Proving Class Equivalence

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. Our technique successfully deals with public and private methods and fields, imperative fields, inheritance, and invocations of callbacks. To the extent of our knowledge this is the first sound and complete proof method of equivalence between classes for such a subset of Java. Using our technique we were able to prove equivalences in examples with higher-order behavior, where previous methods for functional calculi admit limitations [17, 20]. We were also able to show equivalences between classes that expose part of their state using public fields, hide part of their functionality using private methods, and are extensible by the surrounding context. Other reasoning techniques for class-based languages [2, 10] restrict the way a class communicates with and abstracts functionality from its context. We derive our technique following a methodology similar to our previous work on functional [13] and object-based [12] languages, thus showing that this methodology gives useful results in a diversity of languages.

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, conceptually and textually, programs in small units that encode different parts of the entire program behavior. This makes classes attractive for reuse and refactoring. But refactoring 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. To formalize this property we adapt the standard notion of contextual equivalence between expressions from functional languages [16] 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 to consider not just equal, but also related stores. CIU theorems [14] 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. Also to the extent of our knowledge 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 [4, 11]). 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 the Observer pattern 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 [3], or exploit properties of some programs, such as ownership confinement [2]. 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 [7] for reasoning about the behavior of concurrent programs. They were applied in sequential calculi by Abramsky [1] and proven to be a congruence by Howe [8]. Sumii and Pierce later gave a bisimulation proof technique which is sound and complete with respect to contextual equivalence in a language with dynamic sealing [19] and a language with recursive and polymorphic types [20]. Their key innovation was to split the bisimulations into parts, and associate each part with the conditions of knowledge under which that part of the bisimulation holds. Building on that idea we were able to devise a technique for deriving sound and complete definitions of bisimulation from a context-based semantics [6]. We applied this method to derive definitions of sound and complete bisimulation for a lambda calculus with store [13] and for an imperative object calculus [12]. We used these techniques to prove non-trivial equivalences that involve local store and higher-order procedures [15].

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-table contexts 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.

```
class Observable extends Object {
  private Observer[10] ovect;
  private int count;

  Observable(){
    super();
    count:=0;
    this.ovect[1] := null;
    ...
    this.ovect[10] := null
  }
  public void add(Observer o){
    if (count < 10) then
      count:= count+1;
      ovect[counter]:=o
    else unit
  }
  public void notifyAll(Object arg){
    notify(1, arg)
  }

  private void notify(int i, Object arg){
    if (i < count+1) then
      ovect[i].notify(arg);
      this.notify(i+1, arg)
    else unit
  }
}
```

```
class Observable extends Object {
  private Observer o;
  private Observable next;
  private int count;

  Observable(){
    super();
    this.o := null;
    this.next := null;
    this.count := 0
  }
  public void add(Observer o){
    if (this.count < 10) then
      this.count := this.count + 1;
      if (this.next = null) then
        this.o := o;
        this.next := new Observable()
      else
        this.next.add(o)
    else unit
  }
  public void notifyAll(Object arg) {
    if (this.next != null) then
      this.o.notify(arg);
      this.next.notifyAll(arg)
    else unit
  }
}
```

Figure 1. Two implementations of the Observer pattern

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*.

In our language, classes can also hide some of their methods from the context by using the `private` access modifier. As a result our technique deals with a larger set of examples than other reasoning techniques for class-based languages [2, 10]. For example, the two implementations of the Observer pattern shown in Figure 1 would not be equivalent if the methods were all public. Classes can also expose part of their state by making some of their fields public. This creates some extra conditions for related classes with public fields to ensure that their instances have the same behavior for all the values that the context can assign to the public fields.

An interesting aspect of our language is that the context can also extend two related classes and override some of the public methods. One would think that our analysis would have been more fine-grained if our language also allowed a `protected` interface of classes. In the contrary, the protected interface is in effect the same as the public interface. If extending a class and then manipulating protected fields and invoking protected methods can distinguish two class implementations, then so does instantiating the class and then manipulating the same fields and invoking the same methods. The real effect of inheritance to our reasoning is that when public methods are invoked either the original body of the method, or the body of a possible method that overrides it may be called. Our technique, though, makes reasoning about invocations of unknown implementations of methods easy because of the use of an induction hypothesis (see also [12]).

