# 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 presumably 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, protected, and private 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].

Furthermore we use our technique as a tool to understand the exact effect of inheritance on contextual equivalence. We do that by deriving conditions of equivalence for a language without inheritance and compare them to those we get after we extend the language with it. In a similar way we show that adding a cast operator is a conservative extension of the language.

***Categories and Subject Descriptors*** F.3.2 [*Logic and Meanings of Programs*]: Semantics of Programming Languages—operational semantics; D.3.3 [*Programming Languages*]: Language Constructs and Features—procedures, functions and subroutines; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—semantics

***General Terms*** theory, languages

***Keywords*** contextual equivalence, bisimulations, lambda-calculus, higher-order procedures, imperative languages

## 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 reimplementation. 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 to 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* $\mathcal{CT}[\,]$, expressions $e$, and the empty store $\emptyset$, the program configurations $(\mathcal{CT}[C], \emptyset, e)$ and $(\mathcal{CT}[C'], \emptyset, e)$ 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 in 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

```
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;
  private Object 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

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. The exact interaction with contextual equivalence has not been studied before. Inheritance increases the distinguishing power of the context. By extending a class, the context can have access to the protected interface of the class, and more importantly it can override methods of the class.

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 of 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 of adequacy. By comparing the two sets of conditions we were able to give an account as to how inheritance affects class equivalence. By extending the language once more with a cast operator and showing that the conditions of adequacy remain unchanged, we prove that casting is a conservative extension to 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 that our method of deriving conditions of adequacy is incremental when we add new syntactic forms in the language (e.g. the cast expression), but not when we add more complex features like inheritance. In all three languages, though, 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 downcasting on 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 the equivalence of classes. 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.
- Finally we show in a similar way that adding a casting operator is a conservative extension to a class-based 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 of 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 of 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. $\mathcal{J}_1$: A Basic Class-Based Language

We start by defining a small class-based language which we call $\mathcal{J}_1$. 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 $\mathcal{J}_1$ is shown in Figure 2.

The main difference between $\mathcal{J}_1$ and other imperative Java calculi, like Middleweight Java [4] and Classic Java [9], is that $\mathcal{J}_1$ does

|  |  |  |
|---|---|---|
| PROGRAM CONFIGURATIONS: | $pconf \in \mathcal{CT} \times \textsc{Stores} \times \textsc{Expressions}$ |  |
| CLASS TABLES: | $\mathcal{CT} \in \mathcal{P}(\textsc{Class Definitions})$ |  |
| CLASS DEFINITIONS: | $\mathcal{C} ::= \texttt{class}\, C\{\overline{mod\, t\, f}; \mathcal{K}\overline{\mathcal{M}}\}$ |  |
| CONSTRUCTOR DEFINITIONS: | $\mathcal{K} ::= C()\{\overline{\texttt{this}.f := c}\}$ |  |
| METHOD DEFINITIONS: | $\mathcal{M} ::= mod\, t\, m(\overline{t\, x})\{e\}$ |  |
| TYPES: | $t ::= \texttt{void} \mid \texttt{int} \mid \texttt{bool} \mid C$ |  |
| MODIFIERS: | $mod ::= \texttt{public} \mid \texttt{private}$ |  |
| EXPRESSIONS: | $e, d ::= v \mid x$ | Values, Identifiers |
|  | $\mid\ \texttt{new}\, C \mid e.m(\overline{e})$ | Object Instantiation, Method Invocation |
|  | $\mid\ e.f \mid e.f := e \mid \texttt{op}(\overline{e}) \mid \texttt{eq}(l, l)$ | Field Lookup and Update, Operators |
|  | $\mid\ \texttt{let}\, x = e\, \texttt{in}\, e \mid \texttt{if}\, e\, \texttt{then}\, e\, \texttt{else}\, e$ | Let Expression, Conditional |
| VALUES: | $v, u, w ::= c \mid l$ |  |
| CONSTANTS: | $c ::= \texttt{unit} \mid \texttt{null} \mid \texttt{true} \mid \texttt{false} \mid 0 \mid \pm 1 \mid \pm 2 \mid \ldots$ | Unit, Null, Booleans, Integers |
| LOCATIONS: | $l, k$ |  |
| STORES: | $s \in \textsc{Locations} \rightharpoonup \textsc{Stored Objects}$ |  |
| STORED OBJECTS: | $o ::= obj\, C\, \{\overline{f = v}\}$ |  |
| ERRORS: | $\varepsilon ::= \texttt{nerr}$ | Null Error |

**Figure 2.** Syntax of $\mathcal{J}_1$

not have inheritance and casting. We later extend $\mathcal{J}_1$ with inheritance to create the language $\mathcal{J}_2$, and then we add a casting operator to create $\mathcal{J}_3$. $\mathcal{J}_3$ has the same constructs as Middleweight Java, with the addition of constants and access modifiers. $\mathcal{J}_3$ 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 which is assumed to be implemented as one of the operators of the language.

The types of $\mathcal{J}_1$ are class types, as well as the ground types `void`, `int`, and `bool`. There is a `unit` constant of type `void`, `true` and `false` of type `bool`, and the integers of type `int`. The constant `null` has any class type.

$\mathcal{J}_1$ has also a null error (`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.

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 *wfClassHierarchy*$(\mathcal{CT})$. When we add inheritance to the language this predicate will check for a valid class hierarchy.

We will use meta-operations and the dot notation to perform static lookup on class tables and classes. For example $\mathcal{CT}.C$ returns the definition of the class named $C$ from the class table $\mathcal{CT}$, and $\mathcal{CT}.C.\mathsf{fields}$ returns a sequence of all the field definitions in $C$. A complete table of these meta-operations and a description of their functionality is shown in Figure 7 of the Appendix.

Stores are partial maps from locations to stored objects. A *program configuration* is a triple composed of a class table, a store, and a closed expression (written $\mathcal{CT} \vdash s, e$). An *initial configuration* is a program configuration that contains the empty store ($\emptyset$).

The typing rules of $\mathcal{J}_1$ are shown in Figure 3. The typing judgments for expressions have the form $\mathcal{CT}; \Gamma; s \vdash e{:}t$. $\Gamma$ is the type environment. The store $s$ is used to type-check locations; the value stored in a location has the type $s.l.\texttt{type}$. In $\mathcal{J}_1$ this is the class type mentioned in the object itself, but, when we will 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 `null` has any class type defined in the class table. The rest of the typing judgments for expressions are the expected ones for a language like $\mathcal{J}_1$.

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

In Figure 4 we give a small-step semantics for $\mathcal{J}_1$. A small-step $\mathcal{CT} \vdash s, e \to s_1, e_1$ describes a transition from the program configuration $\mathcal{CT} \vdash s, e$ to the configuration $\mathcal{CT} \vdash s_1, e_1$. We also define $\to^*$ to be the reflexive and transitive closure of $\to$, and $\to^{<k}$ the reflexive and up to $k-1$ steps transitive closure of $\to$. We also write $\mathcal{CT} \vdash s, e{\downarrow}$ iff there exists $s_1, w$, such that $\mathcal{CT} \vdash s, e \to^* s_1, w$.

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 $obj\, C\, \{\overline{f = l}\}$, instead of $obj\, C\, \{f_1 = l_1, f_2 = l_2, \ldots, f_n = l_n\}$. We also use the notation $\overline{f = v} \backslash f_i = u$ to denote the sequence $f_1 = v_1, \cdots, f_{i-1} = v_{i-1}, f_i = u, f_{i+1} = v_{i+1}, \cdots$.

## 3. Equivalence and Adequacy in $\mathcal{J}_1$

We want to study contextual equivalence of class definitions in $\mathcal{J}_1$ 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,* $\mathcal{CT}[\,]$*, is a set of class definitions. Placing a class definition, or a sequence of class definitions,*

$$\boxed{\mathcal{CT};\Gamma;s \vdash e{:}t}$$

