A Modular Type System for First-class Composition Inheritance

Marko van Dooren
Wouter Joosen

Report CW 534, January 2009

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

A Modular Type System for First-class Composition Inheritance

Marko van Dooren
Wouter Joosen

Report CW 534, January 2009

Department of Computer Science, K.U.Leuven

Abstract

First-class composition inheritance is a significant improvement over other inheritance techniques with respect to reuse. It allows a class to be used as a component to build other classes, enabling a new kind of reuse. Frequently used characteristics and collaborations, such as bounded values and associations, can be encapsulated in classes and reused conveniently to build other classes.

In this paper, we present the O_2 calculus, which supports first-class composition inheritance. It models multiple subclassing inheritance, first-class composition inheritance, renaming and merging, direct and indirect inheritance, subobject references, and component parameters. Our approach differs from existing calculi in that O_2 is parametric in its inheritance mechanism. We define a signature for the inheritance mechanism that captures its required functions and axioms. The type soundness proof must not be reverified if the inheritance mechanism is modified or replaced, as long as it implements the inheritance signature.

Keywords : inheritance, modularity, type system.
CR Subject Classification : D.2.3, D.2.13, D.3.1, D.3.3

A Modular Type System for First-class Composition Inheritance

Marko van Dooren Wouter Joosen

January 28, 2009

Abstract

First-class composition inheritance is a significant improvement over other inheritance techniques with respect to reuse. It allows a class to be used as a component to build other classes, enabling a new kind of reuse. Frequently used characteristics and collaborations, such as bounded values and associations, can be encapsulated in classes and reused conveniently to build other classes.

In this paper, we present the O_2 calculus, which supports first-class composition inheritance. It models multiple subclassing inheritance, first-class composition inheritance, renaming and merging, direct and indirect inheritance, subobject references, and component parameters. Our approach differs from existing calculi in that O_2 is parametric in its inheritance mechanism. We define a signature for the inheritance mechanism that captures its required functions and axioms. The type soundness proof must not be reverified if the inheritance mechanism is modified or replaced, as long as it implements the inheritance signature.

1 Introduction

A class often consists of application specific functionality written on top of boilerplate code that is written over and over again. Examples of such boilerplate code are associations, constrained values, and infrastructure for managing event listeners. In an object-oriented programming language, one would expect that such concepts can be encapsulated in classes, and reused to build other classes.

First-class composition inheritance [47] is the first mechanism to make this kind of reuse practical. It is essentially a code inheritance relation – called a *component relation* – that is tailored for composition of classes. The crucial difference with other inheritance relations is the first-class nature of the component relation. A component relation can be given a name, which is used for a number of purposes. First, it can be used to access functionality that is not inherited by the reusing class. Second, it can be used to access the part of an object corresponding to a component as a real object. Third, it can be used to create high-level connections between components. A component relation is a class member just like a method or field, so it can be overridden, renamed, and merged. In addition, member names can be parameterized to exploit name patterns, drastically reducing the amount of work needed for renaming.

The main contribution of this paper is the formalization of first-class component relation and the corresponding multiple subclassing inheritance relation. The O_2 calculus models overriding, renaming, and merging of class members including component relations, direct and indirect inheritance, subobject references to treat subcomponents as separate objects, and component parameters for connecting components. The formal semantics of first-class composition inheritance are important for a

number of parties. First, developers of compilers need them for creating vtables and other run-time support mechanisms. The lack of formal semantics can lead to different compilers implementing different semantics, as is the case with C++ [48]. Second, language designers need them to improve the mechanism or port it to another language, or even another programming paradigm. After all, the principle of first-class composition inheritance is based on composition of abstract data types, not just classes. To port the construct to the functional or logic programming paradigms, its precise semantics are important because it must be adapted to fit in with existing language constructs and practices.

The second contribution is the creation of an object-oriented calculus that is parametric in its inheritance mechanism. Adding a new expression and the accompanying evaluation rule to existing object-oriented calculi is relatively easy. The work mostly consists of adding a case to the proofs for preservation and progress, and some of the auxiliary lemmas, as they usually involve induction on the structure of the expression. Changing the inheritance mechanism, however, requires a thorough verification of the entire type soundness proof. Type soundness proofs of existing calculi of object-oriented languages are full of implicit assumptions about the inheritance mechanism. In O_2, we make all assumptions about the inheritance mechanism explicit by defining a signature for the inheritance mechanism. By only using the functions and axioms in this signature, the type soundness proof must not be reverified if the inheritance mechanism is changed or replaced. As long as the signature is implemented correctly, the type soundness proof remains valid. This is an important result since it makes it easier to formalize a new inheritance mechanism, which is a crucial language construct for an object-oriented programming language.

Overview

First, we briefly explain first-class composition inheritance in Section 2. We present the base of the O_2 calculus in Section 3, and the concrete inheritance module for first-class composition inheritance in Section 4. We discuss related work in Section 7, and conclude in Section 8.

2 First-class Composition Inheritance

We start with a short introduction to first-class composition inheritance. We do not give detailed arguments about the advantages of the approach and the underlying motivation as these are not the focus of this paper; these are presented in the paper that introduces first-class composition inheritance [47]. As such, we use a simpler but less compelling example to save space.

The first-class composition relation is essentially a code inheritance relation that can be used to build a class using other classes as components. It will be called a *component relation* in the rest of the paper. The relation can be used to compose all kinds of abstract data types, but we focus specifically on classes. As such, a component is just a class that contains general purpose code. Typical widely used components are associations, bounded values, management of event listeners, etc. The goal is to allow as much code reuse as possible with as little effort as possible.

The left part of Figure 1 shows how the component relation is used to create a class of radios. The dotted lines represent the component relations. A radio has a volume and a frequency, which are both values that must be kept within certain bounds. Instead of implementing both characteristics from scratch, we encapsulate the concept of a bounded value in class `BoundedValue`, and reuse it in `Radio`. The Figure 1 also shows how a programmer thinks of the `Radio` class. Conceptually, it is a class “containing” two separate copies of the `BoundedValue` component. It is as if the methods of `BoundedValue` were each implemented twice in class `Radio`: once for the volume, and once

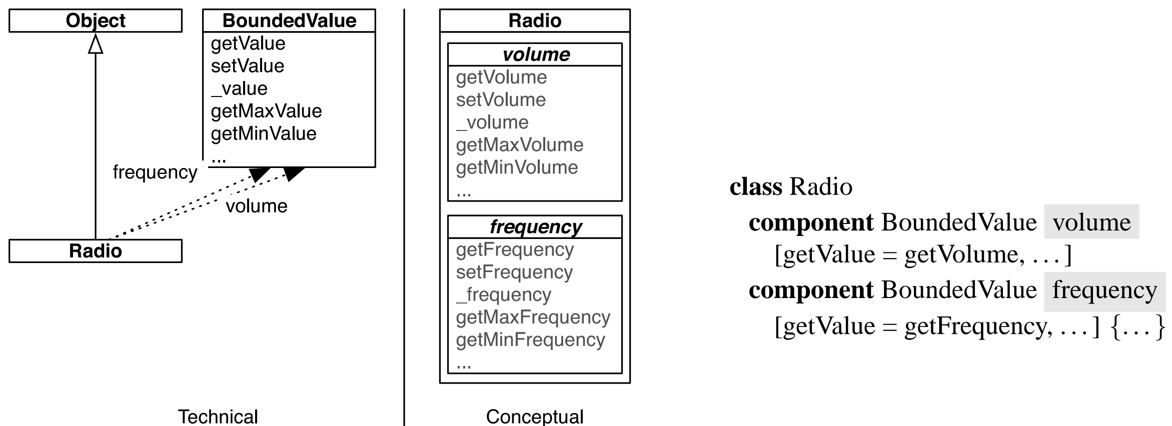


Figure 1: Building a radio with components.

for the frequency. The right part of Figure 1 shows the corresponding component relations of class `Radio`.

Members inherited through a component relation are treated as new members by default, since components are typically used to insert code and usually do not overlap with other components. Therefore, it is a conflict if the same member is inherited via multiple component relations, since they are treated as different members with the same name. The programmer must rename or merge them to solve the conflict.

Renaming parameters [47] provide a very simple macro system to exploit the patterns in the names, which makes renaming much easier. Although they save a lot of work, we do not model them in O_2 because they do not influence the program behavior.

A component relation can have a name, which makes it a class member just like a method or a field. The name allows clients to access indirectly inherited members (Section 2.1), to use subcomponents as if they were separate objects (Section 2.2), and to connect components to each other (Section 2.3). In the example, the components are named `volume` and `frequency`

The subclassing relation of the inheritance mechanism is similar to that of SmartEiffel [10]. It is a multiple inheritance relation that supports renaming, overriding, and merging of class members, but forbids repeated inheritance. The rule-of-dominance is used to simplify conflict resolution as in SmartEiffel and C++ [43].

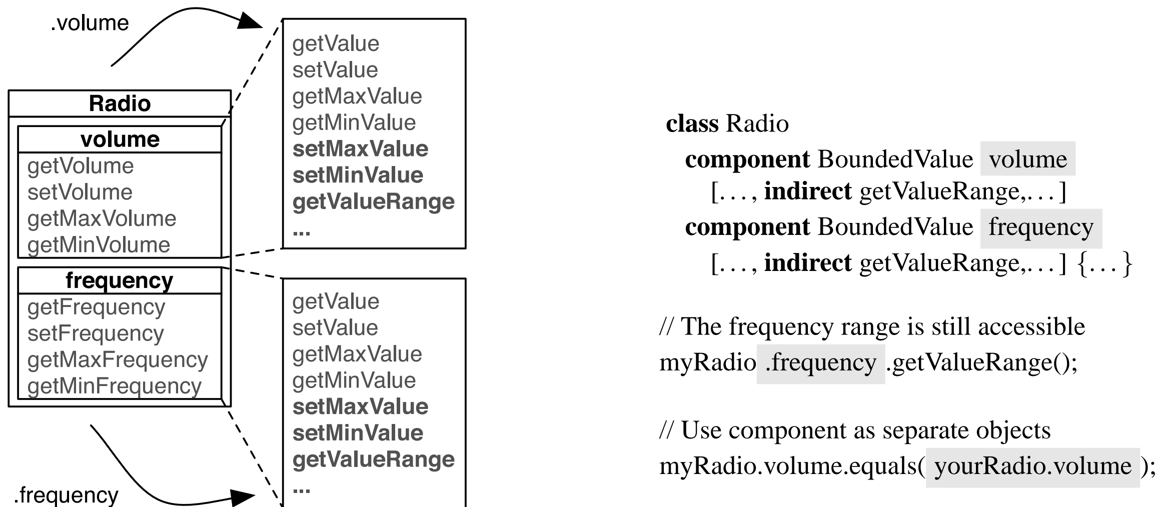


Figure 2: Direct and indirect inheritance.

2.1 Direct and Indirect Inheritance

To prevent interface bloat and unwanted conflicts in the inheriting class, we make a distinction between *direct* and *indirect* inheritance. A directly inherited member is part of the interface of the inheriting class, while an indirectly inherited member is not, and thus cannot cause bloat or conflicts. An indirectly inherited member, however, *can still be accessed* by using the name of the component relation. This is illustrated in Figure 2. The frequency range is not accessible directly in `Radio`, but can be obtained by invoking `myRadio.frequency.getValueRange()`. Of course invoking `radio.volume.getValue()` is the same as invoking `radio.getVolume()`, even if the method is overridden in `Radio`.

Determining how features are inherited is done using `indirect` and `direct` clauses in the configuration block of a component relation, as shown in Figure 2. Members can be placed in groups to directly or indirectly inherit entire groups with a single declaration. Members in the `default` group are inherited directly by default. Member groups have not been modeled in the O_2 calculus.

2.2 Subobject References

With indirect inheritance, a component is used *as if* it were a separate object. To allow even more reuse, we also allow it to be actually used as a separate object that can be passed around. The name of the component acts as a reference to the corresponding subobject, similar to casts in C++.

For example, to use the `equals` method of `BoundedValue` to check if both the volume value and limits of one radio are equal to that of another, we must pass one of the bounded values as an object to the `equals` method of `BoundedValue`. This is done by invoking `myRadio.volume.equals(yourRadio.volume)`.

2.3 Connecting Components

Sometimes, classes must collaborate to do their job. To reuse such a collaboration, we encapsulate the different roles in classes, use them as components in the participating classes, and connect them to each other. We illustrate component connections using a simple collaboration to connect an external

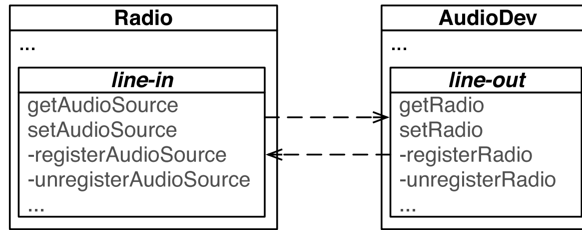


Figure 3: Connecting components.

```
class Association<FROM,TO>
  (otherSide TO → Association<TO,FROM>) {
  FROM getOuter() {...}
  void setObject(TO other) {
    ...
    registerObject(other);
    other @otherSide .registerObject(getOuter());
    ...
  }

class Radio
  component Association<Radio,AudioDev> line-in (line-out)
  ...
class AudioDev
  component Association<AudioDev,Radio> line-out (line-in)
  ...
```

```
void setObject(AudioDev other) {
  ...
  registerObject(other);
  other .line-out .registerObject(getOuter());
  ...
}
```

```
void setObject(Radio other) {
  ...
  registerObject(other);
  other .line-in .registerObject(getOuter());
  ...
}
```

Figure 4: Declaring and using component parameters.

audio device (`AudioDev`) to our `Radio` with a bidirectional association, which we encapsulate in class `Association`.

Since the components collaborate to keep the association in a consistent state, they depend on the names of each other's methods. For example, the `setAudioSource` method of `Radio` must invoke the `registerRadio` method of `AudioDev` to set up the association in the other direction. But since there are quite a few of such method dependencies, it becomes problematic to resolve them all individually. You must provide an abstract method in `Association` for each dependency, and override it in `Radio` and `AudioDev` to delegate the call to the appropriate method of the other class. Note that we cannot use subobject references because we need class-level connections.

To solve this problem, we use *formal component parameters* which are functions $p\ U \rightarrow U'$ that map an object of type U to one of its subobjects of type U'. We then use parameter p to select the appropriate subobject to invoke a method on which the class depends. An actual component parameter is the name of either a component relation or a formal component parameter. If it is a component relation, if must be a component relation of class U – possibly inherited from a superclass – that inherits from U' or one of its subclasses. If it is a formal component parameter, the left-hand side of the constraint must be contravariant, and the right-hand side must be covariant.

Figure 4 shows the corresponding code for the `Association`, `Radio`, and `AudioDev` classes. Class `Association` has two generic parameters representing the types at each end of the association. It uses a formal component parameter `otherSide` to connect to another `Association` component which points in the opposite direction – as indicated by the reversed order of the generic parameters.

To connect the `line-in` component of the `Radio` to the `line-out` component of the `AudioSource`, their names are passed to each other as actual component parameters, as shown

in the bottom part of Figure 4.

To invoke a method on the connected component, we start from an object of type `TO` and apply `@otherSide` to select the connected component. Then, we use the result as a target – of type `Association<TO,FROM>` to invoke the method. This is illustrated in the example by the call to `registerObject`. In this paper, we assume that the outer object at this side of the association is obtained using `getOuter`.

The boxed methods in Figure 4 show how the method works at run-time in `Radio` and `AudioDev`. The component parameter is substituted by the actual component parameter passed to the `Association` component.

3 The Base O_2 Calculus

In this section, we present the base of the O_2 calculus, which models the component relation presented in Section 2 and its associated subclassing relation. The calculus models multiple inheritance for the subclassing relation, repeated inheritance for the first-class component relation, renaming, overriding and merging of members (including component relations), direct and indirect inheritance, subobject references, and component parameters. It does not model renaming parameters and member groups. It also does not model super calls, as we want to focus on the novel features and how members are inherited. On top of that, the different constructs to support super calls, such as `super` calls [9, 44], `inner` calls [20, 29], and method combination rules [11] need knowledge of the concrete inheritance mechanism, which we want to parameterize.

