$Perm_{co}$ – A *Perm*issive Approach to *Co*variant *O*verriding of Subclass Members

Sophia Drossopoulou Dan Yang
Department of Computing, Imperial College
London SW7 2AZ

Technical report nr: 98/4

Abstract

We describe $Perm_{co}$, a permissive approach which allows covariant overriding of instance variables and any overriding of instance methods.

Subclasses are considered subtypes. Instance method access is treated by considering the types of all methods defined for the class of the receiver and all its possible subclasses. Instance variable access is treated by considering the types of the instance variable in all possible subclasses. Thus, $Perm_{co}$ is permissive at the point of redefinition and restrictive at the point of call or access. Closed types describe values which may belong to a class but not to its subclasses, and allow a more precise description of types. Result types may depend on receiver or argument types.

This paper introduces $Perm_{co}$ in terms of an example and compares with related approaches. It then demonstrates its soundness through a subject reduction theorem.

1 Introduction

Component overriding (*i.e.* replacing instance variables or methods in subclasses) allows objects of a subclass to respond to a message slightly differently than objects belonging to the superclass. It is an indispensable feature in object oriented programming languages, but provides challenging problems for the type system.

As demonstrated in [9], if subclasses are connected to subtypes, then one can straightforwardly obtain a sound type system by requiring the overriding components to belong to a subtype of the corresponding components in the superclass. This has the consequence that the argument types of the overriding method have to be *supertypes* of the argument types of the overridden method, *i.e.* we have contravariant overriding, and that the instance variables must have the *same* type as the instance variable of the supertype [4].

However, this proves to be too strong a restriction; as argued *e.g.* by [10, 23, 25], in programming practice situations requiring covariant redefinition often arise. The type theory community has devoted a lot of research to this problem. The solutions offered range from giving up the connection between subclasses and subtypes and applying match-bounded polymorphism [1, 7], multimethods [13], to some combination of static typing with dynamic checking [24].

An informative overview of the solutions to the more specific problem of binary methods (where the type of the argument is identical to that of the the receiver and where it changes

in subclasses together with the receiver) appeared in [5], and it is probable that some of the solutions suggested there are definitive.

However, we believe that the more general problem of component overriding, where the type of the argument is not the same as that of the receiver, has not been definitively settled yet.

This paper outlines $Perm_{co}$ and gives a short survey of other approaches. In $Perm_{co}$ subclasses are considered to be subtypes. $Perm_{co}$ is permissive at the point of redefinition and restrictive at the point of call or access: Any redefinition of method arguments and covariant redefinitions of instance variables are allowed. At the point of method call, or instance variable access we type check by considering all possible subclasses of the type of the receiver. Types indicate the set of classes to whose objects an expression may belong. Closed types, which consider a class but none of its subclasses, support a more precise description of that set, and thus they allow a tighter description of the type of an expression. Result types may depend on the receiver or the argument. $Perm_{co}$ operates under the closed world assumption. However, in the case of introduction of covariant overriding later on in the software lifecycle, $Perm_{co}$ only requires recompilation, as opposed to reprogramming required by other approaches; thus $Perm_{co}$ is more suitable for incremental program and prototype development.

The paper is organised as follows: Section 2 introduces the issues around component overriding through an example. Section 3 outlines $Perm_{co}$ and other approaches to this problem. Section 4 formally defines the approach in terms of a simple calculus and proves its soundness through a subject reduction theorem. In section 5 we draw some conclusions.

2 The Issues

Any solution to covariant component overriding is tightly connected to the solution of method binding and the relation between subclasses and subtypes. We discuss these issues in terms of the example presented in figure 1, using a variant of the syntax in [5].

The class `Car` describes cars which have a `driver`; although not explicitly stated, the instance variable `driver` should belong to the class `Driver`. On the other hand, `RaceCar`, a subclass of `Car` describes racing cars, which require special drivers of class `FastDriver`. The intention here is to override in class `RaceCar` the instance variable `driver` from `Car` in a *covariant* way.

The method `register` defined in `Car`, expecting a `aDriver` argument of class `Driver` is redefined in `RaceCar` expecting a more specific driver, *i.e.* of class `FastDriver` – a case of covariant overriding of method arguments. Thus, `(new Car).register(new Driver)` is safe, whereas `(new RaceCar).register(new Driver)` would raise a run-time exception. This demonstrates that when covariant overriding is supported replacing an object of a class by an object of a subclass is not always safe. This phenomenon was already observed in [15]. Ways of combining covariant overriding with sound typing is the subject of our paper.

The method `testDrive` expects a `Car` argument, and returns it after trying out its properties. Thus, `((new FastDriver).testDive(new RaceCar)).speedLimit` is safe. However, we shall see that $Perm_{co}$ and multimethods are the only type systems which consider this expression type correct.

```
class Car
instance variables
   driver
instance methods
   getDriver
      return driver
   register(aDriver)
      driver := aDriver.
      return self
end Car

class RaceCar inherits Car
instance variables
   speedLimit
instance methods
   register(aDriver)
      driver := aDriver.
      speedLimit := aDriver.getLicense * 10.
      return self
end RaceCar
```

```
class Driver
instance methods
   testDrive(aCar)
      ... try out aCar ....
      return aCar
end Driver


class FastDriver inherits Driver
instance variables
   license
instance methods
   getLicense
      return license
end FastDriver
```

Figure 1: Cars and Drivers – the Problem with Covariant Overriding

2.1 Overriding of Instance Variables and Parameters

We distinguish the following solutions to component overriding:

In the *novariant* solution, as called in [25], a component overrides another component only if they have the same name and the same type. In this solution objects of class `RaceCar` have a driver of class `Driver` *and* a driver of class `FastDriver`, and there is no way to express the intention that a `RaceCar` is a special case of `Car`, where the `driver` has to belong to the class `FastDriver`. Novariance is adopted by C++ and Java.

The *contravariant* solution follows the results from [9], whereby we obtain a simple, sound type system if component types in a subtype are subtypes of the corresponding component types in the supertype. In particular, functional types follow a contravariant subtype rule in the argument type:

$$\mathtt{T}_2 \leq \mathtt{T}_1,\ \mathtt{S}_1 \leq \mathtt{S}_2 \quad \Longrightarrow \quad \mathtt{T}_1 \rightarrow \mathtt{S}_1 \leq \mathtt{T}_2 \rightarrow \mathtt{S}_2$$

which means that the parameter types of the overridden method have to be supertypes of the overridden method. As argued in [4], instance variables require a `set` and a `get` function, therefore, the instance variable types would have to be the same. Thus, for imperative object oriented languages, the novariant solution is a special case of the contravariant solution. In terms of our example, either the redefinition of methods `register` would be forbidden, or `RaceCar` would not be a subtype of `Car`.

In the *covariant* solution, the parameter types of the overriding method are subtypes of the corresponding parameter types of the overridden method, and similarly, the type of an instance variable is a subtype of that being overridden. Eiffel, Beta and O2 adopt the covariant solution.

```
class Car
instance variables
   driver :  <Driver>
instance methods
   getDriver
      '≪ X ≤ <Car>.X ⟶ <Driver> ≫'
      '≪ X ≤ <RaceCar>.X ⟶ <FastDriver> ≫'
      return driver
   register(aDriver)
      '≪ X ≤ [Car].Y ≤ <Driver>.X × Y ⟶ X ≫'
      driver := aDriver.
      return self
end Car

class RaceCar inherits Car
instance variables
   driver:  <FastDriver>
   speedLimit:  <Number>
instance methods
   register(aDriver)
      '≪ X ≤ <RaceCar>.Y ≤ <FastDriver>.X × Y ⟶ X ≫'
      driver := aDriver.
      speedLimit := aDriver.getLicense * 10.
      return self
end RaceCar

class Driver
instance methods
   testDrive(aCar)
      '≪ X ≤ <Driver>.Y ≤ <Car>.X × Y ⟶ Y ≫'
      ... try out aCar ... .
      return aCar
end Driver
```

Figure 2: Cars and Drivers in $Perm_{co}$

2.2 Subclasses vs Subtypes

Objects belong to classes which determine their behaviour. Types – in our approach – describe the set of classes to whose objects evaluation of an expression might lead. The *open* type <C> denotes the type of instances of class C or any subclasses of C. For example, <Car> describes objects of class Car, or RaceCar, or any further subclass of Car. All object oriented languages support open types. In some languages (*e.g.* Eiffel, C++, but not in Java) it is possible to express the type of instances of a class C, but none of its superclasses; in our terminology this is a *closed* type [C]. For example, [Car] only describes objects of class Car.

The concepts of subclass and subtype are closely related but distinguished: Subclasses represent *implementation inheritance*, *i.e.* instances of a subclass inherit variables and methods from a superclass, and therefore they "understand" all messages that an object of a superclass understands. Subtypes represent *specification conformance*, *i.e.* if a type T_1 is a subtype of T_2, then an expression e_1 of type T_1 can replace an expression e_2 of type T_2.

We distinguish the solution whereby subclasses and subtypes are connected from that where they are not (we use $\sqsubseteq$ and $\leq$ to denote subclass and subtype):

$$[\text{unconnect}]\ \frac{\texttt{C}_2 \sqsubseteq \texttt{C}_1}{\texttt{nothing}} \qquad [\text{connect}]\ \frac{\texttt{C}_2 \sqsubseteq \texttt{C}_1}{<\texttt{C}_2> \leq <\texttt{C}_1>}$$

$$[\text{connect}_1]\ \frac{\texttt{C}_2 \sqsubseteq \texttt{C}_1}{\begin{array}{l}<\texttt{C}_2> \leq <\texttt{C}_1> \\ [\texttt{C}_2] \leq [\texttt{C}_1]\end{array}} \qquad [\text{connect}_2]\ \frac{\texttt{C}_2 \sqsubseteq \texttt{C}_1}{\begin{array}{l}<\texttt{C}_2> \leq <\texttt{C}_1> \\ [\texttt{C}_2] \leq <\texttt{C}_1>\end{array}}$$

The first solution, [unconnect], corresponds to structural type equivalence; the subtype hierarchy is separated from the subclass hierarchy. This solution is widely adopted [6, 1]; the subtype relationship holds only when all the components in the subclasses are subtypes of the corresponding components in the superclass. Furthermore, even classes which are not subclasses may form subtypes if their components have appropriate types.

