Reasoning About Class Behavior

Vasileios Koutavas
Northeastern University
vkoutav@ccs.neu.edu

Mitchell Wand
Northeastern University
wand@ccs.neu.edu

Abstract

We present a sound and complete method for reasoning about contextual equivalence between different implementations of classes in an imperative subset of Java. To the extent of our knowledge this is the first such method for a language with unrestricted inheritance, where the context can arbitrarily extend classes to distinguish otherwise equivalent implementations. Similar reasoning techniques for class-based languages [1, 12] don’t consider inheritance at all, or forbid the context from extending related classes. Other techniques that do consider inheritance [3] study whole-program equivalence. Our technique also handles public, private, and protected interfaces of classes, imperative fields, and invocations of callbacks. Using our technique we were able to prove equivalences in examples with higher-order behavior, where previous methods for functional calculi admit limitations [21, 24].

Adding inheritance to a class-based language increases the distinguishing power of the context. Here we show how this extra distinguishing power is reflected in the conditions for equivalence of our technique. Furthermore we show that adding a cast operator is a conservative extension of the language.

1. Introduction

The class is a facility to divide programs into small units that encode different parts of the entire program behavior. This makes classes attractive for reuse and re-implementation. But changing the implementation of a class that is being used in a number of programs comes with the responsibility that the new implementation will not alter the behavior of these programs.

The effect that a change in the implementation of a class has on the behavior of a program that uses it depends greatly on the ways that the program interacts with the class. In a Java-like language (and in the absence of reflection) the surrounding program can interact directly with the class by creating new instances, invoking its public methods, and changing the state of its public fields. It can also interact more indirectly with the class. It can define subclasses that inherit from the original class and instantiate objects, invoke methods, and change the state of fields of these classes. Moreover the subclasses may override methods of the original class and have access to its protected interface.

To formalize the notion of equivalent implementations of classes we adapt the standard notion of contextual equivalence between expressions from functional languages [20] to an equivalence between classes in class-based languages: classes C and C' are contextually equivalent, if and only if, for all class table contexts CT [], expressions e, and the empty store θ, the program configurations (CT [C], θ, e) and (CT [C'], θ, e) have the same operational behavior.

Using this definition directly for proving the equivalence of two sufficiently different implementations of a class is not possible. This is because of the quantification over all class table contexts, but also because it is not strong enough to support an inductive proof which would require us to consider not just equal, but also related stores. CIU theorems [17] ease the quantification over contexts by considering only the evaluation contexts, but they similarly are not strong enough in general to support an inductive proof. Moreover CIU theorems have not been applied to class-based languages.

Another way of reasoning about the behavior of class implementations is by using denotational methods (see [6, 14]). Denotations are usually compositional in the sense that they give the meaning of program fragments without the quantification over contexts. Nevertheless the usual denotational methods distinguish equivalent class implementations that have a different local store behavior. For example the two implementations of a Cell class in Figure 1 would have different denotations because they have different fields. Such equivalences can be dealt with by methods that build logical relations of denotations [5], or exploit properties of some programs, such as ownership confinement [3]. These methods, though, are still not complete with respect to contextual equivalence.

A more natural way to reason about the behavior of two program fragments is by using bisimulations. Bisimulations were introduced by Hennessy and Milner [10] for reasoning about the behavior of concurrent programs. They were applied in sequential calculi by Abramsky [2] and Howe [11] gave a way of proving that they are a congruence. Sumii and Pierce later gave a big-step bisimulation proof technique which is sound and complete with respect to contextual equivalence in a language with dynamic sealing [23] and in a language with recursive and polymorphic types [24]. Their key innovation was to split the sets into parts, and associate each part with the conditions of knowledge under which that part holds. Building on that idea we were able to devise a technique for deriving sound and complete definitions of sets from a context-based semantics [8]. We applied this method to derive definitions of sound and complete big-step bisimulation for a lambda calculus with store [16] and for an imperative object calculus [15]. We used these techniques to prove non-trivial equivalences that involve local store and higher-order procedures [18].

Here we apply the same technique to derive a method for proving equivalence between classes in a subset of Java. One of the differences from our previous work is that, in contrast with expressions, classes are static entities. The contexts of classes are class tables which are also static. These static entities are connected to the dynamic behavior of the program by the instantiations of classes to objects. As a result the conditions that we derive for two classes
to be equivalent are mostly conditions on the possible instances of these classes.

Another difference is that the language we consider here has runtime errors. We treat errors as constants that belong to all types: we require that if an operation on an instance of a class results in an error, then the same operation on the corresponding instance of a related class should also result in the same error.

Inheritance is the most interesting feature of the language that we study here. It increases the distinguishing power of the context: by extending a class, the context can have access to the protected interface of the class, but more importantly it can override some methods of the class and then test the behavior of the rest. We show how this extra distinguishing power is reflected in the conditions for equivalence of our technique.

We start by considering a class-based language with imperative fields and public and private interfaces of classes, but without inheritance. We develop our method for deriving conditions for adequacy between classes for this language and show that adequacy coincides with class equivalence. We then extend the language with inheritance and dynamic dispatch and derive once more conditions for adequacy. By comparing the two sets of conditions we are able to give an account of how inheritance affects class equivalence. By extending the language once more with a cast operator and showing that the conditions for adequacy remain unchanged, we prove that casting is a conservative extension of our language. Moreover we use the Cell example as a case study of proving equivalence in the various versions of the language that we study.

Through these extensions of the language we show how our method can be used incrementally. We also show that in all the extensions we present here the difficulty of proving equivalences does not change significantly.

The contribution of this work is twofold: it gives a sound and complete method of proving equivalence between class implementations in a subset of Java, and also uses this method to study the effect of inheritance and casting on proving contextual equivalence. More specifically:

- To the extent of our knowledge this is the first sound and complete method for proving equivalence between classes in a language where the context can use inheritance to distinguish different implementations of the same class. Similar techniques for class-based languages either don’t consider inheritance at all [1], or don’t allow the context to extend the related classes [12]. Other techniques that do consider inheritance [3] study whole-program equivalence.
- The proof technique we present here can be used to show equivalent classes with different store behavior, where the usual denotational methods admit limitations, and classes that invoke callbacks, a higher-order feature of object-oriented programming.
- We also give an account of how inheritance affects equivalence checking of classes in our technique. We do this by applying our technique to a class-based language without inheritance and to its extension with inheritance, and by deriving the necessary conditions that classes need to satisfy in order to be equivalent in these languages. Equivalence checking has not been studied before in a language with inheritance.
- Finally we show in a similar way that adding a casting operator is a conservative extension of the language.

The structure of the paper is as follows: in Section 2 we give the semantics of a base language with classes, but without inheritance and casting. In Section 3 we apply our method of deriving conditions for equivalence to the base language. In Section 4 we give an example of proving equivalence between two implementations of a Cell class by constructing an appropriate set and proving that it satisfies the derived conditions. In Section 5 we extend the language with inheritance and protected fields and in Section 6 we show how the conditions for equivalence change due to this extension. Section 7 shows that the Cell example is still provable in the second language, showing that interesting equivalences still hold after the addition of inheritance. Section 8 adds casting to the language and Section 9 shows that this doesn’t affect the conditions for equivalence. In Section 10 we show how to prove the equivalence of two classes that invoke callbacks. Finally, in Sections 11 and 12, we discuss related work and conclude.