The O_2 calculus is based on Featherweight Java [24] and ClassicJava [18]. Since the focus of this paper is the inheritance mechanism, the expressions are taken from Featherweight Java. Adding state and assignments as in ClassicJava allows modeling of more realistic programs, but adds too much complexity for the purpose of this paper.

To make the calculus more manageable, we simplify it some more. First, we do not model parametric polymorphism, as it would only make the calculus more complicated without providing a benefit in the context of this paper. Second, all component relations must have a name. Third, for component relations, only members that are renamed are inherited directly. Fourth, all fields must be inherited directly to simplify the definition of the constructor. Fifth, merging members is done by giving them the same name, and overriding them instead of selecting one of the members. Finally, a class cannot directly rename or override members of components that are nested more than one level deep.

3.1 Modularity

To allow the inheritance mechanism of O_2 to be replaced by another, e.g. for traits or mixins, we defined an signature that must be implemented by a concrete inheritance mechanism. The signature defines the signatures of the main lookup functions, and their properties in the form of axioms. The base calculus can only rely on these signatures and axioms to perform lookups.

This does not work the other way around, though. The inheritance module depends on some of the definitions presented in this section. It is possible to also put these definitions in appropriate modules to encapsulate them, but that falls outside the scope of this paper. The calculus is constructed such the inheritance mechanism can be changed without requiring reverification of the soundness proof, as long as the signatures of the inheritance interface remain unchanged, and its axioms hold.

$x ::=$ variable name		$body ::= \{\ ctr\ \overline{def}\ \}$	class body
$p ::=$ component parameter name		$ctr ::= C(\overline{X\ x})\{\overline{\mathbf{this}.f = x;}\}$	constructor
$z, c, f, m ::=$ simple member name		$def ::= path\ val$	member definition
$C, T, U, V ::=$ type name $\mid \mathbf{Object}$		$val ::= T;$	field
$P ::= \langle \overline{cls}, t \rangle$	program	$T(\overline{X\ x})\{t;\}$	method
$cls ::= \mathbf{class}\ T\ (\overline{\alpha})\ \overline{sub}\ body$	class	$\mathbf{component}\ T\ (\overline{\delta})\ conf$	component relation
$\alpha ::= p\ T \rightarrow T'$	formal comp. param.	$t ::= s \mid e$	term
$sub ::= \mathbf{subclass}\ T\ (\overline{\delta})\ conf$	subclass relation	$s ::= var \mid \mathbf{new}\ T(\overline{s}) \mid s.f \mid s.m(\overline{s}) \mid (T)s$	
		$\mid s * c$	subobject reference
$\delta ::= p \mid c$	actual comp. param.	$\mid s\ @\ p$	component param
$conf ::= [\overline{z = z'}]$	renaming clauses	$var ::= x \mid \mathbf{this}$	variable
$path ::= z \mid path\ \diamond\ path'$	member name	$\Gamma ::= \emptyset \mid \Gamma, \overline{var\ :\ T}$	typing environment

Figure 5: Syntax for O_2.

References to elements of the inheritance signature will be marked by a gray background in this section.

3.2 Syntax

Figure 5 shows the syntax of O_2. Formal component parameters (α) have a name and a constraint expressing the type containing the component (T) and the type of the component (T'). Subclass relations can specify a number of actual component parameters which are passed to the super class and a configuration block for renaming. An actual component parameter (δ) is either the name of a formal component parameter which is passed on, or the name of a concrete component relation. The configuration block contains a list of assignments for renaming class members; the left-hand side is the old name and the right-hand side is the new name. The name of a class member ($path$) is either a simple name (z) or a composite name ($path \diamond path'$).

The constructor initializes all fields, as in Featherweight Java. A member definition consists of its name, which can only be a simple name in the surface syntax, plus a value, which is either a field, a method, or a component relation. A component relation is syntactically similar to a subclassing relation. Because of the way def is defined, the name comes before the keyword $\mathbf{component}$ in the calculus.

A term is a surface expression (s) or an elaborated expression (e). Elaborated expressions incorporate static type information and are defined in Section 3.4. Aside from the Featherweight Java expressions, there are two additional expressions: subobject references $\underline{e : T} * c$ and component parameter access $e@\underline{T : p}$, which were discussed in Sections 2.2 and 2.3.

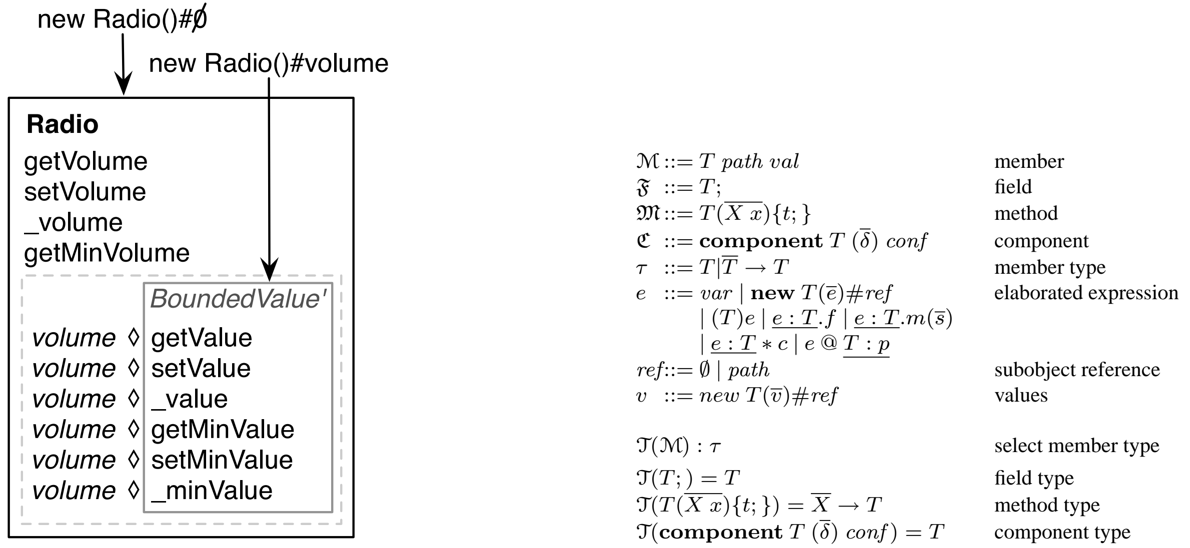


Figure 6: Representation of components, members, and elaborated expressions.

3.3 Component and Member Representation

Directly inherited members are present in the interface of that class. For example, in Figure 6, method `getValue` from `BoundedValue` is inherited as `getVolume`. The field `_value` is inherited as `_volume`.

To represent the component itself, *all* members of the component are added to the inheriting class using the name of the component relation as prefix, essentially flattening the component. To prevent name conflicts, a $\diamond$ symbol is inserted between the member name and the prefix. Since all members, including component relations, must have different names, no conflicts with other components can occur. For example, `getValue` is indirectly inherited by `Radio` as `volume◇getValue`. The lookup mechanism ensures that the lookups of a directly inherited member and the corresponding indirectly inherited member always always return the same result. Therefore, `getVolume` and `volume◇getValue`, and `_volume` and `volume◇_value` will always be the same. As a result, the `volume` component has type `BoundedValue'`, an anonymous subclass of BoundedValue, since changes such as more specific return types in `Radio` show up in the `volume` component. Prefixing is applied recursively to support deeper nested component, for example $a \diamond b \diamond c$.

The right part of Figure 6 shows the internal representation of members and member definitions. A member is represented by the name of its enclosing class, its own name, and its value. The parent type is required because of the support for renaming. The shortcuts for the member values of component relations, fields, and methods are meant to save space in some type rules. The member type selection function is used to hide the syntax of the component relations outside of the inheritance module. The typing and evaluation rules only need its type.

The values are based on those of Featherweight Java: $new\ T(\overline{v})$, but We add a subobject reference: $new\ T(\overline{v})\#ref$. The subobject reference is $\emptyset$ if it references the actual object, and the name of a component relation – which may be a composite name – if it references a subobject. For example, a `Radio` object and its `volume` are represented as `new Radio()#∅` and `new Radio()#volume`.

$\boxed{\vdash_p P \Rightarrow P'}$ P elaborates to P'

[EL_PROG]
$$\frac{1.\ \langle \overline{cls}, s\rangle \vdash_c \overline{cls} \Rightarrow \overline{cls'} \quad 2.\ \langle \overline{cls}, s\rangle, \emptyset \vdash_e s \Rightarrow e : T}{\vdash_p \langle \overline{cls}, s\rangle \Rightarrow \langle \overline{cls'}, e\rangle}$$

$\boxed{P, C \vdash_m def \Rightarrow def'}$

[EL_M]
$$\frac{1.\ P, \Gamma, \overline{x : T} \vdash_s s \Rightarrow e : T}{P, C \vdash_m m\ T\ (\overline{X\ x})\ \{s;\} \Rightarrow m\ T\ (\overline{X\ x})\ \{e;\}}$$

[EL_F]
$$\frac{}{P, C \vdash_m f\ \mathfrak{F} \Rightarrow f\ \mathfrak{F}}$$

[EL_C]
$$\frac{}{P, C \vdash_m c\ \mathfrak{C} \Rightarrow c\ \mathfrak{C}}$$

$\boxed{P, \Gamma \vdash_e s \Rightarrow e : T}$ s elaborates to e with type T in P,Γ

[EL_CALL]
$$\frac{1.\ P, \Gamma \vdash_e s \Rightarrow e : T' \quad 2.\ \mathfrak{L}_P(T', m, T') = T'\ m\ \mathfrak{M} \quad 3.\ \mathfrak{T}(\mathfrak{M}) = \overline{X} \to T \quad 4.\ P, \Gamma \vdash_s \overline{s' \Rightarrow e' : X}}{P, \Gamma \vdash_e s\,.\,m\,(\overline{s'}) \Rightarrow \underline{e : T'}\,.\,m\,(\overline{e'}) : T}$$

[EL_CREF]
$$\frac{1.\ P, \Gamma \vdash_e s \Rightarrow e : T' \quad 2.\ \mathfrak{L}_P(T', c, T') = T'\ c\ \mathfrak{C} \quad 3.\ \mathfrak{T}(\mathfrak{C}) = T}{P, \Gamma \vdash_e s \diamond c \Rightarrow \underline{e : T'} \diamond c : T}$$

[EL_NEW]
$$\frac{1.\ \textbf{class}\ T\ \overline{sub}\ \{\ T\ (\overline{X\ x})\{\overline{\textbf{this.f} = x}\}\ \overline{def}\ \} \in P \quad 2.\ P, \Gamma \vdash_s \overline{s \Rightarrow e : X}}{P, \Gamma \vdash_e \textbf{new}\ T\,(\overline{s}) \Rightarrow \textbf{new}\ T\,(\overline{e})\,\#\,\emptyset : T}$$

$\boxed{P \vdash_c cls \Rightarrow cls'}$

[EL_CLASS]
$$\frac{1.\ cls = \textbf{class}\ T\ (\overline{\alpha})\ \overline{sub}\ \{\ ctr\ \overline{def}\ \} \quad 2.\ cls' = \textbf{class}\ T\ (\overline{\alpha})\ \overline{sub}\ \{\ ctr\ \overline{def'}\ \} \quad 3.\ P, T \vdash_m \overline{def} \Rightarrow \overline{def'}}{P \vdash_c cls \Rightarrow cls'}$$

$\boxed{P, \Gamma \vdash_s s \Rightarrow e : T}$ s elaborates to e subsuming T

[EL_SUB]
$$\frac{1.\ P, \Gamma \vdash_e s \Rightarrow e : T' \quad 2.\ P \vdash T' <: T}{P, \Gamma \vdash_s s \Rightarrow e : T}$$

[EL_CPAR]
$$\frac{1.\ \Gamma \vdash \texttt{this} : C \quad 2.\ \textbf{class}\ C\ (\overline{p\ U \to U'})\ \overline{sub}\ body \in P \quad 3.\ P, \Gamma \vdash_s s \Rightarrow e : U'_k}{P, \Gamma \vdash_e s\,@\,p_k \Rightarrow e@\underline{C : p_k} : U'_k}$$

[EL_READ]
$$\frac{1.\ P, \Gamma \vdash_e s \Rightarrow e : T' \quad 2.\ \mathfrak{L}_P(T', f, T') = T'\ f\ \mathfrak{F} \quad 3.\ \mathfrak{T}(\mathfrak{F}) = T}{P, \Gamma \vdash_e s\,.\,f \Rightarrow \underline{e : T'}\,.\,f : T}$$

Figure 7: Type elaboration.

3.4 Type Elaboration

Because ClassicJava supports syntactic overloading [30] for fields, it uses type elaboration to insert static type information into the program before executing it. We need the same technique because we allow renaming of class members, which means that the we need both the static and dynamic type of the object to which a message was sent.

The type elaboration function inserts static type information into surface expressions. Elaboration starts at the level of a program, and traverses down the lexical structure to finally elaborate the method bodies. Of course, the program expression is elaborated too. The top-down traversal from the program to the method bodies is trivial. Note that $\vdash_m$ adds the formal parameters and the $this$ variable to the typing environment for method bodies. The $\mathfrak{L}_P$ function performs static member lookup and is defined in the inheritance module. Rules EL_READ, EL_CALL, and EL_CREF insert the static type of the receiver into the expression. Rule EL_CPAR inserts the type of the enclosing class; in this case the static type of the receiver can be derived from the types in the constraint of the formal component parameter p_k of class C. Rule EL_NEW only accepts constructor calls for a classes that have no formal component parameters; classes with formal component parameters are considered to be abstract. The rule appends $\emptyset$ to the object to translate the call to the internal representation of the calculus.

$\boxed{P, \Gamma \vdash e : T}$ e has type T in P, Γ

$$\text{(6.1)}\quad \frac{\begin{array}{l} 1.\ \textbf{class}\ T\ (\overline{\alpha_i}^{\,i})\ \overline{sub}\ \{\ ctr\ \overline{def_k}^{\,k}\ \} \in P \\ 2.\ \neg T\ abstract\ in\ P \\ 3.\ ctr\ =\ T\ (\overline{X_j x_j}^{\,j\in 1..n})\ \{\overline{\textbf{this}.f_j = x_j;}^{\,j\in 1..n}\} \\ 4.\ \forall i \in 1..n : P, \Gamma \vdash_s e_i : X_i \\ 5.\ mem_p\ (T) = \overline{\mathcal{M}_j}^{\,j} \\ 6.\ U\ path\ \textbf{component}\ T'\ (\bar{\delta})\ conf \in \overline{\mathcal{M}_j}^{\,j} \end{array}}{P, \Gamma \vdash \textbf{new}\ T\ (\overline{e_i}^{\,i\in 1..n})\ \#\ path : T'}$$

$$\text{(6.2)}\quad \frac{\begin{array}{l} 1.\ \textbf{class}\ T\ (\overline{\alpha_i}^{\,i})\ \overline{sub}\ \{\ ctr\ \overline{def_k}^{\,k}\ \} \in P \\ 2.\ ctr\ =\ T\ (\overline{X_j x_j}^{\,j\in 1..n})\ \{\overline{\textbf{this}.f_j = x_j;}^{\,j\in 1..n}\} \\ 3.\ \neg T\ abstract\ in\ P \\ 4.\ \forall i \in 1..n : P, \Gamma \vdash_s e_i : X_i \end{array}}{P, \Gamma \vdash \textbf{new}\ T\ (\overline{e_i}^{\,i\in 1..n})\ \#\ \emptyset : T}$$

$$\text{(6.3)}\quad \frac{\begin{array}{l} 1.\ P, \Gamma \vdash e : C \\ 2.\ P \vdash C <: T' \\ 3.\ \mathcal{L}_P(C, f, T') = C\ f'\ T\ ; \end{array}}{P, \Gamma \vdash \underline{e : T'}\ .\ f : T}$$

$$\text{(6.4)}\quad \frac{\begin{array}{l} 1.\ P, \Gamma \vdash e : C \\ 2.\ P \vdash C <: T' \\ 3.\ \mathcal{L}_P(C, m, T') = C\ m'\ T\ (\overline{X_i x_i}^{\,i})\ \{t\ ;\} \\ 4.\ \forall i \in 1..n : P, \Gamma \vdash_s e'_i : X_i \end{array}}{P, \Gamma \vdash \underline{e : T'}\ .\ m\ (\overline{e'_j}^{\,j\in 1..n}) : T}$$

$$\text{(6.5)}\quad \frac{\begin{array}{l} 1.\ P, \Gamma \vdash e : C \\ 2.\ P \vdash C <: T' \\ 3.\ \mathcal{L}_P(C, c, T') = C\ c'\ \textbf{component}\ T\ (\bar{\delta})\ conf \end{array}}{P, \Gamma \vdash \underline{e : T'} \diamond c : T}$$

$$\text{(6.6)}\quad \frac{\begin{array}{l} 1.\ \textbf{class}\ C\ (\overline{p\ U_i \rightarrow U'_i})\ \overline{sub}\ body \in P \\ 2.\ P, \Gamma \vdash_s e : U_i \end{array}}{P, \Gamma \vdash e@\underline{C : p_i} : U'_i}$$

$$\text{(6.7)}\quad \frac{1.\ \Gamma \vdash x : T}{P, \Gamma \vdash x : T}$$

$$\text{(6.8)}\quad \frac{}{P, \Gamma \vdash (T)\ e : T}$$

Figure 8: Typing rules for elaborated expressions.