$$\frac{x{:}t \in \Gamma}{\mathcal{CT};\Gamma;s \vdash x{:}t} \qquad \frac{}{\mathcal{CT};\Gamma;s \vdash \texttt{unit}{:}\texttt{void}} \qquad \frac{c \in \{\texttt{true},\texttt{false}\}}{\mathcal{CT};\Gamma;s \vdash c{:}\texttt{bool}} \qquad \frac{c \in \{\pm 1, \pm 2, \cdots\}}{\mathcal{CT};\Gamma;s \vdash c{:}\texttt{int}} \qquad \frac{\texttt{class}\, C\{\cdots\} \in \mathcal{CT}}{\mathcal{CT};\Gamma;s \vdash \texttt{null}{:}C}$$

$$\frac{\begin{array}{c} C \in \mathcal{CT}.\mathsf{classnames} \\ s.l.\mathsf{type} = C \end{array}}{\mathcal{CT};\Gamma;s \vdash l{:}C} \qquad \frac{C \in \mathcal{CT}.\mathsf{classnames}}{\mathcal{CT};\Gamma;s \vdash \texttt{new}\, C{:}C} \qquad \frac{\begin{array}{c} \mathcal{CT};\Gamma;s \vdash \overline{e{:}t_0} \\ \mathsf{op}.\mathit{type} = \overline{t_0} \to t \end{array}}{\mathcal{CT};\Gamma;s \vdash \mathsf{op}(\overline{e}){:}t}$$

$$\frac{\begin{array}{c} \mathcal{CT};\Gamma;s \vdash e{:}C \\ \mathcal{CT};\Gamma;s \vdash \texttt{this}{:}C_0 \\ f \in \mathit{accessible}(\mathcal{CT},C,C_0) \\ \mathit{mod}\, t\, f \in \mathcal{CT}.C.\mathsf{fields} \end{array}}{\mathcal{CT};\Gamma;s \vdash e.f{:}t} \qquad \frac{\begin{array}{c} \mathcal{CT};\Gamma;s \vdash e{:}C \\ \mathcal{CT};\Gamma;s \vdash \texttt{this}{:}C_0 \\ f \in \mathit{accessible}(\mathcal{CT},C,C_0) \\ \mathit{mod}\, t\, f \in \mathcal{CT}.C.\mathsf{fields} \\ \mathcal{CT};\Gamma;s \vdash e_1{:}t \end{array}}{\mathcal{CT};\Gamma;s \vdash e.f := e_1{:}\texttt{void}} \qquad \frac{\begin{array}{c} \mathcal{CT};\Gamma;s \vdash e{:}C \\ \mathcal{CT};\Gamma;s \vdash \texttt{this}{:}C_0 \\ m \in \mathit{accessible}(\mathcal{CT},C,C_0) \\ \mathit{mod}\, \overline{t_0} \to t\, m \in \mathcal{CT}.C.\mathsf{methods} \\ \mathcal{CT};\Gamma;s \vdash \overline{e_0{:}t_0} \end{array}}{\mathcal{CT};\Gamma;s \vdash e.m(\overline{e_0}){:}t}$$

$$\frac{\begin{array}{c} \mathcal{CT};\Gamma;s \vdash e_1{:}t_1 \\ \mathcal{CT};\Gamma,x{:}t_1;s \vdash e{:}t \end{array}}{\mathcal{CT};\Gamma;s \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e{:}t} \qquad \frac{\begin{array}{c} \mathcal{CT};\Gamma;s \vdash e_1{:}\texttt{bool} \\ \mathcal{CT};\Gamma;s \vdash e_2{:}t \\ \mathcal{CT};\Gamma;s \vdash e_3{:}t \end{array}}{\mathcal{CT};\Gamma;s \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3{:}t}$$

$$\boxed{\mathcal{CT} \vdash \mathcal{M}{:}\mathsf{OK\ in}\ C} \qquad \boxed{\mathcal{CT} \vdash \mathcal{C}{:}\mathsf{OK}} \qquad \boxed{\mathcal{CT}{:}\mathsf{OK}} \qquad \boxed{(\mathcal{CT} \vdash s, e){:}\mathsf{OK}}$$

$$\frac{\mathcal{CT};\overline{x{:}t_1},\texttt{this}{:}C;\emptyset \vdash e{:}t}{\mathcal{CT} \vdash \mathit{mod}\; t\, m(\overline{t_1\, x})\{e\}{:}\mathsf{OK\ in}\ C} \qquad \frac{\begin{array}{c} \mathcal{K} = C()\{\overline{\texttt{this}.f := c}\} \\ \mathcal{CT};\emptyset;\emptyset \vdash \overline{c{:}t} \\ \mathcal{CT} \vdash \overline{\mathcal{M}}{:}\mathsf{OK\ in}\ C \end{array}}{\mathcal{CT} \vdash \texttt{class}\, C\{\overline{t\, f};\mathcal{K}\overline{\mathcal{M}}\}{:}\mathsf{OK}} \qquad \frac{\begin{array}{c} \{\overline{\mathcal{C}}\} \vdash \overline{\mathcal{C}}{:}\mathsf{OK} \\ \mathit{wfClassHierarchy}(\{\overline{\mathcal{C}}\}) \end{array}}{\{\overline{\mathcal{C}}\}{:}\mathsf{OK}} \qquad \frac{\begin{array}{c} \mathcal{CT}{:}\mathsf{OK} \\ \mathcal{CT};\emptyset;s \vdash e{:}t \end{array}}{(\mathcal{CT} \vdash s, e){:}\mathsf{OK}}$$

**Figure 3.** Typing of $\mathcal{J}_1$

*in the hole of a class table context corresponds to set union:*

$$\mathcal{CT}[\overline{\mathcal{C}}] \stackrel{\mathit{def}}{=} \mathcal{CT} \cup \{\overline{\mathcal{C}}\}$$

**Definition 3.2.** *$R^{\mathsf{cls}}$ is a relation on classes iff it is a set of pairs of class definitions, such that for all $(\mathcal{C}, \mathcal{C}') \in R^{\mathsf{cls}}$, and all class-table contexts $\mathcal{CT}[\,]$:*

$$\mathcal{CT}[\mathcal{C}]{:}\mathsf{OK} \iff \mathcal{CT}[\mathcal{C}']{:}\mathsf{OK}$$

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, extend the same superclass, and have the same public interface.

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

$$CT[R^{\mathsf{cls}}] \stackrel{\mathit{def}}{=} \{(\mathcal{CT}[\overline{\mathcal{C}}], \mathcal{CT}[\overline{\mathcal{C}'}]) \mid \overline{(\mathcal{C}, \mathcal{C}')} \in R^{\mathsf{cls}}, \\ \mathcal{CT}[\overline{\mathcal{C}}]{:}\mathsf{OK}\}$$

We give the following definition of contextual equivalence for $\mathcal{J}_1$:

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

$$\mathcal{CT} \vdash \emptyset, e\!\downarrow \iff \mathcal{CT}' \vdash \emptyset, e\!\downarrow$$

This definition is hard to use directly. 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 an abstract inductive proof to discover a set of necessary conditions for adequacy. Here we show how to apply this method to $\mathcal{J}_1$.

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

**Definition 3.5.** *A $\mathcal{J}$-relation $\mathbb{R}$ is a set of tuples $(s, s', R^{\ell}, R^{\mathsf{cls}})$, where $s$, $s'$ are stores, $R^{\ell}$ is a relation on object references, and $R^{\mathsf{cls}}$ is a relation on classes.*

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

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

- *related values of type $t$:*

$$V_t[\mathcal{CT}, \mathcal{CT}', s, s', R^{\ell}] \stackrel{\mathit{def}}{=} \{(l, l') \mid (l, l') \in R^{\ell}, \\ \mathcal{CT};\emptyset;s \vdash l{:}t, \\ \mathcal{CT}';\emptyset;s' \vdash l'{:}t\} \\ \cup \{(c, c) \mid \mathcal{CT};\emptyset;\emptyset \vdash c{:}t, \\ \mathcal{CT}';\emptyset;\emptyset \vdash c{:}t\}$$

- *related answers of type $t$:*

$$A_t[\mathcal{CT}, \mathcal{CT}', s, s', R^{\ell}] \stackrel{\mathit{def}}{=} V_t[\mathcal{CT}, \mathcal{CT}', s, s', R^{\ell}] \\ \cup \{(\texttt{nerr}, \texttt{nerr})\}$$

$$\boxed{\mathcal{CT} \vdash s, e \rightarrow s_1, e_1}$$

$$\frac{\mathcal{CT}.C.\mathsf{constr} = C()\{\overline{\mathtt{this}.f := c}\} \qquad l \notin \mathit{Dom}(s)}{\mathcal{CT} \vdash s, \mathtt{new}\, C \rightarrow s[l = \mathit{obj}\, C\,\{\overline{f = c}\}], l}$$

$$\frac{s.l = \mathit{obj}\, C\,\{\overline{f = v}\}}{\mathcal{CT} \vdash s, l.f_i \rightarrow s, v_i} \qquad \frac{}{\mathcal{CT} \vdash s, \mathtt{null}.f_i \rightarrow s, \mathtt{nerr}}$$

$$\frac{\mathcal{CT}.C.m = \mathit{mod}\; t\, m(\overline{t_x\, x})\{e\}}{\mathcal{CT} \vdash s, l.m(\overline{v}) \rightarrow s, [\overline{v/x}, l/\mathtt{this}]e} \qquad \frac{}{\mathcal{CT} \vdash s, \mathtt{null}.m(\overline{v}) \rightarrow s, \mathtt{nerr}}$$

$$\frac{s.l = \mathit{obj}\, C\,\{\overline{f = v}\}}{\mathcal{CT} \vdash s, l.f_i := u \rightarrow s[l \leftarrow \mathit{obj}\, C\,\{\overline{f = v} \mid f_i = u\}], \mathtt{unit}} \qquad \frac{}{\mathcal{CT} \vdash s, \mathtt{null}.f_i := v \rightarrow s, \mathtt{nerr}}$$

$$\frac{}{\mathcal{CT} \vdash s, \mathtt{let}\, x = v\, \mathtt{in}\, e \rightarrow s, [v/x]e} \qquad \frac{}{\mathcal{CT} \vdash s, \mathtt{if}\,\mathtt{true}\,\mathtt{then}\, e_1\, \mathtt{else}\, e_2 \rightarrow s, e_1} \qquad \frac{}{\mathcal{CT} \vdash s, \mathtt{if}\,\mathtt{false}\,\mathtt{then}\, e_1\, \mathtt{else}\, e_2 \rightarrow s, e_2}$$

$$\boxed{\textit{Evaluation Contexts}}$$

$$\begin{aligned}
&E ::= \mathcal{E} \mid E[E] \\
&\mathcal{E} ::= [\,] \mid [\,].f \mid [\,].m(\overline{e}) \mid v.m(\overline{v}, [\,], \overline{e}) \mid [\,].f := e \mid v.f := [\,] \mid \mathtt{let}\, x = [\,]\; \mathtt{in}\, e \mid \mathtt{if}\, [\,]\; \mathtt{then}\, e\, \mathtt{else}\, e \mid \mathtt{op}(\overline{v}, [\,], \overline{e})
\end{aligned}$$

$$\boxed{\mathcal{CT} \vdash s, E[e] \rightarrow s_1, E[e_1]}$$

$$\frac{\mathcal{CT} \vdash s, e \rightarrow s_1, e_1}{\mathcal{CT} \vdash s, E[e] \rightarrow s_1, E[e_1]} \qquad \frac{}{\mathcal{CT} \vdash s, \mathcal{E}[\varepsilon] \rightarrow s_1, \varepsilon}$$

**Figure 4.** Small-step Operational Semantics of $\mathcal{J}_1$

- *and related expressions of type $t$:*

$$\begin{aligned}
&E_t[\mathcal{CT}, \mathcal{CT}', s, s', R^\ell] \overset{def}{=} \\
&\quad \{([\overline{l/x}]e, [\overline{l'/x}]e) \mid FV(e) \subseteq \{\overline{x}\}, \\
&\qquad \overline{(l, l')} \in R^\ell, \\
&\qquad Locs(e) = \emptyset, \\
&\qquad \mathcal{CT}; \emptyset; s \vdash [\overline{v/x}]e{:}t, \\
&\qquad \mathcal{CT}'; \emptyset; s' \vdash [\overline{v'/x}]e{:}t\}
\end{aligned}$$

By requiring that $Locs(e) = \emptyset$ in the above definition we force all $R$-related expressions to use only $R$-related references.

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

**Definition 3.7 (Adequacy).** *$\mathbb{R}$ is adequate if and only if:*

$$\begin{aligned}
&\forall (s, s', R^\ell, R^{\mathsf{cls}}) \in \mathbb{R}. \\
&\quad \forall (\mathcal{CT}, \mathcal{CT}') \in CT[R^{\mathsf{cls}}]. \\
&\quad\quad \forall\, t, \forall (e, e') \in E_t[\mathcal{CT}, \mathcal{CT}', s, s', R^\ell]. \\
&\quad\quad\quad \forall\, s_1, w. \\
&\quad\quad\quad\quad (\mathcal{CT} \vdash s, e \rightarrow^* s_1, w) \\
&\quad\quad\quad\quad \implies \exists\, s_1', w', R_1^\ell. \\
&\quad\quad\quad\quad\quad\quad (\mathcal{CT}' \vdash s', e' \rightarrow^* s_1', w') \\