2. $J_1$: A Basic Class-Based Language

We start by defining a small class-based language which we call $J_1$. This language is a subset of Java containing class definitions, imperative private and public fields, private and public methods, ground types, constants, conditional, and a let expression. However, it does not allow classes to inherit behavior from other classes and to override methods. The syntax of $J_1$ is shown in Figure 2.

The main difference between $J_1$ and other imperative Java calculi, like Middleweight Java [4] and Classic Java [9], is that $J_1$ does not have inheritance and casting. We later extend $J_1$ with inheritance to create the language $J_2$, and then we add a casting operator to create $J_3$. $J_3$ has the same constructs as Middleweight Java, with the addition of constants and access modifiers. $J_3$ doesn’t have the explicit interfaces and mixings of Classic Java.

The values of the language are constants or locations in the store where objects are stored. Stored objects are structures that contain the name of the class which they instantiate, and a binding for each field of the class to a value of the appropriate type. A program can test for pointer equivalence of objects with the operator $\text{eq}$.

The types of $J_1$ are class types, as well as the ground types $\text{void}$, $\text{int}$, and $\text{bool}$. There is a unit constant of type $\text{void}$, $\text{true}$ and $\text{false}$ of type $\text{bool}$, and the integers of type $\text{int}$. The constant $\text{null}$ has any class type.

```
class Cell extends Object {
  private Object c;
  Cell() {this.c = null;}
  public void set(Object o) {this.c = o;}
  public Object get() {this.c;}
}
```

```
class Cell extends Object {
  private Object c1, c2;
  private int n;
  Cell() {this.c1 = null; this.c2 = null; this.n = 0;}
  public void set(Object o) {this.c1 = o; this.c2 = o;}
  public Object get() {
    this.n = this.n + 1;
    if ((this.n) % 2) == 0 then this.c1; else this.c2;
  }
}
```

Figure 1. Two implementations of a Cell class.
$J_1$ has also a null error ($\text{nerr}$) for the case that a program tries to perform an operation on a null value. 

Class definitions state the name of a class, its fields and methods, and also its constructor. The public and private access modifiers in the definitions of fields and methods specify the scope of these names. Public methods and fields are visible to all classes, while private methods and fields are visible only from within the same class. To aid type-checking we assume an annotation of every method call and field usage is performed, in the style of Classic Java [9]. This annotation is evident in the typing rules for method calls, field update, and field dereferencing, and is left implicit in all other places. The meta-function \text{accessible} returns the set of names (fields and methods) of the class given as its second argument that are visible from within the class given as its third argument.

Class tables are sets of classes. Well-formed class tables contain classes with distinct names. We test the well-formedness of a class table by the predicate $\text{wfcClassHierarchy}(\text{CT})$. When we add inheritance later on, this may be the superclass of the object currently stored in a location. In this way stores are used as store typings in the typing judgments. The constant $\text{nnull}$ has all class type defined in the class table. The rest of the typing rules for expressions are the expected ones for a language like $J_1$.

The typing judgments for method, class, and class-table definitions are $\text{CT} \vdash \text{M} : \text{OK}$ in $C$, $\text{CT} \vdash \text{C} : \text{OK}$, and $\text{CT} : \text{OK}$ respectively. $(\text{CT} \vdash s, c) : \text{OK}$ is the typing judgment for program configurations.

In Figure 4 we give a small-step semantics for $J_1$. A small-step $\text{CT} \vdash \text{C}, e \rightarrow_{s, c} \text{s}_1, \text{c}_1$ describes a transition from the program configuration $\text{CT} \vdash \text{C}, e$ to the configuration $\text{CT} \vdash \text{s}_1, e_1$. We also define $\rightarrow^*$ to be the reflexive and transitive closure of $\rightarrow$, and $\rightarrow^c$ the reflexive and up to $k - 1$ steps transitive closure of $\rightarrow$. We also write $\text{CT} \vdash \text{s}, e$ if there exists $\text{s}_1, \text{w}$, such that $\text{CT} \vdash \text{s}, e \rightarrow^c \text{s}_1, \text{w}$.

We use calligraphic font for the meta-identifiers that denote class table, class, constructor, or method definitions. We also use an overbar notation to denote syntactic sequence with arbitrary length. When expanded, all the meta-identifiers in the sequence are annotated with the appropriate subscripts; e.g. we write $\text{obj C} \{f = v\}$ instead of $\text{obj C} \{f_1 = l_1, f_2 = l_2, \ldots, f_s = l_s\}$. We also use the notation $\overline{f = v \backslash f_i = u}$ to denote the sequence $f_1 = v_1, \ldots, f_i-1 = v_{i-1}, f_i = u, f_{i+1} = u_{i+1}, \ldots$.

### 3. Equivalence and Adequacy in $J_1$

We want to study contextual equivalence of class definitions in $J_1$ and its extensions. To do this we need to define class table contexts, relations on classes, and their extension to class tables.

**Definition 3.1.** A class table context, $\text{CT} \mid \cdot$, is a set of class definitions. Placing a class definition, or a sequence of class definitions, in the hole of a class table context corresponds to set union:

$$
\text{CT} \mid \cdot \cup \{C\}
$$

**Definition 3.2.** $\text{R}^{\text{cls}}$ is a relation on classes if it is a set of pairs of class definitions, such that for all $(\langle C, C' \rangle) \in \text{R}^{\text{cls}}$, and all class-
\[ CT; \Gamma; s \vdash e : t \]

\[
\begin{aligned}
\text{x} \in \Gamma &\vdash x : t \\
CT; \Gamma; s \vdash \text{unit: void} \\
c \in \{\text{true}, \text{false}\} &\vdash c : \text{bool} \\
CT; \Gamma; s \vdash \text{c: int} \\
\text{class C\{\ldots\} \in CT} \\
CT; \Gamma; s \vdash \text{null: C} \\
\end{aligned}
\]

\[
\begin{aligned}
C \in \text{CT}. \text{classnames} &\vdash s.l. \text{type} = C \\
CT; \Gamma; s \vdash \text{new C: C} \\
c \in \{\pm 1, \pm 2, \ldots\} &\vdash c : \text{int} \\
CT; \Gamma; s \vdash \text{c: int} \\
\text{class C\{\ldots\} \in CT} \\
\end{aligned}
\]

\[
\begin{aligned}
f \in \text{accessible(CT, C, C_0)} &\vdash \text{mod f \in CT.C.fields} \\
f \in \text{accessible(CT, C, C_0)} &\vdash \text{m \in accessible(CT, C, C_0)} \\
\text{mod f \in CT.C.fields} &\vdash \text{mod f \in CT.C.methods} \\
f \in \text{accessible(CT, C, C_0)} &\vdash \text{m \in accessible(CT, C, C_0)} \\
\text{mod f \in CT.C.methods} &\vdash \text{mod f \in CT.C.methods} \\
\end{aligned}
\]

\[
\begin{aligned}
\text{CT; \Gamma; s \vdash e: C} \\
f \in \text{accessible(CT, C, C_0)} &\vdash \text{mod f \in CT.C.fields} \\
\text{f \in accessible(CT, C, C_0)} &\vdash \text{mod f \in CT.C.methods} \\
\text{CT; \Gamma; s \vdash e: C} \\
\end{aligned}
\]

\[
\begin{aligned}
\text{CT; \Gamma; s \vdash e_1: t} \\
\text{CT; \Gamma; s \vdash e_2: t} \\
\text{if e then e_1 else e_2: t} \\
\end{aligned}
\]

\[
\begin{aligned}
\text{CT; \Gamma; s \vdash e \rightarrow s_1, e_1} \\
\text{CT; \Gamma; s \vdash E[e] \rightarrow s_1, E[e_1]} \\
\end{aligned}
\]