In the next three solutions, subtypes are connected to subclasses: if $\texttt{C}_2$ is a subclass of $\texttt{C}_1$, then the open type $<\texttt{C}_2>$ is a subtype of the open type $<\texttt{C}_1>$. [connect] applies to languages where closed types are not expressible, eg. Java or Beta. In languages where closed types are expressible, either $[\texttt{C}_2]$ is a subtype of $[\texttt{C}_1]$, or $[\texttt{C}_2]$ is a subtype of $<\texttt{C}_1>$. The first, [connect$_1$], is the case where an implicit coercion operation takes place when passing a subclass object where a superclass object is expected; this applies to C++. The latter, [connect$_2$], is the case where entities of open and closed types have the same representation. That is so in LOOM, or Smalltalk where all entities are implicitly pointers. It is *not* so in C++, where open types correspond to pointers, and closed types correspond to entities on the stack.

2.3 Method Binding

Method binding is the process of resolving the method body to be executed when an object receives a message. *Static* binding can be resolved at compile time, thus allowing for efficient implementation but restricting flexibility. All object oriented languages support, at least in some cases, *dynamic binding*, which takes place at run-time, when the receiver is fully evaluated, and its class is known. Dynamic binding is further divided into *single method dispatch*, where only the class of the receiver is taken into account, and *multi-method dispatch*, where binding takes into account the class of receiver and several parameters.

In single method dispatch methods belong to the class where they are defined. The method defined in the nearest superclass of the receiver's class is selected. Single method dispatch is adopted by most languages, eg Smalltalk, C++ and Java.

Multi-methods are global rather than belong to particular classes. When the classes of the receivers are known, then the method whose argument types are the nearest to the actual receiver classes is selected. This solution is adopted by O2[2], CLOS[21] and Rosette [20], and it is formally described by the $\lambda^{\&}$ calculus [13].

2.4 World View, Implications of Covariant Redefinitions

Another important issue is whether an approach operates under the *open world* or the *closed world* assumption. The closed world assumption requires type information about the whole program in order to type check an expression, whereas the open world assumption does not. In particular, adding a subclass under the closed world assumption might require type

checking the complete program again, whereas, under the open world assumption, it would only require type checking the particular class. Seen in isolation, the open world assumption is by far preferable.

However, we believe, that the *implications* of *subsequent* covariant redefinitions is an issue of comparable importance: Consider the implications of a later introduction of a covariant redefinition into the class hierarchy of a program several months or even years after the initial program was developed, type checked, and deployed.

In LOOM, the classes would have to be replaced by parameterized classes, and the use of the corresponding types would have to be replaced by the instantiation of the corresponding parameterized types. Although the original non-parameterized types were subtypes, the new parameterized type instatiations may not be subtypes of each other. Therefore a certain amount of *reprogramming* might be required. In $\lambda^{\&}$ only *a little amount* of reprogramming would be required: some previously normal methods would have to be turned to multimethods. On the other hand, the approaches which do not use explicit class parameterization, *e.g.* $Perm_{co}$, Beta and Eiffel, would only require *renewed type checking* of the program. We feel that this is an advantage in the software lifecycle process.

In terms of our cars and drivers example, consider the case where the classes `Car` and `RaceCar` dealing with engines, horsepower, drivers *etc*, but *not* differentiating the drivers of normal cars from the drivers of racing cars, were developed and imported by other modules. Suppose, that, later on, it was necessary to introduce drivers for `RaceCar`s in the covariant manner. In a LOOM program one would have to replace `Car` and `RaceCar` by parameterized classes, and this change would need to be reflected in all importing modules, even those which did not make use of the vehicles' drivers. Namely, one would need to replace the previously nonparameterized type `CarType` by the type instantiation `CarType(DriverType)`. One would also have to reprogram those parts of the earlier program which were meant to work both for cars *and* race cars, because in the new version of the program they would not, even if these parts were not concerned with drivers – because `RaceCarType(FastDriverType)` does not match `CarType(FriverType)`, *cf.* section 3.4. The necessary changes in the $\lambda^{\&}$ program are less incisive: one would have to turn the method `register` into a multimethod. The $Perm_{co}$ approach would only require type checking of the importing modules. These modules would remain type correct if they did not deal with drivers. Similarly, Beta would require the introduction of a virtual type for the driver in class `Car`, but would not require reprogramming of the use of `Car`, unless the driver was explicitly required.

3 The Approaches

The solutions to method binding, subclasses vs subtypes, *etc* described in the previous section, are not orthogonal: an arbitrary combination would not necessarily give a sound type system. In this section we outline the combinations adopted by C++, Eiffel, multimethods, LOOM, Beta and $Perm_{co}$.

In figure 3 we summarise these approaches in terms of the solution taken. Figure 4 gives type checking results of the different approaches, where $\surd$ means that the expression is type correct, $\times$ means that the expression is type incorrect, ? means that a runtime check is performed, $---$ means that the expression is impossible to express in the language, $\surd_{\texttt{C}}$, $\surd_{\texttt{RC}}$ and $\surd_{\texttt{C,RC}}$ mean that the message is bound to the method defined in class `Car`, `RaceCar` and `Car` *or* `RaceCar` respectively. The types of the variables are given with the figure.

	C++	Effel pol. catcall	$\lambda^{\&}$	LOOM	Beta	$Perm_{co}$
Overriding of Inst. Vars or Parameters	novariant	covariant	covariant, contravariant	covariant	covariant	covariant
Subclasses vs Subtypes	connect1	connect1	connect	unconnect	connect	connect2
Method Binding	single	single	multiple	single	single	single
Result Type Dependence	none	receiver	argument	class parameter	virtual type	receiver or argument
World View	open	closed	open, closed	open	closed	closed
Implication of Covariant Redefinition	impossible	recompile	restricted reprogram	reprogram	restricted reprogram	recompile

Figure 3: Properties of Various Type Systems

3.1 $Perm_{co}$

In $Perm_{co}$ [18, 28] which was originally developed on top of Smalltalk [19, 17], subclasses are connected to subtypes, and because all entities are internally represented as pointers to heap objects, $\mathtt{C_2} \sqsubseteq \mathtt{C_1}$ implies that <$\mathtt{C_2}$> $\leq$ <$\mathtt{C_1}$> and [$\mathtt{C_2}$] $\leq$ <$\mathtt{C_1}$>, but *not* [$\mathtt{C_2}$] $\leq$ [$\mathtt{C_1}$]. Covariant type redefinitions of instance variables and any type redefinitions of method types are allowed. Message expressions are type checked by considering the types of the methods in the receiver *and* all its subtypes. Signatures are used to represent the types of the methods. One or more signatures can be used to represent the type of a method.

The cars example in $Perm_{co}$ is shown in figure 2. Because `<RaceCar>` is a subtype of `<Car>`, the assignment `aFiat = lotus` is legal, and `lotus` may appear wherever a `<Car>` is expected. Similarly, because [`Car`] is a subtype of <`Car`>, `aFiat = twingo` is type correct. On the other hand, <`RaceCar`> is *not* a subtype of `[Car]`, and the assignment `twingo=lotus` is illegal.

The expression `lotus.driver = noddy` is type incorrect and `twingo.driver = john` is type correct - as expected. Furthermore, the expression `lotus.driver = john` is type incorrect; this makes sense, because `john` may point to a `Driver` object at run time. In the same spirit, `aFiat.driver = john` is type incorrect. The type rules of $Perm_{co}$ achieve this by requiring the type of the expression updating an object to be a subtype of the type of the label for all subtypes of the object: in that case they require <`Driver`> to be a subtype of <`Driver`> *and* of <`FastDriver`>.

Similar rules hold for type checking message expressions: consider all possible subtypes of the receiver's type, and all methods defined for them. For example, because `RaceCar` has "exceptional" requirements, for any expression of type `<Car>` we need to take the possibility of `<RaceCar>` into consideration, and to check whether it too, would behave correctly in the context. Hence the expression `aFiat.register(john)` is type incorrect.

Line		C++	Eiffel	$\lambda^{\&}$	LOOM	Beta	$Perm_{co}$
1	`aFiat=lotus`	$\surd$	$\surd$	$\surd$	$\times$	$\surd$	$\surd$
2	`noddy=schumacher`	$\times$	$\times$	$---$	$\times$	$---$	$\times$
3	`twingo.driver = john`	$\surd$	$\surd$	$---$	$\surd$	$---$	$\surd$
4	`aFiat.driver = john`	$\surd$	$\surd$ or $\times$	$---$	$\surd$	?	$\times$
5	`aFiat.driver = schumacher`	$\surd$	$\surd$	$---$	$\times$	$\surd$	$\surd$
6	`lotus.driver = john`	$\times$	$\times$	$---$	$\times$	$\times$	$\times$
7	`lotus.driver = noddy`	$\times$	$\times$	$---$	$\times$	$---$	$\times$
8	`lotus.driver = schumacher`	$\surd$	$\surd$	$---$	$\surd$	$\surd$	$\surd$
9	`twingo.register(john)`	$\surd_{\mathtt{C}}$	$\surd_{\mathtt{C}}$	$---$	$\surd_{\mathtt{C}}$	$---$	$\surd_{\mathtt{C}}$
10	`twingo.register(noddy)`	$\surd_{\mathtt{C}}$	$\surd_{\mathtt{C}}$	$---$	$\surd_{\mathtt{C}}$	$---$	$\surd_{\mathtt{C}}$
11	`aFiat.register(john)`	$\surd_{\mathtt{C}}$	$\surd_{\mathtt{C}}$ or $\times$	$\surd_{\mathtt{C,RC}}$	$\surd_{\mathtt{C}}$	?	$\times$
12	`aFiat.register(noddy)`	$\surd_{\mathtt{C}}$	$\surd_{\mathtt{C}}$ or $\times$	$---$	$\surd_{\mathtt{C}}$	$---$	$\times$
13	`aFiat.register(schumacher)`	$\surd_{\mathtt{C}}$	$\surd_{\mathtt{C}}$ or $\times$	$\surd_{\mathtt{C,RC}}$	$\times$	$\surd_{\mathtt{C,RC}}$	$\surd_{\mathtt{C,RC}}$
14	`lotus.register(noddy)`	$\times$	$\times$	$---$	$\times$	$\times$	$\times$
15	`noddy.testDrive(lotus)` `.speedLimit`	$\times$	$\times$	$\surd$	$\times$	$\times$	$\surd$
16	`schumacher.testDrive(lotus)` `.speedLimit`	$\times$	$\times$	$\surd$	$\times$	$\times$	$\surd$

where the variables have types: `schumacher:<FastDriver>`, `john:<Driver>`, `noddy:[Driver]`, `lotus:<RaceCar>`, `twingo:[Car]`, `aFiat:<Car>`

Figure 4: Type checking cars and drivers expressions in the six approaches

The signature of `testDrive` succinctly expresses that the type of the result is the same as that of the argument. Thus, the type of `noddy.testDrive(lotus)` is <`RaceCar`>, and therefore the expression `noddy.testDrive(lotus).speedLimit` is type correct.