&\quad\quad\quad\quad\quad \wedge ((w, w') \in A_t[\mathcal{CT}, \mathcal{CT}', s_1, s_1', R_1^\ell]) \\
&\quad\quad\quad\quad\quad \wedge ((s_1, s_1', R_1^\ell, R^{\mathsf{cls}}) \in \mathbb{R}) \\
&\quad\quad\quad\quad\quad \wedge (R^\ell \subseteq R_1^\ell)
\end{aligned}$$

*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 this 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 to the existence of an induction.

Adequate relations are sound and complete in the following way.

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

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

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

One could show the equivalence between two class implementations by constructing an appropriate set $\mathbb{R}$ and then prove its adequacy by an induction based on Definition 3.7. Our goal is to improve this by finding sufficient conditions on $\mathbb{R}$ that would make the proof of adequacy go through. These conditions act as a Verification Condition Generator: one provides an invariant (the set $\mathbb{R}$) that presumably proves a particular equivalence, and the conditions say what needs to be proven in order to check the validity of the invariant. The conditions that we derive also give us a better grasp on how the context can distinguish two class definitions.

To do this we investigate a class of inductive proofs based on Definition 3.7 by abstracting over the concrete structure of $\mathbb{R}$ and attempting to prove adequacy. This abstract proof reveals the subcases of the induction that don't go through just by using the induction hypothesis, but also requires that the states of $\mathbb{R}$ satisfy some extra properties. These properties become the *proof obligations*, or verification conditions, for $\mathbb{R}$.

**Conjecture 3.10 (Abstract Proof of Adequacy).** *For some relation $\mathbb{R}$, $\mathbb{R}$ is adequate.*

*Proof.* The proof consists 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:

$$\begin{array}{l}
IH(k) = \\
\quad \forall (s, s', R^\ell, R^{\texttt{cls}}) \in \mathbb{R}. \\
\quad\quad \forall (\mathcal{CT}, \mathcal{CT}') \in CT[R^{\texttt{cls}}]. \\
\quad\quad\quad \forall\, t, \forall (e, e') \in E_t[\mathcal{CT}, \mathcal{CT}', s, s', R^\ell]. \\
\quad\quad\quad\quad \forall\, s_1, w. \\
\quad\quad\quad\quad\quad (\mathcal{CT} \vdash s, e \rightarrow^{<k} s_1, w) \\
\quad\quad\quad\quad\quad \implies \exists\, s_1', w', R_1^\ell. \\
\quad\quad\quad\quad\quad\quad (\mathcal{CT}' \vdash s', e' \rightarrow^{*} s_1', w') \\
\quad\quad\quad\quad\quad\quad \wedge ((w, w') \in A_t[\mathcal{CT}, \mathcal{CT}', s_1, s_1', R_1^\ell]) \\
\quad\quad\quad\quad\quad\quad \wedge ((s_1, s_1', R_1^\ell, R^{\texttt{cls}}) \in \mathbb{R}) \\
\quad\quad\quad\quad\quad\quad \wedge (R^\ell \subseteq R_1^\ell)
\end{array}$$

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 $(e, e') = ([\overline{v/x}]e_0, [\overline{v'/x}]e_0)$, for some $e_0, \overline{v, v'}$, such that $FV(e_0) \subseteq \{\overline{x}\}$, $Locs(e_0) = \emptyset$, $\overline{(v, v')} \in R^\ell$, $\mathcal{CT}; \emptyset; s \vdash [\overline{v/x}]e_0{:}t$, $\mathcal{CT}'; \emptyset; s' \vdash [\overline{v'/x}]e_0{:}t$. We proceed by cases on $e_0$. 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 conditions for $\mathbb{R}$ that we found by the above abstract proof. First we give a notation to write down the inductive cases and the induction hypotheses (one for each direction).

**Definition 3.11 (Inductive Cases).**

$$\begin{array}{l}
R^\ell, R^{\texttt{cls}}, \mathbb{R} \vdash (\mathcal{CT} \vdash s, e{:}t) \sqsubseteq^{<k} (\mathcal{CT}' \vdash s', e'{:}t) \overset{def}{=} \\
\quad \forall\, s_1, w. \\
\quad\quad\quad ((e, e') \in E_t[\mathcal{CT}, \mathcal{CT}', s, s', R^\ell]) \\
\quad\quad \wedge (\mathcal{CT} \vdash s, e \rightarrow^{<k} s_1, w) \\
\quad\quad\quad \implies \exists\, s_1', w', R_1^\ell. \\
\quad\quad\quad\quad (\mathcal{CT}' \vdash s', e' \rightarrow^{*} s_1', w') \\
\quad\quad\quad\quad \wedge ((w, w') \in A_t[\mathcal{CT}, \mathcal{CT}', s_1, s_1', R_1^\ell]) \\
\quad\quad\quad\quad \wedge ((s_1, s_1', R_1^\ell, R^{\texttt{cls}}) \in \mathbb{R}) \\
\quad\quad\quad\quad \wedge (R^\ell \subseteq R_1^\ell)
\end{array}$$

*and* $R^\ell, R^{\texttt{cls}}, \mathbb{R} \vdash (\mathcal{CT} \vdash s, e{:}t) \sqsupseteq^{<k} (\mathcal{CT}' \vdash s', e'{:}t)$ *for the reverse.*

**Definition 3.12 (Inductive Hypotheses).**

$$\begin{array}{l}
IH_{\mathbb{R}}^{L}(k) \overset{def}{=} \\
\quad \forall (s, s', R^\ell, R^{\texttt{cls}}) \in \mathbb{R}. \\
\quad\quad \forall (\mathcal{CT}, \mathcal{CT}') \in CT[R^{\texttt{cls}}]. \\
\quad\quad\quad \forall\, t, (e, e') \in E_t[\mathcal{CT}, \mathcal{CT}', s, s', R^\ell]. \\
\quad\quad\quad\quad R^\ell, R^{\texttt{cls}}, \mathbb{R} \vdash (\mathcal{CT} \vdash s, e{:}t) \sqsubseteq^{<k} \\
\quad\quad\quad\quad\quad\quad\quad\quad (\mathcal{CT}' \vdash s', e'{:}t)
\end{array}$$

*and* $IH_{\mathbb{R}}^{R}(k)$ *for the reverse.*

Our main theorem is the following.

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

1. (Same interfaces) *For all* $(\mathcal{C}, \mathcal{C}') \in R^{\texttt{cls}}$,

$$\begin{array}{rl}
\mathcal{C} = \texttt{class}\, C\{ & \overline{\texttt{public}\, t_1\, f_1;}\, \overline{\texttt{private}\, t_2\, f_2;} \\
& C()\{\overline{\texttt{this}.f_1 := c_1}, \overline{\texttt{this}.f_2 := c_2}\}, \\
& \overline{\texttt{public}\, t_3\, m_3(\overline{t_4\, x_4})\{\cdots\},} \\
& \overline{\texttt{private}\, t_5\, m_5(\overline{t_6\, x_6})\{\ldots\}}\}
\end{array}$$

$$\begin{array}{rl}
\mathcal{C}' = \texttt{class}\, C\{ & \overline{\texttt{public}\, t_1\, f_1;}\, \overline{\texttt{private}\, t_2'\, f_2';} \\
& C()\{\overline{\texttt{this}.f_1 := c_1}, \overline{\texttt{this}.f_2' := c_2'}\}, \\
& \overline{\texttt{public}\, t_3\, m_3(\overline{t_4\, x_4})\{\cdots\},} \\
& \overline{\texttt{private}\, t_5'\, m_5'(\overline{t_6'\, x_6'})\{\ldots\}}\}
\end{array}$$

2. (Related instances) *For all* $(l, l') \in R^\ell$, *there exists* $t$, *such that* $\mathcal{CT}; \emptyset; s \vdash l{:}t$, $\mathcal{CT}'; \emptyset; s' \vdash l'{:}t$.
3. (Enough instances) *For all* $C \in \mathcal{CT}$*.classnames, with*

$$\mathcal{C} = \texttt{class}\, C\{\cdots C()\{\overline{\texttt{this}.f := c}\}, \cdots\}$$

$$\mathcal{C}' = \texttt{class}\, C\{\cdots C()\{\overline{\texttt{this}.f' := c'}\} \cdots\}$$

*and all fresh* $l_1, l_1'$, *there exists* $R_1^\ell \supseteq R^\ell \cup \{(l, l')\}$, *such that*

$$(s[l = obj\, C\, \{\overline{f = c}\}], s'[l' = obj\, C\, \{\overline{f' = c'}\}], R_1^\ell, R^{\texttt{cls}}) \in \mathbb{R}.$$

4. (Related public fields) *For all* $(l, l') \in R^\ell$, *with* $s.l = obj\, C\, \{\overline{f = v}\}$ *and* $s'.l' = obj\, C\, \{\overline{f' = v'}\}$, *and for all* $\texttt{public}\, t_i\, f_i \in \mathcal{C}$*.fields,*

$$(v_i, v_i') \in V_{t_i}[\mathcal{CT}, \mathcal{CT}', s, s', R^\ell].$$

5. (Related updates) *For all* $(l, l') \in R^\ell$, *with* $s.l = obj\, C\, \{\overline{f = v}\}$ *and* $s'.l' = obj\, C\, \{\overline{f' = v'}\}$, *all* $\texttt{public}\, t_i\, f_i \in \mathcal{CT}.C$*.fields, and all* $(u, u') \in V_{t_i}[\mathcal{CT}, \mathcal{CT}', s, s', R^\ell]$,

$$\begin{array}{ll}
(s[l \leftarrow obj\, C\, \{\overline{f = v} \backslash f_i = u\}], & \\
\ s'[l' \leftarrow obj\, C\, \{\overline{f' = v'} \backslash f_i = u'\}], & \\
\ R^\ell, R^{\texttt{cls}}) & \in \mathbb{R}
\end{array}$$

6. (Related public methods) *For all* $(\mathcal{C}, \mathcal{C}') \in R^{\texttt{cls}}$, *all references* $(l, l') \in V_C[\mathcal{CT}, \mathcal{CT}', s, s', R^\ell]$, *all methods* $m$ *with* $\mathcal{CT}.C.m = \texttt{public}\, t_m\, m(\overline{t_x\, x})\{e_3\}$, $\mathcal{CT}'.C.m = \texttt{public}\, t_m\, m(\overline{t_x\, x})\{e_3'\}$, *and for all* $\overline{(v, v')} \in V_{\overline{t_x}}[\mathcal{CT}, \mathcal{CT}', s, s', R^\ell]$,

$$\begin{array}{l}
IH_{\mathbb{R}}^{L}(k) \implies \\
\quad R^\ell, R^{\texttt{cls}}, \mathbb{R} \vdash (\mathcal{CT} \vdash s, l.m(\overline{v}){:}t_m) \sqsubseteq^{<k+1} \\
\quad\quad\quad\quad\quad\quad (\mathcal{CT}' \vdash s', l'.m(\overline{v'}){:}t_m) \\
IH_{\mathbb{R}}^{R}(k) \implies \\
\quad R^\ell, R^{\texttt{cls}}, \mathbb{R} \vdash (\mathcal{CT} \vdash s, l.m(\overline{v}){:}t_m) \sqsupseteq^{<k+1} \\
\quad\quad\quad\quad\quad\quad (\mathcal{CT}' \vdash s', l'.m(\overline{v'}){:}t_m)
\end{array}$$

*Proof.* By recapitulating the proof of Conjecture 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 = class Cell{
        private A c;
        Cell(){this.c := null}
        public void set(A o){this.c := o}
        public A get(){this.c}}

𝒞′ = class Cell{
        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(){
           this.n := this.n + 1;
           if (even(this.n)) then this.c1 else this.c2}}
```

