t_2)) :
not (Client(t_2) *contravariantly overrides* m_1)

Because *top* is (per definition) the greatest element according to the subtype relation, intuitively it should have an empty client interface, and contain all possible methods in its specialisation interface. Therefore *top* should exclude all possible mixin applications, since it contains all possible restrictions. Analogously, because *bottom* is the smallest element according to the subtype relation, intuitively it should have an empty self interface, and contain all possible methods in its client interface. Hence *bottom* should exclude no mixin applications, since it has no restrictions. Keeping all this in mind, we add the following axioms for defining exclusiveness on *top* and *bottom*.

Definition: Exclusion Bottom and Top
$\forall\ M \in \mathfrak{T}_{mixin} :$ *Top* ***excludes*** M
$\forall\ M \in \mathfrak{T}_{mixin} :$ not (*Bottom* ***excludes*** M)

Obviously for a mixin to be applicable to an object, the object should not exclude the mixin application, but furthermore all super calls performed in the methods of the mixin should be captured by the object.

Applicability
$\forall\ T \in \mathfrak{T}^{c}_{object} : \forall\ M \in \mathfrak{T}_{mixin} :$ M ***is applicable to*** T $\Leftrightarrow$
not (T *excludes* M) $\wedge$
$\forall\ m_1 \in$ Set(Super(M)) : $\exists\ m_2 \in$ ClientInterface(T) with Name(m_1) = Name(m_2) :
m_2 *contravariantly overrides* m_1

We also add the following axioms for defining applicability on *top* and *bottom*.

Definition: Applicability Bottom and Top
$\forall\ M \in \mathfrak{T}_{mixin} :$ not (M *is applicable to* *top*)
$\forall\ M \in \mathfrak{T}_{mixin} :$ M *is applicable to* *bottom*

5.4 Object Declaration and Extension

Besides defining when objects are substitutable and when mixins are applicable to objects, definitions are needed to specify how the object's types are obtained. Two cases can be distinguished object declarations and object extensions.

5.4.1 Object Declaration

As explained in section 4.4 the user should have the possibility to explicitly put extra constraints on objects by explicitly extending the self interface. This is done by means of an object declaration of the form: T_2 **define:** T_1 **withSelf:** $Self_{new}$. Because this information is only used to restrict the set of possible extensions and has nothing to do with the actual implementation of the object itself, we relate these self clauses with a special "ghost"-method, that will only be used for this purpose. It cannot be explicitly manipulated by the user. In our model, the information corresponding to this "ghost"-method can be retrieved by means of the following partial function[3]:

[3] Note that GhostSet will always be the empty set or a singleton.

GhostSet : $\mathfrak{T}^c_{object} \rightarrow \mathfrak{P}(\mathfrak{T}_{method})$: $T \mapsto \{ t \in \text{Interface}(T) \mid \text{Name}(\text{Client}(t)) = \texttt{'ghost'} \}$

When declaring an object through an **`define:withSelf:`**-construction, the resulting type can be computed by the following function[4]:

Definition: Object Declaration
If $(T_1,S) \in \mathfrak{T}^c_{object} \times \mathfrak{I}_{spec}$, then **DeclaredType**$(T_1,S) = T_2$ where T_2 is the *`constructedobjecttype`* with interface:
$\text{Interface}(T_2) = \big(\text{Interface}(T_1) \,/\, \text{GhostSet}(T_1) \big) \cup$
$\{ t \in \mathfrak{T}_{method} \mid \text{Client}(t) = \texttt{ghostint} \wedge \text{Self}(t) = \text{Set}^{-1}\big(\text{Set}(S) \cup \{ \text{Self}(t) \mid t \in \text{GhostSet}(T_1) \} \big) \}$

Note that the **`withSelf:`**-clause should be optional and declaring an object without a **`withSelf:`**-clause should result in copying the object on the right-hand side of **`define:`**. The following property can easily be proven.

Property: $\forall\, T \in \mathfrak{T}^c_{object} : \text{DeclaredType}(T,\varepsilon)=T$

It can again be shown by means of a case analysis that DeclaredType is indeed a function from $\mathfrak{T}^c_{object} \times \mathfrak{I}_{spec}$ to $\mathfrak{T}^c_{object}$.

All objects are constructed by applying mixins to an empty RootObject with one of the following types:

RootTypes $= \{ \text{DeclaredType}(\varepsilon,S) \mid S \in \mathfrak{I}_{spec} \} \subset \mathfrak{T}_{object}$

The following property states that RootTypes are maximal object types. The proof is straightforward and left to the reader. Note also that `Top` is an element of RootTypes, but cannot be constructed.

Property:
$\forall\, R \in \text{RootTypes}, \forall\, T \in \mathfrak{T}^c_{object} : \quad R \textit{ is subtype of } T \quad \Rightarrow \quad T \in \text{RootTypes}$

5.4.2 Object Extension

We introduce the set of ***legal mixin applications*** to define how object types are constructed through mixin applications and object declarations.

$\mathfrak{A}_{legal} = \{ (T,M) \in \mathfrak{T}^c_{object} \times \mathfrak{T}_{mixin} \mid M \textit{ is applicable to } T \}$

The result of such a legal mixin application is an object of which the type can be computed by means of the following function[5]:

Definition: Object Extension
If $(T_1,M) \in \mathfrak{A}_{legal}$, then **ExtendedType**$(T_1,M) = T_2$ where T_2 is the *`constructedobjecttype`* with interface:
$\text{Interface}(T_2) =$
$\big(\text{Interface}(T_1) \,/\, \{ t_1 \in \text{Interface}(T_1) \mid \exists\, t_2 \in \text{Interface}(\text{MixinInt}(M)): \text{Name}(\text{Client}(t_1)) = \text{Name}(\text{Client}(t_2)) \} \big)$
$\cup\ \{ t_1 \in \text{Interface}(\text{MixinInt}(M)) \mid \nexists\, m_2 \in \text{Set}(\text{Super}(M)): \text{Name}(\text{Client}(t_1)) = \text{Name}(m_2) \}$
$\cup\ \{ t \in \mathfrak{T}_{method} \mid \exists\, t_1 \in \text{Interface}(\text{MixinInt}(M)): \exists\, m_2 \in \text{Set}(\text{Super}(M)): \exists\, t_3 \in \text{Interface}(T_1):$
$\text{Name}(\text{Client}(t_1)) = \text{Name}(m_2) = \text{Name}(\text{Client}(t_3)) \wedge$
$\text{Client}(t) = \text{Client}(t_1) \wedge \text{Self}(t) = \text{Set}^{-1}(\text{Set}(\text{Self}(t_1)) \cup \text{Set}(\text{Self}(t_3))) \}$

[4] For the same reasons as with the definition of ExtendedType, it is not necessary to define this function on *`top`* and *`bottom`*.
[5] It is not necessary to define this function on *`top`* and *`bottom`* because there are no expressions in our language that correspond to these types, and because the only purpose of ExtendedType is computing the types of new objects defined by the user. *`top`* and *`bottom`* were only introduced in our type system for handling the base cases of inductive definitions.