Our technique for reasoning about class equivalence has the following benefits:

- It is a sound and complete method for proving contextual equivalence of classes and expressions.
- It is able to prove equivalences between classes with different store behavior.
- It is able to prove equivalences of classes that invoke callbacks, which is a higher-order feature in object-oriented languages.
- It successfully deals with inheritance, and with private and public interfaces of classes.
- It is derived by a method which is applicable to a variety languages.

In Section 2 we give the language that we study, and its static and dynamic semantics. In Section 3 we start our method of deriving a proof technique of equivalence by stating a definition of contextual equivalence between classes, and a definition of adequacy for relations on classes; we then show that these two definitions coincide. In Section 4 we give a definition of contextual equivalence for expressions and we connect it to contextual equivalence of classes. In Section 5 we attempt an abstract proof of adequacy for an arbitrary set, from which we find the conditions that this set needs to satisfy in order to be adequate. We then formulate these conditions in a theorem of adequacy, the main theorem of this paper. Section 6 contains two example equivalences between classes and their proofs using the main theorem. Finally, in Sections 7 and 8, we give some related work and our conclusions.

2. The language FWI Java

For the purposes of this work we choose to study an imperative extension of Featherweight Java [9] which we call FWI Java. The syntax of FWI Java is shown in Figure 2. The expressions of the language are the object operations of Featherweight Java (new object, field lookup, method invocation, casting) with the addition of field update. The language has also constants, and a conditional and a let-binding expression. Moreover fields and methods in our language can be declared `public` or `private`.

The types of FWI Java are the class-name 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 values of the language are constants or objects. The latter are structures that contain the name of the class which they instantiate, and a binding for each field of the class and its superclasses to the location in the store in which the value of the field is kept. We assume there is no shadowing of fields, something that can be achieved automatically by changing the names of all the fields to include the name of the class in which they are defined.

FWI Java has also two kinds of errors: a cast error (`cerr`) for invalid casting, and a null error (`nerr`) for the case that a program tries to perform an operation on a `null` value.

Class definitions are similar to those of FW Java, defining the name of the class, the class which it extends, the fields and methods of the class, and the constructor of the class. We assume that there is a root `Object` class, with no fields and methods, which doesn't extend any other class. In FWI Java there are also `public` and `private` access modifiers in the definitions of fields and methods that specify the scope of these names. Public methods and fields are visible to all classes, while private methods and fields are visible only from the same class.

To allow private fields to be completely isolated from the context, we used a different kind of constructors than in FW Java. Instead of exposing as arguments to the constructors all the fields, public and private, we allow a different definition of constructors. The constructor of each class may have an arbitrary number of arguments of any type and it must initialize all the fields in the class, possibly using its arguments. Moreover, just as in FW Java, a call to the constructor of the superclass is made by the keyword `super` before any local initialization of fields. The expressions that are used to provide the arguments to `super` and the values to the field initializations may not refer to the special variable `this` (*DontReferToSelf*(e)), ensuring that no uninitialized field is ever used. As an example the following class definition is valid in our language:

```
class C extends Object{
  public int f;
  private int g;
  C(int x){super(); this.f := *(17, x); this.g := +(17, x); }}
```

Class tables are sets of classes. Well-formed class tables define a tree hierarchy, where the root of the hierarchy is the class $Object$:

$$\texttt{class Object extends } \cdot\{\texttt{Object}()\{\}\}$$

$Object$ is the only class that doesn't extend any other class. Furthermore, all the classes in a well-formed class table have distinct names. We test the well-formedness of a class table by the predicate *wfClassHierarchy*$(\mathcal{CT})$.

We will use meta-operations and the dot notation to perform static lookup on class tables, methods, and fields. 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 and its superclasses. A complete table of these meta-operations and a description of their functionality is shown in Figure 6 of the Appendix.

Stores are partial maps from locations to values. The type of the value stored in location l of store s is obtained by $s.l.\mathsf{type}$. A *program configuration* is a triple composed by 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.

The typing rules of FWI Java are shown in Figure 3. The type judgments for expressions have the form $\mathcal{CT}; \Gamma; s \vdash e{:}t$. Γ is the type environment. The store s is used to type-check the locations referred to by the objects; the value stored in a location has the type $s.l.\mathsf{type}$. 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 FWI Java.