Note that in $Perm_{co}$ replacing an object of a class by an object of a subclass does *not* preserve the types. For example, `twingo.register(john)` is type correct, whereas `(new RaceCar).register(john)` is *not*. Nevertheless, $Perm_{co}$ has the *subtype substitutivity* property, *i.e.* in a type correct expression any sub-expression of type `T` may be replaced by a type correct sub-expression of type `T'`, if `T'` is a subtype of `T`. The above expressions, `twingo.register(john)` and `(new RaceCar).register(john)` do *not* constitute a counterexample, because [`RaceCar`], the type of **new** `RaceCar`, is *not* a subtype of [`Car`], the type of `twingo`. On the other hand, <`FastDriver`> *is* a subtype of <`Driver`>, and the expression `twingo.register(schumacher)` is legal – as required by by the fact that `twingo.register(john)` is legal and the subtype substitutivity property.

3.2 C++

In C++ subclasses are connected to subtypes and component overriding adopts the novariant solution. C++ provides both overriding and static overloading. For example, the method `register` is defined in `Car` with parameter type <`Driver`>, and it is redefined in `RaceCar` with parameter type <`FastDriver`>; this is a case of overloading. While overriding can only

be resolved dynamically, overloading is resolved statically.

For the expressions in figure 4, assume that the variables are declared as: `FastDriver* schumacher`, `Driver* john`, `Driver noddy`, `RaceCar* lotus`, `Car twingo`, `Car* aFiat`, and that the class `Car` has an instance variable `driver` of type `Driver*`, and its subclass `RaceCar` has an instance variable `driver` of type `FastDriver*`. Because pointers and stack objects have different internal representations, the assignment `noddy = schumacher` is type incorrect. The message expression `twingo.register(john)` is statically bound. Also, the expression `aFiat.register(schumacher)` is type correct, because the only method `register` requiring a receiver type of <`Car`> and parameter type <`Driver`> is defined in class `Car` (while the `register` method defined in class `RaceCar` requires different parameter types, and thus overloads and does not override `register` from class `Car`), and at runtime it will be bound to the method `register` from class `Car` even if `aFiat` points to an instance of `RaceCar`.

3.3 Eiffel

Eiffel subclasses are subtypes. Messages are dynamically bound to methods according to the class of the receiver. Instance variable and method redefinitions in subclasses are allowed in a covariant manner. As a result, earlier versions were not sound, [16], and several proposals aim for an upwards compatible, type sound version of Eiffel [15].

A simpler approach in [25] suggests "polymorphic catcalls". Informally, an entity `p` is polymorphic if there is an assignment `p` = `q`, where `q` has a different type than `p`; a call of the form `p f` : ... is a "catcall" if some subclass of the class defining `f` covariantly redefines the parameter types of `f`. Meyer points out that while polymorphic is desirable and catcall is allowed, a polymorphic catcall may cause a runtime error, therefore it is considered a type error.

The method `register` from `RaceCar` covariantly overrides `register` from `Car`. Therefore, if the program contains an assignment like `aFiat = lotus`, then `aFiat.register(john)` and `aFiat.register(schumacher)` would be polymorphic catcalls, and therefore type incorrect. On the other hand, if `aFiat` is not polymorphic, then we know that only the method from `Car` may be called (*i.e.* $\times$ or $\surd_{\mathtt{C}}$, depending on the rest of the program).

The polymorphic catcall solution is less expensive to implement than system-level validity, but it still needs to collect information about variable usage, and some times, *e.g.* for `aFiat.register(schumacher)` it is less permissive than other approaches.

3.4 LOOM

[1, 6, 8, 4] separate inheritance from subtypes and introduce *matching*; in [7] the subtype relation is dropped in favour of matching. A version of matching types enriched with ideas from virtual types is suggested in [22] whereby classes are parameterized in the LOOM style, but can also explicitly refer to each other's type parameters. Figure 5 describes cars and drivers in LOOM.

Roughly, if a class *extends* a superclass, *i.e.* has more components but does not override the types in the superclass, then the object type of the subclass *matches* (represented by $<\!\#$) the object type of the superclass

$$\frac{\tau \ \mathtt{extends} \ \tau'}{\mathtt{C} \vdash \mathtt{ObjectType} \ \tau <\!\# \ \mathtt{ObjectType} \ \tau'}\,\mathtt{ObjectType}(<\!\#)$$

```
class Car(MyDrType<#Driver)
instance variables
   driver:#MyDrType,
instance methods
   getDriver:#MyDrType
      return driver
   register(aDriver:#MyDrType):MyType
      driver := aDriver.
      return self
end Car

class Driver
instance methods
   testDrive(aCar:#Car):#Car
      ... try out aCar ... .
      return aCar
end Driver
```

```
class RaceCar
   inherits  Car(MyDrType<#FastDriverType)
   modifying register
instance variables
   speedLimit:Number
instance methods
   register(aDriver:#MyDrType):MyType
      driver := aDriver
      speedLimit :=
            aDriver.getLicense * 10.
      return self
end RaceCar
```

Figure 5: Cars and Drivers in LOOM

For example, the class `FastDriver` extends `Driver`, therefore the type `FastDriverType` matches the type `DriverType`.

Covariant type redefinitions are supported through `MyType`, or through explicit type parameters to classes. `MyType` represents the type of the receiver. In inherited methods `MyType` changes automatically to the type of the corresponding subclass. Thus, in figure 5, the method `register` returns an object of the same type as the receiver; for a `CarType` receiver the result will be a `CarType`, whereas for a `RaceCarType` receiver the result will be a `RaceCarType`. The "extends" relation considers `MyType` as the same. Type parameters may be declared and used in a class, and restricted in subclasses, for example `MyDrType` in class `Car` is used to describe the type of the instance variable `driver` in the parameterized class `Car`, and is further restricted to match `FastDriver` in class `RaceCar`.

No "matching" relation is defined for such parameterized types, as they are not proper types. Parameterised types may be applied to type arguments, and form proper types. For example, `CarType(FastDriverTYpe)` and `CarType(DriverType)` are proper types, whereas `RaceCarType(DriverType)` is an illegal instantiation. Instantiations of parameterized types are replaced by the type obtained by substituting the formal parameter by the actual type parameter. Thus, `CarType(FastDriverType)` $<\#$`RaceCarType(FastDriverType)` but it does *not* hold that `RaceCarType(FastDriverType)` $<\#$`CarType(DriverType)`, nor does it hold that `CarType(FastDriverType)` $<\#$`CarType(DriverType)`.

For a type τ, the object type $\#\tau$ indicates the values of any type matching τ. Matching is required for type checking calls of inherited methods. In the following rule, if the receiver `o` has type γ, which matches `ObjectType`$\{\mathtt{m} : \tau\}$ (*i.e.* an object type containing a method `m` of type τ), then the message send $\mathtt{o} \Leftarrow \mathtt{m}$ has type $\tau[\gamma/\mathtt{MyType}]$.

$$\frac{\mathtt{C} \vdash \gamma <\# \ \mathtt{ObjectType}\{m : \tau\},\ \mathtt{C}, \mathtt{E} \vdash \mathtt{o} : \gamma}{\mathtt{C}, \mathtt{E} \vdash \mathtt{o} \Leftarrow \mathtt{m} : \tau[\gamma/\mathtt{MyType}]}$$

The types of the LOOM variables in figure 4 are: `schumacher` : #`FastDriverType`, `john` : #`DriverType`, `noddy` : `DriverType`, `aFiat` : #`CarType`(#`DriverType`), `twingo` :

`CarType`(#`DriverType`) and `lotus` : #`RaceCarType`(#`FastDriverType`). The assignment `aFiat = lotus` is illegal, because, as we said earlier, `RaceCarType(FastDriverType)` does *not* match `CarType(DriverType)`. The assignment `aFiat.driver=schumacher` and the expression `aFiat.register(schumacher)` are type incorrect for the same reason. Note that in the examples from figure 4 the methods from class `RaceCar(MyDrType)` are never called, when the receiver is of type #`CarType`(`DriverType`). This should not be a surprise: such a receiver would never at run time be an object of type `RaceCarType(FastDriverType)`. This is so, again because it does *not* hold that `RaceCarType(FastDriverType)` $<\#$ `CarType(DriverType)`

The method `testDrive` in `Driver` accepts any argument that matches `Car`, and returns #`Car`, however the dependence of the result type on the argument type is not expressed. Trying to express that dependence by introducing a type parameter to `Driver` would overshoot the target: namely, one would need to distinguish between `DriverType`(#`CarType`) and `DriverType`(#`RaceCarType`) and it would be impossible to construct one object which would be able to `testDrive` both `RaceCar`s and normal `Car`s.

3.5 Multimethods and $\lambda^{\&}$

Multimethods are supported in CLOS and Rosette [21, 20], and formally studied in the $\lambda^{\&}$-calculus in [13, 11]. In figure 6 we outline the drivers example in the multimethods style of CLOS – the $\lambda^{\&}$ calculus is not imperative, and would look distinctly different.

Dynamically overloaded methods have an *overloaded type*[1] formed by putting together the types of the different branches, *e.g.* $\{\mathtt{U}_1 \to \mathtt{V}_1, \ldots, \mathtt{U}_n \to \mathtt{V}_n\}$. The type of a message expression is evaluated by selecting the best approximating branch from the overloaded type according to the actual parameter types.

An overloaded type $\{\mathtt{U}_1 \to \mathtt{V}_1, \ldots, \mathtt{U}_n \to \mathtt{V}_n\}$ is well formed iff $\mathtt{U}_i \leq \mathtt{U}_j \implies \mathtt{V}_i \leq \mathtt{V}_j$, *i.e.* if the argument type of the first method is a subtype of that of the second method then the return type of the first must be a subtype of that of the second. In fig. 6 the multimethod `register` has type $\{\ \mathtt{Car} \times \mathtt{Driver} \to \mathtt{Car}, \mathtt{RaceCar} \times \mathtt{FastDriver} \to \mathtt{RaceCar}\ \}$. The parameter types of the second branch, $\mathtt{RaceCar} \times \mathtt{FastDriver}$, are subtypes of the parameter types of the first branch, $\mathtt{Car} \times \mathtt{Driver}$, and the return type of the second branch, `RaceCar`, is a subtype of the return type of the first branch, `Car`.