The type of the extended object is the type of the parent object to which newly defined methods in the mixin are added and where the interface of methods that are overridden is adjusted. Recall from section 4.2 why we opted to attach the self-clauses to methods and not to mixins. Here the system uses the extra knowledge provided by our choice to work on the method-level. When a method is overridden without a super call of it being performed, the method can simply be replaced by the new one. When the overriding method does perform a super call, the old method is removed, but the self interface of the old method is added to the self interface of the new one. This is necessary because the super send can cause the method that was removed from the client interface to be called. Therefore, we have to keep track of the restrictions it poses. It is however not necessary (it would even be wrong) to keep this entire method in the client interface, as it can no longer be called directly. By means of a case analysis it can be shown that ExtendedType is indeed a function from $\mathfrak{A}_{legal}$ to $\mathfrak{T}_{object}$.

5.5 System Consistency and Completeness

In Appendix C the entire set of theorems and proofs to show that the system is consistent and complete is given. Here we sketch our approach and give the most important theorems.

5.5.1 Consistency

As the key to type checking dynamic inheritance is a trade-off between applicability and extensibility, it is important to prove that this trade-off is correctly expressed by the type system. We first show that any mixin type excluded by a certain object type, must also be excluded by this object's supertypes. If this were not the case conflicts would arise when assigning a subtype to a supertype and then trying to apply a mixin that is excluded by the subtype and not by the supertype.

Theorem 1: Consistency between substitutability and exclusion
Let $T_1, T_2 \in \mathfrak{T}_{object}$, $M \in \mathfrak{T}_{mixin}$, then:
T_1 *is subtype of* T_2 $\wedge$ T_1 *excludes* M $\Rightarrow$ T_2 *excludes* M

In the same vein we prove that all mixin types that are applicable to a certain object type, are also applicable to this object's subtypes. Otherwise conflicts would arise when assigning a subtype to a supertype and then trying to apply a mixin that is applicable to the supertype and not to the subtype.

Theorem 2: Consistency between substitutability and applicability
Let $T_1, T_2 \in \mathfrak{T}_{object}$, $M \in \mathfrak{T}_{mixin}$, then:
T_1 *is subtype of* T_2 $\wedge$ M *is applicable to* T_2 $\Rightarrow$ M *is applicable to* T_1

The proofs of these theorems can be found in Appendix C.1.

5.5.2 Well-typedness

In Appendix B.1 we define a generic function *LookUp* for looking up `Variable`s or `MixinName`s in their environments VarEnv respectively MixEnv. In Appendix B.2 we describe a generic function TYPE that associates types to language expressions. This function makes use of the *LookUp* function and is only defined on well-typed expressions.

A program is *well-typed* if it will be accepted by our type-checker. Hence the definition of well-typedness provides a sort of formal description of the type checker. In the next section we will then prove that our type system is complete in the sense that every program accepted by it is indeed type-safe. Before we can give an overall definition of when a `Program` is well-typed, we need well-typedness definitions for several subparts of a program.

Since TYPE is only defined on well-typed expressions and the TYPE of a `Variable` or `MixinName` is computed by looking it up in its environment, for a `Variable` or `MixinName` to be well-typed we have to verify whether it is present in the environment.

Definition: **Well-typed variable**
A `Variable` `v` is ***well-typed*** if `v` *is in* VarEnv

Definition: **Well-typed mixin name**
A `MixinName` `M` is ***well-typed*** if `M` *is in* MixEnv

The language syntax shows that on the top level of a program there are three kinds of expressions: object declarations, method sends and assignments. An object declaration is well-typed if all the type definitions in its self-clause are well-typed and if the object that is used as a basis for the declaration is well-typed. Note that it is not necessary to check the consistency of this object with the self-clause, because the self-clause is used to impose additional constrains and has nothing to do with the object.

Definition: **Well-typed object declaration**
An `ObjectDeclaration` o_2 **`define:`** o_1 **`withSelf:`**`[`mt_1`;…;`mt_m`]` is ***well-typed*** if
$\forall$ `i` : mt_i is *well-typed* $\wedge$ o_1 is *well-typed*

An assignment is well-typed if the type of the object on the right-hand side is a subtype of the type of the object on the left-hand side. Note that on the right-hand side a method send or a mixin application can occur as well as a variable, but since the TYPE of all of these is an element of $\mathfrak{T}_{object}$, this is not a problem.

Definition: **Well-typed assignments**
An `Assignment` or `ExtAssignment` o_1 **`:=`** o_2 is ***well-typed*** if
o_1 is *well-typed* $\wedge$ o_2 s *well-typed* $\wedge$ TYPE(o_2) *is a subtype of* TYPE(o_1)

A message send is legal if the method is defined on the receiving object and the argument is of a correct (sub)type.

Definition: **Well-typed message sends**
A `MethodSend` o_1 N`(a)` is ***well-typed*** if
o_1 is *well-typed* $\wedge$ `a` is *well-typed* $\wedge$
$\exists$ m $\in$ ClientInterface(TYPE(o_1)): Name(m) = N $\wedge$ TYPE(`a`) *is a subtype of* ArgType(m)

In the body of a method declaration a fourth sort of expression is allowed, namely expressions of the form **`return`** `x`. For those expressions an extra check is needed to verify that the type of the object that is returned corresponds to the expected result type given in the method declaration. Note that according to the syntax it is possible to include several return-statements in one method body. At run time only the first one that is actually encountered will be executed. It is however necessary to type check them all, as it cannot be detected statically which one will be executed.

Furthermore, for a method declaration to be well-typed all self sends that are performed in its body need to have a corresponding method type in the method's self-clause. It is however allowed to have more methods in this clause than are actually called through a self send.

