Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

# Lifted Java: A Minimal Calculus for Translation Polymorphism

Matthias Diehn Ingesman[a] Erik Ernst[a]

a. Department of Computer Science, Aarhus University, Denmark

**Abstract** To support roles and similar notions involving multiple views on an object, languages like Object Teams and CaesarJ include mechanisms known as lifting and lowering. These mechanisms connect pairs of objects of otherwise unrelated types, and enable programmers to consider such a pair almost as a single object which has both types. In the terminology of Object Teams this is called translation polymorphism. In both Object Teams and CaesarJ the type system of the Java programming language has been extended to support this through the use of advanced language features. The type soundness of translation polymorphism has so far only been proven in a simple special case. This paper presents a simple model that extends Featherweight Java with a general semantics that captures the core operations of translation polymorphism, providing an entire language design space for languages with translation polymorphism. Type soundness is proven for every language in this language design space, and mechanization of the proof in Coq shows that the proof is accurate and complete.

**Keywords** Formal foundations, language design, lifting/lowering, Translation Polymorphism, type systems

## 1 Introduction

In this paper we investigate the mechanisms lifting and lowering that provide a means to connect pairs of objects of otherwise unrelated types; mechanisms that have existed since 1998 [ML98, MSL00, Ost02], but have so far not been proven sound. The Object Teams/Java language (OT/J) [HHM10, Her07] calls them *translation polymorphism* [HHM04].

OT/J is an extension of the Java programming language [GJSB05] that facilitates non-invasive customisation through addition of code instead of modification. This is done by introducing two new types of classes called *teams* and *roles*. Roles solve many of the same problems as aspects [KLM$^+$97, KHH$^+$01], i.e. extension of existing code;

teams provide the means of controlling which roles are active, along with state that is shared between roles in the team. In other words teams provide the context for families of related roles, and in fact teams implement *family polymorphism* [Ern01]. Furthermore, teams can inherit and extend the roles of their super class, a feature known as *virtual classes* [MMP89, EOC06]. Each role is connected to a regular class, the *base class*, through a special `playedBy` relation, making these two objects seem almost like a single object. The mechanisms *lifting* and *lowering* use the `playedBy` relation and provide the translation between roles and base classes. In situations where a role is expected but a base class is given, lifting translates the base class object into the appropriate role. Similarly, if a base class object is expected but a role is given, lowering translates the role into the base class object. In both cases the role and the base are connected via the `playedBy` relation, either through *smart lifting* (OT/J) or through a flexible invariant on the `playedBy` relation (this calculus). In OT/J lifting works across inheritance hierarchies on both the role side and the base side. Smart lifting is an algorithm that lets the run-time system choose the most specific role for a base class. We note that smart-lifting makes it possible to make old code produce errors without modifying it, due to the fact that it tries to always provide the most specific role when lifting. This calculus features a general lifting operation that captures the behaviour of selecting the most specific role as a special case. OT/J is defined in terms of its implementation and a language specification document. A soundness proof for the extensions to the Java programming language type system has not been presented so far. For the full details on OT/J see [HHM10].

This is an extended version of our TOOLS'11 paper [IE11] to which we have added subtyping among roles, generalised the semantics of the calculus, and added the full soundness proof as an appendix. Role subtyping is important to be able to fully describe the lifting operation in ObjectTeams/Java, which dynamically chooses a more specific role than the statically requested. The generalised semantics show that there is in fact a large safe language design space for the semantics of the lifting operation. Thus, the main contributions of this paper are: a minimal calculus of translation polymorphism, along with a full soundness proof of this calculus; and a safe language design space for languages with translation polymorphism. The soundness proof is made using the Coq proof assistant [BC04], on the basis of a Featherweight Java (FJ) [IPW01] soundness proof by De Fraine *et al.* [DF09].

Excluding comments and empty lines, the modifications to the FJ source code amount to ~150 changed lines of code and ~1100 new[1]. To put these numbers into perspective, the original FJ source code is ~700 lines of code. The introduction of roles, in particular the part dealing with role subtyping, had a large impact overall, while lifting and lowering mainly resulted in an increase in the number of cases for the safety properties.

The concepts described in this paper are not specific to OT/J, and thus no previous knowledge of OT/J is required. However, we use some terminology of OT/J which will be explained as it is introduced. The rest of this paper is structured as follows. In section 2 we describe our choice of features for this calculus, give an example program, and describe the way objects are represented. Section 3 presents the calculus and gives the proof of standard type soundness. Section 4 discusses the semantics of lifting in more detail. In section 5 related and future work is discussed, and in section 7 the paper is concluded. Appendix A provides the full soundness proof.

[1]Reported by the 'diffstat' tool.

## 2 The Model

In this section we first argue why we do not model various features of OT/J. After that an example of a program written in the calculus is provided. The example is used to highlight some problems with the lifting operation that demand careful consideration, and we present our solution to these problems. Finally, because our representation of objects is non-standard, we conclude this section by describing objects.

We ignore all features of OT/J that are not at the core of translation polymorphism. Thus the following features are not part of the model: teams, team activation, call-in bindings, and call-out bindings.

Teams are not in the model because the only part they play in relation to translation polymorphism is to contain roles. Instead of being contained in teams roles are top-level classes. It may seem surprising that our model omits teams, because their semantics are at the core of the semantics of OT/J (just like classes containing `cclass`es are at the core of CaesarJ). However, we do not need to model the support for virtual classes in order to establish a universe which is sufficiently rich to support a model of lifting and lowering with a semantics that mirrors the behaviour of full-fledged languages. In fact, the connected pairs of roles and base objects in OT/J can simply be modelled as a *cloud* of objects with a label pointing to the currently *active* one. An object in our calculus is then such a cloud, which is just a finite set of objects of which one is an instance of a normal class (the base object), and the remaining objects are instances of role classes: the set of roles which the base class is currently playing. Such an object cloud works as the base object when its label points to the base object, and as a role object when its label points to one of the role objects. Lowering just means changing the label from one of the role objects to the base object, and lifting means changing the label from the base object to one of the roles in the cloud. In case the base object has not yet played the role which is requested in a lifting operation, a fresh instance of that role is created and added to the cloud. This semantics corresponds to a redistribution of the role objects in OT/J, where each team is responsible for storing existing roles of that team in some internal data structure managed by the language run-time. In this way, not modelling teams is in some sense equivalent to restricting OT/J to a single global and always active team, inside which every role is defined. Without teams there is no need for modelling the team activation constructs. As our aim is to stay close to the implementation of translation polymorphism in OT/J, in which a legal base class is not a role of the same team [HHM10], we do not allow roles to be `playedBy` another role.

Call-in and call-out bindings provide the *Aspect-Oriented Programming* features of OT/J, and are thus unrelated to the core of translation polymorphism. Lifting and lowering do occur inside these bindings, but not in a way that is different from regular method and constructor invocations.

To summarise, translation polymorphism is defined by roles and the operations lifting and lowering. Thus those are the concepts we add to FJ. Roles are restricted in that they cannot have state and cannot be explicitly instantiated. In OT/J roles by default only have a so-called *lifting constructor* for explicit instantiation. Such a constructor requires a single argument which is a fresh instance of the role's declared base class. This semantics corresponds to a call to $\texttt{lift(new C(}\bar{t}\texttt{),R)}$, and a programming language based on this calculus could add syntax for explicit role instantiation that compiles down to such an expression. Fields in roles are inessential because roles may still add non-trivial behaviour to a base object by accessing its fields and methods. Moreover, in a calculus that does not support mutable state, role

```
class Point extends Object {
  int x;
  int y;
  Point(int x, int y) { this.x = x; this.y = y; }
}

class Location extends Object playedBy Point {
  string getCountry() {
    int x = lower(this).x;
    int y = lower(this).y;
    string country = "DK"; // placeholder for (possibly advanced)
                           // computation converting a point in
                           // the plane to the name of a country
    return country;
  }
}

lift(new Point(3,4), Location).getCountry();
```

Figure 1 – Example.

objects with fields would have to initialise their fields to values that could as well be computed when needed. In other words, state could easily be added to roles, but unless the calculus were extended with mutable state as well there would be no reason to do so. This may be an interesting extension in itself, but in line with FJ we claim that a calculus without mutable state is capable of producing a useful analysis of the soundness of an object-oriented language.

## 2.1 Example

Let us demonstrate with an example what a program looks like in our calculus, see Figure 1. The class `Point` is a regular FJ class that describes a point in the plane. `Location` is a role class that is `playedBy Point`, and provides a view of points in the plane as physical locations on a map of the world. A new instance of `Point` is lifted to a `Location`, which makes it possible to call the method `getCountry` on that object. `getCountry` shows how members of the base class are accessed: using the `lower` keyword to retrieve the base class object.