On the other hand, contravariant type redefinitions define the subtype relationship for the parameter types which do not take part in the branch selection. That is, for the parameters which do not take part in dynamic method binding, for two bodies belonging to the same method identifier, if the parameter types of the first are a supertypes of the second, then the return type of the first has to be a subtype of the second.

$$\frac{\mathtt{U}_2 \leq \mathtt{U}_1,\ \mathtt{V}_1 \leq \mathtt{V}_2}{\mathtt{U}_1 \to \mathtt{V}_1 \leq \mathtt{U}_2 \to \mathtt{V}_2} \qquad \frac{\forall \mathtt{i} \in \mathtt{I},\ \exists \mathtt{j} \in \mathtt{J}\ \ \mathtt{U}'_j \to \mathtt{V}'_j \leq \mathtt{U}''_i \to \mathtt{V}''_i}{\{\mathtt{U}'_j \to \mathtt{V}'_j\}_{j \in J} \leq \{\mathtt{U}''_i \to \mathtt{V}''_i\}_{i \in I}}$$

The multi-method approach as in [10] does not support closed types, therefore in figure 4 we do not consider the expressions which mention `twingo` and `noddy`. Furthermore, [10] is a functional calculus (although in [12, 14] imperative versions are suggested), therefore in figure 4 we do not consider assignment expressions either.

[1] Notice that the term "overloading" is used here in a rather unusual sense to describe dynamic method binding, as opposed to the static binding method binding which overloading means for C++.

```
class Car includes
   driver:Driver
   getDriver:Driver is
      return driver
end Car

class RaceCar inherits Car includes
   driver:FastDriver
   speedLimit:Number
   getDriver:FastDriver is
      return driver
end RaceCar

class Driver
   ...
   ...
end Driver
```

```
method register(receiver:Car,
                aDriver:Driver):Car is
   receiver.driver := aDriver.
   return receiver

method register(receiver:RaceCar,
                aDriver:FastDriver):RaceCar is
   receiver.driver := aDriver.
   receiver.speedLimit := aDriver.getLicense*10.
   return receiver

method testDrive(receiver:Driver,
                 aCar:Car):Car is
   ... try out aCar ... .
   return aCar

method testDrive(receiver:Driver,
                 aRaceCar:RaceCar):RaceCar is
   ... try out aRaceCar ...
   return aRaceCar
```

Figure 6: Cars and Drivers in Multimethods

The method call `aFiat.register(john)` will result into calling the branch with types `RaceCar` $\times$ `FastDriver` $\rightarrow$ `RaceCar` if `aFiat` is a `RaceCar` and `john` is a `FastDriver`, otherwise it will call the branch with type `Car` $\times$ `Driver` $\rightarrow$ `Car` (*i.e.* $\surd_{\texttt{C,RC}}$). For the expression `lotus.register(john)` multimethods are the most permissive; the `register` method from `Car` would be bound.

The method `testDrive` and the dependence of the result type on the argument can be expressed through multimethods. Still, there are two disadvantages: Firstly, we use two identical method bodies in order to express the signature of the method. Secondly, we shall need to add as many further method bodies as there will be subclasses of `Car`.

3.6 Beta

Beta [24] allows covariant type redefinition by means of *virtual types*. Virtual types can be understood as symbolic names for types, which may be covariantly overridden in subclasses. Virtual types play a similar role to C++ virtual methods in that they can be redefined in subclasses. For the car example in Figure 7, `MyDrType` is a virtual type which is redefined in the subclass `RaceCar`.

The question whether the assignment `aFiat.driver` = `john` should be permitted is tackled in an "optimistic" way, *i.e.* it is considered type correct, but run-time checks are inserted to ensure that no racing car will be assigned a non licensed driver. Virtual types were incorporated into Java [26]. In [27] a stricter, statically typed version is suggested whereby each runtime check is considered a type error.

Figure 7 outlines the drivers example in Beta. Closed types do not exist in Beta, thus all expressions involving `noddy` or `twingo` are impossible in Beta. Also, `aFiat.driver = john` and `aFiat.register = john` are considered type correct and will be checked at run-time.

The result type of the method `testDrive` does not depend on the argument type, and

```
class Car
typedef MyDrType as Driver
instance variables
   driver:MyDrType
instance methods
   getDriver:MyDrType
      return driver
   register(aDriver:MyDrType):MyType
      driver := aDriver
```

```
class RaceCar inherits Car
typedef MyDrType as FastDriver
instance variables
   speedLimit:Number
instance methods
   register(aDriver:MyDrType):MyType
      driver := aDriver
      speedLimit :=
             aDriver.getLicense * 10.
```

```
class Driver
instance methods
   testDrive(aCar:Car):Car
      ... try out aCar .... .
      return aCar
   register(aDriver:MyDrType):MyType
      driver := aDriver.
```

Figure 7: Cars and Drivers in Beta

therefore `noddy.testDrive(lotus).speedLimit` is a type error. We could have tried to express the dependence by introducing a virtual type into the class `Driver`. However, as for LOOM, this would not solve the problem, since it would not be possible for objects of the same class to `testDrive` ordinary `Car`s and `RaceCar`s.

3.7 Comparison

From the previous discussion and from the table in figure 4, we can see that multi-methods are in some cases the most permissive. However, they require the more expensive multi-method dispatch run-time mechanism. C++ has a sound type system which uses static overloading in lieu of covariant overriding, and thus fails so express the intention that a racing car requires a more special kind of driver than that of a general car. LOOM uses explicit type parameters to allow covariant type redefinitions.

$Perm_{co}$ seems to be nearest to the Beta approach, especially if the [27] idea is followed whereby the run-time checks are replaced by type errors. In that sense, one of the contributions of this paper would be a demonstration of the soundness of their approach. With the reservation, that $Perm_{co}$ does not support explicit names for the virtual types of other classes, and thus is more restricted than Beta.

On the other hand, $Perm_{co}$ allows the dependence of result types on the arguments, and so it can give types to functions to which most of the other approaches could not. Also, by allowing several signatures to one method body, it can express the type of a method like `passenger` in figure 10 in the appendix.

$Perm_{co}$ adopts the closed world view, which is a disadvantage. However, the implications of the introcduction of covariant overriding later in the software lifecycle, is, we believe of comparable importance, especially for prototype development. There, we believe lies an important advantage for $Perm_{co}$.

4 Permissive Component Overriding in terms of $\lambda^{\&,S}$

We use the calculus $\lambda^{\&,S}$ to precisely describe the ideas of $Perm_{co}$, and to demonstrate its soundness.

4.1 $\lambda^{\&,S}$ Syntax

$\lambda^{\&,S}$ is a first order language reflecting only the most essential features of an object oriented language; it is object based and its syntax is similar to that of $\lambda^{\&}$ in [13]. However, $\lambda^{\&,S}$ models single method dispatch, as opposed to multi-method dispatch from [13]. Furthermore, the $\lambda^{\&,S}$ and $\lambda^{\&}$ type systems are different. In $\lambda^{\&}$ covariant redefinition of argument types is required for those arguments whose classes are taken into account for method selection (multimethods, binary methods) and contravariant redefinition of argument types is required for those arguments whose types are not taken into account for method selection. On the other hand, in $\lambda^{\&,S}$ the classes of arguments are *not* taken into account for method selection, and there is *no* restriction to the redefinition of argument types.

Obj	::=	`self`
	|	$\langle$ *Number* $\rangle^{\texttt{Num}}$
	|	$\langle$ *Label=Obj (, Label=Obj)** $\rangle^{Class}$
Expr	::=	*Obj*
	|	*Param*
	|	`FROM` *Obj* `SEL` *Label*
	|	`IN` *Obj* `UPD` *Label* `WITH` *Expr*
	|	*Methods*$\Rightarrow$*Expr*.(*Expr*)*
Method	::=	$\mu\texttt{self}^{Class}(.\lambda\ Param)^*.\ Expr$
Methods	::=	*Method*
	|	+
	|	*Methods* & *Method*
Class, Label, Param	::=	`Identifier`
Number	::=	`1` | `2` | `3` ...

$\lambda^{\&,S}$ is a very basic language. It does not support method parameters, recursion, or the `nil` object. We claim that the language it represents is sufficient for the demonstration of the concept and the soundness, since the omitted features, although important from the programming view point, have usually not exposed difficult typing issues for first order programming languages. $\lambda^{\&,S}$ expressions can be objects, update, select expressions, or message sends.

There exist predefined number objects, *e.g.* $\langle\texttt{4}\rangle^{\texttt{Num}}$. Objects of user defined classes are labelled tuples of further objects, and they carry their class as part of their denotation. For example, $\langle\texttt{driver} = \texttt{john}\rangle^{\texttt{Car}}$ is an object of class `Car`. `self`, the name of the receiver object, and any parameters (*Param*) may appear in method bodies.

The expressions `FROM`...`SEL`... and `IN`...`UPD`...`WITH` are used to select, or to update a label of an object. Notice that the above syntax ensures that label update and selection may only be applied to the receiver self, of a method.

A message send, $\texttt{m}\Rightarrow\texttt{e}_0.\texttt{e}_1...\texttt{e}_n$, consists of one or more method bodies (*i.e.* `m`) being sent to a receiver (*i.e.* $\texttt{e}_0$), and possibly a number of argument expressions (*i.e.* $\texttt{e}_1, ...\texttt{e}_n$). For example, `register`$\Rightarrow$`lotus.schumacher` is a message send. + is a predefined method with the expected behaviour.

$\mu\texttt{self}^{\texttt{C}}.\lambda\texttt{y}_1...\lambda\texttt{y}_\texttt{n}.\texttt{e}$ is a method defined in class `C`, with arguments $\texttt{y}_1, ...\texttt{y}_\texttt{n}$, and method body `e`. We use μ in order to distinguish the receiver from other parameters, because we model single dispatch. Inheritance is described through grouping several methods using the & operator. For example, `getDriver` $= \mu\texttt{self}^{\texttt{Car}}$`.FROM self SEL driver` is a non-overloaded method, while `register` is a dynamically overloaded method, whereby

```
register  =  μself^Car.λaDriver.    IN self UPD driver WITH aDriver
          &
             μself^RaceCar.λaDriver. IN (IN self UPD driver WITH aDriver)
                                     UPD speedLimit WITH ((getLicense⇛aDriver)∗10)
```

4.2 Types and Environment

Expression types, *ExpType*, describe the type of expressions:

ExpType	::=	<TypeError>
	|	*SingleExpType (∨ SingleExpType)**
SingleExpType	::=	<*ClassName*> | [*ClassName*]
ClassName	::=	`identifier`

We call <...> an *open* type, and [...] a *closed* type. For a class `C`, <C> denotes the set of objects belonging to any of the subclasses of `C` (and also to `C`, since we consider the subclass relationship, denoted by $\sqsubseteq$, to be reflexive); whereas [C] denotes only the set of objects that belong to class `C`, but *not* to any of its subclasses. As we discussed already in section 2.1, this distinction between open and closed types can be found in some popular object oriented programming languages.