Definition: Well-typed method declarations

A `MethodDeclaration`
`N(A:ArgType) Method:[me_1;…;me_n] Result:ResultType Self:[mt_1;…;mt_m]`
is ***well-typed*** if

$\forall$ j : mt_j is *well-typed* $\wedge$ `ArgType` is *well-typed* $\wedge$ `ResultType` is *well-typed*

$\wedge$ $\forall$ me_i of the form **return** `Var` : `Var` is *well-typed* $\wedge$ TYPE(`Var`) *is a subtype of* TYPE(`ResultType`)

$\wedge$ $\forall$ me_i of a form other than **return** `Var` : me_i is *well-typed*

$\wedge$ $\forall$ me_i of the form `b` **:= self** `m(a)` :

$\exists$ mt_j : Name(TYPE(mt_j)) = `m` $\wedge$
TYPE(`a`) *is a subtype of* ArgType(TYPE (mt_j)) $\wedge$
ResType(TYPE (mt_j)) *is a subtype of* TYPE(`b`)

Note that in this last clause (as well as in the next definition) we assume again that all methods have an argument and result. The extension of these rules to include the other cases is straightforward.

A mixin declaration is well-typed if every mixin expression in its body is well-typed. Furthermore we require for every method declaration that appears as a mixin expression of this mixin that all super sends performed in the body of this method declaration have a corresponding method type in the super-clause of the mixin-declaration.

Definition: Well-typed mixin Declarations

A `MixinDeclaration` `N Mixin:[me_1;…;me_n] Super:[mt_1;…;mt_m]` is ***well-typed*** if

N *is **not** in* MixEnv $\wedge$ $\forall$ i : me_i is *well-typed* $\wedge$ $\forall$ j : mt_j is *well-typed* $\wedge$

$\wedge$ $\forall$ me_i of the form `M(A:ArgType) Method:[e_1;…;e_p] Result:ResultType Self:[t_1;…;t_q]`:

$\forall$ e_k of the form `b` **:= super** `(a)` :

$\exists$ mt_j : Name(TYPE(mt_j)) = `M` $\wedge$
TYPE(`a`) *is a subtype of* ArgType(TYPE (mt_j)) $\wedge$
ResType(TYPE (mt_j)) *is a subtype of* TYPE(`b`)

A mixin application is well-typed if the type of the mixin is applicable to the type of the object it is applied to.

Definition: Well-typed mixin applications

A `MixinApp` `O M` is ***well-typed*** if

`O` is *well-typed* $\wedge$ `M` is *well-typed* $\wedge$ TYPE(`M`) ***is applicable to*** TYPE(`O`)

A program is well-typed if every one of its mixin declarations and expressions is well-typed.

Definition: Well-typed program

A Program `[mixindec_1;...;mixindec_n; exp_1;...;exp_m]` is ***well-typed*** if

$\forall$ i : $mixindec_i$ is *well-typed* $\wedge$ $\forall$ j : exp_j is *well-typed*

5.5.3 Completeness

The aim of this section is to show that all programs that will be accepted by our type-checker (all well-typed programs) are actually type-safe. Intuitively, a program is defined type-safe if no run time type errors can occur.

Definition: Type-safe

A program is ***type-safe***

$\Leftrightarrow$ No "Message not understood",
"Illegal result type",
"Illegal argument type" or
"Mixin not applicable"
errors will occur during the execution of the program.

Next we prove that the definitions of well-typedness of previous section indeed capture all these errors. We therefore show (see Appendix C.2) that the following property[6] holds.

Property:
If in a `Program` all `MethodSend`s, `ExtMethodSend`s, `MixinApp`s, `MethodDeclaration`s, `MixinDeclaration`s, `Assignment`s and `ExtAssignment`s are well-typed,
them the `Program` is type-safe.

Since a well-typed program only contains well-typed subexpressions, the completeness theorem follows almost immediately from this property. Again we refer to Appendix C.2 for the proof.

Theorem 3: Completeness
Every well-typed program is type-safe.

6 Future Work

It is not enough for a type system to be fine-grained and consistent, it also needs to be understandable for the programmer and easy to work with. In section 6.1 *sets* of mixins are introduced as a means to make the type system more comprehensible. Section 6.2 shortly discusses how the introduction of specialisation interfaces also opens up a lot of interesting, new perspectives in software engineering in general.

6.1 A Practical System

[Hamer&al.92] introduces *classifiers* as a means to implement constraints on generalisation relationships that cannot be expressed using standard (multiple) inheritance. A classifier is a set of mutually exclusive subclasses of some basic class. We want to introduce a complementary feature for grouping mixins with *similar* specialisation interfaces (as opposed to exclusive classes as in [Hamer&al.92]).

The main reasons to do this are practicality and comprehensibility. In the example of section 4.4 the choice between substitutability and extensibility could only be made by the programmer that declares `aMammal`. This implies that a programmer using a mixin library needs to evaluate the specialisation interfaces of the mixins in the library in order to decide what mixins are combinable. This can be rather cumbersome. It is primarily the programmer who defined the library that has this knowledge, not the user. This knowledge has to be made explicit in the mixin library. This can be done by introducing a mechanism that allows a restriction on the combination of mixins to be stated explicitly.

We included such a construct in an experimental version of the type system. We will not introduce it formally here, but shortly discuss how it could work. The type rules express what mixins can be applied after what other mixins without causing type problems. They thus structure the set of mixins in subsets with similar specialisation interfaces. This structure needs to be made explicit. If we focus on the problems discussed above, two sets of mixins are of interest to us: one that contains `MakeBear` and one that contains `MakeHerbivore`. For the sake of the argument we defined sets with only one element in the example.

6 This property does not necessarily work in the opposite direction because in general it is not necessary to check all possible expressions in the program, but only the ones that will actually be executed.

```
BearSet := { MakeBear };
HerbivoreSet := { MakeHerbivore };
```

The constraints on the specialisation interface of `MakeBear`, can then be expressed as an exclusion constraint on these sets:

```
BearSet excludes HerbivoreSet;
```

Intuitively this exclusion means that a mixin of `HerbivoreSet` cannot be applied to an object to which a mixin of `BearSet` was already applied. Variable declarations need to be extended with an optional clause that specifies all sets of applicable mixins. This clause replaces the current **`withSelf:`**-clause, as it serves the same purpose. An object can only be extended with mixins that belong to one of the sets with which it was declared. Again, mammal variables can be defined in several ways, depending on the programmer's choice.

```
aFirstMammal define: rootObject MakeMammal;

aSecondMammal  define: rootObject MakeMammal
               withMixinSets: HerbivoreSet;
```

It is possible to extend `aSecondMammal` with (the only element of the `HerbivoreSet`), while `aFirstMammal` can not be extended. Furthermore, any object created with a mixin contained in `BearSet` (i.e. `MakeBear`) cannot be assigned to `aSecondMammal` since `BearSet` excludes mixins with which `aSecondMammal` can be extended (i.e. `MakeHerbivore` contained in `HerbivoreSet`).