$\boxed{P, \Gamma \vdash_s e : T}$ e subsumes T in P, Γ

$$\text{(7.1)}\quad \frac{\begin{array}{l} 1.\ P, \Gamma \vdash e : U \\ 2.\ P \vdash U <: T \end{array}}{P, \Gamma \vdash_s e : T}$$

Figure 9: Subsumption for elaborated expressions.

3.5 Evaluation Rules

The evaluation rules of O_2 are similar to those of Featherweight Java and are shown in Figure 10. To keep the rules readable, we assume that the class table is stored in a global variable P and the program consists of the class table $\overline{cls}$ and the left-hand side expression e of the evaluation $e \hookrightarrow e'$. Because the type soundness theorems deal with elaborated programs, we need new typing ($P, \Gamma \vdash e : T$) and subsumption ($P, \Gamma \vdash_s e : T$) relations for elaborated expressions, since function $\vdash_e$ in Figure 7 types and elaborates only non-elaborated expressions. These functions are shown in Figures 8 and 9. Remember from Section 3.3 that values have the form $v ::= new\ T(\overline{v})\#ref$.

The subobject reference tracks the (sub)object that is currently in use. The rules for accessing fields and invoking methods are similar to those of Featherweight Java. They are only modified to use the dynamic lookup function $\mathscr{L}_P$ of the inheritance module. The rule for casts is the same as that of Featherweight Java, except for some details of the notation. The rule for evaluating subobject reference expressions is new. It changes the subobject reference into the name ($path$) of the component relation of *actual* type C that corresponds to the component relation with name c in static type T. If we would use $ref \diamond c$ as the new subobject reference, we would have to incorporate the static type T into the subobject reference as well, unnecessarily complicating the values of the calculus.

Note that there is no evaluation rule for $e@\underline{T : p}$. A well-formedness rule forbids occurrences of $e@\underline{T : p}$ in the program expression, and well-formedness must be maintained during the executing. It is the job of the inheritance mechanism to properly substitute such expressions.

$\boxed{e \hookrightarrow e'}$ e evaluates to e' in a single step with $\overline{cls}$ the global class table and $P = \langle \overline{cls}, e \rangle$

[X_READ]
$$\frac{\begin{array}{l}1.\ \textbf{class}\ C\ (\overline{\alpha})\ \overline{sub}\ \{\ C\ (\overline{X\ x})\{\overline{\textbf{this.f} = x}\}\ \overline{def}\ \} \in P \\ 2.\ \mathscr{L}_P(C, ref, f, T) = C\, f'_j\ \mathfrak{F}\end{array}}{\textbf{new}\ C\ (\overline{v})\ \#\ ref : T\ .\ f \hookrightarrow v_j}$$

[C_READ]
$$\frac{1.\ e \hookrightarrow e'}{e : T\ .\ f \hookrightarrow e' : T\ .\ f}$$

[X_CALL]
$$\frac{\begin{array}{l}1.\ \mathscr{L}_P(C, ref, m, T) = C\ path\ V\ (\overline{X\ x})\ \{e_3;\} \\ 2.\ e_4 = [\textbf{new}\ C\ (\overline{v})\ \#\ ref/\textbf{this}\ ,\ \overline{v'/x}]e_3\end{array}}{\textbf{new}\ C\ (\overline{v})\ \#\ ref : T\ .\ m\ (\overline{v'}) \hookrightarrow e_4}$$

[C_CALL]
$$\frac{1.\ e \hookrightarrow e'}{e : T\ .\ m\ (\overline{e''}) \hookrightarrow e' : T\ .\ m\ (\overline{e''})}$$

[X_CAST]
$$\frac{1.\ P, \emptyset \vdash_s \textbf{new}\ C\ (\overline{v})\ \#\ ref\ :\ T}{(T)\ \textbf{new}\ C\ (\overline{v})\ \#\ ref \hookrightarrow \textbf{new}\ C\ (\overline{v})\ \#\ ref}$$

[C_CAST]
$$\frac{1.\ e_1 \hookrightarrow e_2}{(T)\ e_1 \hookrightarrow (T)\ e_2}$$

[X_CREF]
$$\frac{1.\ \mathscr{L}_P(C, ref, c, T) = C\ path\ \mathfrak{C}}{\textbf{new}\ C\ (\overline{v})\ \#\ ref : T \diamond c \hookrightarrow \textbf{new}\ C\ (\overline{v})\ \#\ path}$$

[C_CREF]
$$\frac{1.\ e_1 \hookrightarrow e_2}{e_1 : T \diamond c \hookrightarrow e_2 : T \diamond c}$$

[C_ARG1]
$$\frac{1.\ e_1 \hookrightarrow e'_1}{v : T\ .\ m\ (\overline{v'_i}^{\,i}\ ,\ \overline{e_j}^{\,j \in 1..n}) \hookrightarrow v : T . m(\overline{v'_i}^{\,i}, e'_1, \overline{e_j}^{\,j \in 2..n})}$$

[C_ARG2]
$$\frac{1.\ e'_1 \hookrightarrow e''_1}{\textbf{new}\ C\ (\overline{v'_i}^{\,i}\ ,\ \overline{e'_j}^{\,j \in 1..n})\ \#\ ref \hookrightarrow \textbf{new}\, C(\overline{v'_i}^{\,i}, e''_1, \overline{e'_j}^{\,j \in 2..n})\#ref}$$

Figure 10: The evaluation relation.

3.6 Well-formedness Rules for Type Soundness

Figure 11 shows the well-formedness rule that must hold to prove type soundness. Well-formedness rules that only affect the proofs of the theorems of the inheritance mechanism are split off and moved into the inheritance module.

The rules are similar to those of Featherweight Java. A first difference is the use of the $\mathcal{T}(\mathcal{M})$ to make the type soundness proof mostly independent of the syntax of the component relation. According to its definition in Figure 6 the only relevant part of the syntax is the type. The actual component parameters and configuration block are only needed by the inheritance mechanism. A second difference are the additional rules for the $e@\underline{T : p}$ expression. The formal component parameters must all have different names and the types in their constraints must exist. In addition, $e@\underline{T : p}$ is not allowed in the program expression since there is no evaluation rule for that expression.

Figure 11 also shows the conformance relation for class members. Its definition is used in the soundness proofs to derive properties about the members return by the lookup functions.

3.7 Type Soundness

Type soundness is proved using the standard preservation and progress approach for the elaborated program [49]. The proof is presented in Section 5.

Theorem 3.1 (Type Soundness) *If surface program* $P' = \langle \overline{cls'}, s \rangle$ *elaborates to* $P = \langle \overline{cls}, e \rangle$ *and* $WF(P)$ *and* $P, \Gamma \vdash e : T$ *and* $\langle \overline{cls_i}^{\,i}, e \rangle \hookrightarrow^* \langle \overline{cls_i}^{\,i}, e' \rangle$ *with* e' *a normal form, then either* e' *is a value* v *with* $P, \Gamma \vdash_s v : T$ *or* e' *contains an invalid cast* (U) $\textbf{new}\ C(\overline{e})\#ref$ *with* $P, \Gamma \vdash \textbf{new}\ C(\overline{e})\#ref : T'$ *and* $T' \not<: U$.

The Subject Reduction theorem is slightly modified compared to that of Featherweight Java: we explicitly demand that the program remains well-formed. In Featherweight Java, this follows trivially

$\boxed{WF(P)}$ P is well-formed

$$\frac{1.\ WF_{sound}(P) \quad 2.\ WF_{int}(P)}{WF(P)}$$

$\boxed{WF_{sound}(P)}$

$$\frac{\begin{array}{l}1.\ P = \langle\overline{cls}, e\rangle \\ 2.\ \overline{cls = \textbf{class}\ T\ (\overline{\alpha})\ \overline{sub}\ body} \\ 3.\ T_i = T_j \Leftrightarrow i = j \\ 4.\ P \vdash WF_{c,sound}(\overline{cls}) \\ 5.\ P, \emptyset \vdash e : T \\ 6.\ \neg\exists e', p', T' : e'@\overline{T' : p'} \in e\end{array}}{WF_{sound}(P)}$$

$\boxed{P \vdash WF_{c,sound}(cls)}$

$$\frac{\begin{array}{l}1.\ cls = \textbf{class}\ T\ (\overline{p\ U \to U'})\ \overline{sub_i}^{i\in 1..n}\ \{\ ctr\ \overline{def}\ \} \\ 2.\ \neg T = \textbf{Object} \\ 3.\ (p_i = p_j \Leftrightarrow i = j) \wedge \overline{U}, \overline{U'} \in P \\ 4.\ n > 0 \\ 5.\ ctr = T\ (\overline{X\ x})\{\overline{\textbf{this.f} = x}\} \\ 6.\ \overline{f} = \overline{x} \wedge (x_i = x_j \Leftrightarrow i = j) \\ 7.\ P \vdash WF_{m,sound}(\overline{T\ def})\end{array}}{P \vdash WF_{c,sound}(cls)}$$

$\boxed{P \vdash WF_{m,sound}(\mathcal{M})}$

$$\frac{1.\ \mathcal{T}(\mathfrak{F}) \in P}{P \vdash WF_{m,sound}(T\ f\ \mathfrak{F})} \qquad \frac{1.\ \mathcal{T}(\mathfrak{C}) \in P}{P \vdash WF_{m,sound}(T\ c\ \mathfrak{C})}$$

$$\frac{1.\ U, \overline{X} \in P \quad 2.\ x_i = x_j \Leftrightarrow i = j \quad 3.\ P, \Gamma, \overline{x : X}, \textbf{this} : T \vdash_s e : U}{P \vdash WF_{m,sound}(T\ m\ U\ (\overline{X\ x})\ \{e;\})}$$

$\boxed{P \vdash \mathcal{M} \leq \mathcal{M}'}$ $\mathcal{M}$ conforms to $\mathcal{M}'$

$$\frac{1.\ \mathcal{T}(\mathfrak{F}) = \mathcal{T}(\mathfrak{F}')}{P \vdash C\ path\ \mathfrak{F} \leq C'\ path'\ \mathfrak{F}'} \qquad \frac{\begin{array}{l}1.\ \mathcal{T}(\mathfrak{M}) = \overline{X} \to T \wedge \mathcal{T}(\mathfrak{M}') = \overline{X} \to T' \\ 2.\ P \vdash T <: T'\end{array}}{P \vdash C\ path\ \mathfrak{M} \leq C'\ path'\ \mathfrak{M}'} \qquad \frac{1.\ P \vdash \mathcal{T}(\mathfrak{C}) <: \mathcal{T}(\mathfrak{C}')}{P \vdash C\ path\ \mathfrak{C} \leq C'\ path'\ \mathfrak{C}'}$$

Figure 11: Member well-formedness and conformance.

from the well-formedness of the program, but we must require and prove this explicitly in order to avoid dependencies on the definitions of the lookup mechanism so we can modularize the latter.

Theorem 3.2 (Subject Reduction) *If* $P = \langle\overline{cls_i}^i, e\rangle$ *and* $WF(P)$ *and* $P, \Gamma \vdash e : T$ *and* $\langle\overline{cls_i}^i, e\rangle \hookrightarrow \langle\overline{cls_i}^i, e'\rangle$ *then* $P, \Gamma \vdash_s e' : T$ *and* $WF(\langle\overline{cls_i}^i, e'\rangle)$.

The progress theorem is essentially the same as that of Featherweight Java, but the progress theorem of the latter is actually a lemma that captures the non-trivial conditions that must be proven for the progress theorem to hold. We chose to write the progress theorem in the spirit of [49] since otherwise a part of the proof of progress simply moves to the proof of type soundness, which would no longer trivally from subject reduction and progress.

Theorem 3.3 (Progress) *If* $P = \langle\overline{cls}, e\rangle \wedge WF(P)$ *then either* $e \hookrightarrow e'$ *or* e *is a value, or* e *is or contains an invalid cast* $(U)\textbf{new}\ C(\overline{e''})\#ref$ *with* $P, \Gamma \vdash \textbf{new}\ C(\overline{e''})\#ref : T'$ *and* $P \vdash T' \not<: U$.

3.8 The Inheritance Signature

The inheritance signature defines how the inheritance mechanism of the calculus behaves. Figure 12 shows the main functions of the inheritance mechanism and axioms that capture their properties. A concrete inheritance module must provide the functions, and must prove that the axioms hold. The base calculus only relies on the signatures and axioms of Figure 12.

As a result, any concrete inheritance mechanism can be plugged in without requiring reverification of the type soundness proof, provided that the axioms still hold. The syntax of the component relation

$<:$ $:: P \times T \times T \rightarrow boolean$
$\mathcal{L}$ $:: P \times T \times z \times T \rightarrow \mathcal{M}$
$\mathscr{L}$ $:: P \times T \times ref \times z \times T \rightarrow \mathcal{M}$
$mem_P :: P \times T \rightarrow mem$

Theorem I_1: Subtyping is transitive
Theorem I_2: Result of lookup is well-formed
Theorem I_3: Lookup successful and conform
Theorem I_4: All fields initialized

Figure 12: The signature of an inheritance mechanism.

$\boxed{P \vdash T_1 <: T_2}$ T_1 subtype of T_2

$$\frac{1.\ T \in P}{P \vdash T <: T} \qquad \frac{1.\ P \vdash T_1 <: T_2 \quad 2.\ P \vdash T_2 <: T_3}{P \vdash T_1 <: T_3} \qquad \frac{1.\ \textbf{class}\ T\ (\overline{\alpha})\ \overline{sub}\ body \in P \quad 2.\ \textbf{subclass}\ C\ (\overline{\delta})\ conf \in \overline{sub}}{P \vdash T <: C}$$

Figure 13: The subtyping relation.

can be changed as long the member typing function of Figure 6 is updated. Note that inheritance mechanisms without support for subobject references can simply always use $\emptyset$ and remove expressions $\underline{e : T} \diamond c$ and $e@\underline{T : p}$.

The first axiom states that the subtyping relation must be transitive.

Axiom I_1 (Subtyping Transitive) $(WF(P) \wedge P \vdash T_1 <: T_2 \wedge P \vdash T_2 <: T_3) \Rightarrow P \vdash T_1 <: T_3$

The second axiom demands that the member resulting from a lookup has the dynamic class as its parent, is in the members of that same class, and is well-formed with respect to the partial well-formedness conditions in Figure 11.

Axiom I_2 (Lookup Well-Formed) $WF(P) \wedge (\mathscr{L}_P(C, ref, z, T) = \mathcal{M} \vee \mathcal{L}_P(C, z, T) = \mathcal{M}) \Rightarrow (\mathcal{M} = C\ path\ val \wedge \mathcal{M} \in mem_p(C) \wedge WF_{m,sound}(\mathcal{M}))$

The third axiom states that if a lookup on an outer object succeeds, lookup also succeeds if that object is used as a subobject, and the result of the latter lookup is unique and conforms to the result of the former lookup as defined by $\leq$ in Figure 11.