Union types describe expressions which may evaluate to objects which belong to any of the constituent single types. <TypeError> indicates that at run time the exception `object does not understand message` may be raised.

Type variables may appear in the result type of a signature. The result type of a signature is defined by *ExtExpType*:

ExtExpType	::=	*ExtSingleExpType (∨ ExtSingleExpType)**
ExtSingleExpType	::=	*SingleExpType* | *TypeVariable*
TypeVariable	::=	`Identifier`

Method types are nonempty sets of signatures. Signatures are:

Signature	::=	$\ll$ *Quantifier* Params* $\longrightarrow$ *ExtExpType* $\gg$
Quantifier	::=	*VarIdent* $\leq$ *ExpType* .
Params	::=	*Param (× Param)**
Param	::=	*VarName*
VarIdent, VarName	::=	`Identifier`

In our example, the type of `register` is: $\ll \mathtt{X} \leq [\mathtt{Car}].\mathtt{Y} \leq <\mathtt{Driver}>.\mathtt{X} \times \mathtt{Y} \longrightarrow \mathtt{X} \gg$, $\ll \mathtt{X} \leq <\mathtt{RaceCar}>.\mathtt{Y} \leq <\mathtt{FastDriver}>.\mathtt{X} \times \mathtt{Y} \longrightarrow \mathtt{X} \gg$.

An environment Γ contains the types of variables (*i.e.* $\Gamma_{\mathtt{x}}$), bounds of type variables (*i.e.* $\Gamma_{\mathtt{X}}$), type definitions for the labels of each class (*i.e.* $\Gamma_{\mathtt{C,L}}$), and the subclass hierarchy.

Γ	::=	ϵ	empty
	|	$\Gamma', \mathtt{x} : \mathtt{T}$	x does not appear in Γ' T extended expression type
	|	$\Gamma', \mathtt{X} \leq \mathtt{T}$	X type variable, not appearing in Γ' T expression type
	|	$\Gamma', \mathtt{C} = \langle\langle \mathtt{L}_1 : \mathtt{T}_1, ..., \mathtt{L}_m : \mathtt{T}_m \rangle\rangle$	$\mathtt{T}_i$ expression type, $\mathtt{i} \in \{1, \ldots, \mathtt{m}\}$, C not defined in Γ'
	|	$\Gamma', \mathtt{C}$ INH $\mathtt{C}'$ REDEF $\langle\langle \mathtt{L}_1 : \mathtt{T}_1, ..., \mathtt{L}_m : \mathtt{T}_m \rangle\rangle$ EXT $\langle\langle \mathtt{L}_{m+1} : \mathtt{T}_{m+1}, \ldots, \mathtt{L}_n : \mathtt{T}_n \rangle\rangle$	$\mathtt{C}'$ defined in Γ', C not defined in Γ' $\mathtt{T}_i$ expression type, $\mathtt{i} \in \{1, \ldots, \mathtt{n}\}$ $\mathtt{L}_1, \ldots, \mathtt{L}_m$ labels, appearing in the def. of $\mathtt{C}'$ $\mathtt{L}_{m+1}, \ldots, \mathtt{L}_n$ labels, not appearing in the def. of $\mathtt{C}'$
	|	ResultType = T	T extended object type

For the cars and drivers example the following information would be stored in the environment:
$\Gamma_1 = \mathtt{Car} = \langle\langle \mathtt{driver} : <\mathtt{Driver}> \rangle\rangle$,
RaceCar INH Car REDEF$\langle\langle$driver : <FastDriver>$\rangle\rangle$ EXT $\langle\langle$speedLimit : <Number>$\rangle\rangle$,
Driver $= \langle\langle \ldots \rangle\rangle$,
FastDriver INH Driver REDEF $\langle\langle\rangle\rangle$ EXT $\langle\langle$license : <Num>$\rangle\rangle$.

4.3 Operational Semantics

The term rewrite relationship $\longrightarrow\!\!\triangleright_\Gamma$ describes the evaluation of $\lambda^{\&,S}$ terms:

[Sel] $$\frac{}{\mathtt{FROM}\ \langle ..., \mathtt{lab} = \mathtt{o}, ... \rangle^{\mathtt{C}}\ \mathtt{SEL}\ \mathtt{lab} \longrightarrow\!\!\triangleright_\Gamma \mathtt{o}}$$

[Upd] $$\frac{}{\mathtt{IN}\ \langle .., \mathtt{lab} = \mathtt{o}_1, .. \rangle^{\mathtt{C}}\ \mathtt{UPD}\ \mathtt{lab}\ \mathtt{WITH}\ \mathtt{o}_2 \longrightarrow\!\!\triangleright_\Gamma \langle .., \mathtt{lab} = \mathtt{o}_2, .. \rangle^{\mathtt{C}}}$$

[Con] $$\frac{\mathtt{e}_i \text{ ground for } \mathtt{i} < \mathtt{k} \quad \mathtt{e}_k \longrightarrow\!\!\triangleright_\Gamma \mathtt{e}'_k}{\mathtt{m} \Rightarrow \mathtt{e}_1 ... \mathtt{e}_k ... \mathtt{e}_n \longrightarrow\!\!\triangleright_\Gamma \mathtt{m} \Rightarrow \mathtt{e}_1 ... \mathtt{e}'_k ... \mathtt{e}_n}$$

[MethSel] $$\frac{\langle \ldots \rangle^{\mathtt{C}},\ \mathtt{o}_1, \ldots, \mathtt{o}_n \text{ are ground terms} \quad \mathtt{m}_k = \mu \mathtt{self}^{\mathtt{C}_k} ....,\ \text{for } \mathtt{k} \in 1, ...\mathtt{m} \quad \mathtt{C}_i = \mathcal{M}_{\mathcal{C}}(\mathtt{C}_1, \ldots, \mathtt{C}_m, \mathtt{C}),\ \mathtt{C}_i \neq \perp}{(\mathtt{m}_1 \& ... \& \mathtt{m}_m) \Rightarrow \langle \ldots \rangle^{\mathtt{C}}.\mathtt{o}_1 ... \mathtt{o}_n \longrightarrow\!\!\triangleright_\Gamma \mathtt{m}_i \Rightarrow \langle \ldots \rangle^{\mathtt{C}}.\mathtt{o}_1 ... \mathtt{o}_n}$$

[MethAppl] $$\frac{\mathtt{o}_i \text{ ground for all } \mathtt{i} \leq \mathtt{n}}{\mu \mathtt{self}^{\mathtt{C}}.\lambda \mathtt{x}_1 .... \lambda \mathtt{x}_n . \mathtt{e} \Rightarrow \mathtt{o}_0 ... \mathtt{o}_n \longrightarrow\!\!\triangleright_\Gamma \mathtt{e}[\mathtt{self}/\mathtt{o}_0 ... \mathtt{x}_n/\mathtt{o}_n]}$$

For the following examples, assume that $\texttt{schumacher} = \langle \texttt{license} = 40 \rangle^{\texttt{FastDriver}}$, and $\texttt{lotus} = \langle \texttt{driver} = \texttt{schumacher}, \texttt{speedLimit} = 330 \rangle^{\texttt{RaceCar}}$.

The [Sel]-rule describes the selection of a subcomponent of an object. For example: `FROM lotus SEL driver` $\longrightarrow\!\!\triangleright_\Gamma$ `schumacher`. The [Upd]-rule describes updating a component. For example: `IN schumacher UPD license WITH 100` $\longrightarrow\!\!\triangleright_\Gamma \langle \texttt{license} = 100 \rangle^{\texttt{FastDriver}}$.

When considering a message send, the receiver and then the arguments are evaluated from left to right; cf. the rule [Con].

When the receiver and arguments are ground expressions, then the method whose receiver class is the *minimal superclass* (denoted by $\mathcal{M}_\mathcal{C}(\texttt{C}_1, \ldots, \texttt{C}_\texttt{m}, \texttt{C})$) of the class of the receiver object is selected (cf the rule [MSel]). Thus:

$$\texttt{register} \Rrightarrow \texttt{lotus.schumacher} \longrightarrow\!\!\triangleright_\Gamma \mu\texttt{self}^{\texttt{RaceCar}}.\ldots.\Rrightarrow \texttt{lotus.schumacher}$$

i.e. the `RaceCar` branch of the method `register` is selected, because the *minimal superclass* of the receiver, `RaceCar`, in {`Car`, `RaceCar`} is `RaceCar`.

Once method selection has been resolved, application of a single method causes `self` to be substituted by the receiver object, and all parameters $\texttt{x}_1, \ldots \texttt{x}_\texttt{n}$ to be substituted by the corresponding actual parameters in the method body, cf rule [MethApp]. Term substitution has the usual definition.

For example,

$\texttt{register} \Rrightarrow \langle \ldots \rangle^{\texttt{Car}}.\langle \ldots \rangle^{\texttt{Driver}}$	$\longrightarrow\!\!\triangleright_\Gamma$		[MethSel]
$\mu\texttt{self}^{\texttt{Car}}.\lambda\texttt{aDriver.IN self UPD driver WITH aDriver} \Rrightarrow \langle \ldots \rangle^{\texttt{Car}}.\langle \ldots \rangle^{\texttt{Driver}}$	$\longrightarrow\!\!\triangleright_\Gamma$		[MethAppl]
$\texttt{IN self UPD driver WITH aDriver}[\langle \ldots \rangle^{\texttt{Car}}/\texttt{self}, \langle \ldots \rangle^{\texttt{Driver}}/\texttt{aDriver}]$	=		text.subst
$\texttt{IN} \langle \ldots \rangle^{\texttt{Car}} \texttt{ UPD driver WITH} \langle \ldots \rangle^{\texttt{Driver}}$			

Note that the $\longrightarrow\!\!\triangleright_\Gamma$-relationship is a one to one relation, and therefore, the evaluation of $\lambda^{\&,S}$-terms is deterministic (as is the case in practical programming languages).