In this form this approach is more restrictive than our current type system, but it probably can be refined. On the other hand, besides the more comprehensible notation, one of the strengths of this approach is that it is easier to add extra constraints for other than typing reasons. It could, for example, be useful to add an exclusion constraint on `MakeMale` and `MakeFemale`, purely for design reasons, even when applying the two consecutively wouldn't cause any type errors. The type system only needs to check whether at least all exclusions that should be imposed to prohibit type errors are actually declared.

In general, we feel that the use of mixins as basic components for reuse has not been investigated sufficiently yet. We are convinced that the role of typing in general and specialisation interfaces more specifically will be considerable.

6.2 Types for Software Engineering

Specialisation interfaces were first introduced as a means to enhance reuse. In [Johnson&Russo91] two important techniques for reuse of design: *abstract classes* and *frameworks* are discussed. While an abstract class is a design for a single object, a framework is the design of a *set* of objects that collaborate to carry out a set of responsibilities. Furthermore, they remark that currently frameworks are usually only described by the code and informal descriptions, where ideally the information given by the code should be completed with a formal description of the constraints in a framework. These specify how abstract classes and frameworks can be refined, specialised and combined. One attempt to do this is through *contracts* [Helm&al.90] [Holland92]. Contracts give information about how objects work together in a system. This information includes: the object's interface; which message sends (have to) cause which other message sends; invariants and instantiation requirements.

Specialisation interfaces as we use them in our type system can be used to provide similar information as contracts. One of the main differences is that contracts explicitly describe compositions of different kinds of objects (“ensembles” in [Johnson&Russo91]). One could think of specialisation interfaces as of contracts between objects and their inheritors. We want to investigate how the concept of specialisation interfaces can be extended to play the role of contracts between a number of objects, keeping in mind that specialisation interfaces only use types and do not include semantic information such as invariants.

In a similar vein, we want to investigate how the typing of specialisation interfaces can be used to support the correct reuse of abstract classes. A project strongly related to this is ACTS (Abstract Concrete Type System) [Dodani&Tsai92], a type system that imposes different rules on inheritance between abstract classes and inheritance between concrete classes. We already mentioned their observations that inheritance is used to express two kinds of relationships (the substitutable is-a relationship and abstraction of common behaviour) and that the choice between covariance and contravariance should be directly related to the specific use of inheritance. They therefore propose a type system where inheritance between abstract classes has to comply with the covariance rule and inheritance between concrete classes to the contravariance rule.

To our opinion ACTS does not make a fine enough distinction between different kinds of inheritance due to the absence of specialisation interfaces. Specialisation interfaces can be used to make a distinction between the former two kinds of inheritance and a new one: design inheritance. Similar to abstract classes specialisation interfaces document a part of the design of a class: the layering of methods. This information is not fully exploited by the current type system. The only restriction, involving specialisation interfaces, imposed by the type system is method type correctness. One step in the direction of distinguishing design from plain code inheritance would be an extension of the type rules in order to introduce explicit ‘abstract’ and ‘template’ methods — methods of which the specialisation interface can only be refined.

7 Conclusions

Dynamic object extension can be made type-safe without loosing the flexibility of prototype based languages. The key to typing dynamic inheritance is a trade-off between the set of objects with which an object can be substituted and the set of possible extensions. Specialisation interfaces were introduced in the system to provide extra information necessary to allow the creation of inheritors that are not in a subtype relationship with their parent. Furthermore, the use of specialisation interfaces in this type system opens up a lot of new perspectives in software engineering for flexible object-oriented systems.

We have given a type system for an object-based language with mixin-based inheritance. Its consistency and completeness was proven.

Acknowledgements

We owe our gratitude to Wolfgang De Meuter, Theo D'Hondt, Wim Codenie, Marc Van Limberghen, Tom Mens, Serge Demeyer and Kris De Volder for reading earlier versions of this paper. We would also like to thank Luca Cardelli for his advice on how to prove some of the theorems.

References

[Abadi&Cardelli94] M.Abadi, L. Cardelli: A Theory of Primitive Objects: Second-Order Systems. In Proceedings of European Symposium on Programming '94, pp. 1-25, Springer-Verlag 1994

[Amadio&Cardelli93] R.Amadio, L.Cardelli: *Subtyping Recursive Types*, ACM Transactions on Programming Languages and Systems, 15(4), pp. 575-631, 1993.

[Bracha&Cook90] G. Bracha and W. Cook: *Mixin-based Inheritance*. In Proceedings of ACM Joint OOPSLA/ECOOP'90 Conference Proceedings, pp.303-311, ACM Press 1990.

[Bruce&al.93] K. Bruce, J. Crabtree, T. Murtagh, R. van Gent, A. Dimock, R. Muller: *Safe and Decidable Type Checking in an Object-Oriented Language*. In Proceedings of OOPSLA '93 Conference on Object Oriented Programming, Systems, Languages and Applications, pp. 29-46, ACM Press 1993.

[Canning&al.89] P. Canning, W. Cook, W. Hill, W. Olthoff: *Interfaces for Strongly-Typed Object-Oriented Programming*, In Proceedings of OOPSLA '89 Conference on Object Oriented Programming, Systems, Languages and Applications, pp. 457-467, ACM Press 1989.

[Cardelli&Mitchell89] L.Cardelli, J.C.Mitchell: *Operations on Records*, Mathematical Structures in Computer Science, 1(1):3-48, 1991

[Cardelli&Wegner85] L. Cardelli, P.Wegner: *On Understanding Types, Data Abstraction, and Polymorphism*, Computing Surveys, Vol. 17, No. 4, December 1985

[Cook89] W. Cook: *A Proposal for Making Eiffel Type-Safe*, Proceedings ECOOP '89 European Conference on Object-Oriented Programming, Springer-Verlag 1989.

[Codenie&al.94] W. Codenie, K. De Hondt, T. D'Hondt, P. Steyaert: *Agora: Message Passing as a Foundation for Exploring OO Language Concepts*, SIGPLAN Notices, Volume 29, Number 12, December '94, pp. 48-58.