To illustrate the problem with the smart lifting operation of OT/J consider the following situation: we might have a class `3DPoint` that extends `Point`, and two classes `SpaceLocation` and `SeaLocation` that both extend `Location` and are `playedBy 3DPoint`. In OT/J this could lead to a run-time error due to ambiguity [HHM04], because the smart lifting algorithm would not know whether to lift to `SpaceLocation` or `SeaLocation`, given an instance of `3DPoint` and target role `Location`.

As mentioned in section 1, smart lifting introduces the possibility of making old code fail without modifying it. This is due to the ambiguity mentioned above; a piece of code that looks safe when viewed in isolation might years later become the source of lifting errors because new code can extend old roles, thereby creating an inheritance hierarchy with similar structure as the previous example. A compile-time warning can be given for the new code, but the old code is not necessarily available so the

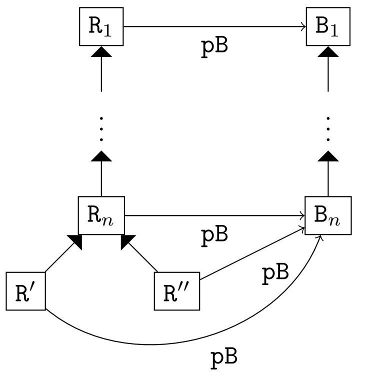

Figure 2 – This hierarchy contains a potential ambiguity. However, when a lifting operation lifting to R$_1$ is given a base class object of type B$_n$ it is always safe to return a role object between R$_1$ and R$_n$ (inclusive).

warning cannot point out which part of the program may fail. This requires a whole program analysis at compile time, which in turn requires that all sources are available. A lifting operation in the old code is now possibly passed a base class object from the new code that makes the lifting operation fail at run-time.

In this calculus our general semantics provide an entire language design space for the lifting operation, where it is possible for the language implementer to choose whether to avoid the ambiguity issue entirely or produce an algorithmic solution that handles it. Figure 2 shows a hierarchy that can cause ambiguity problems, depending on the chosen language design. For this particular hierarchy, given the lifting expression `lift(b`$_n$`,R`$_1$`)`, it is always safe to return a role `R` that is a subtype of the statically required role R$_1$, as long as `R` is a super type of the role R$_n$ after which the hierarchy fans out. Choosing the most specific role in this hierarchy requires some kind of priority system for the roles R′ and R″.

## 2.2 Objects

This calculus uses objects with more structure than what is common among calculi in the FJ family. As mentioned, what we think of as an object is represented by a cloud of objects. In this section we explain in more detail what requirements this cloud must satisfy, and why.

The requirements are in fact influenced by the possible semantics of the lifting operation. The lifting operation is capable of delivering a role whose `playedBy` class is a strict supertype of the class of the base object of the cloud. This means that we may obtain a `Location` role from a `3DPoint` object, even though `Location` specifies that it is `playedBy` a `Point`. The obvious alternative would be to insist that the cloud contains only roles that directly specify the class of the base object in its `playedBy` clause. However, it is necessary in order to preserve type soundness to allow for a flexible invariant. The two situations are illustrated in Figure 3.

Assume we have a class `3DPoint` that extends `Point` from the previous example. The wrapper method for the lifting operation, shown in Figure 4, illustrates the problem. `makeLocation` might be called with a `p` that is an instance of `3DPoint` at run-time. Thus if lifting is unable to lift to roles `playedBy` super types this might get stuck at run-time. This is obviously also a problem for any full-fledged language which

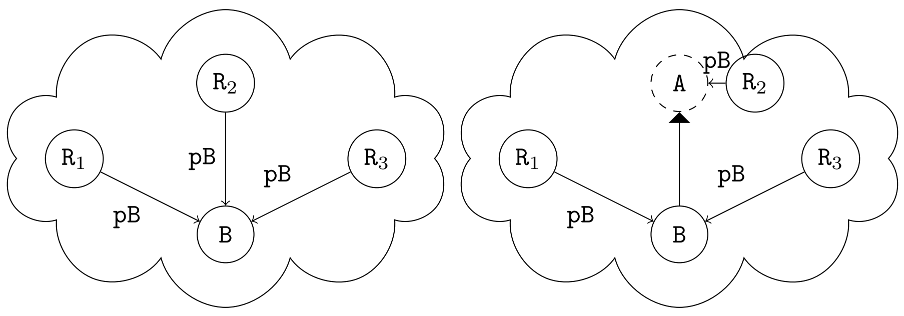

Figure 3 – Left: an object cloud containing only roles directly `playedBy` the base class. Right: an object cloud containing roles `playedBy` super types (`A`) of the base class (`B`).

```
Location makeLocation(Point p) {
  return lift(p, Location);
}
```

Figure 4 – Example

contains our calculus as a sub language. Given that FJ is a sub language of Java, our calculus is essentially a sub language of any language that supports translation polymorphism; hence this property applies to them all.

In the `Point` and `Location` example we included a standard `new` expression for the creation of an object. The formal calculus does not include such an expression; instead it directly creates an object cloud containing a base object and a list of roles. It would be easy to define a surface language that includes traditional `new` expressions and a preprocessing stage that transforms them to cloud creation expressions with an empty role list. In this situation programs would never create clouds with pre-existing roles, they would always come into existence on demand during a lifting operation. However, we note that the actual calculus is safe even without the restriction that all roles are created on demand. We discuss this issue in more detail in section 4.

Before we give the formal definition of the calculus, Figure 5 provides the intuitive relation between the base class type hierarchy and the evaluation and typing rules for lifting and lowering. Both the type of a lifting expression and the result of evaluating it result in a role played by a supertype of the object being lifted. The result of typing a lowering expression is the class that is statically declared as the base class of the role object being lowered; and evaluating a lowering expression results in a subtype of the class that is statically declared as the roles' base.

$$\begin{tikzcd}
\text{Evaluation \& typing} & \text{Lowering} \arrow[r] \arrow[d] & \text{Typing} \\
\text{Lifting} \arrow[u] & \text{Evaluation} &
\end{tikzcd}$$

Figure 5 – The relation between the base class hierarchy and lifting/lowering expressions.

| | | | |
|---|---|---|---|
| *Expressions, values, and evaluation contexts* | | | |
| $\mathtt{e}$ | ::= | $\mathtt{x} \mid \mathtt{[new\ C(\overline{e}),\ \overline{R},\ C]} \mid \mathtt{t.f} \mid$ $\mathtt{t.m(\overline{e})} \mid \mathtt{lift(e,\ R)} \mid$ $\mathtt{lift\text{-}proc(e,\ R)} \mid \mathtt{lower(e)}$ | *expressions* |
| $\mathtt{v,w}$ | ::= | $\mathtt{[new\ C(\overline{v}),\ \overline{R},\ G]}$ | *values* |
| $\mathtt{E}$ | ::= | $\mathtt{[]} \mid \mathtt{[new\ C(\overline{v},\ E,\ \overline{e}),\ \overline{R},\ G]} \mid$ $\mathtt{E.f} \mid \mathtt{E.m(\overline{e})} \mid \mathtt{v.m(\overline{v},\ E,\ \overline{e})} \mid$ $\mathtt{lift(E,\ R)} \mid \mathtt{lift\text{-}proc(E,\ R)} \mid$ $\mathtt{lower(E)}$ | *evaluation contexts* |
| *Member and top-level declarations* | | | |
| $\mathtt{CL}$ | ::= | $\mathtt{class\ C\ extends\ D\ \{\overline{G\ f};\ \overline{M}\}}$ | *classes* |
| $\mathtt{RL}$ | ::= | $\mathtt{class\ R_2\ extends\ R_1\ playedBy\ C\ \{\overline{M}\}}$ | *roles* |
| $\mathtt{M}$ | ::= | $\mathtt{G\ m(\overline{G\ x})\ \{return\ e;\}}$ | *methods* |

Table 1 – The syntax

## 3 Formal Definition of Lifted Java

In this section we present the formal definition of the calculus. We use a sequence notation similar to the one used in the original article on FJ [IPW01], i.e. writing e.g. $\overline{\mathtt{C}}$ means $\mathtt{C}_1 \ldots \mathtt{C}_n$, for some $n \geq 1$. This also applies to binary arguments, such that $\overline{\mathtt{C\ f}}$ means $\mathtt{C}_1\ \mathtt{f}_1 \ldots \mathtt{C}_n\ \mathtt{f}_n$. We use $\bullet$ to denote the empty list. In the following, the meta variables `C` and `D` range over class names; `R` ranges over role names; `G` and `T` can be both class and role names; `f` ranges over field names; `m` ranges over method names; `x` ranges over variable names; `t` ranges over terms; `v` and `w` range over values; `CL` ranges over class declarations; `RL` ranges over role declarations; `M` ranges over method declarations; and `E` ranges over evaluation contexts.

Section 3.1 describes the syntax, section 3.2 the notion of subtyping we use, section 3.3 the auxiliary functions, section 3.4 the small-step semantics, section 3.5 the typing rules, and section 3.6 gives the soundness proof of the calculus.