**Figure 3. Typing of \( J_1 \)**

**Evaluation Contexts**

\[
E ::= \mathcal{E} \mid E[E] \\
\mathcal{E} ::= [] \mid f_1 \mid \ldots \cdot f_n \mid v.m(\overline{v}) \mid v.m(\overline{v}, \overline{v}) \mid \text{let } x = \mathcal{E} \text{ in } \mathcal{E} \mid \text{if } \mathcal{E} \text{ then } e \text{ else } e \mid \text{op}(\overline{v}, \overline{v})
\]

\[
\begin{aligned}
\text{CT} &\vdash s, E[e] \rightarrow s_1, E[e_1] \\
\text{CT} &\vdash s, E[e] \rightarrow s_1, E[e_1] \\
\text{CT} &\vdash s, E[e] \rightarrow s_1, E[e_1] \\
\end{aligned}
\]

**Figure 4. Small-step Operational Semantics of \( J_1 \)**

\[ \text{table contexts } CT[\cdot]: \]

\[ CT[\cdot]: \text{OK} \iff CT[\cdot']: \text{OK} \]

The above definition requires that we relate only class definitions that are interchangeable at compile time; i.e. replacing one with the other in a class-table context doesn’t affect the typing judgment of the program. In practice this means that the related classes have the same name and the same public interface.

**Definition 3.3.** If \( R^{\text{cls}} \) is a relation on classes, then the following is its extension to class tables:

\[ CT[R^{\text{cls}}] \overset{\text{def}}{=} \{ (CT[\overline{C}], CT[\overline{C}]) \mid (\overline{C}, \overline{C}) \in R^{\text{cls}}, CT[\overline{C}]: \text{OK} \} \]
We give the following definition of contextual equivalence for $J_1$:

**Definition 3.4 (Contextual Equivalence ($\equiv$)).** $\equiv$ is the largest relation on classes such that for all $(CT, CT') \in CT[\equiv]$, expressions $e$, and types $t$, such that $CT; \emptyset; \emptyset \vdash e : t$ we have:

$$CT \vdash \emptyset, e \iff CT' \vdash \emptyset, e$$

This definition is hard to use directly to show two classes equivalent. To discover more specific conditions that assumed equivalent classes need to satisfy we follow the method we developed in [16, 15] to derive a usable proof technique of equivalence for imperative higher-order languages. This method relies on the definition of adequate relations that imply contextual equivalence, and a proof construction scheme to discover a set of necessary conditions for adequacy. Here we show how to apply this method to $J_1$.

We will define adequacy as a property of the following relations.

**Definition 3.5.** A $J$-relation $R$ is a set of tuples $(s, s', R^f, R^{e1a})$, where $s, s'$ are stores, $R^f$ is a relation on object references, and $R^{e1a}$ is a relation on classes.

For the definition of adequacy we also need to define $R$-related answers and expressions.

**Definition 3.6.** If $R^e$ is a relation on object references, $CT, CT'$ class tables, and $s, s'$ stores, then we define the following relations:

- related values of type $t$:
  $$V_t[CT, CT', s, s', R^e] \overset{\text{def}}{=} \{(t, t') \mid (t, t') \in R^e, CT; \emptyset; \emptyset \vdash t : t, CT'; \emptyset; \emptyset \vdash t' : t\} \cup \{(s, c) \mid CT; \emptyset; \emptyset \vdash c : t, CT'; \emptyset; \emptyset \vdash c : t\}$$

- related answers of type $t$:
  $$A_t[CT, CT', s, s', R^e] \overset{\text{def}}{=} V_t[CT, CT', s, s', R^e] \cup \{(\text{nerr}, \text{nerr})\}$$

- and related expressions of type $t$:
  $$E_t[CT, CT', s, s', R^e] \overset{\text{def}}{=} \{([\ell/x]e, [\ell'/x]e) \mid \text{FV}(e) \subseteq \{\ell, \ell'\}, (\ell, \ell') \in R^e, \text{Locs}(e) = \emptyset, CT; \emptyset; \emptyset \vdash [\ell/x]e : t, CT'; \emptyset; \emptyset \vdash [\ell'/x]e : t\}$$

The above definition describes when terms are considered to be related. Values are related when they are $R$-related locations, or identical constants; final answers are related when they are related values or identical errors; expressions are related when they are constructed by the same expression context with related values in its holes (which boils down to just related locations since identical constants are subsumed by the identical contexts $e$). Furthermore, by requiring that $\text{Locs}(c) = \emptyset$, the above definition forces all $R$-related expressions to use only $R$-related references.

We now give the definition of adequate $J$-relations. These are $J$-relations from which all derived related program configurations have the same operational behavior.

**Definition 3.7 (Adequacy).** $R$ is adequate if and only if:

$$\forall (s, s', R^f, R^{e1a}) \in R, \forall (CT, CT') \in CT[R^{e1a}], \forall t, \forall (e, e') \in E_t[CT, CT', s, s', R^f], \forall s_1, w, (CT \vdash s, e \rightarrow^* s_1, w) \Rightarrow \exists s_1', w', R^{e1a}_1, (CT' \vdash s', e' \rightarrow^* s_1', w') \land ((w, w') \in A_2[CT, CT', s_1, s_1', R^{e1a}_1]) \land ((s_1, s_1', R^{e1a}_1, R^{e1a} \in R) \land (R^f \subseteq R^{e1a})$$

and the reverse.

In this definition of adequacy, the most interesting quantification is the one on related expressions of type $t$. This quantification is general enough so that we can carry out an induction based on the definition. It covers the case where related methods are invoked on related (and not just equal) objects and are passed related arguments. The quantification over class tables plays a role in the soundness of this definition, but not in the existence of an induction. It is there to merely type-check the quantified expressions. Adequate relations are sound and complete in the following way.

**Theorem 3.8 (Soundness).** If $R$ is adequate and $(\emptyset, \emptyset, \emptyset, R^{e1a}) \in R$ then $R^{e1a} \subseteq \equiv$.

