Vrije Universiteit Brussel
Faculteit Wetenschappen

Static Typing of Dynamic Inheritance

Carine Lucas, Kim Mens, Patrick Steyaert

Techreport vub-prog-tr-95-04

Programming Technology Lab

PROG(WE)

VUB

Pleinlaan 2

1050 Brussel

BELGIUM

Fax: (+32) 2-629-3495

Tel: (+32) 2-629-3308

Anon. FTP: progftp.vub.ac.be

WWW: progwww.vub.ac.be

Static Typing of Dynamic Inheritance

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

WWW: http://progwww.vub.ac.be/prog/

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. The total absence of static type systems for dynamic object extension is a major obstacle for its adoption. We describe a static type system using specialisation interfaces with a trade-off between possible assignments and possible extensions as key principle. We furthermore argue that the introduction of specialisation interfaces in the system opens up a lot of new perspectives in software engineering in general.*

1 Introduction

Recent developments such as subjectivity [Harrison&Ossher93], composition technology and novel prototype-based languages demonstrate that dynamic object extension is an essential feature of object-orientation. The possibility to dynamically extend objects is one of the main characteristics of prototype-based languages [Lieberman86] [Ungar&Smith87]. In contrast with conventional class-based languages where each class statically knows its parent class, in prototype-based languages inheritors only know their parent at run-time, allowing the inheritance hierarchy to be created at run-time. This results in a much more flexible approach.

On the other hand, the total absence of static type systems for dynamic object extension is a major obstacle for its adoption. 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 supertypes by subtypes. As a consequence, it is always possible for a variable to contain an object of a subtype of its formal type. In addition, not all object extensions are type correct. This complicates static type checking dynamic object extension as 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. When allowing objects that are not in a subtype relationship with their parents to be created, some extensions are still 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.

As an example we describe a type system for an object-based language in which it is possible to dynamically extend objects through mixin application[1] [Bracha&Cook90] [Steyaert&al.93].

2 Reuse of Code versus Reuse of Behaviour

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 the possibility of dynamic object extension into account. Furthermore, the use of inheritance has evolved from simply a means to reuse code to a way to achieve reuse of behaviour [Wegner&Zdonik88], which implies *substitutability*. This 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].

The main discriminator between current type systems for object-oriented languages is the choice between covariance and contravariance [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 on the other hand guarantees type safety, but reduces expressiveness. [Dodani&Tsai92] gives a clear discussion of all issues concerned with this choice. 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*

[1] We developed this type system for a toy version of the language Agora [Codenie&al.94]. For reasons of brevity we will not describe the language here; we give our examples in a straightforward syntax.

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. In this case one either 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.