### 3.1 Syntax

As Lifted Java is an extension of FJ the basic syntax is the same, with the following exceptions: a new class definition for roles has been added, called `RL`; a new object creation term replaces the standard object creation term to accommodate our objects with more structure; the value is replaced by the new object creation term; and terms for each of the operations lifting and lowering have been added. The complete syntax can be seen in Table 1.

In the new class definition `RL` the `playedBy` clause is added to the definition of regular classes. Using this class definition results in defining a role class that has the class given by `playedBy` clause as its base class, and the role provided in the `extends` clause as its supertype. Note that `RL` does not specify fields, a consequence of the fact that roles cannot have state.

The new object creation term is used to instantiate classes. It is a record that, when fully evaluated, describes an object. From left to right it consists of a base class

$$\textsc{Extends}\ \frac{\texttt{class}\ \mathtt{G}_2\ \texttt{extends}\ \mathtt{G}_1\ \ldots}{\mathtt{G}_2 <: \mathtt{G}_1} \qquad \textsc{Reflexive}\ \frac{}{\mathtt{G} <: \mathtt{G}} \qquad \textsc{Transitive}\ \frac{\mathtt{G}_3 <: \mathtt{G}_2 \qquad \mathtt{G}_2 <: \mathtt{G}_1}{\mathtt{G}_3 <: \mathtt{G}_1}$$

Figure 6 – Subtyping

instance, a list of role instances, and a label set to the class name of the currently active object. As long as roles do not have state, the list of role instances in the tuple can in fact be simplified, and so we replace it by a list of roles. As mentioned in section 2 this tuple can be viewed as a cloud containing a base class and any number of roles floating around it. The list of role names is only used in the evaluation rules for lifting; rules that may also modify the list of role names if the object is lifted into a role not in the list.

The term `lift(t, R)` lifts the term `t` to the role `R`. Similarly the term `lower(t)` lowers the term `t` to the base class instance in the object cloud. The term `lift-proc(t, R)` is not intended to be used by the programmer; it is an intermediate step in the dynamic semantics of the lifting operation. However, the soundness result shows that using it is in fact harmless. As will be described in more detail in section 3.4, the term `lift(t, R)` is evaluated to the term `lift-proc(t, R)`.

We use the standard notion of evaluation contexts to describe the parts of a term that have yet to be evaluated. We have chosen a left-to-right evaluation order for this calculus because languages using the features of this calculus are imperative, and lazy evaluation would not be relevant in those languages. Also, the proof still works when the evaluation order is different or even non-deterministic.

For the programmer this syntax amounts to more work compared to that of OT/J. We have chosen this approach in order to prioritise a simple calculus with simple proofs rather than simple programs, as is common when working with calculi. In particular we use explicit lifting and lowering operations; this differs from OT/J where lifting and lowering is typically performed implicitly, with the compiler inserting the appropriate method calls. Thus we assume that the preprocessing step that inserts calls to the lifting and lowering operations has been run. Furthermore, accessing members of a roles base class does not happen through a base link, but rather by lowering the object first and accessing the field on the resulting object; and lifting an already lifted object to a new role can only be done by lowering the object first.

## 3.2 Subtyping

Figure 6 shows the subtyping relation we use. It is the standard definition of subtyping where the extends clause on classes and roles defines the direct subtyping relation with a reflexive and transitive closure.

## 3.3 Auxiliaries

Apart from the evaluation rules for roles, lifting, and lowering, the small-step semantics of Lifted Java need two new auxiliary functions defining the behaviour of the `playedBy` relation. The function *fields* is the auxiliary function from FJ [IPW01] and will thus not be described; however, the two functions *mtype* and *mbody* for lookup of a method's type and body have been combined to one function called *method*. In the following we

$$\textsc{PlayedBy}\quad \frac{\texttt{class } \mathtt{R}_2 \texttt{ extends } \mathtt{R}_1 \texttt{ playedBy C } \{\overline{\mathtt{M}}\}}{\mathit{playedBy}(\mathtt{R}_2, \mathtt{C})} \qquad \textsc{PlayedByWide}\quad \frac{\mathtt{D} <: \mathtt{C} \qquad \mathit{playedBy}(\mathtt{R}, \mathtt{C})}{\mathit{playedByWide}(\mathtt{R}, \mathtt{D})}$$

Figure 7 – The *playedBy* relations.

$$\frac{\texttt{class } \mathtt{T}_2 \texttt{ extends } \mathtt{T}_1 \ \ldots\{\ldots;\overline{\mathtt{M}}\} \qquad \mathtt{G}\ \mathtt{m}(\overline{\mathtt{G}\ \mathtt{x}})\ \{\ \texttt{return e;}\ \} \in \overline{\mathtt{M}}}{\mathit{method}(\mathtt{m}, \mathtt{T}_2) = \overline{\mathtt{G}\ \mathtt{x}}.\mathtt{e} : \mathtt{G}}$$

$$\frac{\texttt{class } \mathtt{T}_2 \texttt{ extends } \mathtt{T}_1 \ \ldots\{\ldots;\overline{\mathtt{M}}\} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{\mathit{method}(\mathtt{m}, \mathtt{T}_2) = \mathit{method}(\mathtt{m}, \mathtt{T}_1)}$$

Figure 8 – The *method* function.

will first describe these auxiliary functions, then the evaluation rules, and finally the typing rules.

Before we proceed with defining the new auxiliary functions, we give the definition of the *flexible invariant* on the types of objects in a cloud. As presented in section 2.2 the cloud has the following structure: the base object has a specific type `C`, and the role objects have role types $\mathtt{R}_1 \ldots \mathtt{R}_k$ that are `playedBy` classes $\mathtt{C}_1 \ldots \mathtt{C}_j$, respectively. The intuitively simplest invariant would then be to require that $\mathtt{C}_i = \mathtt{C}$ for all $i$ or that $\mathtt{C}_i$ is the most specific supertype of `C` that plays a role which is $\mathtt{R}_i$, but as noted in subsection 2.2 a more flexible invariant is required to ensure type safety, i.e., where it is just required that $\mathtt{C}_i$ is a supertype of `C`.

The auxiliary functions are defined in Figure 7 and Figure 8. The rule PlayedBy is used to determine whether a role is `playedBy` a given base class, i.e. $\mathit{playedBy}(\mathtt{R}, \mathtt{C})$ holds if and only if the `playedBy` clause of the role definition of `R` mentions the class name `C`. Alone this rule is insufficient for a sound approach to translation polymorphism, as discussed in section 2. Thus, we define the rule PlayedByWide which is the formal definition of the flexible invariant on the *playedBy* relation. It is similar to the PlayedBy rule except that it takes sub-typing into account, i.e. $\mathit{playedByWide}(\mathtt{R}, \mathtt{C})$ holds if and only if the `playedBy` clause of `R` mentions a super type of `C`.

Lookup of method declarations is done using the rule Method, which is a combination of the separate method body (*mbody*) and method type (*mtype*) lookup functions found in [IPW01]. It is a quadruple, written $\overline{\mathtt{G}\ \mathtt{x}}.\mathtt{e} : \mathtt{G}$, of a sequence of arguments with their types $\overline{\mathtt{G}\ \mathtt{x}}$, a body expression `e`, and a return type `G`.

## 3.4 Evaluation

The small-step semantics of Lifted Java are shown in Figure 9. The evaluation rules extend those of FJ to include evaluation of the terms `lift(t,R)`, `lift-proc(t,R)`, `lower(t)`, and evaluation contexts.