**Proof.** Immediate by the definitions of adequacy and contextual equivalence.

**Theorem 3.9 (Completeness).** If $R^{e1a} \subseteq \equiv$ then there exists adequate $R$ with $(\emptyset, \emptyset, \emptyset, R^{e1a}) \in R$.

**Proof.** (Sketch) For any $R^{e1a}$, if there is no such $R$, then it means that there are $(CT, CT') \in CT[R^{e1a}]$ and a well-typed $e$ that, starting from the state $(\emptyset, \emptyset, \emptyset, R^{e1a})$, invalidate Definition 3.7. Then show that we can always construct $e'$ from $e$, such that $CT \vdash \emptyset, e] and $CT' \vdash \emptyset, e]$, or the opposite.

One could show the equivalence between two class implementations by constructing an appropriate set $R$ and then prove its adequacy by an induction based on Definition 3.7. Our goal is to improve this by finding sufficient conditions on $R$ that would make the proof of adequacy go through. These conditions act as a Verification Condition Generator: one provides an invariant (the set $R$) that presumably proves a particular equivalence, and the conditions say what pieces need to be proven in order to check the validity of the invariant. These conditions encode the distinguishing power of the context.

We find a sufficient set of such conditions by investigating a class of inductive proofs based on Definition 3.7. We abstract over the concrete structure of $R$ and attempt to prove adequacy. We push the induction as much as possible just by using the induction hypothesis, and when this is not possible we find the properties that $R$ should satisfy to complete the proof. These properties become the proof obligations, or verification conditions, for $R$.

**Task 3.10.** Given a relation $R$, construct a proof that $R$ is adequate.

**Proof Construction Scheme.** Given an $R$, the adequacy proof would consist of two inductions, one for the forward direction of Definition 3.7 and one for the reverse direction. The induction hypothesis
of the former is:

$$IH(k) =$$

$$\forall s, s', R^i, R^{i+1} \in \mathbb{R},$$

$$\forall (CT, CT') \in CT[R^{i+1}]$$

$$\forall t, (e, e') \in E_t(CT, CT', s, s', R^i)$$

$$\forall s_1, w_1,$$

$$((CT \vdash s, e \leftarrow s_1, w_1)$$

$$\implies \exists s_1', w_1', R^i_1,$$

$$(CT' \vdash s', e' \leftarrow s_1', w_1')$$

$$\land ((s_1, s_1', s_1, w_1) \in V_t)$$

$$\land (R^i \subseteq R^{i+1})$$

We will show that for all k, $IH(k)$ holds. We assume the induction hypothesis for k, and we will show that it holds for $k + 1$.

Let $(e, e') = ((v/x)e_0, (v'/x)e_0)$, for some $e_0, v, v'$, such that $FV(e_0) \subseteq \{\mathcal{F}\}$. $	ext{Locs}(e_0) = \emptyset, (v, v') \in R^i, CT; \emptyset, s \vdash [v/x]e_0, CT'; \emptyset, s' \vdash [v'/x]e_0, CT$. We proceed by cases on $e_0$. We push the proof in each case as much as possible just by using the induction hypothesis. When a case is not proven entirely by the induction hypothesis we find the extra properties that $R$ must satisfy to complete the proof of that case. These properties become the proof obligations for $R$.

Due to limited space we do not show the case analysis of this proof which is similar to the ones in [15, 16]. □

In Theorem 3.13 we summarize all proof obligations for $R$ that we found by the above proof construction scheme. First we give a notation to write down the inductive cases and the induction hypotheses (one for each direction).

**Definition 3.11 (Inductive Cases).**

$$R^i, R^{i+1} \vdash (CT \vdash s, e; t) \subseteq^k (CT' \vdash s', e'; t)$$

$$\forall s_1, u_1,$$

$$(e, e') \in E_t(CT, CT', s, s', R^i)$$

$$\land (CT \vdash s, e \leftarrow s_1, w_1)$$

$$\implies \exists s_1', u_1', R^i_1,$$

$$(CT' \vdash s', e' \leftarrow s_1', u_1')$$

$$\land ((s_1, s_1', s_1, w_1) \in V_t)$$

$$\land (R^i \subseteq R^{i+1})$$

and

$$R^i, R^{i+1}, R \vdash (CT \vdash s, e; t) \subseteq^k (CT' \vdash s', e'; t)$$

for the reverse.

**Definition 3.12 (Inductive Hypotheses).**

$$IH^R(k) \equiv$$

$$\forall s, s', R^i, R^{i+1} \in \mathbb{R},$$

$$\forall (CT, CT') \in CT[R^{i+1}]$$

$$\forall t, (e, e') \in E_t(CT, CT', s, s', R^i).$$

$$R^i, R^{i+1}, R \vdash (CT \vdash s, e; t) \subseteq^k (CT' \vdash s', e'; t)$$

and

$$IH^R(k)$$

for the reverse.

Our main theorem is the following.

**Theorem 3.13 (Adequacy Conditions).** A relation $R$ is adequate if and only if for all states $(s, s', R^i, R^{i+1})$ of $R$ and for all $(CT, CT') \in CT[R^{i+1}]$, the following conditions are satisfied:

1. (Same interfaces) For all $(C, C') \in R^{i+1}$,

$$C = \text{class } C\{\text{publication f1; private l f2;}$$

$$\text{state s; l1 := c1, this.c2 := c2},$$

$$\text{publication t m (m, t x) \{\ldots\}},$$

$$private t m (m, t x) \{\ldots\} \}$$

$$C' = \text{class } C\{\text{publication f1; private l f2;}$$

$$\text{state s; l1 := c1, this.c2 := c2},$$

$$\text{publication t m (m, t x) \{\ldots\}},$$

$$private t m (m, t x) \{\ldots\} \}$$

2. (Related instances) For all $(l, l') \in R^i$, there exists $t$, such that $CT; \emptyset; s \vdash l.t$, $CT'; \emptyset, s' \vdash l'.t$.

3. (Enough instances) For all $C \in CT\text{-classes, with}$

$$C = \text{class } C\{\cdot\cdot\cdot C()\{\text{this.f := c} , \ldots\}$$

$$C' = \text{class } C\{\cdot\cdot\cdot C()\{\text{this.f := c} , \ldots\}$$

and all fresh $l, l'$, there exists $R^i_2 \supseteq R^i \cup \{(l, l')\}$, such that

$$(s[l := \text{obj } C\{\text{t = c}\}],$$

$s[l' := \text{obj } C\{\text{t = c}\}],$$

$R^i_2, R^{i+1} \in \mathbb{R}.$

4. (Related public fields) For all $(l, l') \in R^i$, with $s.l = \text{obj } C\{\text{t = v}\}$$

and $s'.l' = \text{obj } C\{\text{t' = v}\}$, for all public $t, f_i \in C\text{-fields,}$

$$(v, v') \in V_t[CT, CT', s, s', R^i].$$

5. (Related updates) For all $(l, l') \in R^i$, with $s.l = \text{obj } C\{\text{t = v}\}$$