[Dodani&Tsai92] M.Dodani,C. Tsai: ACTS: *A Type System for Object-Oriented Programming Based on Abstract and Concrete Classes*, In Proceedings of ECOOP '92 European Conference on Object-Oriented Programming, pp. 309-328, Springer-Verlag 1992.

[Dony&al.92] C. Dony, J. Malenfant, P. Cointe: *Prototype-based Languages: From a New Taxonomy to Constructive Proposals and Their Validation*. In Proceedings of OOPSLA '92 Conference on Object Oriented Programming, Systems, Languages and Applications, pp. 201-217, ACM Press 1992.

[Hamer&al.92] J. Hamer, J. Hosking, W.B. Mugridge: *Static Subclass Constraints and Dynamic Class Membership Using Classifiers*, Technical Report, University of Auckland, Computer Science Department

[Harrison&Ossher93] W.Harrison, H. Ossher: *Subject-Oriented Programming (A Critique of Pure Objects)*, In Proceedings of OOPSLA '93 Conference on Object Oriented Programming, Systems, Languages and Applications, pp. 411-428, ACM Press 1993.

[Hauck93] F. Hauck: *Inheritance Modeled with Explicit Bindings: An Approach to Typed Inheritance*, In Proceedings of OOPSLA '93 Conference on Object Oriented Programming, Systems, Languages and Applications, pp. 231-239, ACM Press 1993.

[Helm&al.90] R.Helm, I.M.Holland, D.Gangopadhyay: *Contracts: Specifying Behavioral Compositions in Object-Oriented Systems*, In Proceedings of ACM Joint OOPSLA/ECOOP'90 Conference Proceedings, pp.169-180, ACM Press 1990

[Holland92] I.Holland: *Specifying Reusable Components using Contracts*, In Proceedings of ECOOP '92 European Conference on Object-Oriented Programming, pp. 287-308, Springer-Verlag 1992.

[Johnson&Russo91] R.Johnson, V.Russo: *Reusing Object-Oriented Designs*, University of Illinois tech report UIUCDCS, may 1991

[Lamping93] J. Lamping: *Typing the Specialization Interface*, In Proceedings of OOPSLA '93 Conference on Object Oriented Programming, Systems, Languages and Applications, pp. 201-214, ACM Press 1993.

[Lieberman86] H. Lieberman: *Using Prototypical Objects to Implement Shared Behaviour in an Object-Oriented System*. In Proceedings of OOPSLA '86 Conference on Object Oriented Programming, Systems, Languages and Applications, pp. 214-223, ACM Press 1986.

[Lippman93] S. Lippman: *C++ Primer 2nd Edition*, Addison-Wesley Publishing Company, 1993

[Mitchell90] J.C.Mitchell: *Toward a typed foundation for method specialisation and inheritance*, ACM Conference on Principles of Programming Languages, pp. 109-124, 1990

[Moon86] D.A. Moon: *Object-oriented Programming with Flavors*, In Proceedings of OOPSLA '86 Conference on Object Oriented Programming, Systems, Languages and Applications, pp. 1-8, ACM Press 1986

[Nierstrasz93] O. Nierstrasz: *Regular Types for Active Objects*, In Proceedings of OOPSLA '93 Conference on Object Oriented Programming, Systems, Languages and Applications, pp. 1-15, ACM Press 1993

[Palsberg&Schwartzbach94] J. Palsberg and M. Schwartzbach: *Object-Oriented Type Systems*, John Wiley and Sons, 1994

[Pierce92] B. Pierce: *Bounded Quantification is Undecidable*. In Proceedings of 19th ACM Symposium on Principles of Programming Languages, pp. 305-315, ACM Press 1992

[Steyaert&al.93] P. Steyaert, W. Codenie, T. D'Hondt, K. De Hondt, C. Lucas, M. Van Limberghen: *Nested Mixin-Methods in Agora*, In Proceedings of ECOOP '93 European Conference on Object-Oriented Programming, pp. 197-219, Springer-Verlag 1993.

[Ungar&Smith87] D. Ungar, R.B. Smith: *Self: The Power of Simplicity*. In Proceedings of OOPSLA '87 Conference on Object Oriented Programming, Systems, Languages and Applications, pp. 227-242, ACM Press 1987.

[Wegner&Zdonik88] P. Wegner, S. B. Zdonik: *Inheritance as an Incremental Modification Mechanism, or What Like is and Isn't Like*, In Proceedings of ECOOP'88 European Conference on Object-Oriented Programming, pp.55-77, Springer-Verlag 1988.

Appendix A: Syntax

A.1 The Language Syntax

```
Program ::= MixinDeclaration1; …; MixinDeclarationn;
            Expression1; …; Expressionm.

MixinDeclaration ::= MixinName  Mixin: [ MixinExpression ]
                                Super: SpecInterface

Expression ::=    ObjectDeclaration
              |   Assignment

Assignment ::=    Variable := Object

MixinExpression ::=  MethodDeclaration
                  |  ObjectDeclaration
                  |  MixinExpression ; MixinExpression
```

```
MethodDeclaration ::= MethodName ( Variable:Variable )
                          Method:[ MethodExpression1; …; MethodExpressionn ]
                          Result: Variable
                          Self: SpecInterface

ObjectDeclaration ::=  Variable  define: ProtoType
                                 withSelf: SpecInterface

MethodExpression ::=   ObjectDeclaration
                     | ExtAssignment
                     | return Variable

ExtAssignment ::=   Variable := ExtObject

ProtoType   ::=   Variable
                | MixinApp

MixinApp    ::=   Variable MixinName
                | rootObject MixinName

MethodSend  ::= Variable MethodName ( Variable )

ExtendedMethodSend ::=  MethodSend
                     |  self MethodName ( Variable )
                     |  super ( Variable )

Object   ::=   Variable
             | MethodSend
             | MixinApp

ExtObject   ::=   Variable
                | ExtendedMethodSend
                | MixinApp

SpecInterface ::= [ MethodType1; …; MethodTypen ]

MethodType ::=  MethodName ( Variable ) Result: Variable
```

Furthermore, for convenience we assume that `MixinName`, `MethodName` and `Variable` are disjoint sets of names.