Lifting the value `v` to the role `R` is done using three rules. The rule E-Lift is responsible for choosing an arbitrary role among the valid target roles (see Figure 2

$$\textsc{E-Invk}\quad \frac{\mathit{method}(\mathtt{m},\mathtt{G}) = \overline{\mathtt{T}\ \mathtt{x}}.\mathtt{e} : \mathtt{G}_0}{[\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}}),\ \overline{\mathtt{R}},\ \mathtt{G}].\mathtt{m}(\overline{\mathtt{w}}) \rightarrow [\overline{\mathtt{w}}/\overline{\mathtt{x}},\ [\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}}),\ \overline{\mathtt{R}},\ \mathtt{G}]/\mathtt{this}]\mathtt{e}}$$

$$\textsc{E-Field}\quad \frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{G}\ \mathtt{f}}}{[\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}}),\ \overline{\mathtt{R}},\ \mathtt{C}].\mathtt{f}_i \rightarrow \mathtt{v}_i}$$

$$\textsc{E-Lift}\quad \frac{\mathtt{R}_2 <: \mathtt{R}_1 \qquad \mathit{playedByWide}(\mathtt{R}_2,\mathtt{C})}{\mathtt{lift}([\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}}),\ \overline{\mathtt{R}},\ \mathtt{C}],\mathtt{R}_1) \rightarrow \mathtt{lift\text{-}proc}([\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}}),\ \overline{\mathtt{R}},\ \mathtt{C}],\mathtt{R}_2)}$$

$$\textsc{E-Lift-New}\quad \frac{\mathtt{R} \notin \overline{\mathtt{R}}}{\mathtt{lift\text{-}proc}([\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}}),\ \overline{\mathtt{R}},\ \mathtt{C}],\mathtt{R}) \rightarrow [\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}}),\ \overline{\mathtt{R}} +\!+\ \mathtt{R},\ \mathtt{R}]}$$

$$\textsc{E-Lower}\quad \frac{}{\mathtt{lower}([\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}}),\ \overline{\mathtt{R}},\ \mathtt{R}]) \rightarrow [\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}}),\ \overline{\mathtt{R}},\ \mathtt{C}]}$$

$$\textsc{E-Lift-Old}\quad \frac{\mathtt{R} \in \overline{\mathtt{R}}}{\mathtt{lift\text{-}proc}([\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}}),\ \overline{\mathtt{R}},\ \mathtt{C}],\mathtt{R}) \rightarrow [\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}}),\ \overline{\mathtt{R}},\ \mathtt{R}]}$$

$$\textsc{E-Context}\quad \frac{\mathtt{e} \rightarrow \mathtt{e}'}{\mathtt{E}[\mathtt{e}] \rightarrow \mathtt{E}[\mathtt{e}']}$$

Figure 9 – The evaluation rules for Lifted Java

for an example), i.e. it is required that R is in fact a role and that R is playedBy the currently active class object or a supertype of it. Both facts are checked by *playedByWide*. This rule is required for the semantics to be general enough to describe the desired language design space. The rules E-Lift-Old and E-Lift-New apply depending on whether the selected role is in the cloud of v or not. In the first case only the name of the currently active instance is updated (E-Lift-Old), and in the second case the role instance is also added to the cloud of v (E-Lift-New).

Lowering the value v is taken care of by a single rule, E-Lower, that only requires that the name of the currently active object of v is a role. It would be straightforward to make it possible to lower a regular class to itself and still maintain soundness, as long as the typing rule for lowering also allows typing of a lower expression where the active object is the base object. However, to maintain a simple calculus we have decided that lowering should not be smarter than lifting.

The rule E-Context for evaluation contexts provides the necessary evaluation of subexpressions.

For an example of a chain of reductions in this semantics, consider the hierarchy consisting of a single class B and two roles $\mathtt{R}_1$ and $\mathtt{R}_2$ both played by B, where $\mathtt{R}_2$ extends $\mathtt{R}_1$. The following chain of reductions fully evaluates the expression $\mathtt{lift}(\mathtt{aB},\mathtt{R}_1)$:

$$\begin{array}{lll} & \mathtt{lift}([\mathtt{new}\ \mathtt{B}(),\bullet,\mathtt{B}],\mathtt{R}_1) & \\ \rightarrow & \mathtt{lift\text{-}proc}([\mathtt{new}\ \mathtt{B}(),\bullet,\mathtt{B}],\mathtt{R}_1) & [\textsc{E-Lift}] \\ \rightarrow & [\mathtt{new}\ \mathtt{B}(),[\mathtt{R}_1],\mathtt{R}_1] & [\textsc{E-Lift-New}] \end{array}$$

However, given the non-deterministic nature of the semantics there is another possible chain of reductions for this example:

$$\textsc{T-Var}\quad \frac{}{\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x})}$$

$$\textsc{T-Invk}\quad \frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \qquad \mathit{method}(\mathtt{m}, \mathtt{T}_1) = \overline{\mathtt{G}_2\ \mathtt{x}}.\mathtt{e} : \mathtt{T}_2 \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{G}_1} \qquad \overline{\mathtt{G}_1} <: \overline{\mathtt{G}_2}}{\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{T}_2}$$

$$\textsc{T-Field}\quad \frac{\Gamma \vdash \mathtt{e} : \mathtt{C} \qquad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{G}\ \mathtt{f}}}{\Gamma \vdash \mathtt{e.f}_i : \mathtt{G}_i}$$

$$\textsc{T-Lift}\quad \frac{\Gamma \vdash \mathtt{e} : \mathtt{C} \qquad \mathit{playedByWide}(\mathtt{R}, \mathtt{C})}{\Gamma \vdash \mathtt{lift}(\mathtt{e}, \mathtt{R}) : \mathtt{R}}$$

$$\textsc{T-Lift-Proc}\quad \frac{\Gamma \vdash \mathtt{e} : \mathtt{C} \qquad \mathit{playedByWide}(\mathtt{R}, \mathtt{C})}{\Gamma \vdash \mathtt{lift\text{-}proc}(\mathtt{e}, \mathtt{R}) : \mathtt{R}}$$

$$\textsc{T-Lower}\quad \frac{\Gamma \vdash \mathtt{e} : \mathtt{R} \qquad \mathit{playedBy}(\mathtt{R}, \mathtt{C})}{\Gamma \vdash \mathtt{lower}(\mathtt{e}) : \mathtt{C}}$$

$$\textsc{T-New}\quad \frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{G}_1\ \mathtt{f}} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{G}_2} \qquad \overline{\mathtt{G}_2} <: \overline{\mathtt{G}_1} \qquad (\mathit{playedByWide}(\mathtt{G}, \mathtt{C}) \wedge \mathtt{G} \in \overline{\mathtt{R}}) \vee \mathtt{G} = \mathtt{C}}{\Gamma \vdash [\mathtt{new\ C}(\overline{\mathtt{e}}),\ \overline{\mathtt{R}},\ \mathtt{G}] : \mathtt{G}}$$

Figure 10 – Typing rules for Lifted Java

$$\begin{array}{lll} & \mathtt{lift}([\mathtt{new\ B}(), \bullet, \mathtt{B}], \mathtt{R}_1) & \\ \to & \mathtt{lift\text{-}proc}([\mathtt{new\ B}(), \bullet, \mathtt{B}], \mathtt{R}_2) & [\textsc{E-Lift}] \\ \to & [\mathtt{new\ B}(), [\mathtt{R}_2], \mathtt{R}_2] & [\textsc{E-Lift-New}] \end{array}$$

## 3.5 Typing

The typing rules of Lifted Java can be seen in Figure 10 and the wellformedness rules in Figure 11. The FJ typing rules are extended to include well-formedness for roles, typing of the `lift(t, R)` term, typing of the `lift-proc(t, R)` term, and typing of the `lower(t)` term. Furthermore, the typing rule of the new object creation term is updated.

The typing rule for object creation terms, T-New, states that the type of an object is always the class corresponding to the active instance. This can be either the base class `C` or one of the role classes $\mathtt{R}_i$ in the cloud. In order for the rule to apply it is required that the arguments to the constructor of the base class have the correct types, and that the currently active instance is either a role `playedBy` a super type of `C` or that it is `C`.

The rule T-Lift states that a `lift(t, R)` expression has the type of the role lifted to. It is required that the type of the first argument plays the role `R`, or is a subtype of a class that does. The rule T-Lift-Proc is similar, but for the `lift-proc(t, R)` expression.

The T-Lower rule describes the requirements for typing the `lower(t)` term. It states that the `lower` expression has the type of the base class of the currently active instance, and thus requires that the type of the argument is a role. Like with the evaluation rule for the `lower(t)` term it would be straightforward to allow the term to be typed when the argument has the type of a regular class and still maintain soundness, as long as the evaluation rule is also updated to allow evaluation of a `lower`

W-Method

$$\frac{\begin{array}{c}\mathtt{this} : \mathtt{T},\ \overline{\mathtt{x}} : \overline{\mathtt{G_1}} \vdash \mathtt{e_0} : \mathtt{T_0} \\ \mathtt{T_0} <: \mathtt{T_1} \qquad \mathtt{class\ T\ extends\ T'}...\{...\} \qquad \mathit{canOverride}(\mathtt{T'}, \mathtt{m}, \mathtt{T_1})\end{array}}{\mathtt{T_1\ m}(\overline{\mathtt{G_1\ x}})\ \{\ \mathtt{return\ e_0};\ \}\ \mathit{is}\ \mathrm{OK}\ \mathit{in}\ \mathtt{T}}$$

W-Class

$$\frac{\overline{\mathtt{M}}\ \mathit{OK\ in}\ \mathtt{C}}{\mathtt{class\ C\ extends\ D}\{\overline{\mathtt{C\ G}};\ \overline{\mathtt{f}}\}M\ \mathrm{OK}}$$

W-Role

$$\frac{\begin{array}{c}\overline{\mathtt{M}}\ \mathit{is}\ \mathrm{OK}\ \mathit{in}\ \mathtt{R_2} \\ \mathtt{class\ C\ extends\ D}\{\overline{\mathtt{C\ G}};\ \overline{\mathtt{f}}\}m\end{array}}{\mathtt{class\ R_2\ extends\ R_1\ playedBy\ C}\ \{\overline{\mathtt{M}}\}\ \mathrm{OK}}$$

W-Covariance

$$\forall \mathtt{R_1}, \mathtt{R_2}, \mathtt{C_1}, \mathtt{C_2}\ .\ \mathit{playedBy}(\mathtt{R_1}, \mathtt{C_1}) \wedge \mathit{playedBy}(\mathtt{R_2}, \mathtt{C_2}) \wedge \mathtt{R_2} <: \mathtt{R_1} \quad \Rightarrow \quad \mathtt{C_2} <: \mathtt{C_1}$$

Figure 11 – Wellformedness rules

canOverride

$$\mathit{canOverride}(\mathtt{T}, \mathtt{m}, \mathtt{T_1}) := \text{if } \mathit{method}(\mathtt{m}, \mathtt{T}) = \overline{\mathtt{G_2\ x}}.\mathtt{e} : \mathtt{T_2} \text{ then } \mathtt{T_1} <: \mathtt{T_2} \text{ and } \overline{\mathtt{G_1}} = \overline{\mathtt{G_2}}$$

Figure 12 – The *canOverride* predicate.

expression with a value where the active object is the base object.