Axiom I_3 (Lookup Successful and Conform) $(WF(P) \wedge \mathcal{L}_P(U, z, T) = \mathcal{M} \wedge P, \Gamma \vdash_s \textbf{new}\ C(\overline{e})\#ref : U) \Rightarrow (\exists!\ \mathcal{M}' : \mathscr{L}_P(C, ref, z, T) = \mathcal{M}') \wedge (\mathscr{L}_P(C, ref, z, T) \leq \mathcal{M})$

The fourth and last axiom states that all fields of a class, whether declared in that class or inherited through subclassing or component relations, must be initialized in the constructor.

Axiom I_4 (All Fields Initialized) $(\textbf{class}\ C\ (\overline{\alpha})\ \overline{sub}\{C\ (\overline{X\ x})\{\overline{this.x = x}\}\overline{def}\} \in P \wedge \mathscr{L}_P(C, ref, z, T) = C\ path\ U;) \Rightarrow (path = x_i \wedge U = X_i)$

4 An Inheritance Module for First-class Composition Inheritance

In this section, we present the concrete inheritance module of O_2 that implements first-class composition inheritance. Figure 13 shows the subtype relation, which is reflexive and transitive.

4.1 Member Lookup

The lookup procedure consists of tree steps. The well-formedness rules of the inheritance module ensure that there are no ambiguities.

1. Calculate the collection of members of the class, including those of the components.
2. Define relations between the members to impose an order on them.
3. Select the most specific member.

4.1.1 Collecting Class Members

Figure 14 shows the definition of the mem_p function, which computes the members of a given class. Rule MEMBERS_T states that the members of a class are those that are defined locally in the class, and the members inherited through subclassing and component relations. Rule MEMBERS_{object} covers the top class `Object`, which has no members.

A subclass relation inherits a member (rule MEMBERS_{sc}) if it is potentially inherited through subclassing and not overridden, redundant, or dominated.

A subclass relation *potentially* inherits all members of the inherited class (rule POTENTIAL_{sc}). Since all members in a subclass relation are inherited directly, they are transformed by dir_p in rule DIRECT. The dir_p function renames the members and substitutes the formal component parameters and the `this` variable. Because `this` does not require substitution in a subclass relation, however, it is substituted with itself.

A component relation inherits a member (rule MEMBERS_{co}) if it is potentially inherited through components and not overridden. A member $\mathcal{M}$ is overridden if the inheriting class contains a member with the same name, or a component relation whose name c is the first part of the name of $\mathcal{M}$. In the latter case, component c overrides an enclosing component of $\mathcal{M}$, so we inherit the members of component c instead.

A component relation potentially inherits all members of the inherited class indirectly, an also directly if they are renamed (rule POTENTIAL_{co}). Directly inherited members are transformed by dir_p in rule DIRECT. Indirectly inherited members are transformed by ind_p in INDIRECT. The ind_p function also substitutes the formal component parameters and the `this` variable, but additionally prefixes the name of the member with the name of the component relation. In both directly and indirectly inherited members, the `this` variable is substituted with a subobject reference that selects the subobject represented by the current component relation, dispatching all internal self calls to the flattened component. The relations introduced in Section 4.1.2 ensure that such calls are dispatched correctly, even if these methods are overridden.

A member $\mathcal{M}$ is *redundant* (rule REDUNDANT) in two cases. First, it is redundant if an equivalent member $\mathcal{M}'$ is already inherited via a subclass relation that is defined before the current relation. In this case, the name and value of $\mathcal{M}$ and $\mathcal{M}'$ are equal according to the well-formedness rules, so we can remove all versions except the one inherited through the left-most subclass relation that inherits the member. Second, it is redundant if an equivalent member is inherited through a component relation of the subclass. In this case, that component relation overrides the component relation where $\mathcal{M}$ originated from. Because the component relation through which $\mathcal{M}'$ is inherited potentially uses different actual component parameters, we use $\mathcal{M}'$ and mark $\mathcal{M}$ as redundant.

A member is *dominated* (definition DOMINATED) if the class already inherits another member that overrides that member. In this case, it is safe to use the definition of the overriding member and discard the overridden members. As in the redundancy rule, members inherited through component relations are also taken into account.

$$[\text{MEMBERS}_T] \quad \frac{1.\ \textbf{class}\ T\ (\overline{\alpha})\ \overline{sub}\ \{\ ctr\ \overline{def}\ \} \in P}{mem_p(T) = \overline{T\ def} \cup mem_{p,st}(T) \cup mem_{p,co}(T)}$$

$$[\text{MEMBERS}_{object}] \quad \frac{}{mem_p(\textbf{Object}) = \emptyset}$$

$$[\text{MEMBERS}_{sc}] \quad \frac{1.\ \textbf{class}\ T\ (\overline{\alpha})\ \overline{sub}\ body \in P \quad 2.\ \overline{sub}\ =\ \overline{\textbf{subclass}\ C\ (\overline{\delta})\ conf}}{mem_{p,st}(T) = \{dir_p(\mathcal{M}, T, \textbf{this}, \overline{\delta_i}, conf_i) | \mathcal{M} \in pot_{p,st}(T, sub_i) \wedge inh_P(\mathcal{M}, sub_i, T)\}}$$

$$[\text{POTENTIAL}_{sc}] \quad \frac{}{pot_{p,st}(T, \textbf{subclass}\ C\ (\overline{\delta})\ conf) = mem_p(C)}$$

$$[\text{MEMBERS}_{co}] \quad \frac{\begin{array}{l} 1.\ \textbf{class}\ T\ (\overline{\alpha})\ \overline{sub}\ \{\ ctr\ \overline{z\ val}\ \} \in P \wedge \overline{z\ val} = \overline{z_1\ \mathfrak{C}} \cup \overline{z_2\ \mathfrak{F}} \cup \overline{z_3\ \mathfrak{M}} \\ 2.\quad \overline{\mathcal{M}'} = \{\mathcal{M}' | \mathcal{M}' \in pot_{p,co}(T, z_{1\,i}\ \mathfrak{C}_i) \wedge \mathcal{M}' = T\ path'\ val' \wedge \\ \qquad (\neg path' = z_i) \wedge (\neg path' = z_{1\,i} \diamond path'')\} \end{array}}{mem_{p,co}(T) = \overline{\mathcal{M}'}}$$

$$[\text{POTENTIAL}_{co}] \quad \frac{\begin{array}{l} 1.\quad \overline{\mathcal{M}'} = \{\mathcal{M}' | C\ path\ val \in mem_p(C) \wedge (path = z_i \vee path = z_i \diamond path') \wedge \\ \qquad dir_p(C\ path\ val, T, \underline{\textbf{this}: T} \diamond c, \overline{\delta}, [\overline{z = z'}]) = \mathcal{M}'\} \end{array}}{pot_{p,co}(T, c\ \textbf{component}\ C\ (\overline{\delta})\ [\overline{z = z'}]) = ind_p(mem_p(C), T, z\ \mathfrak{C}) \cup \overline{\mathcal{M}'}}$$

$$[\text{INHERITED}_{sc}] \quad \frac{\begin{array}{l} 1.\ \textbf{class}\ T\ (\overline{\alpha})\ \overline{sub}\ \{\ ctr\ \overline{z\ val}\ \} \in P \\ 2.\ ((\neg path' \in \overline{z}) \wedge (\neg dom_p(T\ path'\ val', T))) \wedge (\neg red_p(T\ path'\ val', sub_i, T)) \end{array}}{inh_P(T\ path'\ val', sub_i, T)}$$

$$[\text{REDUNDANT}] \quad \frac{\begin{array}{l} 1.\ \textbf{class}\ T\ (\overline{\alpha})\ \overline{sub}\ \{\ ctr\ \ \overline{z_1\ \mathfrak{C}}\ \ \overline{z_2\ \mathfrak{F}}\ \ \overline{z_3\ \mathfrak{M}}\ \} \in P \\ 2.\ ((\mathcal{M}' \in pot_{p,st}(T, sub_k) \wedge k\ <\ i) \vee \mathcal{M}' \in pot_{p,co}(C, z_{1\,i}\ \mathfrak{C}_i)) \wedge\ P \vdash \mathcal{M} \equiv \mathcal{M}' \end{array}}{red_p(\mathcal{M}, sub_i, T)}$$

$$[\text{DOMINATED}] \quad \frac{\begin{array}{l} 1.\ \textbf{class}\ T\ (\overline{\alpha})\ \overline{sub}\ \{\ ctr\ \ \overline{z_1\ \mathfrak{C}}\ \ \overline{z_2\ \mathfrak{F}}\ \ \overline{z_3\ \mathfrak{M}}\ \} \in P \\ 2.\ (\mathcal{M}' \in pot_{p,st}(T, sub) \vee \mathcal{M}' \in pot_{p,co}(T, z_{1\,i}\ \mathfrak{C}_i)) \wedge\ P \vdash \mathcal{M}' \ggg \mathcal{M} \end{array}}{dom_p(\mathcal{M}, T)}$$

$$[\text{DIRECT}] \quad \frac{1.\ rename(path, [\overline{z = z'}]) = path' \quad 2.\ transform_p(T, val, C, e, \overline{\delta}) = val'}{dir_p(C\ path\ val, T, e, \overline{\delta}, [\overline{z = z'}]) = T\ path'\ val'}$$

$$[\text{INDIRECT}] \quad \frac{1.\ transform_p(T, val, C, \underline{\textbf{this}: T} \diamond c, \overline{\delta}) = val'}{ind_p(C\ path\ val, T, c\ \textbf{component}\ C\ (\overline{\delta})\ conf) = T\ c \diamond path\ val'}$$

Figure 14: Class members.

$$\boxed{rename(path, conf) = path'}$$

[REN_BASIC] $$\frac{}{rename(z_i, [\overline{z = z'}]) = z'_i}$$

[REN_COMP] $$\frac{}{rename(z_i \diamond path, [\overline{z = z'}]) = z'_i \diamond path}$$

[REN_BASIC_NO] $$\frac{1.\ \neg z'' \in \overline{z}}{rename(z'', [\overline{z = z'}]) = z''}$$

[REN_COMP_NO] $$\frac{1.\ \neg z'' \in \overline{z}}{rename(z'' \diamond path, [\overline{z = z'}]) = z'' \diamond path}$$

$$\boxed{transform_p(C, val, T, e, \overline{\delta}) = val'}$$ val in T becomes val' in C, after substitution of e and δ

[TR_M] $$\frac{1.\ \textbf{class}\ T\ (\overline{p\ V \rightarrow V'})\ \overline{sub}\ body \in P \quad 2.\ \llbracket \overline{\delta}/\overline{p} \rrbracket_C [e/\textbf{this}] t = t''}{transform_p(C, U\ (\overline{X\ x})\ \{t;\}, T, e, \overline{\delta}) = U\ (\overline{X\ x})\ \{t'';\}}$$

[TR_F] $$\frac{}{transform_p(C, \mathfrak{F}, T, e, \overline{\delta}) = \mathfrak{F}}$$

[TR_C] $$\frac{1.\ \textbf{class}\ T\ (\overline{p\ V \rightarrow V'})\ \overline{sub}\ body \in P}{transform_p(C, \mathfrak{C}, T, e, \overline{\delta}) = [\overline{a}/\overline{p}]\mathfrak{C}}$$

$$\boxed{\llbracket \overline{\delta}/\overline{p} \rrbracket_C\, t = t'}$$ replacing $\overline{p}$ with $\overline{\delta}$ of class C in t gives t'

[ACTUAL] $$\frac{1.\ \textbf{class}\ T\ (\overline{p\ U \rightarrow U'})\ \overline{sub}\ body \in P \quad 2.\ \delta_k = c}{\llbracket \overline{\delta}/\overline{p} \rrbracket_C\, e@\underline{T : p_k} = \underline{\llbracket \overline{\delta}/\overline{p} \rrbracket_C\, e : U_k} \diamond c}$$

[FORMAL] $$\frac{1.\ \delta_k = p'_i}{\llbracket \overline{\delta}/\overline{p} \rrbracket_C\, e@\underline{T : p_k} = \llbracket \overline{\delta}/\overline{p} \rrbracket_C\, e@\underline{C : p'_i}}$$

Figure 15: Member transformation.

Transformation of Members

Figure 15 shows the functions that transform members when they are inherited. They take care of renaming the members, and substituting formal component parameters and the `this` variable.

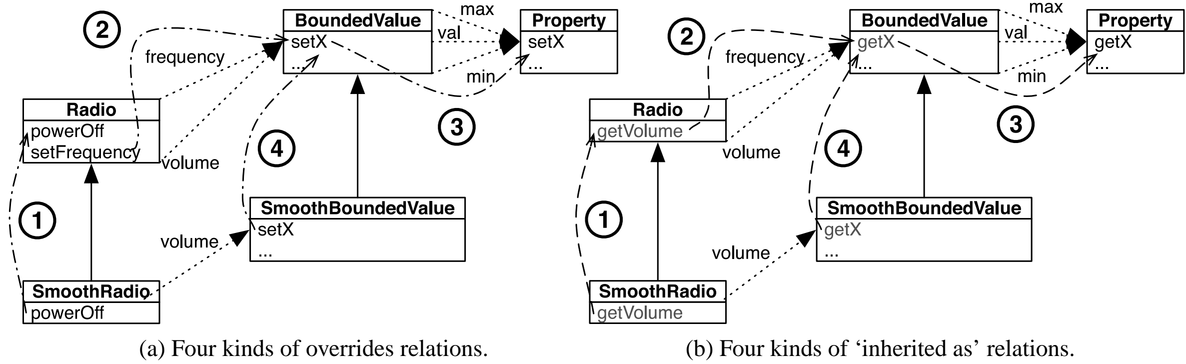
The renaming function performs two kinds of renamings. First, it gives a new name to members that are renamed (rule REN_BASIC). Second, it gives a new name to members of a component if that component is renamed (rule REN_COMP). For example, if in a subclass of `Radio` the `volume` component is renamed to `noisiness`, then `volume`$\diamond$`getValue` is renamed to `noisiness`$\diamond$`getValue`. If there is no match (rules REN_BASIC_NO and REN_COMP_NO), nothing is changed.

Function $transform_p$ substitutes formal component parameters and the `this` variable. In method bodies (rule TR_M), a regular substitution is done for the `this` variable, and a special substitution for invocations on component parameters ($e@\underline{T : p}$) is delegated to the $\llbracket \overline{\delta}/\overline{p} \rrbracket_C$ function. Fields remain unmodified (rule TR_F), and in component relations, a regular substitution replaces the formal component parameters with the actual parameters (rule TR_C).

Function $\llbracket \overline{\delta}/\overline{p} \rrbracket_C$ performs substitution of formal component parameters in expressions. Only the interesting rules are displayed to save space; the other rules simply descend into their child expressions. If the actual component parameter is the name of a component relation c (rule ACTUAL), the expression is changed to a subobject reference ($\diamond c$). In addition, the elaborated type is changed to U_k, which is the left-hand type of the constraint of the substituted formal component parameter. If the actual component parameter is the name of a formal component parameter from the inheriting class (rule FORMAL), a regular substitution is done and the elaborated type is updated. The well-formedness rules on actual component parameters ensure that type soundness is preserved.

4.1.2 Member Relations

To filter the methods inherited through an inheritance relation and to select the most specific method in the typing and evaluation relations, we must impose an order on members by adding relations



(a) Four kinds of overrides relations.

(b) Four kinds of 'inherited as' relations.

Figure 16: Relations between members.

between them. There are two base relations, and two relations built on top of them. Figure 17 shows the definitions. To simplify the rules, we ignore the order of the member declarations of a class.

The base relations are $\mathcal{M}_1$ *overrides* $\mathcal{M}_2$ ($P \vdash \mathcal{M}_1 \ggg \mathcal{M}_2$), and $\mathcal{M}_2$ *is inherited as* $\mathcal{M}_1$ ($P \vdash \mathcal{M}_1 \leftarrow \mathcal{M}_2$). The relations are of course mutually exclusive, and in both cases renaming can be performed.

An overrides relation is mainly introduced in four different situations. Figure 16a uses an artificial example to illustrate the four different situations. Overrides relations are shown by the mixed dashed and dotted arrows. In the example, the `set` method of the frequency component is overridden as `setFrequency`, and lowers the volume when changing the frequency to prevent unpleasant surprises when switching channels. A `SmoothRadio` uses a special volume control that can only make smooth changes. It therefore overrides the `volume` component with a more specialized `SmoothBoundedValue` which only allows gradual changes. In addition, it overrides `powerOff` to turn down the volume if the radio is turned off. The value and boundaries of `BoundedValue` are programmed using `Property` components which provide a field, a getter, and a setter. The labels in Figures 16a and the rule names in Figure 17 match those in the description below.