A.2 The Type Syntax

```
methodint   ::=   methodname objecttype1 objecttype2
ghostint    ::=   'ghost' bottom top

clientint   ::=   methodint
                | ghostint
specint     ::=   ε
                | methodint specint

methodtype  ::=   clientint specint

constructedobjecttype   ::=   ε
                            | methodtype constructedobjecttype

objecttype  ::=   constructedobjecttype
                | top
                | bottom

mixintype   ::=   constructedobjecttype specint
```

Appendix B: Environments and Typing

B.1 Environments

Our type system needs some environments to record the types of the mixins and objects that have been declared. We will not describe these structures in detail. We simply assume that there is an environment VarEnv (possibly structured in subsections) associating declared variables with their types, an environment MixEnv associating declared mixins with their types, and a generic function *LookUp* for looking up variables in VarEnv or mixinnames in MixEnv. Note that, while for mixinnames *LookUp* simply needs to look up the mixin's type in MixEnv, for looking up variables in VarEnv it also needs to take scoping into account. We also need a function *MethodLookUp* for finding the type associated with a method of a given name in an object of a given type.

Of course we also need to explain how all these environments will be constructed. However, since we did not introduce environments formally, and because the construction of the environments is quite straightforward, we only give one (informal) example. When declaring a mixin

`Name` **`Mixin:`** `[`o_1`;...;`o_m`;`m_1`;...;`m_n`]` **`Super:`** `SpecInterface`

the type of the declared mixin is associated with its `Name` in the environment MixEnv. Furthermore, all object declarations o_1, ..., o_m are added to VarEnv (at the correct scoping level).

B.2 Typing

We now introduce a generic function TYPE to make the correlation between the syntax of our language (Appendix A.1) and our type expressions (Appendix A.2). TYPE is defined on the following language constructs, and can only be computed if these language constructs or *well-typed* (see section 5.5.2).

Variables:

TYPE (`Variable`) = *Lookup*(`Variable`,VarEnv)

So in order to determine the type of a well-typed variable, we simply need to look it up in the environment of variables. The same goes for (well-typed) mixin names:

Mixin names:

TYPE (`MixinName`) = *Lookup*(`MixinName`,MixEnv)

Mixin declarations:

TYPE (`Name` **`Mixin:`**`[`o_1`;...;`o_m`;`m_1`;...;`m_n`]` **`Super:`** `SpecInterface`)

= TYPE(m_1) ... TYPE(m_n) TYPE(`ExtSpecInterface`) $\in \mathfrak{T}_{mixin}$

where `ExtSpecInterface` = Set^{-1}(Set(`SpecInterface`) $\cup$ SuperExtension)

where SuperExtension = { m TYPE(`a`) TYPE(`b`) |

$\exists\ m_i$ of the form `M (A:ArgType)` **`Method:`**`[`e_1`;…;`e_p`]` **`Result:`**`ResultType` **`Self:`**`[`t_1`;…;`t_q`]` :

$\exists\ e_k$ of the form `b` **`:= self`** `m(a)` :

`not` $\exists\ m_j$: of the form `M’ (A:ArgType)` **`Method:`**`[`e_1`;…;`e_p`]` **`Result:`**`ResultType` **`Self:`**`[`t_1`;…;`t_q`]` :

m = M’ }

Note that the o_i’s and m_i’s can actually appear intertwined and that the mutual order of the m_i’s is of no importance.

The artificial extension of the super-clause is necessary because not all self sends that are performed in a method body are calls of methods defined on the same level. It is also possible to perform self sends of methods defined in the parent. For these methods the same checks as for methods called through super sends are necessary. Therefore, these methods are added to the super-clause here.

Method declarations:

TYPE (`Name(A:`Var_1`)` **`Method:`**`[`M_1`;...;`M_n`]` **`Result:`**Var_2 **`Self:`**`SpecInterface`)

= `Name` TYPE(Var_1) TYPE(Var_2) TYPE(`SpecInterface`) $\in \mathfrak{T}_{method}$

SpecInterfaces:

TYPE (`[mt1; ... ;mtn]`) = TYPE(`mt1`) ... TYPE(`mtn`) $\in \mathfrak{J}_{spec}$

Methodtype declarations:

TYPE (`Name(Variable1) Result: Variable2`)

= `Name` TYPE(`Variable1`) TYPE(`Variable2`) $\in \mathfrak{J}^{m}_{client}$

ObjectDeclarations:

TYPE (`Variable define: ProtoType withSelf: SpecInterface`)

= DeclaredType(TYPE(`ProtoType`), TYPE(`SpecInterface`)) $\in \mathfrak{T}^{c}_{object}$

Mixin applications:

TYPE (`Variable MixinName`)

= ExtendedType(TYPE(`Variable`),TYPE(`MixinName`)) $\in \mathfrak{T}^{c}_{object}$

TYPE (`rootObject MixinName`)

= ExtendedType(ε,TYPE(`MixinName`)) $\in \mathfrak{T}^{c}_{object}$

Method sends:

TYPE (`Variable1 MethodName(Variable2)`)

= ResType(*MethodLookUp*(`MethodName`,TYPE(`Variable1`))) $\in \mathfrak{T}_{object}$

Appendix C: Concistency and Completeness

C.1 Consistency

Lemma 1: Alternative definition of object types.

$T \in \mathfrak{T}^{c}_{object} \iff \forall t \in \text{Interface}(T) : t \in \mathfrak{T}_{method} \wedge$
$\forall t_1, t_2 \in \text{Interface}(T) : \text{Name}(\text{Client}(t_1)) = \text{Name}(\text{Client}(t_2)) \Rightarrow t_1 = t_2$

Proof: *Follows from the definition of `constructedobjecttype`s (keeping in mind the restrictions on them).*

Lemma 2: Alternative formulation of Subtyping Rule

$\forall T_1, T_2 \in \mathfrak{T}^{c}_{object} : T_1$ *is subtype of* $T_2 \iff$

$\forall m_2 \in \text{ClientInterface}(T_2) : \exists! m_1 \in \text{ClientInterface}(T_1)$ with $\text{Name}(m_1) = \text{Name}(m_2)$:

ArgType(m_2) *is subtype of* ArgType(m_1) $\wedge$ ResType(m_1) *is subtype of* ResType(m_2)

$\wedge$ $\forall m_1 \in \text{SelfInterface}(T_1) : \exists m_2 \in \text{SelfInterface}(T_2) : \text{Name}(m_1) = \text{Name}(m_2)$

$\wedge$ $\forall m_1 \in \text{SelfInterface}(T_1) : \forall m_2 \in \text{SelfInterface}(T_2)$ with $\text{Name}(m_1) = \text{Name}(m_2)$:

ArgType(m_1) *is subtype of* ArgType(m_2) $\wedge$ ResType(m_2) *is subtype of* ResType(m_1)

Proof: *Follows immediately from the definitions.*

Lemma 3: Transitivity of subtyping

$\forall T_1, T_2, T_3 \in \mathfrak{T}_{object} : T_1$ *is subtype of* $T_2 \wedge T_2$ *is subtype of* $T_3 \Rightarrow T_1$ *is subtype of* T_3

Proof: *Induction on the derivations.*

One can show, by induction on the depth of the tree of recursive applications of the subtyping rule, that:

T_1 *is subtype of* T_2

$\Rightarrow \forall T_3 \in \mathfrak{T}_{object}$: T_2 *is subtype of* $T_3 \Rightarrow T_1$ *is subtype of* T_3

T_3 *is subtype of* $T_1 \Rightarrow T_3$ *is subtype of* T_2

from which transitivity follows immediately.