The rule for role typing (W-Role) is similar to the rule for regular class typing (W-Class), except for the fact that the class specified in the `playedBy` clause must exist in the program. W-Method is the rule for method typing; the definition of the *canOverride* predicate can be seen in Figure 12. Notice how this rule allows methods on classes to have signatures that include roles.

Finally, we have an extra wellformedness criterion, W-Covariance. It specifies that the `playedBy` relation must be covariant.

## 3.6 Safety Properties

Under the assumption that the program is well-formed, the following safety properties hold for the calculus presented in the previous section:

**Theorem A.1** (*preservation*). *If* $\bullet \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e} \rightarrow \mathtt{e'}$ *then there exists some* $\mathtt{T'}$ *such that* $\bullet \vdash \mathtt{e'} : \mathtt{T'}$ *and* $\mathtt{T'} <: \mathtt{T}$.

**Theorem A.2** (*progress*). *If* $\bullet \vdash \mathtt{e} : \mathtt{T}$ *then* $\mathtt{e}$ *is either a value or* $\mathtt{e} \rightarrow \mathtt{e'}$ *for some* $\mathtt{e'}$.

**Corollary A.3** (*type soundness*). *If* $\bullet \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e} \rightarrow^* \mathtt{e'}$ *where* $\mathtt{e'}$ *is a normal form, then* $\mathtt{e'}$ *is a value and* $\bullet \vdash \mathtt{e'} : \mathtt{T'}$, *where* $\mathtt{T'} <: \mathtt{T}$.

Corollary 3.3 follows easily from the preservation and progress theorems, following the pattern introduced in [WF94]. The complete proof can be seen in Appendix A. The Coq implementation available at [EI11] provides a lot of extra details and furthermore provides confidence that the proof is correct.

Note that we have been able to simplify the proofs by assuming empty type environments in the preservation theorem. The resulting property is still sufficient to prove the standard type soundness result, which is expressed as Corollary 3.3. Hence, the weaker preservation property is sufficient to show the desired soundness result, and consequently the extra work required to show preservation with non-empty environments would be superfluous. This technique was used by De Fraine *et al.* both in their implementation of the $A$ Calculus [DFES10] and in the implementation of FJ [DF09] which we use as a basis.

## 4 Discussion

In this section we will discuss three things: our choices with regard to the semantics of lifting and lowering; the case of unrestricted roles in object creation expressions as mentioned in section 2.2; and the flexible invariant.

**Lifting.** In OT/J lifting is smart, i.e. it will produce a role with the dynamically most specific type rather than the statically known type. This can lead to ambiguity, the reason for which is that a base object might be lifted to a role that is extended by two otherwise unrelated roles. If the object cloud of the base object does not already contain a role of the requested type, such a role should now be created. In this situation it is ambiguous which of the two unrelated roles is the most specific, and thus which of them the smart lifting algorithm should select. In OT/J this causes an exception at run-time, and it may happen in a piece of code that was compiled without warnings or errors, possibly long before the two unrelated roles were written.

We have chosen a more general semantics for lifting whereby an arbitrary subtype of the statically known role type is used, as long as it does not specify a more specific type than that of the object being lifted in its `playedBy` relation. This captures every possible choice of role, from the statically known role down to the most specific role(s). Our soundness proof shows that this semantics is sound. Smart lifting, where the most specific role is always chosen or an exception is raised, can thus be seen as a special case of our semantics. We argue that there are many sound ways to remove the ambiguity problem in this language design space, e.g.: always returning the statically given role; the non-deterministic approach taken in our calculus; approaches based on taking the most specific type that does not cause ambiguities; or using programmer declared precedence are among the possible choices. It is a main contribution of this work to clarify that this ambiguity problem can be solved by choosing any language design within this language design space.

Subtyping among roles makes covariance of the `playedBy` relation a necessity, i.e. when refining the `playedBy` clause in a subtype $\mathtt{R}_2$ of $\mathtt{R}_1$ the base of $\mathtt{R}_2$ must be a subtype of the base of $\mathtt{R}_1$. This is the same situation as in OT/J. To see why the covariance of the `playedBy` relation is necessary, consider the hierarchies in Figure 13. Anywhere an $\mathtt{R}_2$ is expected an $\mathtt{R}_3$ can be passed in at runtime; if that code tries to lower the $\mathtt{R}_3$ object and expects to be able to access members of a $\mathtt{B}_2$ instance, the computation is unable to take another step.

Lifting and lowering is always explicit in our calculus, using the special functions `lift` and `lower`, whereas they are generally added by the compiler in OT/J. This means more work for programmers using our calculus, but since it would be easy to add the necessary calls to these functions in a preprocessing step, there is no need to have implicit lifting and lowering as part of the calculus. In fact the OT/J compiler takes this approach, automatically inserting calls to lifting and lowering methods.

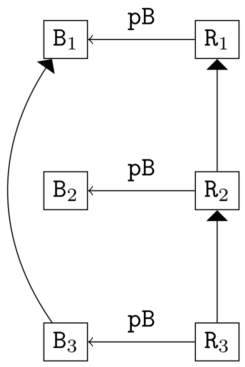

Figure 13 – A base and role hierarchy that breaks type safety by destroying the preservation property.

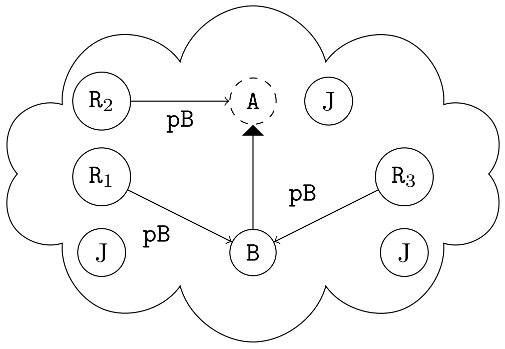

Figure 14 – The cloud as implemented in the model. `J` marks junk role names.

**Flexible invariant.** An interesting property of our calculus is that it employs a flexible invariant for the types of objects in a cloud, and the soundness proof shows that this is a safe thing to do. We introduced a *widePlayedBy* relation in the calculus in order to express this invariant. The important fact to note is that almost any choice of semantics for the lifting operation from the above-mentioned language design space would require a more or less flexible invariant in the sense defined here.

**Objects.** From the calculus syntax in section 3.1, it is clear that there is no restriction on the role names that can be in the object cloud of an object creation expression. Programmers could therefore write programs that contain object creation expressions including roles that do not have a *widePlayedBy* relation to the class of the base object, let us call them *junk roles.* Intuitively this creates the problem that the cloud contains roles that are not `playedBy` the given base object, not even via a superclass! Figure 14 illustrates this situation. It may seem dangerous to allow programs to run when some objects contain junk roles, but this is in fact benign. The undeniable argument is that the Coq soundness proof works for a formalisation that allows junk roles to exist; the associated intuition is that these junk roles are unreachable because roles can only come into play when being selected by a lifting operation — this will never happen for a junk role.

## 5 Related Work

The AspectJ language [KHH$^+$01] was the first to introduce *Aspect-Oriented Programming* [KLM$^+$97] in a general purpose programming language. OT/J supports a different style of AOP which is also quite powerful. However, aspects are at the other end of OT/Js features compared to our focus on translation polymorphism, and thus we will not treat them further.

CaesarJ [AGMO06] solves the same scenario as OT/J, i.e. non-invasive customisation through addition instead of modification. The following are the similarities that are relevant with respect to our work. Like in OT/J, virtual classes and family polymorphism are added to the language. The equivalents to roles and base classes are called *wrappers* and *wrappees*. To translate an object of a wrappee type to an object of a type wrapping it (lifting), a *wrapper constructor* is called with the wrappee object as an argument. The translation from wrapper to wrappee (lowering) is done using an explicit *wrappee* link. We will not go into detail with CaesarJ, but simply note that the model and observations in this paper apply to that language as well.