and $s'.l' = \text{obj } C\{\text{t' = v}\}$, for all public $t, f_i \in C\text{-fields,}$

and all $(u, u') \in V_t[CT, CT', s, s', R^i]$, such that

$$(s[l := \text{obj } C\{\text{t = c}\}],$$

$s[l' := \text{obj } C\{\text{t' = c}\}],$$

$$R^i, R^{i+1} \in \mathbb{R}.$$

6. (Related public methods) For all $(C, C') \in R^{i+1}$, all references $(l, l') \in V_t[CT, CT', s, s', R^i]$, all methods $m$ with $CT.C.m = \text{public } t.m (t.x) \{e\}$,

$$CT'.C.m = \text{public } t.m (t.x) \{e'\},$$

and for all $(v, v') \in V_t[CT, CT', s, s', R^i]$, such that

$$IH^R(k) \implies$$

$$R^i, R^{i+1}, R \vdash (CT \vdash s, l.m(\emptyset); m) \subseteq^k (CT' \vdash s', l.m(\emptyset); m)$$

$$IH^R(k) \implies$$

$$R^i, R^{i+1}, R \vdash (CT \vdash s, l.m(\emptyset); m) \subseteq^k (CT' \vdash s', l.m(\emptyset); m)$$

Proof. By recapitulating the proof construction scheme of Task 3.10. □

The first condition of Theorem 3.13 requires that related classes have the same public interface. This ensures that the classes are not distinguishable at compile time. The second condition requires that related objects respect the class types. The rest of the conditions correspond to the primitive operations that the context can perform on the objects in order to distinguish them. Thus the third condition corresponds to instantiating a new object by the context, the fourth condition corresponds to dereferencing a public field, the fifth to updating a public field, and the last to invoking a public method.

The above theorem contains a top-level quantification over all possible class tables that contain the related classes. This is necessary in order to specify the well-typed values, and to use the reduction relation. As we will see in the example that follows, this quantification does not introduce any difficulty in the proofs of equivalence. This is because we never need to reason about the behavior of
methods and classes defined in the class-table contexts since these cases are handled by the induction hypothesis.

4. The Cell Example

Here we give two implementations of a `Cell` class that store objects of some class `A` that is provided by the context. The first implementation of `Cell` is the usual one, while the other uses two private fields to keep the stored object, and a counter to decide which one to return when the get method is invoked. These implementations have sufficiently different store behavior and the usual denotational models would assign different denotations and thus distinguish them [25, 19].

\[ C = \text{class Cell} \{
\text{private A c;}
\text{Cell()} \{ \text{this.c} := \text{null} \}
\text{public void set(A o) \{ this.c := o \}}
\text{public A get()} \{ \text{this.c} \}
\} \]

\[ C' = \text{class Cell} \{
\text{private A c1, c2;}
\text{private int n;}
\text{Cell()} \{ \text{this.c1} := \text{null};
\text{this.c2} := \text{null};
\text{this.n} := 0 \}
\text{public void set(A o) \{ \text{this.c1} := o; \text{this.c2} := o; \}}
\text{public A get()} \{ \text{this.n} := \text{this.n} + 1;
\text{if (even(this.n)) then this.c1}
\text{else this.c2} \} \}
\]

To prove the above two class implementations equivalent we construct the following set:

\[ R = \{ (s, s', R', R'^{\text{cls}}) \mid \exists CT, CT', D, f_1, v_1, v_2, l_s, l'_s, l_c, l'_c, m : \]

\[ s = l_D \text{obj Cell}(c = f_1(v_1)) \]
\[ l_s = \text{obj A} \end{equation}
\[ l_c = \text{obj Cell}(c = l_s) \]
\[ s' = l'_D \text{obj Cell}(c = l'_s) \]
\[ l'_s = \text{obj A} \end{equation}
\[ l'_c = \text{obj Cell}(c = l'_s) \]
\[ R^{\text{cls}} = \{ (C, C') \} \]
\[ R' = \{ (l_D, l'_D), (l_s, l'_s), (l_c, l'_c) \}
\[ \{ (v_1, v'_1), (v_2, v'_2) \in V_C[CT, CT', s, s', R'] \}
\[ (CT, CT') \in CT[R'^{\text{cls}}] \}
\]

We choose this particular set by inspecting the conditions of Definition 3.13. Condition 1 is obviously satisfied by `C` and `C'`. To satisfy conditions 2 and 3 we add in `R'` the related references to any class of all possible class tables. These are objects of the classes `Cell` and `A`, as well as, objects of any other class `D` that may be defined and instantiated by the context. The values in the fields of the instances of `D` and `A` are related in `V_C[CT, CT', s, s', R']`, since these are fields of identical classes in the class tables.

We also require that the values stored in the private fields of `Cell` to be related references of `A` objects. This is an invariant of the equivalence between the two implementations of `Cell` and is going to help us prove condition 6.

To prove condition 6 we consider an arbitrary tuple `(s, s', R', R'^{\text{cls}}) \in R`, and an arbitrary pair of related class tables `(CT, CT') \in CT[R'^{\text{cls}}]`. The only pair of related class definitions in `R'^{\text{cls}}` is `(C, C')`. For any `(l_s, l'_s) \in V_{\text{cell}}[CT, CT', s, s', R']` with `s.l_s = \text{obj Cell}(c = l_s)`, `s'.l'_s = \text{obj Cell}(c = l'_s, c2 = l'_s, n = m)`, we consider all `public t_s \rightarrow t_m \in CT.Cell.methods`. These are the methods get and set.

In the case of the get we have `l_s \rightarrow t = \text{void} \rightarrow A`. Furthermore:

\[ CT \vdash s, l_s \text{get()} \rightarrow^* s, l_s \]
\[ CT' \vdash s', l'_s \text{get()} \rightarrow^* s'[l'_s \leftarrow \text{obj Cell}(c = l'_s, c2 = l'_s, n = m + 1)], l'_s \]

Moreover:

\[ (l_s, l'_s) \in V_C[CT, CT', s, s', R'] \]
\[ (s, s'', R', R'^{\text{cls}}) \in R \]

where `s'' = s'[l'_s \leftarrow \text{obj Cell}(c = l'_s, c2 = l'_s, n = m + 1)]`

Similarly for the case of set we have `l_s \rightarrow t = \text{unit}`. Let `(l_s, l'_s) \in V_C[CT, CT', s, s', R']`. We have:

\[ CT \vdash s, o \text{set}(l_s) \rightarrow^* s[l_s \leftarrow \text{obj Cell}(c = l_s)], \text{unit} \]
\[ CT' \vdash s', o' \text{set}(l'_s) \rightarrow^* s'[l'_s \leftarrow \text{obj Cell}(c = l'_s, c2 = l'_s, n = m)], \text{unit} \]
\[ (s[l_s \leftarrow \text{obj Cell}(c = l_s)], s'[l'_s \leftarrow \text{obj Cell}(c = l'_s, c2 = l'_s, n = m)], R', R'^{\text{cls}}) \in R \]

5. \( J_2 \): An Extension of \( J_1 \) With Inheritance

We now extend \( J_1 \) by adding the feature of class inheritance. We also add the protected access modifier for fields and methods. Protected fields and methods are accessible from the same class and its subclasses. We do not yet add a cast operation to the language, allowing, thus, only implicit (and type-safe) upcasting of objects. We assume there is no shadowing of fields, something that can be accomplished automatically by adding the name of the class as part of the name of each field. The differences in the syntax, typing, and operational semantics of \( J_1 \) and \( J_2 \) are shown in Figure 5.