The base case is handled by the subtyping rules for *`top`* and *`bottom`*. ❑

Note: Although we did not explicitly introduce a notion of recursive types into our model, we think that it is feasible to do so. For example, this inductive proof can be generalised in order to cope with recursive types as well, because a recursive type is still a finite syntactic construction. See [Amadio&Cardelli93] for a finitary type system for recursive types with folding and unfolding.

Corollary 1: Transitivity of contravariance and interface containment
The relations "*contravariantly overrides*" and "*contains*" are transitive.

Proof:
The transitivity of "*contains*" follows from the transitivity of "*contravariantly overrides*" which follows immediately from the transitivity of "*is subtype of*". ❑

Corollary 2:
$\forall$ $m_1, m_2, m_3 \in \mathfrak{I}_{client}$ with Name(m_1) = Name(m_2) = Name(m_3) :
(not (m_3 *contravariantly overrides* m_1) $\wedge$ m_2 *contravariantly overrides* m_1)
$\Rightarrow$ not (m_3 *contravariantly overrides* m_2)

Proof: *Follows by contraposition from Corollary 1.*
Suppose by contraposition that m_3 *contravariantly overrides* m_2
then we have to show that either m_3 *contravariantly overrides* m_1
or not (m_2 *contravariantly overrides* m_1)
There are two trivial cases:
1) not (m_2 *contravariantly overrides* m_1) $\Rightarrow$ nothing left to prove.
2) m_2 *contravariantly overrides* m_1 $\Rightarrow$ m_3 *contravariantly overrides* m_1
because of our initial assumption m_3 *contravariantly overrides* m_2
and because of the transitivity of *contravariantly overrides* (Corollary 1). ❑

Corollary 3:
$\forall$ $m_1, m_2, m_3 \in \mathfrak{I}_{client}$ with Name(m_1) = Name(m_2) = Name(m_3) :
(not (m_3 *contravariantly overrides* m_2) $\wedge$ m_3 *contravariantly overrides* m_1)
$\Rightarrow$ not (m_1 *contravariantly overrides* m_2)

Proof: *Follows by contraposition from Corollary 1.*
The proof is completely analogous to that of Corollary 2. ❑

Theorem 1: Consistency between substitutability and exclusion
$\forall$ $T_1, T_2 \in \mathfrak{T}_{object}$, $\forall$ $M \in \mathfrak{T}_{mixin}$:
(T_1 *is subtype of* T_2 $\wedge$ T_1 *excludes* M) $\Rightarrow$ T_2 *excludes* M

Proof:
Since the proof is trivial if T_1 or T_2 is equal to `top` or `bottom`, we only give the proof for $T_1, T_2 \in \mathfrak{T}^c_{object}$.
Using the definition of exclusion we know:
T_1 *excludes* M
$\Rightarrow$ $\exists$ $m_1 \in$ SelfInterface(T_1) : $\exists$ t $\in$ Interface(MixinInt(M)) with Name(m_1) = Name(Client(t)) :
not (Client(t) *contravariantly overrides* m_1) (1)
Furthermore, the subtyping rule and definition of interface containment yield:
T_1 *is subtype of* T_2
$\Rightarrow$ SelfInterface(T_2) *contains* SelfInterface(T_1)
$\Rightarrow$ $\forall$ $m_1 \in$ SelfInterface(T_1) : $\exists$ $m_2 \in$ SelfInterface(T_2) : Name(m_1) = Name(m_2)
$\wedge$ $\forall$ $m_1 \in$ SelfInterface(T_1) : $\forall$ $m_2 \in$ SelfInterface(T_2) with Name(m_1) = Name(m_2) :
m_2 *contravariantly overrides* m_1
$\Rightarrow$ $\forall$ $m_1 \in$ SelfInterface(T_1) : $\exists$ $m_2 \in$ SelfInterface(T_2) with Name(m_1) = Name(m_2) :
m_2 *contravariantly overrides* m_1 (2)
Next, by combining (1) and (2) we get
$\Rightarrow$ $\exists$ $m_1 \in$ SelfInterface(T_1) : $\exists$ t $\in$ Interface(MixinInt(M)) with Name(m_1) = Name(Client(t)) :
not (Client(t) *contravariantly overrides* m_1) $\wedge$
$\exists$ $m_2 \in$ SelfInterface(T_2) with Name(m_1) = Name(m_2) :
m_2 *contravariantly overrides* m_1

$\Rightarrow$ $\exists\, m_1 \in$ SelfInterface(T_1), $\exists\, m_2 \in$ SelfInterface(T_2), $\exists\, t \in$ Interface(MixinInt(M)), with Name(m_1) = Name(Client(t)) = Name(m_2) :

not (Client(t) *contravariantly overrides* m_1) $\wedge$
m_2 *contravariantly overrides* m_1

From which follows by Corollary 2:

$\Rightarrow$ $\exists\, m_2 \in$ SelfInterface(T_2) : $\exists\, t \in$ Interface(MixinInt(M)) with Name(m_2) = Name(Client(t)) :

not (Client(t) *contravariantly overrides* m_2)

$\Rightarrow$ T_2 *excludes* M (using the definition of exclusion) ❑

Theorem 2: Consistency between substitutability and applicability
$\forall\, T_1, T_2 \in \mathfrak{T}_{object}$, $\forall\, M \in \mathfrak{T}_{mixin}$:
(T_1 *is subtype of* T_2 $\wedge$ M *is applicable to* T_2) $\Rightarrow$ M *is applicable to* T_1

Proof: *By contraposition.*

Since the proof is trivial if T_1 or T_2 is equal to `top` or `bottom`, we only give the proof for $T_1, T_2 \in \mathfrak{T}^c_{object.}$

Suppose by contraposition that not (M *is applicable to* T_1) (a)

then we have to show that either not (T_1 *is subtype of* T_2)
or not (M *is applicable to* T_2)

If not (T_1 *is subtype of* T_2), then there is nothing left to prove.