*Expanders* [WSM06] is a technique for statically scoped object adaptation that bears some similarity to the roles of OT/J. Like OT/J it is implemented as an extension to the Java language, called eJava. Both the concept of roles and that of expanders provides the means to augment existing objects with new state and behaviour. There are some differences however, as the following points illustrate. Unlike roles, expanders cannot be instantiated explicitly. Roles can completely redefine methods of their base while expanders cannot even override methods of the classes they augment (method overloading is possible, however). Expanders provide explicit expanding only to deal with multiple imported expanders specifying the same method. Otherwise, once an expander is imported (*used*) in a context the compiler infers the places where object expansion is necessary. In OT/J objects are lifted implicitly most of the time as well, and like with expanders it is possible to declare places of explicit lifting. This is not used to disambiguate however, but to specify methods that operate on roles but require the caller to provide a base object. The expanders approach has been proven sound using an extension of Featherweight Java called Featherweight eJava.

*Wide classes* [Ser99] is another concept that serves much of the same purpose as the roles of OT/J. The analogous operations to lifting and lowering are called *widening* and *narrowing* when dealing with wide classes. The main differences between these concepts are: When lifting a base object to a role the new object is in another hierarchy entirely, whereas when widening an object it is still in the same class hierarchy (it is widened to one of its subtypes); and widening an object does not create a new object, it simply adds fields and methods directly on the existing object. This last difference also means that a widened object is the same as the object it was before, i.e. equality is preserved, which is not the case for roles.

Finally, we should mention that even though lifting and lowering enables objects to "change their class dynamically", this is still very different from the large number of mechanisms known from highly dynamic languages whereby classes are directly modified. For instance, programming activities in Smalltalk [GR83] typically imply modifications of classes and their instances, and import operations in Ruby and Python may also change classes (known as 'monkey patching'), e.g. [Ben08]. Although it is statically typed, C# partial classes [Mic] are similar in that they can be considered to be a class modification mechanism. Common to all these mechanisms is that they allow classes to be modified or extended, but they do not allow objects to include just some modifications, they must all in synchrony include all modifications. With translation polymorphism two instances of the same class may have different sets of roles at the same time, which makes this very different from a class modification mechanism, even a highly dynamic one.

## 6 Future Work

For the calculus presented in this paper simplicity is a major feature, because it isolates the core of translation polymorphism. A more elaborate model would be interesting to explore in order to address the problems with ambiguity in smart lifting directly, for instance demonstrating that a certain class of priority mechanisms could enable lifting to produce a most specific role in some sense, and remain free of run-time errors.

Adding mutable state as well as role state could provide a better view of the interaction between roles and classes. Investigating how this calculus would fit as an extension of Welterweight Java [OW10] could be interesting in this regard. Welterweight Java is a minimal imperative and stateful calculus for Java-like languages. [Sum09] notes that some extensions are only unsound in the presence of state. He demonstrates this by extending Featherweight Generic Java [IPW01] with existential types, showing that the extension is sound. Using a high-level example that further extends Featherweight Generic Java with mutable state he then shows that existential types with state is unsound. The standard argument is that an immutable calculus in the style of Featherweight Java is useful for proving soundness of the type system of mutable languages as well, and he only demonstrates that existential types are unsound in the presence of state. Type-wise, our calculus is closer to plain Java with no potential for mixing completely unrelated types like he does in his example with existential types, and thus we believe the soundness proof of our calculus translates to a calculus with mutable state as well.

## 7 Conclusion

Translation polymorphism, also known as lifting and lowering, is a language mechanism which enables multiple objects, organised into pairs of base and role objects, to act almost as if they were single objects supporting multiple unrelated interfaces. This paper demonstrates for the first time that the core semantics of translation polymorphism with role inheritance is provably type sound, and that there is in fact an entire language design space of safe choices for this semantics. The results in this paper were achieved by means of a very simple formal calculus that models lifting and lowering independently of the advanced features that are typically present in languages supporting translation polymorphism, such as virtual classes and family polymorphism. The completeness and correctness of the soundness proof of this calculus has been verified mechanically by means of the Coq proof assistant. Consequently, translation polymorphism can now be considered safe.

## A Lemmas and Theorems

In this appendix we present the full soundness proof of the calculus, using the same notation as in the rest of this paper. The source code available at [EI11] is an implementation of this proof using the Coq proof assistant and has a lot of extra details.

The following sections each contain a part of the soundness proof, keeping related lemmas in the same section. As is usual for paper proofs we omit some details in favour of conciseness and readability. The proofs contain the necessary details to repeat them, e.g. which lemmas are used and whether it is a proof by induction. Each lemma in this proof is annotated with the name of the corresponding lemma(s) in the Coq code. When no proof is stated immediately following a lemma it means that the proof is straightforward and uses no other lemmas. Most lemmas are also found in a proof of FJ, their proofs adjusted to take into account the role hierarchy. The new lemmas are A.5, A.7, A.10, and all lemmas in section A.4.

Lemma A.4 and A.5 are very similar; this is because they serve the same purpose but for two different, yet similar, class hierarchies. It would be possible to state each pair of lemmas as one lemma, but in order to avoid greater complexity we do not do that. In the Coq model the lemmas that depend on these two lemmas are all split in two even though their proofs are almost identical, so here we present them as a single lemma and annotate them with the names of the corresponding two lemmas in the Coq model. The Coq source is a machine verified version of this proof, and we invite the reader interested in every little detail to download and explore the Coq implementation.

The main difference between this proof and its Coq implementation, apart from the fact that the implementation specifies everything in full detail, is in readability. Coq imposes some extra work which is not interesting for the proof, but is a necessary cost of working with the type system. One example of this is the subtyping relation $\mathtt{G}_2 <: \mathtt{G}_1$ which in the coq source has three representations depending on whether $\mathtt{G}_1$ and $\mathtt{G}_2$ are class labels, role labels, or types, requiring a number of auxiliary conversion and inversion lemmas. These extra lemmas and definitions make it harder to spot the important details. Furthermore, because a Coq proof consists of a series of applications of tactics that all operate on the current proof context, such a proof can be hard to read and understand without using a tool like CoqIDE[2] or ProofGeneral[3] to step through it interactively.

## A.1 Uniqueness of Types

The following lemmas show that types in this calculus are unique. Lemma A.3 is the most general result and although it is not used for proving type safety we present it here because it is an interesting property by itself. Something to note about the Coq model of the proof of A.1 is that it requires two separate lemmas, one for methods on roles and one for methods on classes. This is because the method lookup function *method* has to be split into two when modelled in Coq.

**Lemma A.1.** *(method_fun, method_fun_r) If* $\mathit{method}(\mathtt{m}, \mathtt{G}) = \overline{\mathtt{G_i}\ \mathtt{x_i}}.\mathtt{e_i} : \mathtt{G_i}$ *for* $i \in \{1, 2\}$ *then* $\overline{\mathtt{G_1}\ \mathtt{x_1}}.\mathtt{e_1} : \mathtt{G_1} = \overline{\mathtt{G_2}\ \mathtt{x_2}}.\mathtt{e_2} : \mathtt{G_2}$.

**Proof**. By induction in the proof of $\mathit{method}(\mathtt{m}, \mathtt{G}) = \overline{\mathtt{G_1}\ \mathtt{x_1}}.\mathtt{e_1} : \mathtt{G_1}$ and case analysis in the proof of $\mathit{method}(\mathtt{m}, \mathtt{G}) = \overline{\mathtt{G_2}\ \mathtt{x_2}}.\mathtt{e_2} : \mathtt{G_2}$. □

**Lemma A.2.** *(fields_fun) If* $\mathit{fields}\,(\mathtt{C}) = \overline{\mathtt{T_i}\ \mathtt{x_i}}$ *for* $i \in \{1, 2\}$ *then* $\overline{\mathtt{T_1}\ \mathtt{x_1}} = \overline{\mathtt{T_2}\ \mathtt{x_2}}$.

**Proof**. By induction in the proof of $\mathit{fields}\,(\mathtt{C}) = \mathtt{T_1}\ \mathtt{x_1}$ and inversion in the proof of $\mathit{fields}\,(\mathtt{C}) = \mathtt{T_2}\ \mathtt{x_2}$. The fact that `Object` is not defined in the program in used to

[2]Bundled with the Coq proof assistant.
[3]Available at `http://proofgeneral.inf.ed.ac.uk/`

establish a contradiction in two of the four resulting cases. The remaining two cases are straightforward. □

**Lemma A.3.** *(typing_fun) If* $\Gamma \vdash \texttt{e} : \texttt{T}$ *and* $\Gamma \vdash \texttt{e} : \texttt{T}'$ *then* $\texttt{T} = \texttt{T}'$.