4.4 Type Rules

The following rules determine subtypes, where $\sqsubseteq$ stands for subclass and $\leq$ stands for subtype. If Γ=Γ', `C` INH `C'` REDEF... EXT..., Γ'', then $\Gamma \vdash \texttt{C} \sqsubseteq \texttt{C}'$. The subtype relationship is based on the suclass relationship:

[SubCl-1] $\dfrac{\Gamma \vdash \texttt{C}_1 \sqsubseteq \texttt{C}_2}{\Gamma \vdash \texttt{<C}_1\texttt{>} \leq \texttt{<C}_2\texttt{>}}$ [TypErr] $\dfrac{}{\Gamma \vdash \texttt{T} \leq \texttt{<TypeError>}}$

[SubCl-2] $\dfrac{\Gamma \vdash \texttt{C}_1 \sqsubseteq \texttt{C}_2}{\Gamma \vdash [\texttt{C}_1] \leq \texttt{<C}_2\texttt{>}}$ [Trans] $\dfrac{\Gamma \vdash \texttt{T}_1 \leq \texttt{T}_2}{\Gamma, \texttt{X} \leq \texttt{T}_1 \vdash \texttt{X} \leq \texttt{T}_2}$

[UnTyp] $\dfrac{\text{for all } \texttt{i} \in \{1, \ldots, \texttt{n}\} \text{ exists } \texttt{j} \in \{1, \ldots, \texttt{m}\} \text{ with } \Gamma \vdash \texttt{T}_\texttt{i} \leq \texttt{S}_\texttt{j}}{\Gamma \vdash \texttt{T}_1 \vee \ldots \vee \texttt{T}_\texttt{n} \leq \texttt{S}_1 \vee \ldots \vee \texttt{S}_\texttt{m}}$

We can prove that the $\leq$ relationship is a partial order. All rules enjoy the subformula property, no judgement matches the consequence of more than one rule, and the requirements of each rule are simpler than the consequence; therefore, the $\leq$ relationship is deterministically computable.

In our system, covariant redefinition of the types for subclass labels is allowed. This means that the type of a subclass label can be a subtype of that of the corresponding label in the superclass. This requirement is expressed in the following definition:

Definition 1 *An environment* Γ *is* well formed*, iff for any classes* $\mathtt{C}$ *and* $\mathtt{C}'$*,* $\Gamma \vdash \mathtt{C} = \langle\langle \ldots, \mathtt{L} : \mathtt{T}, \ldots\rangle\rangle$*,* $\Gamma \vdash \mathtt{C}' = \langle\langle \ldots, \mathtt{L} : \mathtt{T}', \ldots\rangle\rangle$*,* $\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}'$ *implies that* $\Gamma \vdash \mathtt{T} \leq \mathtt{T}'$*.*

From now, we implicitly expect environments to be well formed.

Figure 8 describes the types of $\lambda^{\&,S}$ expressions. The first three rules are rather straightforward.

The fourth rule, [Update] deals with component updating when covariant redefinition of instance variables is allowed in subclasses. It allows updating label L in an object o by expression $\mathtt{e}'$, provided that $\mathtt{T}'$, the type of $\mathtt{e}'$, is a subtype of the type of the label L in all subclasses of the class of o. This rule makes the expression aFiat.driver := john type incorrect, because for [RaceCar], a subtype of <Car>, the type of the label driver is <FastDriver>, which is not a supertype of john's type.

[Var] $\dfrac{}{\Gamma \vdash \mathtt{x} : \Gamma_{\mathtt{x}}}$

[Object] $\dfrac{\begin{array}{l}\mathtt{L}_1, ..., \mathtt{L}_n \text{ are all labels of C} \\ \Gamma \vdash \mathtt{e}_i : \mathtt{T}_i \\ \Gamma \vdash \mathtt{T}_i \leq \Gamma_{\mathtt{C},\mathtt{L}_i}\end{array}}{\Gamma \vdash \langle \mathtt{L}_1 = \mathtt{e}_1, ...\mathtt{L}_n = \mathtt{e}_n\rangle^{\mathtt{C}} : [\mathtt{C}]}$

[Select] $\dfrac{\Gamma \vdash \mathtt{o} : <\mathtt{C}> \text{ or } \Gamma \vdash \mathtt{o} : [\mathtt{C}]}{\Gamma \vdash \mathtt{FROM\ o\ SEL\ L} : \Gamma_{\mathtt{C},\mathtt{L}}}$

[Update] $\dfrac{\begin{array}{l}\Gamma \vdash \mathtt{e}' : \mathtt{T}' \\ \Gamma \vdash \mathtt{o} : \mathtt{T} \\ \Gamma \vdash [\mathtt{C}] \leq \mathtt{T} \text{ implies } \Gamma \vdash \mathtt{T}' \leq \Gamma_{\mathtt{C},\mathtt{L}}\end{array}}{\Gamma \vdash \mathtt{IN\ o\ UPD\ L\ WITH\ e}' : \mathtt{T}}$

[Abstr] $\dfrac{\begin{array}{l}\Gamma \vdash \mathtt{T}_0 \leq <\mathtt{C}> \\ \Gamma, \mathtt{Y}_0 \leq \mathtt{T}_0, .., \mathtt{Y}_n \leq \mathtt{T}_n, \mathtt{self} : \mathtt{Y}_0, \mathtt{y}_1 : \mathtt{Y}_1, .., \mathtt{y}_n : \mathtt{Y}_n \vdash \mathtt{exp} : \mathtt{R}\end{array}}{\Gamma \vdash \mu\mathtt{self}^{\mathtt{C}}.\lambda\mathtt{y}_1 ... \lambda\mathtt{y}_n.\mathtt{exp} :: \ll \mathtt{Y}_0 \leq \mathtt{T}_0 . \ldots \mathtt{Y}_n \leq \mathtt{T}_n . \mathtt{Y}_0 \times \ldots \times \mathtt{Y}_n \longrightarrow \mathtt{R} \gg}$

[Over] $\dfrac{\begin{array}{l}\Gamma \vdash \mu\mathtt{self}^{\mathtt{C}_i}.\lambda\mathtt{y}_1 ... \lambda\mathtt{y}_n.\mathtt{exp}_i :: \mathtt{sig}_{i,j}\ \mathtt{j} \in \{1, \ldots, \mathtt{m}_i\} \\ \mathtt{S}_i = \{\mathtt{sig}_{i,1}, ...\mathtt{sig}_{i,\mathtt{m}_i}\} \text{ complete signature set for } \mathtt{C}_i,\ \mathtt{i} \in \{1, \ldots, \mathtt{n}\}\end{array}}{\Gamma \vdash \mu\mathtt{self}^{\mathtt{C}_1}.\lambda\mathtt{y}_1 ... \lambda\mathtt{y}_n.\mathtt{exp}_1 \& \ldots \& \mu\mathtt{self}^{\mathtt{C}_n}.\lambda\mathtt{y}_1 ... \lambda\mathtt{y}_n.\mathtt{exp}_n : \vee_{\mathtt{C}_1,\ldots,\mathtt{C}_n}\mathtt{S}_i}$

[Union] $\dfrac{\begin{array}{l}\Gamma \vdash \mathtt{e}_i : \mathtt{T}_{i,1} \vee \ldots \vee \mathtt{T}_{i,\mathtt{n}_i}\ \mathtt{i} \in \{0, \ldots, \mathtt{n}\} \\ \Gamma, \mathtt{y}_0 : \mathtt{T}_{0,\mathtt{j}_0}, \ldots, \mathtt{y}_n : \mathtt{T}_{n,\mathtt{j}_n} \vdash \mathtt{m}\Rightarrow\mathtt{y}_0 . \ldots . \mathtt{y}_n : \mathtt{T}_{\mathtt{j}_0,\ldots,\mathtt{j}_n}\end{array}}{\Gamma \vdash \mathtt{m}\Rightarrow\mathtt{e}_0.\mathtt{e}_1 \ldots \mathtt{e}_n : \vee_{\mathtt{j}_i \in \{1,\ldots,\mathtt{n}_i\}}\mathtt{T}_{\mathtt{j}_0,\ldots,\mathtt{j}_n}}$

[MessSend] $\dfrac{\begin{array}{l}\mathtt{T}_i \text{ single type } \mathtt{i} \in \{0, \ldots, \mathtt{n}\} \\ \Gamma \vdash \mathtt{e}_i : \mathtt{T}_i\ \mathtt{i} \in \{0, \ldots, \mathtt{n}\} \\ \Gamma \vdash \mathtt{m} : \mathtt{S}\end{array}}{\Gamma \vdash \mathtt{m}\Rightarrow\mathtt{e}_0 \ldots \mathtt{e}_n : \mathcal{A}(\mathtt{S}, \mathtt{T}_0, \ldots, \mathtt{T}_n)}$

Figure 8: Type Rules for $\lambda^{\&,S}$

The remaining rules describe types of methods and message sends. They make use of the terms *complete signature set*, $\vee_{\mathtt{C}_1,\ldots,\mathtt{C}_n}\mathtt{S}_i$, and the function $\mathcal{A}(\mathtt{S}, \mathtt{T}_0, \ldots, \mathtt{T}_n)$, which are defined in [28]; here we give an outline.

Rules [Abstr] and [Over] describe the type of methods. A signature of a *single* method body, $\mu\mathtt{self}^C ... \mathtt{exp}$, given in class C, expressed by $\Gamma \vdash \mu\mathtt{self}^{\mathtt{C}} ... \mathtt{exp} :: \mathtt{sig}$, is calculated in

the usual manner, [Abstr]. A single method body may have more than one signatures.[2] These signatures are collected into a *complete signature set* for the class where the method body appears. For example, for classes C1, C2, C3 with C1 $\sqsubseteq$ C2 $\sqsubseteq$ C3, and a method defined in class C2 the signature set {X $\leq$ [C2].....} is not complete, whereaS the set {X $\leq$ <C2>....} is. The type of a dynamically overloaded method with method bodies in classes C_1, ... C_n is the *consistent union* $\vee_{C_1,...,C_n} S_i$, which is the union of all signatures from S_i, except for signatures whose receiver is a subtype of the receiver of another method body's receiver class. In Appendix 6 we further clarify the concepts of complete signature sets and consistent union.

The type of message expressions ([MessSend]) is $\mathcal{A}(S, T_0, \ldots, T_n)$, which, given a set of signatures S and types $T_0, \ldots, T_n$, is determined by:

• $\mathcal{R}(S, T_0, \ldots, T_n)$, the *relevant signature set*, are those signatures of S whose i-th element is in the subtype relationship with T_i. For example $\ll$ X $\leq$ <Car>.Y $\leq$ <FastDriver>.X $\times$ Y $\longrightarrow$ X $\gg$ and $\ll$ X $\leq$ <RaceCar>.Y $\leq$ <Driver>.X $\times$ Y $\longrightarrow$ X $\gg$ are relevant signatures for <RaceCar> and <Driver>.