Therefore we can assume that T_1 *is subtype of* T_2 (b)

and try to prove that not (M *is applicable to* T_2). (c)

Because of the definition of applicability, (a) is equivalent with

T_1 *excludes* M $\vee$ (a_1)

$\exists\, m \in$ Set(Super(M)): $\forall\, m_1 \in$ ClientInterface(T_1) with Name(m) = Name(m_1):
not (m_1 *contravariantly overrides* m) (a_2)

and for the same reason showing (c) is equivalent with showing

T_2 *excludes* M $\vee$ (c_1)

$\exists\, m \in$ Set(Super(M)): $\forall\, m_2 \in$ ClientInterface(T_2) with Name(m) = Name(m_2):
not (m_2 *contravariantly overrides* m) (c_2)

Hence it suffices to show that (a_1) $\Rightarrow$ (c_1) and (a_2) $\Rightarrow$ (c_2)

- (a_1) $\Rightarrow$ (c_1) follows immediately from Theorem 1 by making use of (b).
- Proof of (a_2) $\Rightarrow$ (c_2):

  According to Lemma 2 we have

  $\forall\, m_2 \in$ ClientInterface(T_2) : $\exists\, m_1 \in$ ClientInterface(T_1) with Name(m_1) = Name(m_2) :
  m_1 *contravariantly overrides* m_2

  Combining this with (a_2) and using Corollary 3 we get (c_2). ❑

C.2 Completeness

Property:
If in a `Program` all `MethodSend`s, `ExtMethodSend`s, `MixinApp`s, `MethodDeclaration`s, `MixinDeclaration`s, `Assignment`s and `ExtAssignment`s are well-typed,
then the `Program` is type-safe.

(Sketch of the) Proof:

Using the definition of type-safety, we only need to show that for a program of the specified form all errors are indeed captured by the well-typedness constraints.

(1) *"Mixin not applicable" errors* occur when one tries to extend an object with a mixin that is not applicable to the object. As can be seen from the definition of well-typed mixin applications such errors can be avoided by verifying whether all mixin applications are well-typed.

(2) *"Message not understood" errors* occur when an object receives a request to execute a method that it does not understand.

(a) As a first requirement for capturing such errors we want the type of an object to be what we expect it to be. I.e. if we declare a variable as an object that understands a given set of messages, and later on we assign a new object to this variable, then we must make sure that the variable still has the "same" type. More specifically the object in the variable still understands the same messages (and perhaps more). This requirement is captured in the subtype condition of the definition of well-typed `Assignment`s and well-typed `ExtAssignment`s.

(b) Furthermore, for every `MethodSend` of the form

`Variable`$_1$ `MethodName(Variable`$_2$`)`

we must check if the expected type of `Variable`$_1$ indeed contains a method with the given name. This is checked in the first condition of well-typed `MethodSend`s.

(c) For self and super calls the situation is somewhat less obvious. Let us first take a look at super sends of the form **`super`**`(Variable)`. Looking closely at the language syntax we see that such `ExtMethodSend`s always occur inside an `ExtAssignment` which occurs inside the body of a `MethodDeclaration` (which in turn occurs in the body of some `MixinDeclaration`). We then need to check that a super call of this method is actually possible. In order to do this, it is verified in the definition of well-typed mixin declarations whether the corresponding method is mentioned in the **`Super:`**-part of the mixin declaration. If this is the case, the "is applicable to" constraint in the definition of well-typed mixin applications will see to it that all methods in the **`Super:`**-clause are actually implemented in the object to which the mixin is applied.

(d) Finally, we show that self calls are always understood. There are two possibilities. Either a self call is performed of a method that is declared in the same mixin, or a self call is made of a method that is not declared in the same mixin. In the latter case, we have to make sure that this method will actually be implemented by an object to which the mixin will be applied. This is indeed so, because when computing the TYPE of a mixin declaration, for all self sends to methods that are not implemented in the same mixin, the corresponding method types are added to the `specint` of the `mixintype`. As is the case for super calls, the presence of these methods in the object to which this mixin is sent will be checked by the "is applicable to" constraint in the definition of well-typed mixin applications.

(3) *"Illegal argument type" errors* occur when a request to execute a method is sent to an object, and the message is understood, but has a wrong argument type.

(a) For ordinary `MethodSend`s the correctness of the argument type is checked in the second condition of the definition of well-typed message sends.

(b) For super calls the correctness of the argument type is checked by comparing it with the argument type of the corresponding method in the super interface. This is checked in the definition of well-typed mixin declarations.

(c) For self calls the correctness of the argument type is checked by comparing it with the argument type of the corresponding method in the self interface. This occurs in the definition of well-typed method declarations.

(4) *"Illegal result type" errors* occur when a message send returns a result that does not correspond with the expected result. These errors can easily be avoided by checking if every method expression of the form `return Variable` in a `MethodDeclaration` actually satisfies the restriction that TYPE(`Variable`) is a subtype of TYPE(`ResulType`) where `ResulType` describes the expected result of the method expression. This test occurs in the definition of well-typed method declarations.

So we can indeed conclude that if all `MethodSend`s, `ExtMethodSend`s, `MixinApp`s, `MethodDeclaration`s, `MixinDeclaration`s, `Assignment`s and `ExtAssignment`s in a program are well-typed, then the program is type-safe. ❑

Theorem 3: Completeness
Every well-typed program is type-safe.

Proof:

Using the previous property we simply show that if a program is well-typed, then all `MethodSend`s, `ExtMethodSend`s, `MixinApp`s, `MethodDeclaration`s, `MixinDeclaration`s, `Assignment`s and `ExtAssignment`s in the `Program` are well-typed. The proof of this is very straightforward and is based on the general observation that a well-typed syntactic expression only contains well-typed subexpressions. E.g. consider the example of `Assignment`s.

According to the language syntax, an `Assignment` can only occur as a top-level expression. Suppose that our well-typed program contains an `Assignment` that is not well-typed. This is impossible, since this would mean that the program contains a top-level expression that is not well-typed, which is in contradiction (see definition of well-typed program) with the fact that the program is well-typed. So our initial assumption must be incorrect and thus all `Assignment`s in the program are well-typed.

The proof for the other subexpressions of a program is completely analogous, although they might involve some more steps. ❑