To encode inheritance we have added the rule of subsumption and the subtyping judgments imposed by the class hierarchy and the reflexive and transitive property. The rest of the typing rules of the form `CT; \Gamma; s \vdash e : t` have only trivial changes, assuming that the implementation of the meta-function `accessible handles protected fields and methods in the right way. Furthermore we have changed the rule for type-checking method definitions to check that subclasses override only public and protected methods. A valid class hierarchy is now considered to be a tree, with an empty class `Object` as its root. In the typing judgments of \( J_2 \), the meta-function `w/classHierarchy` is true exactly when these conditions hold for a set of classes.

The operational semantics of \( J_2 \) are mostly the same as of \( J_1 \), with the exception of the steps that involve object instantiation. In \( J_2 \), when a new object is created, the fields from all the superclasses must be initialized.

6. Adequacy in \( J_2 \)

In this section we study adequacy for \( J_2 \). The definition of contextual equivalence we gave earlier (Definition 3.4) still holds for \( J_2 \). It conceals though the fact that the context has what seems to be more distinguishing power. It can extend related classes and then use these extensions to distinguish the two sides. We apply
our method for deriving conditions for adequacy to study this extra distinguishing ability of the context. This reveals the effect of inheritance to the equivalence of classes.

The outline of our technique is the same as before. We reason about the same sets of tuples \( R \), and we use the same definitions for \( \Psi, \Delta, \), and \( E_1 \) (Definition 3.6). The latter sets, though, are larger than before. In \( J_2 \), the type of an object is the name of the class it instantiates, but, because of subsumption, it may also be the name of any of its superclasses. This is the place where inheritance appears in our technique.

The definition of adequacy (Definition 3.7) remains unchanged for \( J_2 \), and, as a consequence, the induction hypothesis of the corresponding proof construction scheme also remains the same. When unwinding this proof, and taking cases on the structure of \( e_0 \), we need to consider sub-cases introduced by inheritance. For example when method \( m \) is defined in a class \( C, e_0 = x.m() \), and \( x \) is substituted by an object reference, then this object may be an instantiation of a subclass \( D \) of \( C \). It also means that \( m \) may have been overridden between \( C \) and \( D \). More explicitly the conditions for adequacy that we derive for \( J_2 \) are the following:

1. (Same interfaces) For all \((C, C') \in R^{13a}\),
   \[
   C = \text{class } C \{ \text{public } f_1; \text{private } f_2 \};
   \]
   \[
   C' = \text{class } C' \{ \text{public } f_1'; \text{private } f_2';
   \]

2. (Related instances) For all \((l, l') \in R^t \), there exists \( t \), such that \( CT; \emptyset; s \vdash t.t, CT'; \emptyset; s' \vdash t'.t \).
3. (Enough instances) For all \( C \in CT_{\text{classnames}} \) with \( CT.C.\text{fields} = \text{mod } f \), and \( CT'.C.\text{fields} = \text{mod } f' \) and all fresh \( l, l'\), there exists \( R^t \geq R^t \cup \{ (l, l') \} \) such that

   \[
   (s|l = \text{obj } C \{ f = e \}, s'|l = \text{obj } C' \{ f' = e' \}, R^t, R^{13a}) \in R,
   \]

where \( \tau \) and \( \overline{\tau} \) are the initial values of the fields in the constructors of \( C \) and its superclasses.

---

**Theorem 6.1 (Adequacy Conditions for \( J_2 \)).** A relation \( R \) is adequate if and only if for all states \((s, s', R^t, R^{13a}) \) of \( R \), and for all \((CT, CT') \in CT[R^{13a}]\), the following conditions are satisfied:
4. (Related public and protected methods) For all \((l, l') \in R'\), with \(s.l = \text{obj } C \{f = v\}\) and \(s'.l' = \text{obj } C \{f' = v'\}\), and for all \(\mod t, f_i \in C\). fields, with \(\mod \in \{\text{public}, \text{protected}\}\):

\[(v, v') \in V_i[CT, CT', s, s', R']\.

5. (Related updates) For all \((l, l') \in R'\), with \(s.l = \text{obj } C \{f = v\}\) and \(s'.l' = \text{obj } C \{f' = v'\}\), all \(\mod t, f_i \in C.T.C\). fields, with \(\mod \in \{\text{public}, \text{protected}\}\), and all \((u, u') \in V_i[CT, CT', s, s', R']\):

\[(s[l \mapsto \text{obj } C \{f = v\}] f_i = u)]

\[(s'[l' \mapsto \text{obj } C \{f = v\}] f_i = u)]

\[R_i, R^{\text{cls}} \in \mathbb{R}\]

6. (Related public and protected methods) For all \((C, C') \in R^{\text{cls}}\), all references \((l, l') \in V[C]CT, CT', s, s', R']\), all methods \(m\) defined in \(C\) with \(\mod t, m(t_x)\{v\}_s\), \(CT'.m = \mod t, m(t_x)\{v\}_s\), and \(\mod \in \{\text{public}, \text{protected}\}\), and for all \((v, v') \in V[C]CT, CT', s, s', R']\):

\[IH_X^S(k) \implies R_i, R^{\text{cls}} \vdash (CT \vdash s, l, m(\overline{\pi}): \mod m_t) \models^{<k+1} (CT' \vdash s', l', m(\overline{\pi}): \mod m_t)\]

\[IH_X^S(k) \implies R_i, R^{\text{cls}} \vdash (CT \vdash s, l, m(\overline{\pi}): \mod m_t) \models^{<k+1} (CT' \vdash s', l', m(\overline{\pi}): \mod m_t)\]

The differences between these conditions and the conditions of Theorem 3.13 are the following:

- Condition 1 requires that related classes have the same protected interface. This is because the context can extend a class and have access to its protected fields and methods.
- In condition 3 new objects contain all the fields of their superclasses.
- Condition 4 requires that also protected fields contain related values. If we assume that we have two presumably equivalent classes \(C\) and \(C'\) that contain a protected field \(f\) which may not contain related values, then the context can distinguish the two implementations by extending them with the following class:

```java
class D extends C {
    public t getf() {this.f}
}
```

Then by instantiating the invoking methods of \(C\) until \(f\) does not contain related values in the two implementations, and then invoking the method `getf`, it can distinguish the two sides.