• $\mathcal{M}(S, T_0, \ldots, T_n)$, the *minimal signature set*, contains those signatures from $\mathcal{R}(S, T_0, \ldots, T_n)$ which *match* $T_0, \ldots, T_n$ *best*. For example $\ll$ X $\leq$ <RaceCar>.Y $\leq$ <FastDriver>.X $\times$ Y $\longrightarrow$ X $\gg$ matches <RaceCar>, <FastDriver> better than $\ll$ X $\leq$ <Car>.Y $\leq$ <Driver>.X $\times$ Y $\longrightarrow$ X $\gg$.

• If $\mathcal{M}(S, T_0, \ldots, T_n)$ does not *cover* S and $T_0, \ldots, T_n$, then $\mathcal{A}(S, T_0, \ldots, T_n)$ = <TypeError>. A signature set does not *cover* types $T_0, \ldots, T_n$ iff there exists a T'_0, a subtype of T_0, and a signature in the set whose receiver type is T'_0, and none of the signatures in the set has receiver T'_o, and all argument types are supertypes of $T_1, \ldots, T_n$. For example, the set {$\ll$ X $\leq$ <Car>.Y $\leq$ <Driver>.X $\times$ Y $\longrightarrow$ X $\gg$, $\ll$ X $\leq$ <RaceCar>.Y $\leq$ <FastDriver>.X $\times$ Y $\longrightarrow$ X $\gg$} does not cover <Car>, <Driver>, because for <RaceCar> the argument type <FastDriver> is not a supertype of <Driver>. Intuitively this expresses that if the receiver expression reduces to a value of type <RaceCar>, the argument may still be <Driver>, which is not covered by the argument type of the signature.

• If $\mathcal{M}(S, T_0, \ldots, T_n)$ covers S and $T_0, \ldots, T_n$, then $\mathcal{A}(S, T_0, \ldots, T_n)$ returns the *application* of the signatures in $\mathcal{M}(S, T_0, \ldots, T_n)$ to the types $T_0, \ldots, T_n$. For example {$\ll$ X $\leq$ <RaceCar>.Y $\leq$ <Driver>.X $\times$ Y $\longrightarrow$ Y $\gg$} covers [RaceCar], [FastDriver].
Therefore, $\mathcal{A}(S, [\text{RaceCar}], [\text{FastDriver}])$ is [FastDriver].

4.4.1 Concepts required for the type rules

The concepts required for the type rules are mostly straightforward. Ideas similar to applicable signatures, minimal signatures, and covering signature sets can be found in widely spread programming languages, such aS C++ or Java.

Less common are the notions of complete signature set and consistent union. The requirement for these two concepts stems from the fact that in $Perm_{co}$ we allow more than one signature for one method body, and that signatures may have a receiver which is a sub-

[2] For example, the method body m1 defined below has signatures $\ll$ X $\leq$ <A>.Y $\leq$ <B>.X $\times$ Y $\longrightarrow$ <Num> $\gg$ and $\ll$ X $\leq$ <A>.Y $\leq$ <C>.X $\times$ Y $\longrightarrow$ <Boolean> $\gg$.

```
class A
  methods
    m1(y)
      return y.m2
```

```
class B
  methods
    m2
      return 15
```

```
class C
  methods
    m2
      return true
```

class of, and not the same as the class containing the method body. These possibilities make $Perm_{co}$ more flexible, and allows more expressions to be type correct.

This extra flexibility was important to us, because our original aim was to allow as many expressions to be type correct as possible, especially as we developed $Perm_{co}$ with Smalltalk programs in mind, which have been constructed without type checkers and thus tend to have rather peculiar types – if any.

On the other hand, we could have chosen to allow less flexibility and thus obtain a simpler type system. Good programming style probably would suggest that this extra flexibility is not that desirable: it makes sense to require only one signature per body, and that this signature should have the class of the definition as its receiver. We believe, that under these restrictions the straightforwardly calculated signature of one body would form a complete signature set, and the union of signatures of overloaded method bodies would be consistent. Thus, these two concepts would not be required, the [Union] type rule could be simpliified to the set theoretic union, and the definition of covering signature set would also be simplified.

Thus, we believe that it is possible to design a slightly stricter and simpler version of $Perm_{co}$. With the increased interest in virtual types for Java [27], this avenue of research is becoming more appealing.

4.5 Soundness

We demonstrate the soundness of our approach by proving the following *subject reduction theorem*. Note that the theorem states that any well formed non ground expression will be rewritten, and thus it excludes the possibility of the runtime error `method not understood`. It also excludes `null access` exceptions, because we have not modelled `nil`.

Theorem

If $\mathtt{e}$ is a closed, non-ground expression, $\mathtt{T}$ is an object type, $\Gamma \vdash \mathtt{e} : \mathtt{T}$, $\mathtt{T} \neq <\mathtt{TypeError}>$, then there exists an expression $\mathtt{e}'$ and a type $\mathtt{T}'$, such that $\mathtt{e} \longrightarrow\!\!\triangleright_{\Gamma} \mathtt{e}'$, $\Gamma \vdash \mathtt{e}' : \mathtt{T}'$ and $\Gamma \vdash \mathtt{T}' \leq \mathtt{T}$.

Proof The full proof can be found in [28]. The following support lemmas are used, for a set of signatures $\mathtt{S}$, and types $\mathtt{T_0}, \ldots, \mathtt{T_n}$, $\mathtt{T'_0}, \ldots, \mathtt{T'_n}$:

- $\Gamma \vdash \mathtt{T'_i} \leq \mathtt{T_i},\ \mathtt{i} \in \{\mathtt{0}, \ldots, \mathtt{n}\} \implies \mathcal{R}(\mathtt{S}, \mathtt{T'_0}, \ldots, \mathtt{T'_n}) \subseteq \mathcal{R}(\mathtt{S}, \mathtt{T_0}, \ldots, \mathtt{T_n})$
- $\Gamma \vdash \mathtt{T'_i} \leq \mathtt{T_i},\ \mathtt{i} \in \{\mathtt{0}, \ldots, \mathtt{n}\}$:
  $\mathcal{M}(\mathtt{S}, \mathtt{T_0}, \ldots, \mathtt{T_n})$ covers $\mathtt{T_0}, \ldots, \mathtt{T_n}$ and $\mathtt{S} \implies \mathcal{M}(\mathtt{S}, \mathtt{T'_0}, \ldots, \mathtt{T'_n}) \subseteq \mathcal{M}(\mathtt{S}, \mathtt{T_0}, \ldots, \mathtt{T_n})$
- $\Gamma \vdash \mathtt{T'_i} \leq \mathtt{T_i}$, $\mathtt{i} \in \{\mathtt{0}, \ldots, \mathtt{n}\}$:
  $\mathcal{M}(\mathtt{S}, \mathtt{T_0}, \ldots, \mathtt{T_n})$ covers $\mathtt{T_0}, \ldots, \mathtt{T_n}$ and S $\implies \mathcal{M}(\mathtt{S}, \mathtt{T'_0}, \ldots, \mathtt{T'_n})$ covers $\mathtt{T'_0}, \ldots, \mathtt{T'_n}$ and S
- $\Gamma \vdash \mathtt{T'_i} \leq \mathtt{T_i}$, $\mathtt{i} \in \{\mathtt{0}, \ldots, \mathtt{n}\}$, and if signature `sig` is relevant to $\mathtt{T_0}, \ldots, \mathtt{T_n}$:
  $\Gamma \vdash \mathtt{sig} \circ \mathtt{T'_0}, \ldots, \mathtt{T'_n} \leq \mathtt{sig} \circ \mathtt{T_0}, \ldots, \mathtt{T_n}$
- For closed types $\mathtt{T_0}, \ldots, \mathtt{T_n}$, $\mathtt{T_0} = [\mathtt{C}]$, an overloaded method `meth`, where
  $\mathtt{meth} = \mu\mathtt{self}^{\mathtt{C_1}}.\mathtt{e_1}\&\ldots\&\mu\mathtt{self}^{\mathtt{C_k}}.\mathtt{e_k}$, $\Gamma \vdash \mu\mathtt{self}^{\mathtt{Cj}}.\mathtt{e_j} : \mathtt{S_j},\ \mathtt{j} \in \{\mathtt{1}, \ldots, \mathtt{n}\}$, $\Gamma \vdash \mathtt{method} : \mathtt{S}$, $\mathcal{M}_{\mathcal{C}}(\mathtt{C_1}, \ldots, \mathtt{C_n}, \mathtt{C}) = \mathtt{C_i}$. If $\mathcal{M}(\mathtt{S}, \mathtt{T_0}, \ldots, \mathtt{T_n})$ covers $\mathtt{T_0}, \ldots, \mathtt{T_n}$ and $\mathtt{S}$, then:
  $\mathcal{M}(\mathtt{S}, \mathtt{T_0}, \ldots, \mathtt{T_n}) = \mathcal{M}(\mathtt{S_i}, \mathtt{T_0}, \ldots, \mathtt{T_n})$ and $\mathcal{M}(\mathtt{S_i}, \mathtt{T_0}, \ldots, \mathtt{T_n})$ covers $\mathtt{T_0}, \ldots, \mathtt{T_n}$ and $\mathtt{S}$
- $\Gamma,\ \mathtt{x} : \mathtt{X},\ \mathtt{X} \leq \Gamma_{\mathtt{X}} \vdash \mathtt{exp} : \mathtt{R}$, $\Gamma \vdash \mathtt{exp}' : \mathtt{T}$, $\Gamma \vdash \mathtt{T} \leq \Gamma_{\mathtt{X}}$, then there exists a type $\mathtt{T}'$ so that:
  $\Gamma \vdash \mathtt{exp}[\mathtt{o}'/\mathtt{x}] : \mathtt{T}'$ and $\Gamma \vdash \mathcal{B}(\mathtt{T}') \leq \mathcal{B}(\mathtt{R}[\mathtt{T}/\mathtt{X}])$

Using these results, the theorem can be proven by structural induction on the deduction $\Gamma \vdash \mathtt{e} : \mathtt{T}$.

5 Conclusions

Our approach is permissive in that it aims to allow the programmer as much label/method overriding as possible, and in that it aims to consider type correct as many programs as possible. It does this without application of data flow analysis. Thus it should be suitable for type checking legacy software, and should provide an appropriate alternative to the current suggestions for type checking Eiffel [15, 25].

However, restrictions on separate compilation need to be imposed, *i.e.* subclasses defined in separately compiled modules and redefining the types of inherited methods or labels in a non-contravariant manner would invalidate the types of previously type checked modules importing the superclass.

On the other hand, when non-contravariant redefinitions are introduced fairly late in the software development cycle, our permissive approach only requires *recompilation* of importing modules, whereas other approaches require *reprogramming* of importing modules. The possiblility to express dependence on the type of an argument gives $Perm_{co}$ additional expressive power.