For the overrides relation, the main cases are:

1. *Subclass relation :* within a subclass relation, a class member $T\ n\ val$ overrides another member $\mathcal{M}'$ if $\mathcal{M}'$ would have been inherited with name n. In Figure 16a, this rule ensures that `SmoothRadio.powerOff` overrides `Radio.powerOff`.

2. *Component relation :* within a component inheritance relation, a class member $T\ n\ val$ overrides another member $T\ c\diamond path'\ val'$ if the member with name $path'$ would have been inherited with name n. The rule does not introduce a relation with $T'\ path'\ val'$ of the inherited class because that relation cannot encode the inheritance path. This would lead to ambiguities because the component relation supports repeated inheritance. Instead, it introduces a relation with the corresponding member of the flattened component by prefixing with the name of the component relation and a $\diamond$. In Figure 16a, this rule ensures that `Radio.setFrequency` overrides `SmoothRadio◇frequency.setX`.

3. *Copying for inheritance relations :* all overrides relations between the members of the target class of a component relation are copied to the flattened component. This situation occurs if the target class of component relation has components. In Figure 16a, this rule ensures that `SmoothRadio◇volume.setX` overrides `SmoothRadio◇volume◇val.setX`. Such relations are also copied for members that are inherited through subclassing.

4. *Copying for component content :* if a component relation overrides another component relation, overrides relations between members of both target classes are copied to the corresponding members of the flattened components. In Figure 16a, this rule ensures that `SmoothRadio◇volume.setX` overrides `Radio◇volume.setX`

An 'inherited as' relation is added in situations similar to that of the overrides relation. The relation is illustrated in Figure 16b by the dashed arrows.

For the 'inherited as' relation, the main cases are:

1. *Subclass relation:* for a subclass relation, all members that are not overridden are considered to be renamed even if their name does not change. In Figure 16b, this rule ensures that `Radio.getVolume` is inherited as `SmoothRadio.getVolume`.

2. *Component relation:* for a component relation, all indirectly inherited members in the flattened component are inherited as their corresponding directly inherited members in the inheriting class if they are not overridden. In Figure 16b, this rule ensures that `Radio◇volume.getX` is inherited as `Radio.getVolume`.

3. *Copying for inheritance relations:* all 'inherited as' relations between the members of the target class of a component relation are copied to the flattened component. In Figure 16b, this rule ensures that `SmoothRadio◇volume◇val.getX` is inherited as `SmoothRadio◇volume.getX`. Such relations are also copied for members that are inherited through subclassing.

4. *Copying for component content:* if a component relation overrides another component relation, *equivalence* relations between members of both target classes are translated to 'inherited as' relations between the corresponding members of the flattened components. In Figure 16b, this rule ensures that `Radio◇volume.getX` is inherited as `SmoothRadio◇volume.getX`. Note that we must use an equivalence relation between the original members to trigger the REDUNDANT rule in Figure 14 if the overriding component has the same type as the overridden component, and all members of the inherited class are thus equal. The same is done if there is an 'inherited as' relation between both components.

The relation $\mathcal{M}$ *equivalent to* $\mathcal{M}'$ ($P \vdash \mathcal{M} \equiv \mathcal{M}'$) is the reflexive and symmetric closure of the 'inherited as' relation.

The relation $\mathcal{M}$ *related to* $\mathcal{M}'$ ($P \vdash \mathcal{M} \leftrightsquigarrow \mathcal{M}'$) specifies when two members are related in a subclass hierarchy. This relation is the symmetric closure of the 'inherited as' and overrides relations – the latter is not symmetric.

$\boxed{P \vdash \mathcal{M} \vdash \mathcal{M}' \ggg}$ $\mathcal{M}$ overrides $\mathcal{M}'$

$$[\text{O_SC}] \quad \frac{\begin{array}{l} 1.\ \textbf{class}\ T\ (\overline{\alpha})\ \overline{\textbf{subclass}\ C\ (\overline{\delta})\ conf}\ \{\ ctr\ \overline{z_0\ val_0}\ \} \in P \\ 2.\ \mathcal{M}' \in mem_p(C_i) \\ 3.\ dir_p(\mathcal{M}', T, \textbf{this}, \overline{\delta}, conf) = T\ z_k\ val_1 \end{array}}{P \vdash\ T\ z_k\ val_k \ggg \mathcal{M}'}$$

$$[\text{O_CO}] \quad \frac{\begin{array}{l} 1.\ \textbf{class}\ T\ (\overline{\alpha})\ \overline{sub}\ \{\ ctr\ \overline{z_0\ val_0}\ \} \in P \wedge c\ \textbf{component}\ C\ (\overline{\delta})\ [\overline{z' = z''}] \in \overline{z_0\ val_0} \\ 2.\ C\ z'_i\ val_1 \in mem_p(C) \\ 3.\ dir_p(C\ z'_i\ val_1, T, \underline{\textbf{this}: T} \diamond c, \overline{\delta}, [\overline{z' = z''}]) = T\ z_{0\,k}\ val_2 \end{array}}{P \vdash\ T\ z_{0\,k}\ val_{0\,k} \ggg T\ c \diamond z'_i\ val_1}$$

$$[\text{O_SC}_{copy}] \quad \frac{\begin{array}{l} 1.\ \textbf{class}\ T\ (\overline{\alpha})\ \overline{sub}\ body \in P \\ 2.\ sub_i\ =\ \textbf{subclass}\ C_i\ (\overline{\delta})_i\ conf_i \\ 3.\ \mathcal{M}_1, \mathcal{M}_2 \in mem_p(C_i) \\ 4.\ inh_P(\mathcal{M}_3, sub_i, T) \wedge P \vdash \mathcal{M}_3 \leftarrow \mathcal{M}_1 \\ 5.\ inh_P(\mathcal{M}_4, sub_i, T) \wedge P \vdash \mathcal{M}_4 \leftarrow \mathcal{M}_2 \\ 6.\ P \vdash \mathcal{M}_1 \ggg \mathcal{M}_2 \end{array}}{P \vdash \mathcal{M}_3 \ggg \mathcal{M}_4}$$

$$[\text{O_CO}_{copy}] \quad \frac{\begin{array}{l} 1.\ \textbf{class}\ T\ (\overline{\alpha})\ \overline{sub}\ \{\ ctr\ \overline{def}\ \} \in P \\ 2.\ c\ \textbf{component}\ C\ (\overline{\delta})\ conf \in \overline{def} \\ 3.\ C\ path_1\ val_1 \in mem_p(C) \\ 4.\ C\ path_2\ val_2 \in mem_p(C) \\ 5.\ P \vdash\ C\ path_1\ val_1 \ggg C\ path_2\ val_2 \end{array}}{P \vdash\ T\ c \diamond path_1\ val_1 \ggg T\ c \diamond path_2\ val_2}$$

$$[\text{O_CONTENT}_{copy}] \quad \frac{\begin{array}{l} 1.\ P \vdash\ T_1\ path_1\ \mathfrak{C}_1 \ggg T_2\ path_2\ \mathfrak{C}_2 \\ 2.\ \mathcal{T}(\mathfrak{C}_1)\ path_3\ val_3 \in mem_p(\mathcal{T}(\mathfrak{C}_1)) \wedge \mathcal{T}(\mathfrak{C}_2)\ path_4\ val_4 \in mem_p(\mathcal{T}(\mathfrak{C}_2)) \\ 3.\ P \vdash \mathcal{T}(\mathfrak{C}_1)\ path_3\ val_3 \ggg \mathcal{T}(\mathfrak{C}_2)\ path_4\ val_4 \end{array}}{P \vdash\ T_1\ path_1 \diamond path_3\ val_1 \ggg T_2\ path_2 \diamond path_4\ val_4}$$

$$[\text{O_TRANS}_1] \quad \frac{\begin{array}{l} 1.\ P \vdash \mathcal{M}_1 \ggg \mathcal{M}_2 \\ 2.\ P \vdash \mathcal{M}_2 \ggg \mathcal{M}_3 \end{array}}{P \vdash \mathcal{M}_1 \ggg \mathcal{M}_3} \qquad [\text{O_TRANS}_2] \quad \frac{\begin{array}{l} 1.\ P \vdash \mathcal{M}_1 \equiv \mathcal{M}_2 \\ 2.\ P \vdash \mathcal{M}_2 \ggg \mathcal{M}_3 \end{array}}{P \vdash \mathcal{M}_1 \ggg \mathcal{M}_3} \qquad [\text{O_TRANS}_3] \quad \frac{\begin{array}{l} 1.\ P \vdash \mathcal{M}_1 \ggg \mathcal{M}_2 \\ 2.\ P \vdash \mathcal{M}_2 \equiv \mathcal{M}_3 \end{array}}{P \vdash \mathcal{M}_1 \ggg \mathcal{M}_3}$$

$\boxed{P \vdash \mathcal{M} \leftarrow \mathcal{M}'}$ $\mathcal{M}'$ is inherited as $\mathcal{M}$

$$[\text{I_SC}] \quad \frac{\begin{array}{l} 1.\ \textbf{class}\ T\ (\overline{\alpha})\ \overline{\textbf{subclass}\ C\ (\overline{\delta})\ conf}\ body \in P \\ 2.\ \mathcal{M}' \in mem_p(C_i) \\ 3.\ dir_p(\mathcal{M}', T, \textbf{this}, \overline{\delta}, conf) = \mathcal{M} \\ 4.\ inh_P(\mathcal{M}', sub_i, T) \end{array}}{P \vdash \mathcal{M} \leftarrow \mathcal{M}'}$$

$$[\text{I_TRANS}] \quad \frac{\begin{array}{l} 1.\ P \vdash \mathcal{M}_1 \leftarrow \mathcal{M}_2 \\ 2.\ P \vdash \mathcal{M}_2 \leftarrow \mathcal{M}_3 \end{array}}{P \vdash \mathcal{M}_1 \leftarrow \mathcal{M}_3}$$

$$[\text{I_CO}] \quad \frac{\begin{array}{l} 1.\ \textbf{class}\ T\ (\overline{\alpha})\ \overline{sub}\ \{\ ctr\ \overline{z_0\ val_0}\ \} \in P \wedge c\ \textbf{component}\ C\ (\overline{\delta})\ [\overline{z' = z''}] \in \overline{z_0\ val_0} \\ 2.\ C\ z'_i\ val_1 \in mem_p(C) \\ 3.\ dir_p(C\ z'_i\ val_1, T, \underline{\textbf{this}: T} \diamond c, \overline{\delta}, [\overline{z' = z''}]) = T\ z_2\ val_2 \\ 4.\ \neg z_2 \in \overline{z_0} \end{array}}{P \vdash\ T\ z_2\ val_2 \leftarrow T\ c \diamond z'_i\ val_1}$$

$$[\text{I_SC}_{copy}] \quad \frac{\begin{array}{l} 1.\ \textbf{class}\ T\ (\overline{\alpha})\ \overline{sub}\ body \in P \\ 2.\ sub_i\ =\ \textbf{subclass}\ C_i\ (\overline{\delta})_i\ conf_i \\ 3.\ \mathcal{M}_1, \mathcal{M}_2 \in mem_p(C_i) \\ 4.\ P \vdash \mathcal{M}_1 \leftarrow \mathcal{M}_2 \\ 5.\ inh_P(\mathcal{M}_1, sub_i, T) \wedge P \vdash \mathcal{M}_3 \leftarrow \mathcal{M}_1 \\ 6.\ inh_P(\mathcal{M}_2, sub_i, T) \wedge P \vdash \mathcal{M}_4 \leftarrow \mathcal{M}_2 \end{array}}{P \vdash \mathcal{M}_3 \leftarrow \mathcal{M}_4}$$

$$[\text{I_CO}_{copy}] \quad \frac{\begin{array}{l} 1.\ \textbf{class}\ T\ (\overline{\alpha})\ \overline{sub}\ \{\ ctr\ \overline{def}\ \} \in P \\ 2.\ c\ \textbf{component}\ C\ (\overline{\delta})\ conf \in \overline{def} \\ 3.\ C\ path_1\ val_1 \in mem_p(C) \\ 4.\ C\ path_2\ val_2 \in mem_p(C) \\ 5.\ P \vdash\ C\ path_1\ val_1 \leftarrow C\ path_2\ val_2 \end{array}}{P \vdash\ T\ c \diamond path_1\ val_1 \leftarrow T\ c \diamond path_1\ val_2}$$

$$[\text{I_CONTENT}_{copy}] \quad \frac{\begin{array}{l} 1.\ P \vdash\ T_1\ path_1\ \mathfrak{C}_1 \ggg T_2\ path_2\ \mathfrak{C}_2 \vee P \vdash\ T_1\ path_1\ \mathfrak{C}_1 \leftarrow T_2\ path_2\ \mathfrak{C}_2 \\ 2.\ \mathcal{T}(\mathfrak{C}_1)\ path_3\ val_3 \in mem_p(\mathcal{T}(\mathfrak{C}_1)) \wedge \mathcal{T}(\mathfrak{C}_2)\ path_4\ val_4 \in mem_p(\mathcal{T}(\mathfrak{C}_2)) \\ 3.\ P \vdash \mathcal{T}(\mathfrak{C}_1)\ path_3\ val_3 \equiv \mathcal{T}(\mathfrak{C}_2)\ path_4\ val_4 \end{array}}{P \vdash\ T_1\ path_1 \diamond path_3\ val_1 \leftarrow T_2\ path_2 \diamond path_4\ val_4}$$

$\boxed{P \vdash \mathcal{M} \equiv \mathcal{M}'}$ $\mathcal{M}$ is equivalent to $\mathcal{M}'$: symmetric and reflexive closure of $P \vdash \mathcal{M} \leftarrow \mathcal{M}'$

$\boxed{P \vdash \mathcal{M} \leftrightsquigarrow \mathcal{M}'}$ $\mathcal{M}$ is related to $\mathcal{M}'$: symmetric closure of $P \vdash \mathcal{M} \equiv \mathcal{M}'$ and $P \vdash \mathcal{M} \ggg \mathcal{M}'$

Figure 17: Member relations.

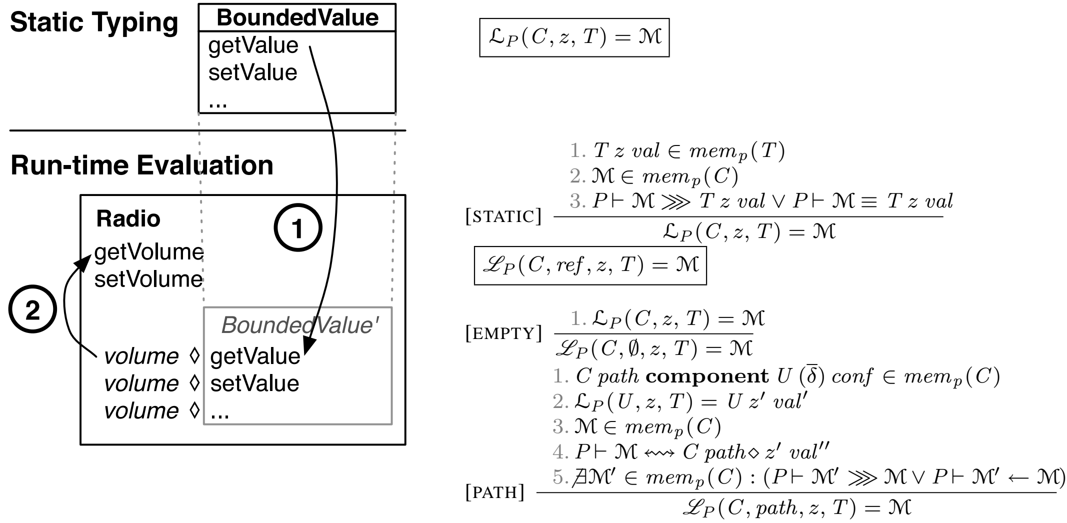


Figure 18: Member lookup.