- Similarly condition 5 requires that related updates should also apply to protected fields. If two implementations of a class \(C\) are not operationally equivalent when one of their protected fields (e.g. \(f\)) is updated with equivalent values, then the context can distinguish these implementations by extending them with the class:

```java
class D extends C {
    public void setf(t x) {this.f := x}
}
```

- In a similar fashion, in condition 6 we need to test the behavior of any related protected (in addition to public) method \(m\) that is defined in any \((C, C') \in R^{\text{cls}}\). These methods may be invoked on objects that instantiate subclasses of \(C\) and \(C'\). The case that \(m\) is overridden by a subclass defined in the context is handled by the induction hypothesis. The only non-trivial case is when \(m\) is defined in a class in \(R^{\text{cls}}\). Furthermore, it may be the case that \(m\) invokes internally another method \(g\) of the same object which has been overridden by a subclass defined in the context. Since \(g\) is overridden, it must be that \(g\) is a public or protected method, and internal calls to them are handled by the induction hypothesis in our method.

From the above we conclude that adding inheritance to \(J_1\) does not affect the local reasoning of condition 6 when using our technique. This is because of the wide application of the induction hypothesis to factor out many of the sub-cases.

For classes that don't use protected fields and methods, we observe that conditions 1-5 of Theorem 3.13 imply the corresponding conditions of Theorem 6.1. This is not true, though, for condition 6, where in Theorem 6.1 method invocations are tested on objects that instantiate any subclass of related classes, while in Theorem 3.13 are tested only on objects of related classes. For example the following two classes are equivalent in \(J_1\) but not in \(J_2\):

```java
C = class C {
    C() {}
    public bool m1() {true}
    public bool m2() {this.m1()}
}
```

```java
C' = class C {
    C() {}
    public bool m1() {true}
    public bool m2() {true}
}
```

A context containing the following subclass of \(C\) can distinguish the two implementations:

```java
class D extends C {
    D() {super()}
    public bool m1() {false}
}
```

Therefore, extending the language with inheritance gives more distinguishing power to the context. By overriding methods the context can break invariants that hold for public and protected methods and are needed by other methods. The only way for a class to maintain these invariants is to declare the methods that need to satisfy them private.

7. The Cell Example in \(J_2\)

We adapt the two implementations of a Cell from Section 4 to the language \(J_2\), and we show that the equivalence still holds.

```java
C = class Cell extends Object {
    private A c;
    Cell() {this.c := null}
    public void set(A o) {this.c := o}
    public A get() {this.c}
}
```

```java
C' = class Cell extends Object {
    private A c1, c2;
    private int n;
    Cell() {
        this.c1 := null;
        this.c2 := null;
        this.n := 0
    }
    public void set(A o) {
        this.c1 := o;
        this.c2 := o
    }
    public A get() {
        if even(this.n) then this.c1
        else this.c2
    }
}
```

To prove the above two class implementations equivalent we construct the following set:
\[ R = \{(s, s', R, R^{\text{clus}}) \mid \exists CT, CT', I_d, I_p, D, f_1, v_1, v'_1, l_B, l_D, l_C, l_C', m, f_3, v_3, v'_3 : \]
\[ s = [l_D = \text{obj } D \{ f_1 = v_1 \}] \]
\[ [l_B = \text{obj } B \{ f_2 = v_2 \}] \]
\[ [l_C = \text{obj } C \{ c = l_D, f_3 = v_3 \}] \]
\[ s' = [l_D = \text{obj } D \{ f_1 = v'_1 \}] \]
\[ [l_B = \text{obj } B \{ f_2 = v_2 \}] \]
\[ [l_C = \text{obj } C \{ c = l_D, f_3 = v_3 \}] \]
\[ R^{\text{clus}} = \{ \langle \{ \langle C, C' \rangle \} \} \]
\[ R' = \{ \langle l_D, I_p, l_B, I_p, l_C, I_C' \} \}
\[ \{v_1, v'_1, v_2, v'_2, v_3, v'_3\} \in V_{\langle CT, CT' \rangle, s, s', R'} \]
\] 

Again we constructed this set by inspecting the conditions of adequacy for \( J_2 \). The resulting set differs from the one in Section 4 at the bold-faced parts. In this set we need to take into account the subclasses of \( A \) and \( \text{Cell1} \), which may have extra fields. Furthermore, when we prove condition 6 for \( \sigma \) and \( \sigma' \) we must consider that the rest of the methods may have been overridden. In the case of \( \text{Cell1} \) this does not affect the equivalence.

Even with considering arbitrary subclasses of \( \text{Cell1} \), proving the adequacy conditions for \( J_2 \) is not substantially harder than proving them for \( \sigma \).

8. \( J_3 \): Adding Downcasing to \( J_2 \)

Our last extension adds a cast expression to \( J_2 \), creating language \( J_3 \). The casting operator allows both explicit upcasing and downcasing. The latter is a non-type-safe operation which introduces one more error to the language, \( \text{cerr} \). The extra syntax, typing rules and operational semantics for casting are shown in Figure 6.

9. Adequacy in \( J_3 \)

The addition of a casting operation has minimal effect on the technical machinery we have set up so far. The first thing we need to do is to add the new error to the answers of type \( t \).

Definition 9.1. If \( R^t \) is a relation on object references, \( CT, CT' \) class tables, and \( s, s' \) stores, then we define the following relation on answers of type \( t \):

\[ A_t \{ CT, CT', s, s', R^t \} \triangleq \{ V_{\langle CT, CT' \rangle, s, s', R^t} \cup \{ \langle \text{cerr}, \text{cerr} \rangle, \langle \text{cerr}, \text{errer} \rangle \} \}
\]

The rest of the definitions remain the same throughout our technique.

The second and last thing we need to do is to consider the new cast expression is in the proof construction scheme of Task 3.10 for \( J_3 \). We need to consider one more case for \( e_0 \); namely the case when \( e_0 = \langle C \rangle e_1 \). On the left-hand side this expression will reduce to an answer, which means that also \( e_1 \) will reduce to some answer in less steps. Thus, by applying the induction hypothesis at a smaller evaluation sequence, we conclude that \( e_1 \) will terminate on the right-hand side as well, and the final answers will be related. The final step on both sides will either result to related values (successful casting), or will produce the same error (because related objects have the same class name, and the class hierarchies have the same structure on both sides).

By the above, we conclude that the addition of the cast expression in the language doesn’t change Theorem 6.1 which holds also for \( J_3 \). The conditions of that theorem distinguish inequivalent implementations of classes that use the casting operation. Furthermore this means that casting is a conservative extension to the language.