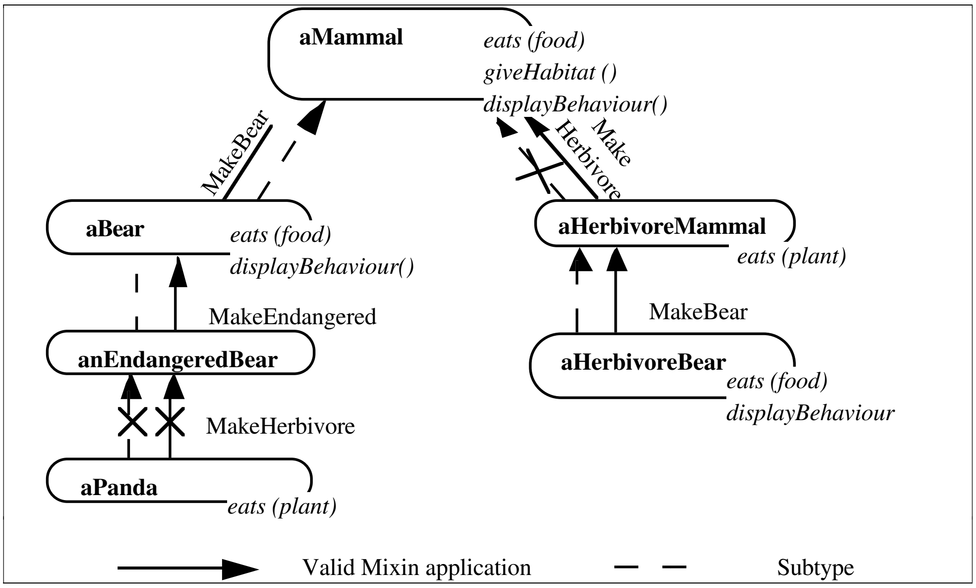

Figure 1

Figure 1 illustrates this on the example of an information system in a zoo. We consider the subsystem concerning mammals. `aMammal` contains 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 describes 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. Listing 1 shows a possible implementation of `MakeBear`.

```
mixin MakeBear is method  eats (someFood:food) is ...
                  method  displayBehaviour ()  is self eats(fish);
                                                  super (); …
```

Listing 1

Then objects are created through mixin application. For instance, `aBear` is created by applying `MakeBear` to `aMammal`. 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 `aHerbivore-Mammal`. 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).

3 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, or even even to apply MakeHerbivore before 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. 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 (its specialisation interface).

Remark that appending the self clause could have been achieved either 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) (which wasn't the case earlier in the example, and in the figure). 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). 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, such that the displayBehaviour of MakeBear 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 implemented *by the parent.* Mixins can do super sends, even though their future parents are unknown. Therefore, when applying a mixin to an object it should be verified whether all messages that are called through super sends in the mixin are implemented in the parent object. The mixin is therefore extended with a super clause[2]. Listing 2 gives `MakeBear` extended with self and super clauses. The type checker verifies whether all methods called through self and super sends are given in the intended clauses[3].

```
mixin MakeBear is method eats (someFood:food) is  ...
                  method displayBehaviour()   is  self eats(fish);
                                                  super (); …
                                                withSelf eats(food) end
               withSuper displayBehaviour() end
```

Listing 2

By adding these 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 Typing Dynamic Object Extension

All the issues raised up until now are applicable to static inheritance, as well as dynamic inheritance. Dynamic extension does however raise some particular problems. As substitutability of supertypes by subtypes is a general characteristic of object-oriented type systems, the actual type of an object can always be a subtype of its formal type. The key problem of typing dynamic object extension is that a static type checker cannot know the exact type of the object that is being extended. As far as a static type checker can determine, in listing 3 an object with type `aMammal` is extended with herbivore behaviour. The actual type of this mammal object can, however, also be `aBear`, due to the assignment of `aBear` to `aMammal`. As the extension of `aBear` with herbivore behaviour is not type correct, the program in listing 3 should be rejected.

```
variable aMammal is rootObject MakeMammal;
variable aBear is aMammal MakeBear;
bool ifTrue: aMammal := aBear;
variable herbie is aMammal MakeHerbivore.
```

Listing 3

5 Substitutability versus Dynamic Extensibility

The conflict between the above assignment of `aBear` to `aMammal` 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` is compatible with `aMammal`, with respect to the specialisation interface it is not (i.e. `aMammal` allows more extensions than `aBear`). It is therefore clear that the above typing

[2] Note that we add the super clause at the mixin level and not at the method level. As we only allow super calls of methods with the same name as the method performing the call, introducing super clauses on the method level wouldn't provide us with any extra information (only perhaps a shorter notation). The system could be adjusted to allow other super calls, but we feel it is good programming practice to perform only such super calls.

[3] We do not allow using **self** or **super** at the right-hand side of an assignment or as a parameter. Therefore the self and super clauses could be generated, but for clarity reasons we add them explicitly.

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 the 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 `aMammal := aBear`, 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 to extend the specialisation interface of an object with additional restrictions, i.e. an artificially added self clause (listing 4)[4]. It is obviously superfluous to relate the methods in this extra self clause to a certain method.

```
variable aMammal is rootObject MakeMammal withSelf eats (food) end
```

Listing 4

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. When the programmer wants to allow the `MakeHerbivore` extension, it is e.g. still possible to assign an endangered mammal to `aMammal` as this does not conflict with the extension. For the same reason it is still possible to apply `MakeEndangered` to `aMammal`, when `aMammal := aBear` 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.