4.1.3 Lookup

We can now describe member lookup using the members of a class, and the relations between them. We illustrate the procedure for resolving `getValue` on type `BoundedValue` if the object is the `volume` subobject of the `Radio` class of Figure 18. We assume `getValue` of the `volume` component is overridden in `Radio` as `getVolume`. In the first phase, the procedure looks for the member of the type of the subobject that corresponds to `getValue`. Since that type is the same as the static type – `BoundedValue` – the result is still `getValue`. In the second phase, that method is then prefixed with the name of the component such that it becomes a method of the type of the real object, resulting in `volume◇getValue`. The procedure then looks for the most specific version of the member in `Radio`. In this case, `getVolume` overrides `volume◇getValue`, and thus `getVolume` is the final result.

The $\mathcal{L}_P$ function, defined by STATIC, performs the first phase of the lookup. It selects the most specific member given a simple member name z, a static type T, and a dynamic type C. Because duplication is forbidden for subclassing and the lookup is done on a simple name, the relation can simply select the member in C which is related to the member with name z in T. Such a member always exists because for every member $\mathcal{M}$ of a class, there is a member $\mathcal{M}'$ in the subclass such that $\mathcal{M}'$ overrides $\mathcal{M}$, or $\mathcal{M}$ is inherited as $\mathcal{M}'$.

The second relation performs the second phase of the lookup. It performs the actual lookup given a simple member name z, a subobject reference ref, the static type T of the subobject reference, and the dynamic type C of the real object. In case of an empty subobject reference (rule EMPTY), the lookup is directly delegated to $\mathcal{L}_P$. In case of a non-empty subobject reference (rule PATH), first the component corresponding to the subobject reference ($path'$) is selected from the actual type. This subobject has static type T and dynamic type U with $U <: T$. Then, $\mathcal{L}_P$ is used to select the member of the dynamic component type U that is related to member z of the static type T. This member corresponds to member $path' \diamond z$ of dynamic type C – it is part of the flattened component with name $path'$. Finally, we select the most specific related member from C with respect to overrides and 'inherited as' relations.

$\boxed{P, T \vdash \delta \leq \alpha}$ in T, δ conforms to α

$$(22.1)\quad \frac{1.\ \mathcal{L}_P(U, c, U) = U\ c\ \textbf{component}\ U''\ (\overline{\delta})\ conf \quad 2.\ P \vdash U'' <: U'}{P, T \vdash c \leq p\ U \rightarrow U'}$$

$$(22.2)\quad \frac{1.\ P \vdash T \rightarrow \textbf{class}\ T\ (\overline{p_i'\ V_i \rightarrow V_i'}^{\,i})\ \overline{sub_j}^{\,j}\ body \quad 2.\ P \vdash U <: V_k \quad 3.\ P \vdash V_k' <: U'}{P, T \vdash p_k' \leq p\ U \rightarrow U'}$$

Figure 19: Conformance relations.

$\gamma ::= \delta \mid T \diamond \delta$ elaborated component parameter

$\boxed{\alpha \vdash \gamma \Rightarrow \gamma'}$

$$(23.1)\quad \frac{}{p\ T_1 \rightarrow T_2 \vdash c \Rightarrow \underline{T_1 \diamond c}}$$

$$(23.2)\quad \frac{1.\ \neg\exists c : \gamma = c}{\alpha \vdash \gamma \Rightarrow \gamma}$$

$\boxed{P \vdash \gamma \vdash \gamma' \equiv}$

$$(24.1)\quad \frac{1.\ \mathcal{L}_P(T, c, T) = \mathcal{M} \quad 2.\ \mathcal{L}_P(T', c', T') = \mathcal{M}' \quad 3.\ P \vdash \mathcal{M} \leftrightsquigarrow \mathcal{M}'}{P \vdash \underline{T \diamond c} \vdash \underline{T' \diamond c'} \equiv}$$

$$(24.2)\quad \frac{}{P \vdash \underline{T \diamond p} \vdash \underline{T \diamond p} \equiv}$$

$\boxed{P \vdash params(\overline{\gamma_i}^{\,i}, T, T') = \overline{\gamma_j'}^{\,j}}$

$$(25.1)\quad \frac{\begin{array}{l} 1.\ \neg T\ =\ T' \\ 2.\ P \vdash T \rightarrow \textbf{class}\ T\ (\overline{\alpha_i}^{\,i\in 1..n})\ \overline{sub_j}^{\,j\in 1..o}\ body \\ 3.\ \forall i \in 1..n : \alpha_i \vdash \gamma_i \Rightarrow \gamma_i' \\ 4.\ \forall j \in 1..o : sub_j\ =\ \textbf{subclass}\ U_j\ (\overline{\delta})_j\ conf_j \\ 5.\ \overline{\gamma_l''}^{\,l}\ =\ [\overline{\gamma_i'/\alpha_i}^{\,i\in 1..n}]\ (\overline{\delta})_k \\ 6.\ P \vdash params(\overline{\gamma_l''}^{\,l}, U_k, T') = \overline{\gamma_o'''}^{\,o} \end{array}}{P \vdash params(\overline{\gamma_i}^{\,i\in 1..n}, T, T') = \overline{\gamma_o'''}^{\,o}}$$

$$(25.2)\quad \frac{\begin{array}{l} 1.\ P \vdash T \rightarrow \textbf{class}\ T\ (\overline{\alpha_i}^{\,i\in 1..n})\ \overline{sub_j}^{\,j}\ body \\ 2.\ \forall i \in 1..n : \alpha_i \vdash \gamma_i \Rightarrow \gamma_i' \end{array}}{P \vdash params(\overline{\gamma_i}^{\,i\in 1..n}, T, T) = \overline{\gamma_i'}^{\,i\in 1..n}}$$

Figure 20: Flow of actual component parameters.

4.2 Well-formedness for Inheritance

4.2.1 Auxiliary Definitions

Figure 20 shows the flow of actual component parameters through the subtyping hierarchy. This flow is used in the well-formedness rules to ensure that a class does not pass different actual component parameters to a single formal component parameter via different subtyping paths.

To perform the computation, an elaborated component parameter γ is introduced. The elaboration is required because names of component relations (c) and formal component parameters (p) are declared relative to a type. Names of component relations and formal component parameters are elaborated separately. Definition 23 shows how names of component relations are elaborated. If γ is a non-elaborated name, it is prefixed with the left-hand type of the constraint of the corresponding formal component parameter. Otherwise, γ remains unmodified. Names of formal component parameters (p) are elaborated in the well-formedness rule when the function is invoked.

Equivalence of elaborated component parameters is defined in definition 24. Two elaborated parameters are equivalent if they reference related components or formal component parameters of the same type. Requiring that the names reference related components suffices because duplication is not allowed for subtyping, ensuring that during lookup the same component is selected in both cases.

$\boxed{WF_{int}(P)}$ P is well-formed for lookup

$$\text{(26.1)}\quad \frac{\begin{array}{l} 1.\ P = \langle \overline{cls}, e \rangle \\ 2.\ P \vdash WF_{c,int}(\overline{cls}) \end{array}}{WF_{int}(P)}$$

$\boxed{P \vdash WF_{c,int}(cls)}$ cls is well-formed for lookup in P

$$\text{(27.1)}\quad \frac{\begin{array}{l} 1.\ cls = \textbf{class}\ T\ (\overline{p_i\ U_i \rightarrow U'_i}^{i \in 1..g})\ \overline{sub_j}^{j \in 1..n}\ \{\ ctr\ \overline{z_k\ val_k}^{k \in 1..o}\ \} \\ 2.\ \forall j \in 1..n : P \vdash ValidST(T, sub_j) \\ 3.\ \forall k \in 1..o : \forall \mathcal{M} : (P \vdash\ T\ z_k\ val_k \ggg \mathcal{M} \Rightarrow P \vdash\ T\ z_k\ val_k \leq \mathcal{M}) \\ 4.\ mem_p(T) = \overline{T\ path'_l\ val'_l}^{l \in 1..h} \\ 5.\ \forall \mathcal{M} \in mem_p(C) : (P \vdash\ T <: C \Rightarrow \exists T\ z\ val \in mem_p(T) : P \vdash\ T\ z\ val \leftrightsquigarrow \mathcal{M}) \\ 6.\ \forall i, j \in 1..h : ((\neg i\ =\ j) \Rightarrow (\neg path'_i = path'_j)) \\ 7.\ \quad \forall \mathcal{M}, \mathcal{M}' \in mem_p(T) : (\mathcal{M} \neq \mathcal{M}' \wedge P \vdash \mathcal{M} \leftrightsquigarrow \mathcal{M}') \Rightarrow \\ \qquad\qquad ((P \vdash \mathcal{M} \ggg \mathcal{M}' \vee P \vdash \mathcal{M}' \ggg \mathcal{M}) \vee (P \vdash \mathcal{M} \leftarrow \mathcal{M}' \vee P \vdash \mathcal{M}' \leftarrow \mathcal{M})) \\ 8.\ P \vdash NoSTLoops(cls) \\ 9.\ P \vdash NoCOLoops(cls) \\ 10.\ P \vdash ConsistentCParams(cls) \\ 11.\ P \vdash NoSubtypeDup(cls) \\ 12.\ P \vdash FieldInit(T) \end{array}}{P \vdash WF_{c,int}(cls)}$$

$\boxed{P \vdash FieldInit(T)}$

$$\text{(28.1)}\quad \frac{\begin{array}{l} 1.\ P \vdash T \rightarrow \textbf{class}\ T\ (\overline{\alpha_i}^{\,i})\ \overline{sub}\ \{\ ctr\ \overline{def}_k^{\,k}\ \} \\ 2.\ mem_p(T) = \overline{\mathcal{M}} \\ 3.\ \overline{T\ f_j\ U_j\ ;}^{j \in 1..n} = \{T\ f\ U\ ;\ \mid\ T\ f\ U\ ; \in \overline{\mathcal{M}}\} \\ 4.\ ctr\ =\ T\ (\overline{U_j\, x_j}^{j \in 1..n})\ \{\overline{\mathbf{this}.f_j = x_j;}^{j \in 1..n}\} \end{array}}{P \vdash FieldInit(T)}$$

Figure 21: Well-formedness for inheritance, part 1.

Definition 25 computes which parameters a type T passes to one of its supertypes T' if its formal component parameters are substituted with elaborated component parameters $\overline{\gamma_i}^i$. Rule 25.1 first elaborates the names of component relations in the incoming parameters, given $\overline{\gamma'_i}^i$. It then substitutes the formal component parameters in a subtype relation sub_k and recurses. Note that the computation can only be done if $T <: T'$, because otherwise no suitable k can be found, and no rule can process `Object` as that class cannot be defined according to the well-formedness rules. A well-formedness rule ensures that the choice of k is irrelevant since the result is the same for any valid k. Rule 25.2 terminates the computation.

4.2.2 Well-formedness Rules

To prove the inheritance theorems, we must impose additional well-formedness rules. These are shown in Figures 21 and 22.

First, all members of a class as defined by $mem_P(C)$ must have different names, and the constructor must initialize all fields of the class. Further, if $\mathcal{M}$ overrides $\mathcal{M}'$, then $P \vdash \mathcal{M} \leq \mathcal{M}'$ must hold.

Component relations and subclass relations have most of their rules in common. For both relations, the inherited class must exist, the actual component parameters must be valid for the corresponding formal component parameters of the inherited class, and no method can be renamed more than once in the configuration block. In addition, no loops are allowed in the graph containing the

$P \vdash NoSTLoops(cls)$

$$\text{(29.1)}\quad \frac{\begin{array}{l} 1.\ cls = \textbf{class}\ T\ (\overline{\alpha_i}^{\,i})\ \overline{sub}\ body \\ 2.\ \overline{sub} = \overline{\textbf{subclass}\ U_j\ (\overline{\delta})_j\ conf_j}^{\,j\in 1..n} \\ 3.\ \forall k \in 1..n : \neg P \vdash U_k <: T \end{array}}{P \vdash NoSTLoops(cls)}$$

$P \vdash NoCOLoops(cls)$

$$\text{(30.1)}\quad \frac{\begin{array}{l} 1.\ cls = \textbf{class}\ T\ (\overline{\alpha_i}^{\,i})\ \overline{sub_j}^{\,j}\ \{\ ctr\ \overline{def_k}^{\,k\in 1..n}\ \} \\ 2.\ \overline{z_l \mathfrak{C}_l}^{\,l\in 1..o} = \{z | \exists k \in 1..n : z\ \mathfrak{C} = def_k\} \\ 3.\ \forall l \in 1..o : z_l\ \mathfrak{C}_l = c_l\ \textbf{component}\ C_l\ (\overline{\delta})_l\ conf_l \\ 4.\ \forall l \in 1..o : \neg P \vdash C_l <: T \\ 5.\ \forall l \in 1..o : \neg P \vdash C_l < T \end{array}}{P \vdash NoCOLoops(cls)}$$

$P \vdash ValidST(T, sub)$

$$\text{(31.1)}\quad \frac{1.\ sub = \textbf{subclass}\ \textbf{Object}\ (\,)\ [\ ]}{P \vdash ValidST(T, sub)}$$

$$\text{(31.2)}\quad \frac{\begin{array}{l} 1.\ sub = \textbf{subclass}\ U\ (\overline{\delta_i}^{\,i\in 1..n})\ [\overline{z = z'}] \\ 2.\ P \vdash T \rightarrow \textbf{class}\ U\ (\overline{\alpha_i}^{\,i\in 1..n})\ \overline{sub}\ body \\ 3.\ \forall i \in 1..n : P, T \vdash \delta_i \leq \alpha_i \\ 4.\ z_i = z_j \Leftrightarrow i = j \end{array}}{P \vdash ValidST(T, sub)}$$

$P \vdash ConsistentCParams(cls)$

$$\text{(32.1)}\quad \frac{\begin{array}{l} 1.\ cls = \textbf{class}\ T\ (\overline{p_i\ V_i \rightarrow V'_i}^{\,i})\ \overline{sub_j}^{\,j\in 1..n}\ body \\ 2.\ cls \in P \\ 3.\ \forall j \in 1..n : sub_j = \textbf{subclass}\ U_j\ (\overline{\delta})_j\ conf_j \\ 4.\ \ \forall T' \in P : \forall k, l \in 1..n : \\ \quad (P \vdash U_k <: T' \wedge P \vdash U_l <: T') \implies \\ \qquad (P \vdash params([\overline{T \diamond p_i / p_i}^{\,i}]\ (\overline{\delta})_k, U_k, T') = \overline{\gamma_o}^{\,o} \wedge \\ \qquad P \vdash params([\overline{T \diamond p_i / p_i}^{\,i}]\ (\overline{\delta})_j, U_l, T') = \overline{\gamma'_o}^{\,o} \wedge \\ \qquad \forall l \in 1..o : P \vdash \gamma_l \vdash \gamma'_l \equiv) \end{array}}{P \vdash ConsistentCParams(cls)}$$

$P \vdash NoSubtypeDup(cls)$

$$\text{(33.1)}\quad \frac{\begin{array}{l} 1.\ cls = \textbf{class}\ T\ (\overline{\alpha_i}^{\,i})\ \overline{sub_j}^{\,j\in 1..n}\ body \\ 2.\ cls \in P \\ 3.\ \forall j \in 1..n : pot_{p,st}(T, sub_j) = \overline{\mathcal{M}}_j \\ 4.\ \forall T_1\ z_1\ val_1, T_2\ z_2\ val_2 \in \bigcup^{j\in 1..n} \overline{\mathcal{M}}_j : (P \vdash T_1\ z_1\ val_1 \leftrightsquigarrow T_2\ z_2\ val_2 \Rightarrow z_1 = z_2) \end{array}}{P \vdash NoSubtypeDup(cls)}$$

$P \vdash WF_{m,int}(\mathcal{M})$

$$\text{(34.1)}\quad \frac{}{P \vdash WF_{m,int}(T\ f\ U\ ;)}$$

$$\text{(34.2)}\quad \frac{}{P \vdash WF_{m,int}(T\ m\ U\ (\overline{X_i x_i}^{\,i\in 1..n})\ \{e\ ;\})}$$

$$\text{(34.3)}\quad \frac{\begin{array}{l} 1.\ \textbf{class}\ C\ (\overline{\alpha})\ \overline{sub}\ body \in P \\ 2.\ P, T \vdash \overline{\delta} \leq \overline{\alpha} \\ 3.\ z_i = z_j \Leftrightarrow i = j \\ 4.\ \forall C\ f\ U\ ; \in mem_p(C) : f \in \overline{path} \end{array}}{P \vdash WF_{m,int}(T\ c\ \textbf{component}\ C\ (\overline{\delta_i}^{\,i\in 1..n})\ [\overline{z = z'}])}$$

Figure 22: Internal program well-formedness.

classes as nodes, and all component relations and subclass relations as directed edges.