10. An Example With Callbacks

We consider the adaptation to \( J_3 \) of an example from Meyer and Sieber [18]. We have two implementations of a \( \text{Counter} \) class, one containing a private counter \( g \), initialized to zero and then increased by two only by the method \( \text{inc} \). There is also a method \( \text{callP} \), which takes as an argument an instance of some class \( A \), provided by the context. \( A \) is assumed to have a method \( P \) which can accept as arguments instances of the class \( \text{Counter} \). By proving this equivalence we show that \( g \) can be accessed only through the method \( \text{inc} \), and thus it will always contain an even number. To do that we will need to reason about the behavior of calls to \( P \), the implementation of which is unknown to us. We manage this by using the induction hypothesis.

\[ C = \text{class Counter extends Object} \{
\begin{align*}
& \text{private int } g; \\
& \text{public int callP(A o) \{ o.P(this); if even(this.g) then 0 else 1 \} public void inc() \{ this.g := this.g + 2 \}}
\end{align*}
\]

\[ C' = \text{class Counter extends Object} \{
\begin{align*}
& \text{C() \{} \text{public int callP(A o) \{ o.P(this); 0 \} public void inc() \{ unit} \}
\end{align*}
\]

We construct the following set:

\[ R = \{(s, s', R', R^{\text{clus}}) \mid \exists CT, CT', I_d, I_p, D, f_1, v_1, v'_1, l_B, l_D, l_C, l_C', m, f_3, v_3, v'_3 : \]
\[ s = [l_D = \text{obj } D \{ f_1 = v_1 \}] \]
\[ [l_B = \text{obj } B \{ f_3 = v_3 \}] \]
\[ s' = [l_D = \text{obj } D \{ f_1 = v'_1 \}] \]
\[ [l_B = \text{obj } B \{ f_3 = v'_3 \}] \]
\[ R^{\text{clus}} = \{ \langle \{ \langle C, C' \rangle \} \} \]
\[ R' = \{ \langle l_D, I_p, l_B, I_p, l_C, I_C' \} \}
\[ \{v_1, v'_1, v_2, v'_2, v_3, v'_3\} \in V_{\langle CT, CT' \rangle, s, s', R'} \]
\[ (CT, CT') \in CT[R^{\text{clus}}] \]
\[ CT \vdash B <t A \]
\[ CT \vdash C <t \text{Counter} \]

The tuples of \( R \) consist of related stores that contain related objects of the classes \( B, C, D \). \( B \) represents any possible subclass of the class \( A \) and \( C \) any possible subclass of class \( \text{Counter} \). \( D \) are all the other classes that may be defined by the context.

We need to show that \( R \) satisfies Theorem 6.1. The interesting case is to show condition 6 for method \( \text{callP} \).

Consider arbitrary \( (s, s', R', R^{\text{clus}}) \in R \) and \( (CT, CT') \in CT[R^{\text{clus}}] \). For any \( \langle l_C, l_C' \rangle \in V_{\text{Subtree}}[CT, CT', s, s', R'] \) with \( s.l_C = \text{obj } \text{Counter} \{ g = 2n, \ldots \} \), \( s'.l_C' = \text{obj } \text{Counter} \{ \ldots \} \) and any \( \langle l_B, l_B' \rangle \in V_{\langle CT, CT' \rangle, s, s', R'} \), we need to show for any \( k \):

\[ IH_{\text{callP}}^{k}(k) \Rightarrow R', R^{\text{clus}}, R \vdash (CT \vdash s, l_C, \text{callP}(l_B):\text{int}) \sqsubseteq_{k+1} (CT' \vdash s', l_C', \text{callP}(l_B'):\text{int}) \]
and the reverse.
Furthermore, we can apply the induction hypothesis and get that there exist $s_1$ and $c$:

$$CT; \Gamma; s \vdash e : t$$

$$CT; \Gamma; s \vdash e : D$$

$$CT; \Gamma; s \vdash (C): e : C$$

Thus the invariant that the field $s$ and by this we are done.

11. Related Work

Banerjee and Naumann in [3] present a method for reasoning about whole-program equivalence in a subset of Java, similar to $\mathcal{F}_3$. Their technique is based on a denotational model in which they build a simulation relation of denotations. They use a notion of confinement to restrict certain pointers to the heap, and show that if two class tables are confined and there is a simulation between their denotations, then these class tables are equivalent. It is not clear if this technique is complete. This is because the authors don’t show that there is a simulation relation for any two equivalent and confined class tables, but also because the technique seems helpful to reason only about confined class tables. Furthermore it is not obvious how this technique can be extended to contextual equivalence of classes, a stronger property than whole program equivalence. Our method gives a sound and complete proof technique for exactly this stronger property. Using our technique we were able to show contextual equivalence for all of their examples.

Another technique for reasoning about contextual equivalence in a class-based language is the one from Jeffrey and Rathke in [12], which follows their work on concurrent objects [13]. They study a Java-like language for which they define a semantic trace equivalence and show that it is sound and complete with respect to testing equivalence. This is an elegant technique which can be used to prove equivalences like the adaptation of our examples to their language. Doing these proofs with their method would require to show that two modules have the same traces, which we believe would result in introducing bisimulations similar to ours and doing a full-blown inductive proof as the one shown in Task 3.10. The difference with our work is that they study a significantly different language. The most important feature of that language, that ours does not have, is a package system that restricts the interaction of classes with their context. Classes are not visible through the package barriers. Therefore they are not extensible by classes in other packages and the state of each class is guaranteed to be private. Only interfaces and instances of classes that implement these interfaces are shared between packages. With these restrictions the interaction of classes with the context becomes an interaction of messages. When packages have the same interface and they communicate through the same messages with the context, then these packages are equivalent. This technique is not applicable to $\mathcal{F}_3$ where the interaction of classes with their context is more complex, and furthermore it doesn’t study the effect of inheritance on class equivalence.

In the same spirit as Jeffrey and Rathke, Abrahám et al. [1] give a fully abstract trace semantics, with respect to may testing, for a concurrent class-based language. Their work focuses on classes that don’t support inheritance, much like our $\mathcal{F}_1$ language, and all fields and methods are public.

12. Conclusions and Future Work

We have presented a sound and complete method for reasoning about class equivalence in a subset of Java. This is the first such method that deals with inheritance, as well as public and private interfaces of classes and imperative fields. We were able to use this method to prove equivalences in a number of interesting examples. We have also studied the effect of inheritance on the equivalence checking of classes with our technique. Moreover we have shown that adding a cast operator in a language with inheritance is a conservative extension.

We have seen that our method can deal with the null-pointer and cast exceptions of Java. We would like to investigate further in this direction and see if our technique can prove contextual equivalence in a language with more advanced control effects (e.g. [7]).
In the future we would also like to see whether our method can benefit from ideas in closely related areas, like Separation Logic [22].

References


A. Appendix

<table>
<thead>
<tr>
<th>C.classname</th>
<th>Returns the class name defined by the class definition C.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CT.C.super</td>
<td>Returns the name of the immediate superclass of C.</td>
</tr>
<tr>
<td>CT.C.constr</td>
<td>Returns the constructor definition of class C.</td>
</tr>
<tr>
<td>CT.C.constr.type</td>
<td>Returns the constructor type of class C.</td>
</tr>
<tr>
<td>CT.C.fields</td>
<td>Returns a sequence of all of the field definitions (public and private) of class C and all its superclasses.</td>
</tr>
<tr>
<td>CT.C.methods</td>
<td>Returns a sequence of all the method definitions (public and private) that can be invoked on an instance of class C.</td>
</tr>
<tr>
<td>CT.C.m.defclass</td>
<td>Traverses up the class hierarchy, starting from class C and returns the first name of the class in which method m is defined.</td>
</tr>
</tbody>
</table>

Figure 7. Meta-Functions