**Proof**. By induction in the typing derivation $\Gamma \vdash \texttt{e} : \texttt{T}$ and inversion in the typing derivation $\Gamma \vdash \texttt{e} : \texttt{T}'$. This results in ten cases, two of which deserve extra attention:
Case 1: Two field accesses. This uses A.2.
Case 2: Two methods. This uses A.1. □

## A.2 Wellformedness

These lemmas define various relations between the wellformedness rules.

**Lemma A.4.** *(ok_class_meth) If* `class D extends C {`$\overline{\texttt{D f}}$;$\overline{\texttt{M}}$`}` *is* OK *and* `T m(`$\overline{\texttt{T x}}$`) {return e;}` $\in \overline{\texttt{M}}$ *then* m *is* OK *in* D.

**Lemma A.5.** *(ok_role_meth) If* `class` $\texttt{R}_2$ `extends` $\texttt{R}_1$ `playedBy B {`$\overline{\texttt{M}}$`}` *is* OK *and* `T m(`$\overline{\texttt{T x}}$`) {return e;}` $\in \overline{\texttt{M}}$ *then* m *is* OK *i* $\texttt{R}_2$.

**Lemma A.6.** *(ok_ctable_class) If* CT *is not circular and* `CT(C) = class C extends D {`$\overline{\texttt{G f}}$;$\overline{\texttt{M}}$`}` *then* `class C extends D {`$\overline{\texttt{G f}}$;$\overline{\texttt{M}}$`}` *is* OK.

**Lemma A.7.** *(ok_rtable_role) If* RT *is not circular and* `RT(R2) = class` $\texttt{R}_2$ `extends` $\texttt{R}_1$ `playedBy C {`$\overline{\texttt{M}}$`}` *then* `class` $\texttt{R}_2$ `extends` $\texttt{R}_1$ `playedBy C {`$\overline{\texttt{M}}$`}` *is* OK.

**Lemma A.8.** *(method_implies_typing, method_implies_typing_role) If* `G m (`$\overline{\texttt{G}'\ \texttt{x}}$`) {return e;}` *is* OK *in* $\texttt{G}_2$ *then there exists a* $\texttt{G}_1$ *such that* $\texttt{G}_2 <: \texttt{G}_1$ *and* $\Gamma, \texttt{this} : \texttt{G}_1 \vdash \texttt{e} : \texttt{G}$.

**Proof**. By induction in the proof of `G m (`$\overline{\texttt{G}'\ \texttt{x}}$`) {return e;}` is OK in $\texttt{G}_2$, resulting in two cases. The first case uses A.4 and A.5. The second case uses the transitivity and extends properties of the subtyping relation. □

**Lemma A.9.** *(sub_implies_dom) If* $\texttt{R}_2 <: \texttt{R}_1$ *and* $\texttt{R}_1 \in$ *dom* RT *then* $\texttt{R}_2 \in$ *dom* RT.

**Lemma A.10.** *(in_RT_implies_playedBy) If* $\texttt{R} \in$ *dom* RT *then there exists a* C *such that playedBy(*R, C*)*.

**Proof**. Straightforward, observing that the *playedBy* relation is part of the role table. □

## A.3 Preservation of Properties in Subtypes

These are lemmas establishing that a type is replaceable with any of its subtypes.

**Lemma A.11.** *(sub_field) If* $\texttt{D} <: \texttt{C}$ *and* $\texttt{G g} \in \mathit{fields}(\texttt{C})$ *then* $\texttt{G g} \in \mathit{fields}(\texttt{D})$.

**Lemma A.12.** *(gt_field) If* $\Gamma \vdash \texttt{e} : \texttt{G}_2$ *where* $\texttt{G}_2 <: \texttt{G}_1$ *and* $\texttt{G}_3\ \texttt{f} \in \mathit{fields}(\texttt{G}_1)$ *then* $\Gamma \vdash \texttt{e.f} : \texttt{G}_3$.

**Proof**. Uses A.11. □

**Lemma A.13.** *(gt_sub) If* $\Gamma \vdash \texttt{e} : \texttt{G}_2$ *where* $\texttt{G}_2 <: \texttt{G}_1$ *then* $\Gamma \vdash \texttt{e} : \texttt{G}_1$.

**Lemma A.14.** *(sub_fields) If* $\mathtt{D} <: \mathtt{C}$ *and* $\mathit{fields}(\mathtt{C}) = \overline{\mathtt{G}_1\ \mathtt{f}}$ *then there exists* $\overline{\mathtt{G}_2\ \mathtt{g}}$ *such that* $\mathit{fields}(\mathtt{D}) = \overline{\mathtt{G}_1\ \mathtt{f}};\ \overline{\mathtt{G}_2\ \mathtt{g}}$.

**Proof.** Straightforward proof by induction in the subtyping derivation $\mathtt{D} <: \mathtt{C}$. ☐

**Lemma A.15.** *(sub_mtype, sub_mtype_r) If* $\mathtt{T}_2 <: \mathtt{T}_1$ *and* $\mathtt{G}_1\ \mathtt{m}(\overline{\mathtt{G}\ \mathtt{x}})\ \{\mathtt{return}\ \mathtt{e}_1;\}$ *is* OK *in* $\mathtt{T}_1$ *then there exists a* $\mathtt{G}_2$ *where* $\mathtt{G}_2 <: \mathtt{G}_1$ *and there exists an* $\mathtt{e}_2$ *such that* $\mathtt{G}_2\ \mathtt{m}(\overline{\mathtt{G}\ \mathtt{x}})\ \{\mathtt{return}\ \mathtt{e}_2;\}$ *is* OK *in* $\mathtt{T}_2$.

**Proof.** By induction in the subtyping derivation $\mathtt{T}_2 <: \mathtt{T}_1$. The reflexive and transitive cases are straightforward. The extends case is split into two sub cases:
Case: m is a method on D. Uses A.4 and A.5.
Case: m is not a method on D. Straightforward. ☐

**Lemma A.16.** *(gt_meth, gt_meth_r) If* $\Gamma \vdash \mathtt{e}_0 : \mathtt{G}_2$ *where* $\mathtt{G}_2 <: \mathtt{G}_1$, $\mathtt{G}_3\ \mathtt{m}(\overline{\mathtt{G}_3\,\mathtt{x}})\ \{\mathtt{return}\ \mathtt{e}_1;\}$ *is* OK *in* $\mathtt{G}_1$*, and* $\overline{\mathtt{G}_4} < \overline{\mathtt{G}_3}$*, then* $\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{G}_4$ *where* $\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{G}_4}$ *and* $\mathtt{G}_4 <: \mathtt{G}_3$.

**Proof.** Uses A.13 and A.15. ☐

## A.4 Properties Using Covariance of PlayedBy

The assumption that the *playedBy* relation is covariant allows us to infer some necessary information about the interaction between classes and roles.

**Lemma A.17.** *(covariant_extraction) If playedBy(*$\mathtt{R}_1$, $\mathtt{C}_1$*), playedBy(*$\mathtt{R}_2$, $\mathtt{C}_2$*), and* $\mathtt{R}_2 <: \mathtt{R}_1$*, then* $\mathtt{C}_2 <: \mathtt{C}_1$.

**Proof.** Using A.7 and the assumption that the *playedBy* relation is covariant. ☐

**Lemma A.18.** *(infer_playedBy) If playedBy(*$\mathtt{R}_1$, $\mathtt{C}_1$*) and* $\mathtt{R}_2 <: \mathtt{R}_1$ *then there exists a* $\mathtt{C}_2$ *such that* $\mathtt{C}_2 <: \mathtt{C}_1$ *and playedBy(*$\mathtt{R}_2$, $\mathtt{C}_2$*).*

**Proof.** By induction in the subtyping relation $\mathtt{R}_2 <: \mathtt{R}_1$. The reflexive case is straightforward.
Case: Transitive. Uses A.7, A.17, A.9 and A.10.
Case: Extends. Uses A.17. ☐

## A.5 Term Substitutivity

A lemma showing that substituting sub-expressions in an expression preserves subtyping.

**Lemma A.19.** *(term_substitutivity) If* $\overline{\mathtt{G}_1\ \mathtt{x}} \vdash \mathtt{e} : \mathtt{G}_1$ *and* $\Gamma \vdash \overline{\mathtt{d}} : \overline{\mathtt{G}_2}$ *where* $\overline{\mathtt{G}_2} <: \overline{\mathtt{G}_1}$ *then* $\Gamma \vdash \mathtt{e}\left[\overline{\mathtt{x}} \to \overline{\mathtt{d}}\right] : \mathtt{G}_2$ *for some* $\mathtt{G}_2$ *where* $\mathtt{G}_2 <: \mathtt{G}_1$.