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

$$\begin{aligned}
\mathbb{R} = \{(s, s', R^{\ell}, R^{\mathtt{cls}}) \mid \exists\, & \mathcal{CT}, \mathcal{CT}', \overline{l_D, l'_D, D}, \overline{f_1, v_1, v'_1},\\
& \overline{l_S, l'_S}, \overline{f_2, v_2, v'_2}, \overline{l_C, l'_C, m} :\\
s \ = \ & \overline{[l_D = obj\, D\,\{\overline{f_1 = v_1}\}]}\\
& \overline{[l_S = obj\, \mathtt{A}\,\{\overline{f_2 = v_2}\}]}\\
& \overline{[l_C = obj\, \mathtt{Cell}\,\{c = l_S\}]}\\
s' = \ & \overline{[l'_D = obj\, D\,\{\overline{f_1 = v'_1}\}]}\\
& \overline{[l'_S = obj\, \mathtt{A}\,\{\overline{f_2 = v'_2}\}]}\\
& \overline{[l_C = obj\, \mathtt{Cell}\,\{c1 = l'_S, c2 = l'_S, n = m\}]}\\
R^{\mathtt{cls}} = \ & \{(\mathcal{C}, \mathcal{C}')\}\\
R^{\ell} = \ & \{\overline{(l_D, l'_D)}, \overline{(l_S, l'_S)}, \overline{(l_C, l'_C)}\}\\
& \overline{(v_1, v'_1)}, \overline{(v_2, v'_2)} \in V_t[\mathcal{CT}, \mathcal{CT}', s, s', R^{\ell}]\\
& (\mathcal{CT}, \mathcal{CT}') \in CT[R^{\mathtt{cls}}]\}
\end{aligned}$$