For component relations, there is an additional constraint that requires that all fields are inherited directly to satisfy Axiom I_4. For subclass relations, there is an additional constraint to prevent duplication of members in the inheritance hierarchy. In addition, for every member in the superclass with a simple name, there must be a related member in the subclass with a simple name. This preserves the direct interface of the class. For all different members $\mathcal{M}, \mathcal{M}' \in mem_p(T)$, that are related ($P \vdash \mathcal{M} \leftrightsquigarrow \mathcal{M}'$), there must either exist an overrides relation or an 'inherited as' relation between them in either direction: $P \vdash \mathcal{M} \ggg \mathcal{M}' \vee P \vdash \mathcal{M}' \ggg \mathcal{M} \vee P \vdash \mathcal{M} \leftarrow \mathcal{M}' \vee P \vdash \mathcal{M}' \leftarrow \mathcal{M}$. If that is not the case, there is an ambiguity that cannot be resolved by the rule of dominance, which means that lookup rules STATIC and PATH of Figure 18 cannot select the single most specific member. In case of rule STATIC, this well-formedness rule ensures that no duplication is allowed in a subclass hierarchy, by requiring that related members are given the same name, and that the members are overridden if no single most specific version is inherited. In case of rule PATH, this well-formedness rule ensures that in the scenario in Figure 16a, `setFrequency` is overridden in `SmoothRadio`, and that in the scenario in Figure 16b, the `volume` component relation of `SmoothRadio` directly inherits `SmoothBoundedValue.getX` as `getVolume`.

For actual component parameters, there are two constraints. First, they must satisfy the constraint of the corresponding formal component parameter $p\ U \rightarrow U'$ in the inherited class. If the actual parameter is the name of a component relation (c), class U must have a component relation with name c and type C with $C <: U'$. If the actual parameter itself is a component parameter $p'\ V \rightarrow V'$ from the inheriting class, then $U <: V$ and $V' <: U'$ must hold. Second, for each class C, all actual component parameters that are passed to the same formal component parameter $p\ U \rightarrow U'$ of class T via different subclass paths must be equal. This rule is similar to the rule in Java that requires that all actual parameters that a class passes to the same generic parameter via different inheritance paths must be equal. This rule allows us to prune redundant members in Figure 14 since the transformation of identical definitions with the same actual component parameters produce identical new definitions. Note that we do not require that the actual component parameters of a component relation are equal to the corresponding actual parameters of the component relations that it overrides. This is not required for type soundness.

5 Proof of Type Soundness

In this section we give the proof of type soundness for the base calculus. The proof for the concrete inheritance module for first-class composition inheritance is presented in Section 6.

5.1 Subject Reduction

Theorem 5.1 (Subject Reduction) *If* $P = \langle \overline{cls_i}^i, e\rangle$ *and* $WF(P)$ *and* $P, \Gamma \vdash e : T$ *and* $\langle \overline{cls_i}^i, e\rangle \hookrightarrow \langle \overline{cls_i}^i, e'\rangle$ *then* $P, \Gamma \vdash_s e' : T$ *and* $WF(\langle \overline{cls_i}^i, e'\rangle)$.

PROOF 5.1.

Case: field access $e = \underline{\textbf{new}\ C(\overline{e})\#ref : T'}.f$ From $P, \Gamma \vdash \underline{\textbf{new}\ C(\overline{e})\#ref : T'}.f : T$, we know that:

1. $P, \Gamma \vdash \textbf{new}\ C(\overline{e})\#ref : U$
2. $P \vdash U <: T'$

3. $\mathcal{L}_P(U, f, T') = U\ f'\ T$

From $e \hookrightarrow e'$, and Theorem 6.13, we know that:

1. **class** $C\ (\overline{\alpha})\ \overline{sub}\{C(\overline{Xx})\{\overline{\mathbf{this}.f = x;}\}\ \overline{def}\} \in P$
2. $\mathscr{L}_P(C, ref, f, T') = C\ f'_j\ X_j$
3. $e' = e_j$.

We must prove that $P, \Gamma \vdash_s e_j : T$. From Theorem I_3 we know that the result of the dynamic lookup conforms to that of the static lookup. Therefore $P \vdash X_j <: T$. It then follows that $P, \Gamma \vdash_s e_j : T$. Because the program was already well-formed, and no new elements were added, it remains well-formed.

Case: method invocation $e = \underline{\mathbf{new}\ C(\overline{e})\#ref : T'}.m(\overline{e'})$ From $P, \Gamma \vdash \underline{\mathbf{new}\ C(\overline{e})\#ref : T'}.m(\overline{e'} : T$, we know that:

1. $P, \Gamma \vdash \mathbf{new}\ C(\overline{e})\#ref : U$
2. $P \vdash U <: T'$
3. $\mathcal{L}_P(U, f, T') = U\ m'\ T(\overline{Xx})\{e_2;\}$
4. $P, \Gamma \vdash_s \overline{e'} : \overline{X}$

From $e \hookrightarrow e'$, and Theorem I_3, we know that

1. $\mathscr{L}_P(C, ref, f, T') = C\ path\ V(\overline{Xy}\{e_3\})$
2. $e' = [\mathbf{new}\ C(\overline{e})\#ref/\mathbf{this}, \overline{e'}/\overline{y}]e_3$.

From Theorem I_3, we know that $P \vdash V <: T$. From Theorem I_2, we know that $P, \Gamma, \overline{y : X}, \mathbf{this} : C \vdash e_3 : V'$ and $P \vdash V' <: V$. From Lemma 5.2, we thus know that $P, \Gamma \vdash_s [\mathbf{new}\ C(\overline{e})\#ref/\mathbf{this}, \overline{e'}/\overline{y}]e_3 : T$. From Theorem I_2 and $WF_{sound}(P)$, we also know that no expression of the form $e@\underline{T : p}$ is in e_3, and thus not in e', which means that the program is still well-formed.

Case: subobject reference $e = \underline{\mathbf{new}\ C(\overline{e})\#ref : T'} \diamond c$ From $P, \Gamma \vdash \underline{\mathbf{new}\ C(\overline{e})\#ref : T'} \diamond c : T$, we know that:

1. $P, \Gamma \vdash \mathbf{new}\ C(\overline{e})\#ref : U$
2. $P \vdash U <: T'$
3. $\mathcal{L}_P(U, c, T') = c'\ \mathbf{component}\ T\ (\overline{\delta})\ conf$

From $e \hookrightarrow e'$, and Theorem I_3, we know that

1. $\mathscr{L}_P(C, ref, f, T') = C\ path\ \mathbf{component}\ V\ (\overline{\delta})\ conf$
2. $e' = \mathbf{new}\ C(\overline{e})\#path$

From Theorem I_3, we know that $P \vdash V <: T$. From Theorem I_2, we know that the result of the dynamic lookup is a member of class C, and $path \neq \emptyset$. Therefore, from the typing judgement, it follows that $P, \Gamma \vdash_s \mathbf{new}\ C(\overline{e})\#path : T$.

Case: cast Follows trivially from $P, \Gamma \vdash (T)e : T$ and $e \hookrightarrow e'$.

Case: congruence rules Follow from induction on the derivation $e'' \hookrightarrow e'''$, where e'' is the subexpression being evaluated.

Lemma 5.2 (Term substitution preserves typing)
$P, \Gamma, \overline{x} : \overline{X} \vdash e : T \wedge P, \Gamma \vdash_s \overline{e'} : X \Rightarrow P, \Gamma \vdash_s [\overline{e'/x}]e : T$

PROOF 5.2. Proven by induction on the derivation $P, \Gamma, \overline{x} : \overline{X} \vdash e : T$

Case: field access $\underline{e : T'}.f$ From $P, \Gamma, \overline{x} : \overline{X} \vdash \underline{e : T'}.f : T$ we know that:

1. $P, \Gamma \vdash e : C$
2. $P \vdash C <: T'$
3. $\mathcal{L}_P(C, f, T') = U\ f'\ T$

From induction and the definition of $P, \Gamma \vdash_s e : T$, we know that:

1. $P, \Gamma \vdash [\overline{e'/x}]e : U$
2. $P \vdash U <: T'$

From Theorem I_3 and Theorem 6.13 and the definition of $\leq$, we know that $\mathcal{L}_P(U, f, T') = U\ f''\ T''$ and $T'' = T$. Therefore, $P, \Gamma \vdash_s [\overline{e'/x}]e : T$.

Case: method invocation $\underline{e : T'}.m(\overline{e'})$ From $P, \Gamma \vdash \underline{e : T'}.m(\overline{e'} : T)$, we know that:

1. $P, \Gamma \vdash e : C$
2. $P \vdash C <: T'$
3. $\mathcal{L}_P(C, f, T') = U\ m'\ T(\overline{Xx})\{e_2;\}$
4. $P, \Gamma \vdash_s \overline{e'} : \overline{X}$

From induction and the definition of $P, \Gamma \vdash_s e : T$, we know that:

1. $P, \Gamma \vdash_s [\overline{e''/x}]e : T'$
2. $P, \Gamma \vdash_s [\overline{e''/x}]e' : \overline{X}$

From Theorem I_3 andthe definition of $\leq$, we know that $\mathcal{L}_P(U, m, T') = U\ m''\ T''(\overline{Vx})$ and $P \vdash T'' <: T$. Therefore, $P, \Gamma \vdash_s [\overline{e''/x}]\underline{e : T'}.m(\overline{e'}) : T$.

Case: subobject reference Similar to the previous cases.

Case: constructor invocations Straightforward induction, since the type of the constructed object does not change. Therefore, it must only be verified that the preconditions of the typing rules still apply. This is straightforward.

Case: cast Straightforward induction.

Case: component parameter Straightforward induction because the elaborated type is not that of the expression, that that of the component parameter..

Case: variable Follows directly from Lemma 5.3.

Lemma 5.3 (Weakening) $P, \Gamma \vdash e : T \Rightarrow P, \Gamma, x : X \vdash e : T$

PROOF 5.3. Straightforward

Theorem 5.4 (Progress) *If* $P = \langle \overline{cls}, e \rangle \wedge WF(P)$ *then either* $e \hookrightarrow e'$ *or* e *is a value, or* e *is or contains an invalid cast* $(U)\mathbf{new}\ C(\overline{e''})\#ref$ *with* $P, \Gamma \vdash \mathbf{new}\ C(\overline{e''})\#ref : T'$ *and* $P \vdash T' \not<: U$.

PROOF 5.4.

Case: field access From Theorem I_3, it follows that the lookup succeeds. From Theorem 6.13, it follows that it will have a simple name. Therefore, the evaluation can proceed.

Case: method invocation Follows straight from Theorem I_3.

Case: subobject reference Follows straight from Theorem I_3.

Case: others Straightforward.

6 Proof of Inheritance Axioms

In this section, we prove that the axioms of the inheritance signature hold for the concrete inheritance module that implements first-class composition inheritance. The theorems are the same as the inheritance axioms.

Theorem 6.1 (Subtyping Transitive) $(WF(P) \wedge P \vdash T_1 <: T_2 \wedge P \vdash T_2 <: T_3) \Rightarrow P \vdash T_1 <: T_3$

PROOF 6.1. Follows directly from the definition.

Theorem 6.2 (Lookup Well-Formed) $WF(P) \wedge (\mathscr{L}_P(C, ref, z, T) = \mathcal{M} \vee \mathcal{L}_P(C, z, T) = \mathcal{M}) \Rightarrow (\mathcal{M} = C\ path\ val \wedge \mathcal{M} \in mem_p(C) \wedge WF_{m,sound}(\mathcal{M}))$

PROOF 6.2. Follows directly from Lemmas 6.3 and 6.4.

Lemma 6.3 $(WF(P) \wedge \mathscr{L}_P(C, ref, z, T) = \mathcal{M}) \Rightarrow (\mathcal{M} = C\ path\ val \wedge \mathcal{M} \in mem_p(C) \wedge WF_{m,sound}(\mathcal{M}))$

PROOF 6.3. If $ref = \emptyset$, the proof follows straight from Lemma 6.4. If $ref = path$, it follows from Lemma 6.5.

Lemma 6.4 $(WF(P) \wedge \mathcal{L}_P(C, z, T) = \mathcal{M}) \Rightarrow (\mathcal{M} = C\ path\ val \wedge \mathcal{M} \in mem_p(C) \wedge WF_{m,sound}(\mathcal{M}))$

PROOF 6.4. Follows directly from Lemma 6.5.

Lemma 6.5 $(WF(P) \wedge \mathcal{M} \in mem_P(C) \Rightarrow (\mathcal{M} = C\ path\ val \wedge \mathcal{M} \in mem_p(C) \wedge WF_{m,sound}(\mathcal{M}))$

PROOF 6.5. Since the program is well-formed, and member are only transformed through dir_P, the proof follows directly from Lemma 6.6.

Lemma 6.6
$P \vdash WF(val) \wedge transform_p(C, val, T, e, \overline{\delta}) = val' \wedge \overline{\delta} \leq component\ parameters\ \overline{p}\ of\ T \wedge P, \Gamma \vdash_s e : C \Rightarrow WF(val')$

PROOF 6.6.

Case: field Fields are not changed.

Case: component Regular transformation is done, and δ must be valid actual component parameters.

Case: method Follows from Lemmas 5.2 and 6.7.

Lemma 6.7 *$[\![\delta/p]\!]_e C = e'$ preserves typing in a well-formed program.*

PROOF 6.7. Straightforward induction using the fact that the program is well-formed.

Theorem 6.8 (Lookup Successful and Conform) $(WF(P) \ \wedge \ \mathcal{L}_P(U, z, T) \ = \ \mathcal{M} \ \wedge \ P, \Gamma \ \vdash_s$ **new** $C(\overline{e})\#ref : U) \Rightarrow (\exists!\, \mathcal{M}' : \mathscr{L}_P(C, ref, z, T) = \mathcal{M}') \wedge (\mathscr{L}_P(C, ref, z, T) \leq \mathcal{M})$

PROOF 6.8.

$ref = \emptyset$**:** It follows from Lemma 6.9 that the result of the dynamic lookup is the same as that of the static lookup, and thus it exists.

$ref = path$**:** From Lemma 6.9, we know that the static lookup give exactly one result $\mathcal{M}'$. From the definition of mem_P and its auxiliary functions, we know that the result of the dynamic lookup is a transformation of $\mathcal{M}'$. From Lemma 6.10, we know that the result of the dynamic lookup conforms to that of the static lookup. From the well-formedness rules, we know that there exists a $\mathcal{M}$ such that it overrides the flattened version of $\mathcal{M}'$, or $\mathcal{M}'$ is inherited as $\mathcal{M}$, so there is at least one result. From the definitions of these relations, we know that they are mutually exclusive, so the result is unique.

Lemma 6.9 *$\mathcal{L}_P$ is a function.*

PROOF 6.9. Because we perform the lookup on a simple name, we know from the well-formedness rules that the selected member is either defined in the dynamic type, or inherited through subclassing. The proof then follows from Lemma 6.11.

Lemma 6.10 *Member transformation implies conformance.*

PROOF 6.10.

Case: field: Fields are not transformed.

Case: component: The substitution of the component parameters has no influence on conformance.

Case: method: The transformation of the method body has no influence on conformance.

Lemma 6.11 (Subclassing leaves unique trail of member relations) $P \vdash C <: T \wedge T\ z\ val \in mem_P(T) \Rightarrow \exists! \mathcal{M} \in mem_P(C) : P \vdash \mathcal{M} \ggg T\ z\ val \vee P \vdash \mathcal{M} \equiv T\ z\ val$

PROOF 6.11.

Case: C=T In this case, the proof is trivial.

Case: $C \neq T$ In this case, the proof follows from induction on the intermediate class X in the inheritance hierarchy and Lemma 6.12.