6 The Type System

Because of space restrictions we only sketch the two main rules here[5].

6.1 Type Checking Substitutability

If you consider the client interface of an object to be the set of all methods and its self interface to be the union of all self clauses attached to these methods, then subtyping can be

[4] Note that this might not be the most practical approach. Other notations are possible, we just add them explicitly as extra self clauses here to make our point.

[5] For the full set of rules please contact the first author at clucas@vnet3.vub.ac.be.

defined through interface containment on both these sets. An interface A contains an interface B if all methods in A are also in B and the methods of A contravariantly override those in B. Note the reverse direction on the containment relationship on client and self interfaces.

Subtyping Rule

$$\frac{\Gamma;\ \Omega\ \vdash\ \omega_1\ \texttt{containsClient}\ \omega_2 \qquad \Gamma;\ \Omega\ \vdash\ \omega_2\ \texttt{containsSelf}\ \omega_1}{\Gamma;\ \Omega\ \vdash\ \omega_1\ <:\ \omega_2}$$

6.2 Type Checking Dynamic Extensibility

As explained in the example, a mixin is not allowed to extend an object (= the object excludes the mixin), if the mixin covariantly overrides a method that is invoked through a self send in the object.

Exclusion Rule

$$\frac{\exists\, i,j:\ m_i = n_j \text{ and not } \varphi_j <: \phi_i}{\Gamma;\ \Omega\ \vdash\ \omega\ \texttt{excludes}\ \gamma}$$

where $\omega = \{\ldots,\ m{:}\phi\ \textbf{withSelfType}\ \{m_i{:}\phi_i\}_{i=1,\ldots k}\ ,\ldots\}$
and $\gamma = \{n_j{:}\ \psi_j\ \textbf{withSelfType}\ \sigma_j\}_{j=1,\ldots,k}\ \textbf{withSuperType}\ \sigma$

A mixin is applicable to an object if it isn't excluded by it and if the object implements all methods required by the mixin's super interface.

7 Types for Software Engineering

Specialisation interfaces were first introduced as a means to enhance reuse. As mentioned earlier, inheritance was traditionally used to describe two kinds of reuse: reuse of behaviour and reuse of code. Later the object-oriented reuse community emphasised reuse of design as another major factor, with abstract classes and frameworks as the most important techniques to achieve it [Johnson&Russo91]. Finding a clear and expressive way to describe how a framework can be reused and to which constraints adaptations should comply is one of the most compelling problems in the development of reusable software. While currently the possibilities for reuse are mostly described in an informal way, we want show how type information can be used to achieve a more controllable model of modification. One way to do this is through *contracts* [Helm&al.90] [Holland92]. Specialisation interfaces as we use them in our type system can be used to provide similar information. 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.

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. 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. We first taxonomised different kinds of reuse and different operations performed through incremental modification. We distinguish three kinds of reuse: reuse of code, reuse of behaviour and reuse of design and four operations that are performed through incremental modification: extension, concretisation, generalisation and refinement. Now we investigate what constraints should be put on these operations to achieve the distinct kinds of reuse. In the current system, method type correctness is the only restriction involving specialisation interfaces. One step in the direction of distinguishing design inheritance from plain code inheritance is an extension of the type rules introducing explicit 'abstract' and 'template' methods.

8 Conclusions

Dynamic object extension can be made type safe without loosing the flexibility of prototype-based languages. The key to it 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 new perspectives in software engineering for flexible object-oriented systems.

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.

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

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

[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.

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

[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.

[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.

[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.

[Lucas&al.95] C.Lucas, K.Mens, P.Steyaert: Typing Dynamic Inheritance: A Trade-Off between Substitutability and Extensibility, Submitted to Theory and Practice of Object Systems, Special Issue on Typing, John Wiley and Sons, 1995

[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.