Further work includes developing parameterized types for the system. Also, one could consider incorporating the ideas from [22, 27] whereby the virtual types or type arguments may be referred to from outside their class.

Finally, we would like to develop a slightly less powerful, but simplified version of $Perm_{co}$ along the lines described in section 4.4.1.

6 Acknowledgements

We are grateful to Giuseppe Castagna and the anonymous FOOL3 referees for useful comments on a previous version of this paper. Stuart Kent and Peter Burton made many constructive suggestions for the clarification of the issues discussed.

References

[1] Martin Abadi and Luca Cardelli. On subtyping and matching. *ACM Transactions on Programming Languages and Systems*, 18(4):401–423, 1996.

[2] Francois Bancilhon and Claude Delobel. *Implementing an Object-Oriented Database System: The story of O2.* Morgan Kafmann, 1992.

[3] Kim Bruce. Typing in Object Oriented Languages: Achieving Expressibility and Safety. Technical report, Williams College, 1996.

[4] Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On Binary Methods. *Theory and Practice of Object-oriented Systems*, pages 221–242, 1995.

[5] Kim Bruce, Jon Crabtree, Thomas Murtagh, Robert van Gent, Allyn Dimock, and Robert Muller. Safe and decidable type checking in an object oriented language. In *Proceedings of OOPSLA*, pages 29–46, 1993.

[6] Kim Bruce, Leaf Petersen, and Adrian Fiech. Subtyping is not a Good Match for Object-Oriented Languages. In *ECOOP'97*. Springer, June 1997.

[7] Kim Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A Type-safe Polymorphic Object Oriented Language. In *Proceedings of ECOOP 95*, pages 27–51, 1995.

[8] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2-3):138–164, February 1988.

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

[10] Giuseppe Castagna. A meta-language for typed object-oriented languages. *Theoretical Computer Science*, 151(2):297–352, November 1995.

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

[12] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A Calculus for Overloaded Functions with Subtyping. *Information and Computation*, 117(2):115–135, February 1995.

[13] Guiseppe Castagna and John Boyland. Parasitic methods: an implementation of multi-methods in Java. In *OOPSLA*, 1997.

[14] William R. Cook. A Proposal for Making Eiffel Type-safe. *The Computer Journal*, 32(4):305–311693, 1989.

[15] William R. Cook and Jens Palsberg. A Denotational Semantics of Inheritance and its Correctness. In *Proceedings of OOPSLA*, pages 433–443, 1989.

[16] Sophia Drossopoulou and Stephan Karathanos. Static typing for Dynamic Binding. In *BCS-FACS Special Christmas Meeting on Object Orientation*, 1993.

[17] Sophia Drossopoulou and Dan Yang. Permissive types, July 1996. FOOL 3, New Brunswick, New Jersey.

[18] Sophia Drossopoulou, Dan Yang, and Stephan Karathanos. Static Types for Smalltalk. In Steven Goldsack and Stuart Kent, editors, *Formal Methods and Object Technology*, pages 262–286. Springer Verlag, 1996.

[19] Carnot Research Group. Rosette reference manual. at http://www.mcc.com/projects/carnot/resette/.

[20] S.K. Keene. *Object-Oriented Programming in COMMON LISP: A Programming Guide to CLOS*. Addison Wesley, 1989.

[21] Philip Wadler Kim Bruce, Martin Odersky. A Statically Safe Alternative to Virtual Types. In *FOOL5, San Diego*, 1998.

[22] Ole L. Madsen. Open Issues in Object Oriented Programming - a Scadinavian Perspective. *Software Practice and Experience*, 25, December 1995.

[23] Ole Lehrman Madsen, Birger Moeller-Pedersen, and Kristen Nygaard. *Object Oriented Programming in the Beta Programming Language*. Object-Oriented Programming. Addison Wesley, 1993.

[24] Bertrand Meyer. Static Typing and Other Mysteries of Life. In *Keynote Lecture, OOPSLA 95*, 1995.

[25] Kresten K. Thorup. Genericity in Java with Virtual Types. In *ECOOP*, 1997.

[26] Mads Torgersen. Virtual Types are Statically Safe. In *FOOL5, San Diego*, 1998.

[27] Dan Yang. *Type Checking Smalltalk*. PhD thesis, Imperial College of Science, Technology and Medicine, University of London, 1997. Available at: http://www.doc.ic.ac.uk/ yd/thesis.

Appendix Notes on *Complete Signature Set* and *Consistent Union*

In section 4.4 we used the concepts of *complete signature set* and *consistent union* for the type of message expressions, without a complete definition. In this appendix we give the definitions and motivation for these concepts. More can be found in [28].

Appendix.1 Complete Signature Set

For a method defined in class C, individual signatures are inferred according to the [Abstr]-rule. The *type* of a method is a set of such signatures, which has to be complete. Such a set is *complete* iff for any subclass of C, there exists at least one signature in the set applicable to that class. Any incomplete set of signatures can be made complete by adding the signature $\ll$ X $\leq$ <C>.... $\longrightarrow$ <TypeError> $\gg$.

The example in figure 9 demonstrates the need for the requirement of complete signature sets. The issue is the proper type for the method passenger in class Van. From [Abstr]

```
class Car
instance methods
   passenger
      ' ≪ X ≤ <Car>.X ⟶ <Num> ≫'
      return 4
end Car

class Van inherits Car
instance methods
   passenger
      ' ≪ X ≤ <Van>.X ⟶ <TypeError> ≫,
      ≪ X ≤ <SmallVan>.X ⟶ <Num> ≫'
      return self.extraSeat
end Van

class SmallVan inherits Van
instance methods
   extraSeat
      ' ≪ X ≤ <SmallVan>.X ⟶ <Num> ≫'
      return 2
end SmallVan
```

Figure 9: Example of incomplete signature sets

we get the signature: $\ll$ X $\leq$ <SmallVan>.X $\longrightarrow$ <Num> $\gg$. However, the set consisting of this signature alone is *not* complete, because the signature is not applicable to type <Van>. The complete signature set is {$\ll$ X $\leq$ <Van>.X $\longrightarrow$ <TypeError> $\gg$, $\ll$ X $\leq$ <SmallVan>.X $\longrightarrow$ <Num> $\gg$}.

If we allowed the incomplete signature set {$\ll$ X $\leq$ <SmallVan>.X $\longrightarrow$ <Num> $\gg$} to be the type of passenger in Van, and combined it with the signature $\ll$ X $\leq$ [Car].X $\longrightarrow$ <Num> $\gg$ of passenger in Car, then the overloaded method passenger = μself$^{\text{Car}}$ ... & μself$^{\text{Van}}$ ... would have type {$\ll$ X $\leq$ [Car].X $\longrightarrow$ <Num> $\gg$, $\ll$ X $\leq$ <SmallVan>.X $\longrightarrow$ <Num> $\gg$}, and the type system would consider aVan.passenger to have type <Num> although execution of the expression (**new** Van).passenger creates a runtime exception.

Appendix.2 Consistent Union

When we put together the signatures of different branches of a method defined in classes C_1, ... C_n, we use the *consistent union* $\vee_{C_1,\ldots,C_n} S_i$. A signature of a method body defined in class

```
class  Car
instance methods
   called
      ‘ ≪ X ≤ <Car>.X ⟶ <TypeError> ≫,
      ≪ X ≤ <RentVan>.X ⟶ <Num> ≫’
      return self.name
end  Car

class  Van inherits  Car
instance methods
   called
      ‘ ≪ X ≤ <Van>.X ⟶ <TypeError> ≫,
      ≪ X ≤ <SmallVan>.X ⟶ <Char> ≫’
      return self.band
end Van

class SmallVan inherits Van
instance methods
   band
      ‘ ≪ X ≤ <SmallVan>.X ⟶ <Char> ≫’
      return ’f’
end  SmallVan

class RentVan inherits SmallVan
instance methods
   name
      ‘ ≪ X ≤ <RentVan>.X ⟶ <Num> ≫’
      return 4
end  SmallVan
```

Figure 10: Example of inconsistent signature sets

$\mathtt{C_i}$, with a receiver type `T` is not included into the consistent union, if `T` is a subtype of $\mathtt{C_j}$, ie of one of the classes where another body of the method appears.

The example in figure 10 demonstrates how inconsistent signature unions may cause problems. The method `called` has a definition in class `Car`, and another definition in class `Van`, *i.e.* $\mathtt{called} = \mu\mathtt{self}^{\mathtt{Car}} \ldots \& \mu\mathtt{self}^{\mathtt{Van}} \ldots$. The first body, *i.e.* $\mu\mathtt{self}^{\mathtt{Car}} \ldots$, has type $\{\ll \mathtt{X} \leq \mathtt{<Car>.X} \longrightarrow \mathtt{<TypeError>} \gg, \ll \mathtt{X} \leq \mathtt{<RentVan>.X} \longrightarrow \mathtt{<Num>} \gg\}$, which is complete. The second body, *i.e.* $\mu\mathtt{self}^{\mathtt{Car}} \ldots$, has type $\{\ll \mathtt{X} \leq \mathtt{<Van>.X} \longrightarrow \mathtt{<TypeError>} \gg, \ll \mathtt{X} \leq \mathtt{<SmallVan>.X} \longrightarrow \mathtt{<Char>} \gg\}$, which is also a complete signature set. The set $\{\ll \mathtt{X} \leq \mathtt{<Car>.X} \longrightarrow \mathtt{<TypeError>} \gg, \ll \mathtt{X} \leq \mathtt{<RentVan>.X} \longrightarrow \mathtt{<Num>} \gg, \ll \mathtt{X} \leq \mathtt{<SmallVan>.X} \longrightarrow \mathtt{<Char>} \gg\}$ is the "straightforward union" of these types; the consistent union is $\{\ll \mathtt{X} \leq \mathtt{<Car>.X} \longrightarrow \mathtt{<TypeError>} \gg, \ll \mathtt{X} \leq \mathtt{<SmallVan>.X} \longrightarrow \mathtt{<Char>} \gg\}$, which is a subset of the straightforward union. The second signature of `called` in class `Van`, $\ll \mathtt{X} \leq \mathtt{<RentVan>.X} \longrightarrow \mathtt{<Num>} \gg$ has the receiver type `<RentVan>`, which corresponds to a subclass of the class `Van` where the method `called` is redefined; therefore this signature is not included in the consistent union.

If, instead, we considered the type of `called` to be the straightforward union of the signature sets, then the expression (**new** `RentVan`).`called` would have type `<Num>`. However, evaluation of this expression will return `'f'` which has type [`Char`].