**Proof.** The proof builds on a mutual induction result which states that if $\Gamma \vdash \overline{\mathtt{d}} : \overline{\mathtt{G}_2}$ and $\overline{\mathtt{G}_1\ \mathtt{x}} \vdash \mathtt{Z} : \tau$ then $\Gamma \vdash \mathtt{Z}\left[\overline{\mathtt{x}} \to \overline{\mathtt{d}}\right] : \tau'$ where Z stands for an expression or a list of expressions, and $\tau$ and $\tau'$ stand for types or lists of types where the latter is a (list of) subtype(s) of the former. The interesting cases are shown here:
Case: Field lookup. Uses A.12.
Case: Method invocation. Uses A.16.
Case: Lowering. Uses A.18. ☐

## A.6 Type Soundness

The following lemmas and theorems combine the lemmas from the previous sections to prove the type safety of the calculus.

**Lemma A.20.** *(preservation_over_ec) If we have an evaluation context* $\mathtt{E}$*,* $\bullet \vdash \mathtt{E}\,[\mathtt{e}] : \mathtt{G}_1$ *and the implication* $\bullet \vdash \mathtt{e} : \mathtt{G}_1 \rightarrow \bullet \vdash \mathtt{e} : \mathtt{G}_2$ *then* $\bullet \vdash \mathtt{E}\,[\mathtt{e}'] : \mathtt{G}_2$ *where* $\mathtt{G}_2 <: \mathtt{G}_1$*.*

**Proof**. By case analysis in the evaluation context derivations followed by inversion in the typing derivation $\bullet \vdash \mathtt{E}\,[\mathtt{e}] : \mathtt{G}_1$.
Case: Fields. Uses A.12.
Case: Methods. Uses A.16.
Case: Lowering. Uses A.18. □

**Theorem A.1.** *(preservation) If* $\bullet \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e} \rightarrow \mathtt{e}'$ *then there exists some* $\mathtt{T}'$ *such that* $\bullet \vdash \mathtt{e}' : \mathtt{T}'$ *and* $\mathtt{T}' <: \mathtt{T}$*.*

**Proof**. By induction in the evaluation derivations $\mathtt{e} \rightarrow \mathtt{e}'$. Most cases are straightforward, except the following:
Case: Methods. Uses A.1, A.8, A.19 and lemma A.13.
Case: Evaluation Contexts. Uses A.20. □

**Theorem A.2.** *(progress) If* $\bullet \vdash \mathtt{e} : \mathtt{T}$ *then* $\mathtt{e}$ *is either a value or* $\mathtt{e} \rightarrow \mathtt{e}'$ *for some* $\mathtt{e}'$*.*

**Proof**. By induction in the typing derivation $\bullet \vdash \mathtt{e} : \mathtt{G}$. Most of the resulting cases are straightforward, except for the expressions `lift(e, R)` and `lift-proc(e, R)`.
Case: `lift(e, R)`. Split into two sub-cases depending on the type of `e`. Observe that when the type is a role the proof is by contradiction because only base objects can be lifted!
Case: `lift-proc(e, R)`. Split into two sub-cases depending on whether `R` is in the object cloud or not. □

**Corollary A.3.** *(safety) If* $\Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e} \rightarrow^* \mathtt{e}'$ *where* $\mathtt{e}'$ *is a normal form, then* $\mathtt{e}'$ *is a value and* $\Gamma \vdash \mathtt{e}' : \mathtt{T}'$ *where* $\mathtt{T}' <: \mathtt{T}$*.*

**Proof**. Follows from theorem A.1 and theorem A.2. □

## References

[AGMO06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. *Transactions on Aspect-Oriented Software Development I. LNCS*, 3880:135–173, 2006.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*, volume XXV of *Texts in Theoretical Computer Science*. Springer, 2004.

[Ben08] Edward Benson. *The Art of Rails (Programmer to Programmer)*. Wrox Press Ltd., Birmingham, UK, UK, 2008.

[DF09] Bruno De Fraine. *Language Facilities for the Deployment of Reusable Aspects*. PhD thesis, Vrije Universiteit Brussel, 2009. Available at `http://soft.vub.ac.be/soft/_media/members/brunodefraine/phd.pdf`.

[DFES10] Bruno De Fraine, Erik Ernst, and Mario Südholt. Essential AOP: the A calculus. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 101–125, Berlin, Heidelberg, 2010. Springer-Verlag.

[EI11] Erik Ernst and Matthias Diehn Ingesman. Coq source for Lifted Java, 2011. Available at `http://users-cs.au.dk/mdi/liftedJavaCoq.tar.gz`.

[EOC06] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 270–282, New York, NY, USA, 2006. ACM. Available from: `http://doi.acm.org/10.1145/1111037.1111062`, `doi:10.1145/1111037.1111062`.

[Ern01] Erik Ernst. Family polymorphism. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 303–326, London, UK, UK, 2001. Springer-Verlag. Available from: `http://dl.acm.org/citation.cfm?id=646158.680013`.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[Her07] Stephan Herrmann. A precise model for contextual roles: The programming language Object Teams/Java. *Appl. Ontol.*, 2:181–207, 2007.

[HHM04] Stephan Herrmann, Christine Hundt, and Katharina Mehner. Translation polymorphism in Object Teams. Technical report, Technical University Berlin, 2004.

[HHM10] Stephan Herrmann, Christine Hundt, and Marco Mosconi. *OT/J Language Definition*, version 1.3 edition, 2010.

[IE11] Matthias Diehn Ingesman and Erik Ernst. Lifted java: a minimal calculus for translation polymorphism. In *Proceedings of the 49th international conference on Objects, models, components, patterns*, TOOLS'11, pages 179–193, Berlin, Heidelberg, 2011. Springer-Verlag.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001. Available from: `http://doi.acm.org/10.1145/503502.503505`, `doi:10.1145/503502.503505`.

[KHH$^+$01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, volume 1241 of *Lecture*

*Notes in Computer Science*, pages 220–242. Springer Berlin / Heidelberg, 1997. Available from: `http://dx.doi.org/10.1007/BFb0053381`, `doi:10.1007/BFb0053381`.

[Mic] Microsoft. Partial classes and methods (c♯ programming guide). `http://msdn.microsoft.com/en-us/library/wa80x488(v=vs.110).aspx`.

[ML98] Mira Mezini and Karl Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 97–116, New York, NY, USA, 1998. ACM. Available from: `http://doi.acm.org/10.1145/286936.286950`, `doi:10.1145/286936.286950`.

[MMP89] O. L. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '89, pages 397–406, New York, NY, USA, 1989. ACM. Available from: `http://doi.acm.org/10.1145/74877.74919`, `doi:10.1145/74877.74919`.

[MSL00] Mira Mezini, Linda Seiter, and Karl Lieberherr. Component integration with pluggable composite adapters. In *Software Architectures and Component Technology: The State of the Art in Research and Practice.* Kluwer Academic Publishers, 2000.

[Ost02] Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 89–110, London, UK, 2002. Springer-Verlag.

[OW10] Johan Östlund and Tobias Wrigstad. Welterweight java. In *Proceedings of the 48th international conference on Objects, models, components, patterns*, TOOLS'10, pages 97–116, Berlin, Heidelberg, 2010. Springer-Verlag.

[Ser99] Manuel Serrano. Wide classes. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, ECOOP '99, pages 391–415, London, UK, 1999. Springer-Verlag. Available from: `http://dl.acm.org/citation.cfm?id=646156.679848`.

[Sum09] Alexander J. Summers. Modelling java requires state. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*, FTfJP '09, pages 10:1–10:3, New York, NY, USA, 2009. ACM. Available from: `http://doi.acm.org/10.1145/1557898.1557908`, `doi:10.1145/1557898.1557908`.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, November 1994. Available from: `http://dx.doi.org/10.1006/inco.1994.1093`, `doi:10.1006/inco.1994.1093`.

[WSM06] Alessandro Warth, Milan Stanojević, and Todd Millstein. Statically scoped object adaptation with expanders. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 37–56, New

York, NY, USA, 2006. ACM. Available from: http://doi.acm.org/10.1145/1167473.1167477, doi:10.1145/1167473.1167477.

## About the authors

**Matthias Diehn Ingesman** is a graduate student at Aarhus University, Denmark. His primary scientific interests are in the area of programming languages research. Contact him at mdi@cs.au.dk.

**Erik Ernst** is an associate professor at Aarhus University, Denmark. His primary area of interest is programming language design, implementation, and static analysis. He has introduced the notions of family polymorphism, propagating combination (deep mixin composition), and generalized virtual classes, and contributed to the introduction of genericity in Java and the foundational analysis of wildcarded types. Contact him at eernst@cs.au.dk.

**Acknowledgments** We have received very constructive feedback from Stefan Herrmann, the creator of ObjectTeams/Java, who actually adjusted the language in light of our results, and also from anonymous reviewers of this paper and earlier versions thereof.