Lemma 6.12 (Direct subclassing leaves unique trail of member relations)
class C $(\overline{\alpha})$ $\overline{sub}\{C(\overline{X x})\{\overline{\mathbf{this}.f = x;}\}$ $\overline{def}\} \in P \wedge$ **subclass** T $(\overline{\delta}) conf \in \overline{sub} \wedge T\ z\ val \in mem_P(T) \Rightarrow \exists! \mathcal{M} \in mem_P(C) : P \vdash \mathcal{M} \ggg T\ z\ val \vee P \vdash \mathcal{M} \leftarrow T\ z\ val$

PROOF 6.12. This follows directly from the definitions of the overrides and the 'inherited as' relations.

Theorem 6.13 (All Fields Initialized) (**class** C $(\overline{\alpha})$ $\overline{sub}\{C\ (\overline{X\ x})\{\overline{this.x = x}\}\overline{def}\} \in P \wedge \mathscr{L}_P(C, ref, z, T) = C\ path\ U;) \Rightarrow (path = x_i \wedge U = X_i)$

PROOF 6.13. Follows directly form the well-formedness rules that require that all fields are inherited directly, and that the constructor initializes all fields with a simple name.

7 Related Work

In the paper that introduced first-class composition inheritance [47], we focused on comparing the mechanism with other language constructs. In this paper, we focus on a comparison of its type system O_2 with other calculi for inheritance mechanisms.

Odersky et. al. present HM(X) [35], which is a framework for Hindley/Milner type systems with constraints, which is parametric in its constraint system X. The authors show that under certain conditions of X, the type inference algorithm of HM(X) always computes the principal type of a term. The authors also present a concrete constraint system that works with subtyping.

Nystrom et. al. present the X10 programming language [34], which is an object-oriented programming language that, similar to HM(X), is parametric in its constraint system. They formalize the core of the language in CFJ, which is based on Featherweight Java. CFJ adds constrained types and methods to Featherweight Java. CFJ is defined is terms of a constraint system C which provides a number of predefined relations in addition to custom formulas and predicates. To ensure decidability of constraint checking, the constrains must be pure functions of the properties of the base class. The properties of a class are final instance fields defined in a special clause of the class.

Fagorzi and Zucca present the R calculus to formalize component based systems [16]. A component is represented by a module, which can be replaced at run-time. The R calculus is parametric in its core language. The parameterization is thus done in the other direction compared to HM(X), CFJ, and O_2. The authors provide two calculi R^{rcv}_{hide} and R^{rcv}_{cstr} to deal with unintentional name conflicts. The former calculus solves the problem hiding functionality that was not required, the latter calculus solves it using a more sophisticated type system that does constraint checking.

Although the parameterizations in HM(X), CFJ, and R have a difference purpose than that of O_2, the general idea is the same. All four approaches define a calculus that guarantees certain properties as long the parameterized part satisfies a predefined condition, such that the calculus can easily be reused. An interesting challenge is to combine the approach of R with the other approaches, and create a calculus that is parametric in both the core language, and the inheritance mechanism.

Igarashi et. al. present Featherweight Java in [24], which is a core calculus for Java, and was the starting point for the O_2 calculus. To keep the calculus simple, it does not support assignments, interfaces, or syntactic overloading. The authors extend their own calculus to support parametric polymorphism, resulting in Featherweight Generic Java, which is much more complicated. While support for parametric polymorphism is crucial for first-class composition inheritance in a real programming language, we had to use the core calculus to keep things manageable.

Flatt et. al. present ClassicJava in [18], which is another widely used core calculus for Java. Unlike Featherweight Java, however, it does support assignment, interfaces, and syntactic overloading. The authors use type elaboration to incorporate the static type of expressions into the run-time model since they support syntactic overloading and `super` calls. We borrowed this technique to support renaming. The authors extend ClassicJava with support for mixins, resulting in MixedJava. To avoid ambiguities in MixedJava, they replaced the subclass relation with a *viewable as* relation that only holds if there is only one path between both operands in the type graph. To correctly resolve method invocations in case of name conflicts, a value consists of an object and a chain of mixins called a *view*, which is similar to subobject references in CoreC++ [48] and O_2. Instead of congruence rules, the evaluation rules use an expression context to define the order in which (sub)expression must be evaluated.

Wasserrab et. al. present the CoreC++ calculus, which formalizes multiple inheritance in C++ [48]. The subobjects that result from repeated inheritance are represented by a list of super classes, similar to [26, 27], which works for C++ because directly inheriting twice from the same class is not allowed. This is similar to our approach, only we use the list of names of the component relations to represent a specific subobject – we do not allow repeated inheritance for subclass inheritance. CoreC++ supports assignment because it is required to correctly formalize method and field lookup in C++, as it interacts with casts and assignments. Upcasts in C++ are similar to the use of subobject references in O_2, but in C++ they can cause ambiguities at run-time because of repeated inheritance in the subclassing hierarchy, as there is no way to tell which subobject must be chosen. The authors made a machine checked proof in Isabelle/HOL to prove the type soundness of CoreC++.

Reppy and Turon formalize trait based metaprogramming in [39], where they add deep hiding and renaming to traits [41]. Their Meta-Trait Java calculus (MTJ) is based on Featherweight Generic Java. Names of trait members can be parameterized, which is similar to renaming in O_2. Their type system is a hybrid of a nominal and a structural type system because of the structural nature of traits. Types are denoted as $N \diamond \sigma$ where N is the name of the type and σ its structural signature. The nominal type system is used for the argument types and the return type of a method, while the structural type system is used for field accesses and method invocations. To deal with the presence of name parameters in a type, they introduce a limited form of dependent types. A trait can impose a constraint on the class that reuses that trait by imposing a bound on the special type $ThisType$. This is similar to self types in Scala [36]. A similar construct can be used to get rid of the `getOuter` method of Figure 4, though it must be adapted to cope with a hierarchy of subobjects since $ThisType$ always references the type of the outer object. Because traits can be flattened, the lookup mechanism of the host language does not have to be modified. Therefore they can easily express the semantics of MTJ as a translation to FGJ during which traits are flattened. We cannot use this approach for O_2 because the lookup mechanism differs significantly from that of FJ. The resulting translation would be far more difficult to understand than the current O_2 calculus.

Ducasse et. al. introduce freezable traits in [12] to prevent forced merging of methods during the composition of traits. Normally, all methods in a trait are late bound, and because invocations are only based on method signatures, this makes it impossible to resolve certain name conflicts. By using the *freeze* operator, a method is removed from the interface of a trait, and bound statically within that trait. As a result, the trait still uses the correct method when composed with another trait that defines a method with the same signature. The *defrost* operator reverses the effect. Freezing, however does not work if different versions of the same method are inherited via different paths because that could cause invocations of the original method to become ambiguous in the trait where that method is defined. In that case, the conflict must be resolved through exclusion or overriding the involved methods. With first-class composition inheritance, this problem cannot occur for two reasons. First, methods are always bound within the same inheritance path in case of a component relation, so merging is

never mandatory. Second, members from different component relation are inherited once for each component relation, but once they are merged they can never be separated again. Therefore, no ambiguity can arise. The semantics of the accompanying FreezableTraits calculus is expressed as a translation to the SmalltalkLite calculus [4], which is based on ClassicJava. SmalltalkLite supports single inheritance, message passing, field access. and update, and *self* and *super* sends. Because of the support for super sends, static information must be incorporated in the run-time program, but in SmalltalkLite this elaboration is done in the evaluation rules. This approach, however, is based on the assumption that method bodies remain unchanged throughout the inheritance hierarchy, making it hard to replace the inheritance mechanism if that assumption does not hold – as is the case with first-class composition inheritance.

The calculi for inheritance mechanisms of object-oriented programming languages discussed above do not modularize the inheritance mechanism, which means that the type soundness proofs must be completely verified from scratch if the inheritance mechanism is changed or replaced by a different inheritance mechanism. Given that the inheritance mechanism is a crucial part of an object-oriented programming language, we consider this to be an important contribution.

Virtual classes [29] provide techniques such as family polymorphism [13] and higher-order hierarchies [37, 14] that are orthogonal to typical inheritance mechanisms. A family is a set of classes that depend on each other for the extended functionality. Though already in use for a long time [29], soundness of virtual classes has only recently been proved by Ernst et. al. [36]. Recent work has been done by Gasiunas et. al. to make the approach more flexible [19], and by Igarashi and Viroli to achieve the same without a dependent type system [25].

Tobin-Hochstadt and Allen present the MCJ core calculus of metaclasses [45], which allows the construction of arbitrary hierarchies with '*instance of*' relations. The 'instance of' relation can capture aspects of the world that cannot otherwise be expressed. In addition to the *extends* relation for subtyping and code inheritance, they use a *kind* relation to declare that a class is an instance of another class. The MCJ calculus is based on Featherweight Generic Java.

8 Conclusion

First-class composition inheritance is the first technique to make composition of classes practical. It allows general purpose behavior and collaborations to be encapsulated in classes and reused as components to build other classes, increasing code reuse.

In this paper, we presented the O_2 calculus. The calculus formalizes first-class composition inheritance, multiple subclass inheritance, overriding, renaming, and merging of class members, direct and indirect inheritance, subobject references, and component parameters for connecting components. Furthermore, the O_2 calculus is parametric in its inheritance mechanism. Therefore, the type soundness proof must not be reverified if the inheritance mechanism is modified or replaced by another. This is an important improvement given the importance of inheritance in object-oriented programming.

Acknowledgments

We are very grateful to Adriaan Moors, Erik Ernst, and Dave Clarke who gave a lot of feedback on the text and the calculus.

References

[1] D. Ancona, S. Fagorzi, and E. Zucca. Mixin modules for dynamic rebinding. In *in: TGC 2005 -Symposium on Trustworthy Global Computing, Lecture Notes in Computer Science*, pages 279–298. Springer, 2005.

[2] I. Attali, D. Caromel, and S. O. Ehmety. A natural semantics for eiffel dynamic binding. *ACM Trans. Program. Lang. Syst.*, 18(6):711–729, 1996.

[3] D. Bardou and C. Dony. Split objects: a disciplined use of delegation within objects. In *OOPSLA*, pages 122–137. ACM Press, 1996.

[4] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful traits and their formalization. *Comput. Lang. Syst. Struct.*, 34(2-3):83–108, 2008.

[5] G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP*, pages 303–311, New York, NY, USA, 1990. ACM Press.

[6] G. Bracha and G. Lindstrom. Modularity meets inheritance. In *IEEE International Conference on Computer Languages*, pages 282–290, 1992.

[7] C. Chambers. The Cecil language specification and rationale: Version 3.2. 2004.

[8] C. Chambers. The Diesel language specification and rationale: Version 0.2. 2006.

[9] C. Chambers, D. Ungar, B.-W. Chang, and U. Hölzle. Parents are shared parts of objects: inheritance and encapsulation in SELF. *Lisp Symb. Comput.*, 4(3):207–222, 1991.

[10] D. Colnet, G. Marpons, and F. Merizen. Reconciling subtyping and code reuse in object-oriented languages: Using inherit and insert in SmartEiffel. In *ICSR*, 2006.

[11] L. G. DeMichiel and R. P. Gabriel. The common lisp object system: An overview. In *ECOOP*, pages 151–170, 1987.

[12] S. Ducasse, R. Wuyts, A. Bergel, and O. Nierstrasz. User-changeable visibility: resolving unanticipated name clashes in traits. In *OOPSLA*, pages 171–190, 2007.

[13] E. Ernst. Family polymorphism. In J. L. Knudsen, editor, *ECOOP*, LNCS 2072, pages 303–326, Heidelberg, Germany, 2001. Springer-Verlag.

[14] E. Ernst. Higher-order hierarchies. In L. Cardelli, editor, *ECOOP*, LNCS 2743, pages 303–329, Heidelberg, Germany, July 2003. Springer-Verlag.

[15] E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. In *POPL*, pages 270–282, New York, NY, USA, 2006. ACM.

[16] S. Fagorzi and E. Zucca. A calculus of components with dynamic type-checking. *Electron. Notes Theor. Comput. Sci.*, 182:73–90, 2007.

[17] M. Flatt, R. B. Findler, and M. Felleisen. Scheme with classes, mixins, and traits. In *APLAS*, pages 270–289, 2006.

[18] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *POPL*, pages 171–183. ACM Press, 1998.

[19] V. Gasiunas, M. Mezini, and K. Ostermann. Dependent classes. In *OOPSLA*, pages 133–152, New York, NY, USA, 2007. ACM.

[20] D. S. Goldberg, R. B. Findler, and M. Flatt. Super and inner: together at last! In *Proceedings of OOPSLA '04*, pages 116–129, 2004.

[21] J. Gordon S. Novak. GLISP users' manual. Technical report, Stanford, CA, USA, 1982.

[22] J. Gordon S. Novak. Data abstraction in GLISP. In *Symposium on Programming language issues in software systems*, pages 170–177, New York, NY, USA, 1983. ACM.

[23] J. Gosling et al. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.

[24] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–450, 2001.

[25] A. Igarashi and M. Viroli. Variant path types for scalable extensibility. In *OOPSLA*, pages 113–132, New York, NY, USA, 2007. ACM.

[26] J. Jonathan G. Rossie and D. P. Friedman. An algebraic semantics of subobjects. In *OOPSLA*, pages 187–199, 1995.

[27] J. Jonathan G. Rossie, D. P. Friedman, and M. Wand. Modeling subobject-based inheritance. In *ECOOP*, pages 248–274, 1996.

[28] J. L. Keedy, C. Heinlein, and G. Menger. Inheriting multiple and repeated parts in Timor. *JOT*, 3(10):99–120, 2004.

[29] O. L. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA*, pages 397–406, New York, NY, USA, 1989. ACM.

[30] B. Meyer. Overloading vs. object technology. *Journal of Object-Oriented Programming*, October 2001.

[31] M. Mezini and K. Ostermann. Integrating independent components with on-demand remodularization. In *OOPSLA*, pages 52–67, New York, NY, USA, 2002. ACM.

[32] N. Nystrom, S. Chong, and A. C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA*, pages 99–115, New York, NY, USA, 2004. ACM Press.

[33] N. Nystrom, X. Qi, and A. C. Myers. J&: nested intersection for scalable software composition. In *OOPSLA*, pages 21–36, New York, NY, USA, 2006. ACM.

[34] N. Nystrom, V. Saraswat, J. Palsberg, and C. Grothoff. Constrained types for object-oriented languages. In *OOPSLA*, pages 457–474, New York, NY, USA, 2008.

[35] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theor. Pract. Object Syst.*, 5(1):35–55, 1999.

[36] M. Odersky and M. Zenger. Scalable component abstractions. In *OOPSLA*, pages 41–57, New York, NY, USA, 2005. ACM.

[37] H. Ossher and W. Harrison. Combination of inheritance hierarchies. In *OOPSLA*, pages 25–40, New York, NY, USA, 1992. ACM Press.

[38] K. Ostermann. Dynamically composable collaborations with delegation layers. In *ECOOP*, pages 89–110, London, UK, 2002. Springer-Verlag.

[39] J. H. Reppy and A. Turon. Metaprogramming with traits. In *ECOOP*, pages 373–398, 2007.

[40] M. Sakkinen. Disciplined inheritance. In *ECOOP*, pages 39–56, 1989.

[41] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *ECOOP*, pages 248–274. Springer Verlag, July 2003.

[42] Y. Smaragdakis and D. S. Batory. Implementing layered designs with mixin layers. In *ECOOP*, pages 550–570, London, UK, 1998. Springer-Verlag.

[43] B. Stroustrup. *The C++ programming language (3rd ed.).* Addison-Wesley, 1991.

[44] Technical Group 4 of Technical Committee 39. *ECMA-367 Standard: Eiffel Analysis, Design and Programming Language*. ECMA International, 2005.

[45] S. Tobin-Hochstadt and E. Allen. A core calculus of metaclasses, 2005. FOOL.

[46] M. van Dooren and W. Joosen. A modular type system for first-class composition inheritance. Technical report, K.U.Leuven, December 2008. http://www.cs.kuleuven.be/~marko/technical.pdf.

[47] M. van Dooren and E. Steegmans. A higher abstraction level using first-class inheritance relations. In *ECOOP*, pages 425–449, 2007.

[48] D. Wasserrab, T. Nipkow, G. Snelting, and F. Tip. An operational semantics and type safety proof for multiple inheritance in C++. In *OOPSLA*, pages 345–362, 2006.

[49] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.