We choose this particular $\mathbb{R}$ by inspecting the conditions of Theorem 3.13. Condition 1 is obviously satisfied by $\mathcal{C}$ and $\mathcal{C}'$. To satisfy conditions 2 and 3 we add in $R^{\ell}$ 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 $V_t[\mathcal{CT}, \mathcal{CT}', s, s', R^{\ell}]$, 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 $(s, s', R^{\ell}, R^{\mathtt{cls}}) \in \mathbb{R}$, and an arbitrary pair of related class tables $(\mathcal{CT}, \mathcal{CT}') \in CT[R^{\mathtt{cls}}]$. The only pair of related class definitions in $R^{\mathtt{cls}}$ is $(\mathcal{C}, \mathcal{C}')$. For any $(l_i, l'_i) \in V_{\mathtt{Cell}}[\mathcal{CT}, \mathcal{CT}', s, s', R^{\ell}]$ with $s.l = obj\, \mathtt{Cell}\,\{c_i = l_{Si}\}$, $s'.l' = obj\, \mathtt{Cell}\,\{c1 = l_{Si}, c2 = l_{Si}, n = l_{ni}\}$, we consider all $\mathtt{public}\, t_x \rightarrow t m \in \mathcal{CT}$.`Cell`.methods. These are the methods `get` and `set`.

In the case of `get` we have $\overline{t_x} \rightarrow t = \mathtt{void} \rightarrow \mathtt{A}$. Furthermore:

$$\begin{aligned}
& \mathcal{CT} \ \vdash s, l.\mathtt{get}() \rightarrow^{*} s, l_{Si}\\
& \mathcal{CT}' \vdash s', l'.\mathtt{get}()\\
& \quad \rightarrow^{*} s'[l'_i \leftarrow obj\, \mathtt{Cell}\,\{c1 = l_{S}{}'_i, c2 = l_{S}{}'_i, n = m{+}1\}], l_{S}{}'_i
\end{aligned}$$

where $s'.l'_i = obj\, \mathtt{Cell}\,\{c1 = l_{S}{}'_i, c2 = l_{S}{}'_i, n = m\}$. Moreover:

$$(l_{Si}, l_{S}{}'_i) \in V_{\mathtt{A}}[\mathcal{CT}, \mathcal{CT}', s, s'[l_{ni} \leftarrow m{+}1], R^{\ell}]$$

and

$$\begin{aligned}
& (s, \ s'[l'_i \leftarrow obj\, \mathtt{Cell}\,\{c1 = l_{S}{}'_i, c2 = l_{S}{}'_i, n = m{+}1\}],\\
& \ R^{\ell}, R^{\mathtt{cls}}) \qquad\qquad\qquad\qquad\qquad\qquad \in \mathbb{R}
\end{aligned}$$

Similarly for the case of `set` we have $\overline{t_x} \rightarrow t = \mathtt{A} \rightarrow \mathtt{void}$ . Let $(l_s, l'_s) \in V_{\mathtt{A}}[\mathcal{CT}, \mathcal{CT}', s, s', R^{\ell}]$. We have:

$$\begin{aligned}
& \mathcal{CT} \ \vdash s, o.\mathtt{set}(u)\\
& \quad \rightarrow^{*} s[l_i \leftarrow obj\, \mathtt{Cell}\,\{c = l_s\}], \mathtt{unit}\\
& \mathcal{CT}' \vdash s', o'.\mathtt{set}(u')\\
& \quad \rightarrow^{*} s'[l'_i \leftarrow obj\, \mathtt{Cell}\,\{c1 = l'_s, c2 = l_s i', n = m\}], \mathtt{unit}
\end{aligned}$$

and

$$\begin{aligned}
& (s[l_i \leftarrow obj\, \mathtt{Cell}\,\{c = l_s\}],\\
& \ s'[l'_i \leftarrow obj\, \mathtt{Cell}\,\{c1 = l'_s, c2 = l_s i', n = m\}],\\
& \ R^{\ell}, R^{\mathtt{cls}}) \qquad\qquad\qquad\qquad\qquad \in \mathbb{R}
\end{aligned}$$

## 5. $\mathcal{J}_2$: An Extension of $\mathcal{J}_1$With Inheritance

We now extend $\mathcal{J}_1$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 $\mathcal{J}_1$and $\mathcal{J}_2$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 $\mathcal{CT}; \Gamma; s \vdash e{:}t$ 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 $\mathcal{J}_2$, the meta-function *wfClassHierarchy* is true exactly when these conditions hold for a set of classes.

The operational semantics of $\mathcal{J}_2$are mostly the same as of $\mathcal{J}_1$, with the exception of the steps that involve object instantiation. In $\mathcal{J}_2$, when a new object is created, the fields from all the superclasses must be initialized.

## 6. Adequacy in $\mathcal{J}_2$

In this section we study adequacy for $\mathcal{J}_2$. The definition of contextual equivalence we gave earlier (Definition 3.4) still holds for $\mathcal{J}_2$. It conceals though the fact that the context has, what it seems to be, more distinguishing power. It can extend related classes and then use these extensions to distinguish the two sides. We apply

$$\begin{array}{rl}\textsc{Class Definitions:} & \mathcal{C},\mathcal{D} ::= \texttt{class}\ C\ \texttt{extends}\ D\{\overline{mod\,t\,f};\mathcal{K}\overline{\mathcal{M}}\}\\ \textsc{Modifiers:} & mod ::= \cdots \mid \texttt{protected}\end{array}$$

$$\boxed{\mathcal{CT};\Gamma;s\vdash e{:}t} \qquad \boxed{\mathcal{CT}\vdash t_1 <: t_2}$$

$$\frac{\begin{array}{c}\mathcal{CT};\Gamma;s\vdash e{:}t_1\\ \mathcal{CT}\vdash t_1 <: t\end{array}}{\mathcal{CT};\Gamma;s\vdash e{:}t} \qquad \frac{}{\mathcal{CT}\vdash t <: t} \qquad \frac{\mathcal{CT}\vdash t_1 <: t_2 \qquad \mathcal{CT}\vdash t_2 <: t_3}{\mathcal{CT}\vdash t_1 <: t_3} \qquad \frac{\mathcal{CT}.C.\mathsf{super} = D}{\mathcal{CT}\vdash C <: D}$$

$$\boxed{\mathcal{CT}\vdash\mathcal{M}{:}\mathsf{OK\ in}\ C}$$

$$\frac{\mathcal{CT};\overline{x{:}t_1},\texttt{this}{:}C;\emptyset\vdash e{:}t \qquad \mathcal{CT}.C.\mathsf{super} = D \qquad \mathit{overridable}(\mathcal{CT},mod\,\overline{t_1}\rightarrow t\,m,D)}{\mathcal{CT}\vdash mod\ t\,m(\overline{t_1\,x})\{e\}{:}\mathsf{OK\ in}\ C}$$

$$\boxed{\mathcal{CT}\vdash s,e\rightarrow s_1,e_1}$$

$$\frac{l\notin Dom(s)}{\mathcal{CT}\vdash s,\texttt{new Object}\rightarrow s[l = obj\ Object\,\{\}],l} \qquad \frac{\texttt{class}\ C\ \texttt{extends}\ D\{\cdots C()\{\texttt{super}();\overline{\texttt{this}.f := c}\}\cdots\}\in\mathcal{CT}}{\mathcal{CT}\vdash s,\texttt{new}\ C\rightarrow s,(\texttt{new}\ D;\overline{\texttt{this}.f := c})_C}$$

$$\frac{}{\mathcal{CT}\vdash s[l = obj\ D\,\{\overline{f_D = c_D}\}],(l;\overline{\texttt{this}.f := c})_C \rightarrow s[l = obj\ C\,\{\overline{f_D = c_D},\overline{f = c}\}],l}$$

$$\boxed{\textit{Evaluation Contexts}}$$

$$\mathcal{E} ::= \cdots \mid ([\,]\,;\overline{\texttt{this}.f := c})_C$$

**Figure 5.** Syntax, typing, and operational semantics of $\mathcal{J}_2$(differences from $\mathcal{J}_1$)

our method for deriving conditions of 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 $\mathbb{R}$, and we use the same definitions for $V_t$, $A_t$, and $E_t$ (Definition 3.6). The latter sets, though, are larger than before. In $\mathcal{J}_2$an object has type the name of the class that it instantiates, but, because of subsumption, it can also have the type 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 $\mathcal{J}_2$, and, as a consequence, the induction hypothesis of the corresponding abstract proof 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 $\mathcal{J}_2$are the following:

**Theorem 6.1 (Adequacy Conditions for $\mathcal{J}_2$).** *A relation $\mathbb{R}$ is adequate if and only if for all states $(s,s',R^\ell,R^{\mathrm{cls}})$ of $\mathbb{R}$ and for all $(\mathcal{CT},\mathcal{CT}')\in CT[R^{\mathrm{cls}}]$, the following conditions are satisfied:*

1. (Same interfaces) *For all $(\mathcal{C},\mathcal{C}')\in R^{\mathrm{cls}}$,*

$$\begin{array}{rl}\mathcal{C} = \texttt{class}\ C\{ & \overline{\texttt{public}\ t_1\,f_1};\overline{\texttt{private}\ t_2\,f_2};\\ & \overline{\texttt{protected}\ t_3\,f_3};\\ & C()\{\overline{\texttt{this}.f_1 := c_1},\overline{\texttt{this}.f_2 := c_2},\\ & \quad\overline{\texttt{this}.f_3 := c_3}\},\\ & \overline{\texttt{public}\ t_4\,m_4(\overline{t_5\,x_5})\{\cdots\}},\\ & \overline{\texttt{private}\ t_6\,m_6(\overline{t_7\,x_7})\{\ldots\}},\\ & \overline{\texttt{protected}\ t_8\,m_8(\overline{t_9\,x_9})\{\ldots\}}\}\end{array}$$

$$\begin{array}{rl}\mathcal{C}' = \texttt{class}\ C\{ & \overline{\texttt{public}\ t_1\,f_1};\overline{\texttt{private}\ t_2'\,f_2'};\\ & \overline{\texttt{protected}\ t_3\,f_3};\\ & C()\{\overline{\texttt{this}.f_1 := c_1},\overline{\texttt{this}.f_2' := c_2'},\\ & \quad\overline{\texttt{this}.f_3 := c_3}\},\\ & \overline{\texttt{public}\ t_4\,m_4(\overline{t_5\,x_5})\{\cdots\}},\\ & \overline{\texttt{private}\ t_6'\,m_6'(\overline{t_7'\,x_7'})\{\ldots\}},\\ & \overline{\texttt{protected}\ t_8\,m_8(\overline{t_9\,x_9})\{\ldots\}}\}\end{array}$$

2. (Related instances) *For all $(l,l')\in R^\ell$, there exists $t$, such that $\mathcal{CT};\emptyset;s\vdash l{:}t$, $\mathcal{CT}';\emptyset;s'\vdash l'{:}t$.*
3. (Enough instances) *For all $C\in\mathcal{CT}.\mathsf{classnames}$, with $\mathcal{CT}.C.\mathsf{fields} = \overline{mod\,t\,f}$ and $\mathcal{CT}'.C.\mathsf{fields} = \overline{mod'\,t'\,f'}$ and all fresh $l_1$, $l_1'$, there exists $R_1^\ell\supseteq R^\ell\cup\{(l,l')\}$ such that*

$$(s[l = obj\ C\,\{\overline{f = c}\}],s'[l' = obj\ C\,\{\overline{f' = c'}\}],R_1^\ell,R^{\mathrm{cls}})\in\mathbb{R}.$$

*where $\overline{c}$ and $\overline{c'}$ are the initial values of the fields in the constructors of $C$ and its superclasses.*

4. (Related public and protected fields) *For all* $(l, l') \in R^{\ell}$, *with* $s.l = obj\, C\,\{\overline{f = v}\}$ *and* $s'.l' = obj\, C\,\{\overline{f' = v'}\}$, *and for all* $mod\, t_i\, f_i \in \mathcal{C}.\textit{fields}$, *with* $mod \in \{\texttt{public}, \texttt{protected}\}$,
$$(v_i, v_i') \in V_{t_i}[\mathcal{CT}, \mathcal{CT}', s, s', R^{\ell}].$$
5. (Related updates) *For all* $(l, l') \in R^{\ell}$, *with* $s.l = obj\, C\,\{\overline{f = v}\}$ *and* $s'.l' = obj\, C\,\{\overline{f' = v'}\}$, *all* $mod\, t_i\, f_i \in \mathcal{CT}.C.\textit{fields}$, *with* $mod \in \{\texttt{public}, \texttt{protected}\}$, *and all* $(u, u') \in V_{t_i}[\mathcal{CT}, \mathcal{CT}', s, s', R^{\ell}]$,
$$\begin{array}{ll}(s[l \leftarrow obj\, C\,\{\overline{f = v}\backslash f_i = u\}], & \\ \;s'[l' \leftarrow obj\, C\,\{\overline{f' = v'}\backslash f_i = u'\}], & \\ \;R^{\ell}, R^{\texttt{cls}}) & \in \mathbb{R}\end{array}$$
6. (Related public methods) *For all* $(\mathcal{C}, \mathcal{C}') \in R^{\texttt{cls}}$, *all references* $(l, l') \in V_C[\mathcal{CT}, \mathcal{CT}', s, s', R^{\ell}]$, *all methods* $m$ *defined in* $C$ *with* $\mathcal{CT}.C.m = mod\, t_m\, m(\overline{t_x\, x})\{e_3\}$, $\mathcal{CT}'.C.m = mod\, t_m\, m(\overline{t_x\, x})\{e_3'\}$, *and* $mod \in \{\texttt{public}, \texttt{protected}\}$, *and for all* $\overline{(v, v')} \in V_{\overline{t_x}}[\mathcal{CT}, \mathcal{CT}', s, s', R^{\ell}]$,
$$\begin{array}{l} IH^{L}_{\mathbb{R}}(k) \implies \\ \quad R^{\ell}, R^{\texttt{cls}}, \mathbb{R} \vdash (\mathcal{CT} \vdash s, l.m(\overline{v}){:}t_m) \sqsubseteq^{<k+1} \\ \qquad\qquad\qquad (\mathcal{CT}' \vdash s', l'.m(\overline{v'}){:}t_m) \\ IH^{R}_{\mathbb{R}}(k) \implies \\ \quad R^{\ell}, R^{\texttt{cls}}, \mathbb{R} \vdash (\mathcal{CT} \vdash s, l.m(\overline{v}){:}t_m) \sqsupseteq^{<k+1} \\ \qquad\qquad\qquad (\mathcal{CT}' \vdash s', l'.m(\overline{v'}){:}t_m) \end{array}$$

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 $\mathcal{C}$ and $\mathcal{C}'$ 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:

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

  Then by instantiating $D$, 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:

  ```
  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 $(\mathcal{C}, \mathcal{C}') \in R^{\texttt{cls}}$. These methods may be invoked on objects that instantiate subclasses of $\mathcal{C}$ and $\mathcal{C}'$. 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 $R^{\texttt{cls}}$. 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 $\mathcal{J}_1$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 $\mathcal{J}_1$but not in $\mathcal{J}_2$:

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

```
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 $\mathcal{J}_2$

We adapt the two implementations of a Cell from Section 4 to the language $\mathcal{J}_2$, and we show that the equivalence still holds.

```
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}}

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(){
          this.n := this.n + 1;
          if (even(this.n)) then this.c1 else this.c2}}
```

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

$$
\begin{array}{l}
\mathbb{R} = \{(s, s', R^\ell, R^{\texttt{cls}}) \mid \exists \mathcal{CT}, \mathcal{CT}', \overline{l_D, l'_D, D}, \overline{f_1, v_1, v'_1},\\
\qquad\qquad\qquad \overline{l_B, l'_B, B}, \overline{f_2, v_2, v'_2}, \overline{l_C, l'_C, m}, \overline{f_3, v_3, v'_3} :\\
\quad s \;\, = \overline{[l_D = obj\, D\,\{\overline{f_1 = v_1}\}]}\\
\qquad\quad \overline{[l_B = obj\, \boldsymbol{B}\,\{\overline{f_2 = v_2}\}]}\\
\qquad\quad\;\; \overline{[l_C = obj\, \boldsymbol{C}\,\{c = l_B, \overline{\boldsymbol{f_3 = v_3}}\}]}\\
\quad s' = \overline{[l'_D = obj\, D\,\{\overline{f_1 = v'_1}\}]}\\
\qquad\quad \overline{[l'_B = obj\, \boldsymbol{B}\,\{\overline{f_2 = v'_2}\}]}\\
\qquad\quad\;\; \overline{[l_C = obj\, \boldsymbol{C}\,\{c1 = l'_B, c2 = l'_B, n = m, \overline{\boldsymbol{f_3 = v'_3}}\}]}\\
\quad R^{\texttt{cls}} = \{(\mathcal{C}, \mathcal{C}')\}\\
\quad R^\ell = \{\overline{(l_D, l'_D)}, \overline{(l_B, l'_B)}, \overline{(l_C, l'_C)}\}\\
\quad \overline{(v_1, v'_1)}, \overline{(v_2, v'_2)}, \overline{\boldsymbol{(v_3, v'_3)}} \in V_{\overline{t}}[\mathcal{CT}, \mathcal{CT}', s, s', R^\ell]\\
\quad (\mathcal{CT}, \mathcal{CT}') \in CT[R^{\texttt{cls}}]\\
\quad \boldsymbol{\mathcal{CT} \vdash B <: \mathbf{A}}\\
\quad \boldsymbol{\mathcal{CT} \vdash C <: \mathbf{Cell}}\}
\end{array}
$$

Again we constructed this set by inspecting the conditions of adequacy for $\mathcal{J}_2$. 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 `Cell`, which may have extra fields. Subclasses of `Cell` may have also overridden the methods `set` and `get`. Even in this case the equivalence holds, since the implementation of each of the two methods doesn't depend on the other. In the case, for example, that the method `set` called internally the method `get`, inheritance may had broken the equivalence, as we discussed in the previous section.

However, even considering arbitrary subclasses of `Cell`, proving the adequacy conditions for $\mathcal{J}_2$is not substantially harder than proving them for $\mathcal{J}_1$.

## 8. $\mathcal{J}_3$: Adding Downcasting to $\mathcal{J}_2$

Our last extension adds a cast expression to $\mathcal{J}_2$, creating language $\mathcal{J}_3$. The casting operator allows both explicit upcasting and downcasting. The latter is a non type-safe operation which introduces one more error to the language, `cerr`. The extra syntax, typing rules and operational semantics for casting are shown in Figure 6.

## 9. Adequacy in $\mathcal{J}_3$

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 set of answers of type $t$.

**Definition 9.1.** *If $R^\ell$ is a relation on object references, $\mathcal{CT}$, $\mathcal{CT}'$ class tables, and $s$, $s'$ stores, then we define the following relation on answers of type $t$:*

$$
\begin{array}{rl}
A_t[\mathcal{CT}, \mathcal{CT}', s, s', R^\ell] \overset{def}{=} & V_t[\mathcal{CT}, \mathcal{CT}', s, s', R^\ell]\\
& \cup\, \{(\texttt{cerr}, \texttt{cerr}), (\texttt{nerr}, \texttt{nerr})\}
\end{array}
$$

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

The second and last place where we need to consider the new cast expression is in the abstract proof of Conjecture 3.10 for $\mathcal{J}_3$. We need to consider one more case for $e_0$; namely the case when $e_0 = (C)e_1$. This case, though, is handled immediately by the induction hypothesis and thus it does not add any extra conditions of adequacy on the set $\mathbb{R}$. Theorem 6.1 holds also for $\mathcal{J}_3$.

From the analysis discussed above we conclude that downcasting (and casting in general) does not give to the context any extra distinguishing power. Therefore its presence does not affect contextual equivalence.

## 10. An Example With Callbacks

We consider the adaptation to $\mathcal{J}_3$of an example from Meyer and Sieber [18]. We have two implementations of a `Counter` class, one containing a private counter `g`, initialized to zero and then increased by two only by the method `inc`. There is also a method `callP`, which takes as an argument an instance of an some class `A`, provided by the context. `A` is assumed to have a method `P` which can accept as arguments instances of the class `Counter`. By proving this equivalence we show that `g` can be accessed only through the method `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 = class Counter extends Object{
      private int g;
      C(){this.g := 0}
      public int callP(A o){
        o.P(this);
        if even(this.g) then 0 else 1}
      public void inc(){this.g := this.g + 2}}

𝒞′ = class Counter extends Object{
      C(){}
      public int callP(A o){o.P(this); 0}
      public void inc(){unit}}
```

We construct the following set:

$$
\begin{array}{l}
\mathbb{R} = \{(s, s', R^\ell, R^{\texttt{cls}}) \mid \exists \mathcal{CT}, \mathcal{CT}', \overline{l_D, l'_D, D}, \overline{f_1, v_1, v'_1},\\
\qquad\qquad\qquad \overline{l_B, l'_B, B}, \overline{f_2, v_2, v'_2}, \overline{l_C, l'_C, n}, \overline{f_3, v_3, v'_3} :\\
\quad s \;\, = \overline{[l_D = obj\, D\,\{\overline{f_1 = v_1}\}]}\\
\qquad\quad \overline{[l_B = obj\, B\,\{\overline{f_2 = v_2}\}]}\\
\qquad\quad\;\; \overline{[l_C = obj\, C\,\{g = 2n, \overline{f_3 = v_3}\}]}\\
\quad s' = \overline{[l'_D = obj\, D\,\{\overline{f_1 = v'_1}\}]}\\
\qquad\quad \overline{[l'_B = obj\, B\,\{\overline{f_2 = v'_2}\}]}\\
\qquad\quad\;\; \overline{[l_C = obj\, C\,\{\overline{f_3 = v'_3}\}]}\\
\quad R^{\texttt{cls}} = \{(\mathcal{C}, \mathcal{C}')\}\\
\quad R^\ell = \{\overline{(l_D, l'_D)}, \overline{(l_B, l'_B)}, \overline{(l_C, l'_C)}\}\\
\quad \overline{(v_1, v'_1)}, \overline{(v_2, v'_2)}, \overline{(v_3, v'_3)} \in V_{\overline{t}}[\mathcal{CT}, \mathcal{CT}', s, s', R^\ell]\\
\quad (\mathcal{CT}, \mathcal{CT}') \in CT[R^{\texttt{cls}}]\\
\quad \mathcal{CT} \vdash B <: \texttt{A}\\
\quad \mathcal{CT} \vdash C <: \texttt{Counter}\}
\end{array}
$$

The tuples of $\mathbb{R}$ 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 `Counter`. $D$ are all the other classes that may be defined by the context.

We need to show that $\mathbb{R}$ satisfies Theorem 6.1. The interesting case is to show condition 6 for method `callP`.

Consider arbitrary $(s, s', R^\ell, R^{\texttt{cls}}) \in \mathbb{R}$, and $(\mathcal{CT}, \mathcal{CT}') \in CT[R^{\texttt{cls}}]$. For any $(l_{Ci}, l'_{Ci}) \in V_{\texttt{Counter}}[\mathcal{CT}, \mathcal{CT}', s, s', R^\ell]$ with $s.l_{Ci} = obj\,\texttt{Counter}\,\{g = 2n, \cdots\}$, $s'.l'_{Ci} = obj\,\texttt{Counter}\,\{\cdots\}$, and any $(l_{Bj}, l'_{Bj}) \in V_{\texttt{A}}[\mathcal{CT}, \mathcal{CT}', s, s', R^\ell]$, we need to show for any $k$:

$$
\begin{array}{l}
IH^L_{\mathbb{R}}(k) \implies\\
\quad R^\ell, R^{\texttt{cls}}, \mathbb{R} \vdash (\mathcal{CT} \vdash s, l_{Ci}.\texttt{callP}(l_{Bj})\texttt{:int}\,) \sqsubseteq^{<k+1}\\
\qquad\qquad\qquad\qquad (\mathcal{CT}' \vdash s', l'_{Ci}.\texttt{callP}(l'_{Bj})\texttt{:int}\,)
\end{array}
$$

and the reverse.

$$\begin{array}{rll} \text{EXPRESSIONS:} & e ::= \cdots \mid (C)e & \text{Cast Expression} \\ \text{ERRORS:} & \varepsilon ::= \cdots \mid \texttt{cerr} & \text{Cast Error} \end{array}$$

$$\boxed{\mathcal{CT};\Gamma;s\vdash e{:}t}$$

$$\frac{\begin{array}{c}\mathcal{CT};\Gamma;s\vdash e{:}D \\ \mathcal{CT}\vdash D <: C\end{array}}{\mathcal{CT};\Gamma;s\vdash (C)\overline{e}{:}C} \qquad \frac{\begin{array}{c}\mathcal{CT};\Gamma;s\vdash e{:}D \\ \mathcal{CT}\vdash C <: D \qquad C\neq D\end{array}}{\mathcal{CT};\Gamma;s\vdash (C)\overline{e}{:}C}$$

$$\boxed{\mathcal{CT}\vdash s,e\to s_1,e_1}$$

$$\frac{\mathcal{CT}\vdash C <: D}{\mathcal{CT}\vdash s,(D)obj\,C\,\{\overline{f=l}\}\to s,obj\,C\,\{\overline{f=l}\}} \qquad \frac{}{\mathcal{CT}\vdash s,(D)\texttt{null}\to s,\texttt{null}} \qquad \frac{\mathcal{CT}\vdash C \not<: D}{\mathcal{CT}\vdash s,(D)obj\,C\,\{\overline{f=l}\}\to s,\texttt{cerr}}$$

*Evaluation Contexts*

$$\mathcal{E} ::= \cdots \mid (C)[\,]$$

**Figure 6.** Syntax, Typing, and Operational Semantics of $\mathcal{J}_3$(differences with $\mathcal{J}_2$)

We assume $IH^L_{\mathbb{R}}(k)$ and for any $s_1$ and $c$:

$$\begin{array}{l} \mathcal{CT}\ \vdash s, l_{Ci}.\texttt{callP}(l_{Bj}) \to^{<k+1} s_1, c \\ \Longrightarrow \mathcal{CT}\ \vdash s, l_{Bj}.\texttt{P}(l_{Ci}) \to^{<k} s_1, \texttt{unit} \end{array}$$

Because the last derivation has fewer than $k$ steps and

$$\begin{array}{l} (s, s', R^\ell, R^{\texttt{cls}}) \in \mathbb{R}, \\ (l_{Bj}.\texttt{P}(l_{Ci}), l'_{Bj}.\texttt{P}(l'_{Ci})) \in E_{\texttt{void}}\,[\mathcal{CT}, \mathcal{CT}', s, s', R^\ell] \end{array}$$

we can apply the induction hypothesis and get that there exist $s'_1$, $R^\ell$ such that:

$$\begin{array}{l} \mathcal{CT}' \vdash s', l'_{Bj}.\texttt{P}(l'_{Ci}) \to^* s'_1, \texttt{unit} \\ (s_1, s'_1, R^\ell_1, R^{\texttt{cls}}) \in \mathbb{R}, \\ R^\ell \subseteq R^\ell_1 \end{array}$$

Thus the invariant that the field g will contain an even number in store $s_1$, after the call to P, is maintained and therefore $c = 0$. Furthermore

$$\mathcal{CT}' \vdash s', l'_{Ci}.\texttt{callP}(l'_{Bj}) \to^* s'_1, 0$$

and by this we are done.

In this example we didn't need to worry about the implementation of A and its method P, since our use of the induction hypothesis abstracts over them.

## 11. Related Work

Banerjee and Naumann in [3] present a method for reasoning about whole-program equivalence in a subset of Java, similar to $\mathcal{J}_3$. 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 contextually equivalent 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 the abstract proof of Conjecture 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 though the *same* messages with the context, then these packages are equivalent. This technique is not applicable to $\mathcal{J}_3$ where the interaction of classes with their context is more complex, and furthermore it doesn't study the effect of inheritance to class equivalence.

In the same spirit as Jeffrey and Rathke, Ábrahám 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 $\mathcal{J}_1$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 successfully 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 used our technique to study the effect of inheritance on contextual equivalence. 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

[1] Erika Ábrahám, Marcello M. Bonsangue, Frank S. de Boer, and Martin Steffen. Object connectivity and full abstraction for a concurrent calculus of classes. In Zhiming Liu and Keijiro Araki, editors, *ICTAC*, volume 3407 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 2004.

[2] Samson Abramsky. The lazy lambda calculus. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[3] Anindya Banerjee and David A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *J. ACM*, 52(6):894–960, 2005.

[4] Gavin Bierman, Matthew Parkinson, and Andrew Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, Cambridge University Computer Laboratory, April 2003.

[5] Nina Bohr and Lars Birkedal. Relational reasoning for recursive types and references. Submitted for publication, May 2006.

[6] William R. Cook. A denotational semantics of inheritance. Technical Report CS-89-33, 1989.

[7] M. Felleisen, D. P. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.

[8] Matthias Felleisen. *The Calculi of Lambda-v-cs Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Indiana University, 1987.

[9] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. *Lecture Notes in Computer Science*, 1523:241–269, 1999.

[10] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In *ICALP*, pages 299–309, 1980.

[11] Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, February 1996.

[12] A. S. A. Jeffrey and J. Rathke. Java jr.: Fully abstract trace semantics for a core Java language. In *Proc. European Symposium on Programming*, volume 3444 of *Lecture Notes in Computer Science*, pages 423–438. Springer-Verlag, 2005.

[13] Alan Jeffrey and Julian Rathke. A fully abstract may testing semantics for concurrent objects. *Theor. Comput. Sci.*, 338(1-3):17–63, 2005.

[14] Samuel Kamin. Inheritance in smalltalk-80: A denotational definition. In *Proceedings 15th Annual ACM Symposium on Principles of Programming Languages*, pages 80–87, 1988.

[15] Vasileios Koutavas and Mitchell Wand. Bisimulations for untyped imperative objects. In Peter Sestoft, editor, *Proc. ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 146–161, Berlin, Heidelberg, and New York, 2006. Springer-Verlag.

[16] Vasileios Koutavas and Mitchell Wand. Small bisimulations for reasoning about higher-order imperative programs. In *Proceedings 33rd ACM Symposium on Programming Languages*, pages 141–152, January 2006.

[17] Ian A. Mason and Carolyn L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.

[18] Albert R. Meyer and Kurt Sieber. Towards fully abstract semantics for local variables: Preliminary report. In *Proceedings 15th Annual ACM Symposium on Principles of Programming Languages*, pages 191–203, 1988.

[19] Robert Milne and Christopher Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, London, 1976. Also Wiley, New York.

[20] James H. Morris, Jr. *Lambda Calculus Models of Programming Languages*. PhD thesis, MIT, Cambridge, MA, 1968.

[21] Andrew Pitts and Ian Stark. Operational reasoning for functions with local state. In Andrew Gordon and Andrew Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 227–273. Publications of the Newton Institute, Cambridge University Press, 1998.

[22] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[23] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for dynamic sealing. In *Proceedings 31st Annual ACM Symposium on Principles of Programming Languages*, pages 161–172, New York, NY, USA, 2004. ACM Press.

[24] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for type abstraction and recursion. In *Proceedings 32nd Annual ACM Symposium on Principles of Programming Languages*, pages 63–74, New York, NY, USA, 2005. ACM Press.

[25] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, MA, 1993.

## A. Appendix

| | |
|---|---|
| $\mathcal{C}$.classname | Returns the class name defined by the class definition $\mathcal{C}$. |
| $\mathcal{CT}.C$.super | Returns the name of the immediate superclass of $C$. |
| $\mathcal{CT}.C$.constr | Returns the constructor definition of class $C$. |
| $\mathcal{CT}.C$.constr.type | Returns the constructor type of class $C$ |
| $\mathcal{CT}.C$.fields | Returns a sequence of all of the field definitions (public and private) of class $C$ and all its superclasses. |
| $\mathcal{CT}.C$.methods | Returns a sequence of all the method definitions (public and private) that can be invoked on an instance of class $C$. |
| $\mathcal{CT}.C.m$.defclass | Traverses up the class hierarchy, starting from class $C$ and returns the first name of the class in which method $m$ is defined. |

**Figure 7.** Meta-Functions