$\mathcal{CT} \vdash t_1 <: t_2$ are the subtyping judgments imposed by the class-hierarchy and the reflexive and transitive property. 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 FWI Java. A small-step $\mathcal{CT} \vdash s, e \ \rightarrow\ s_1, e_1$ 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 $\rightarrow^*$ to be the reflexive and transitive closure of $\rightarrow$, and $\rightarrow^{<k}$ the reflexive and up to $k-1$ steps transitive closure of $\rightarrow$. We also write $\mathcal{CT} \vdash s, e{\downarrow}$ iff there exists s_1, w, such that $\mathcal{CT} \vdash s, e \rightarrow^* s_1, w$.

We write in calligraphic font 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 $obj\, C\, \{f_1 = l_1, f_2 = l_2, \ldots, f_n = l_n\}$.

3. Equivalence and Adequacy

We reason about the behavior of class implementations. Thus 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, in the hole of a class table context corresponds to set union:*

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

Definition 3.2. *$R^{\mathtt{cls}}$ is a relation on classes iff it is a set of pairs of class definitions, such that for all $(\mathcal{C}, \mathcal{C}') \in R^{\mathtt{cls}}$, 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 type 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 $R^{\mathtt{cls}}$ is a relation on classes, then the following is its extension to class tables:*

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

We give the following definition of contextual equivalence for FWI Java:

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 does not give rise to a usable proof technique for equivalence. We will instead give a definition of adequacy which can be used in an inductive proof, and then show how adequacy coincides with contextual equivalence.

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

PROGRAM CONFIGURATIONS:	$pconf \in \mathcal{CT} \times \text{STORES} \times \text{EXPRESSIONS}$	
CLASS TABLES:	$\mathcal{CT} \in \mathcal{P}(\text{CLASS DEFINITIONS})$	
CLASS DEFINITIONS:	$\mathcal{C} ::= \texttt{class}\ C\ \texttt{extends}\ D\{\overline{mod\ t\ f}; \mathcal{K}\overline{\mathcal{M}}\}$	
CONSTRUCTOR DEFINITIONS:	$\mathcal{K} ::= C(\overline{t\,x})\{\texttt{super}(\overline{e}); \overline{\texttt{this}.f := e}\}$	
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
	$\mid\ e.f \mid e.m(\overline{e})$	Field Lookup, Method Invocation
	$\mid\ \texttt{new}\, C(\overline{e}) \mid (C)e$	Class Instantiation, Object Cast
	$\mid\ e.f := e \mid \texttt{op}(\overline{e})$	Field Update, Arithmetic Operators
	$\mid\ \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	Let Expression, Conditional
VALUES:	$v, u, w ::= c \mid o$	
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
OBJECTS:	$o ::= obj\, C\{\overline{f = l}\}$	
ERRORS:	$\varepsilon ::= \texttt{cerr} \mid \texttt{nerr}$	Cast Error, Null Error
LOCATIONS:	l, k	
STORES:	$s \in \text{LOCATIONS} \rightharpoonup \text{VALUES}$	

Figure 2. Syntax of FWI Java

Definition 3.5. *A relation $\mathbb{R}$ is a set of tuples $(s, s', R^{\texttt{val}}, R^{\texttt{cls}})$, where s, s' are stores, $R^{\texttt{val}}$ is a relation on objects, and $R^{\texttt{cls}}$ is a relation on classes.*

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

Definition 3.6. *If $R^{\texttt{val}}$ is a relation on objects, $\mathcal{CT}$, $\mathcal{CT}'$ class tables, and s, s' stores, then the following is a relation on values of type t:*

$$V_t[\mathcal{CT}, \mathcal{CT}', s, s', R^{\texttt{val}}] \stackrel{def}{=} \{(v, v') \mid (v, v') \in R^{\texttt{val}} \cup Id_{const},\ \mathcal{CT}; \emptyset; s \vdash v{:}t,\ \mathcal{CT}'; \emptyset; s' \vdash v'{:}t\} \cup \{(\texttt{cerr}, \texttt{cerr}), (\texttt{nerr}, \texttt{nerr})\}$$

Definition 3.7. *If $R^{\texttt{val}}$ is a relation on objects, $\mathcal{CT}$, $\mathcal{CT}'$ class tables, and s, s' stores, then the following is a relation on closed expressions of type t:*

$$E_t[\mathcal{CT}, \mathcal{CT}', s, s', R^{\texttt{val}}] \stackrel{def}{=} \{([\overline{v/x}]e, [\overline{v'/x}]e) \mid \overline{FV(e) \subseteq \{\overline{x}\}},\ \overline{(v, v') \in R^{\texttt{val}}},\ Obj(e) = \emptyset,\ \mathcal{CT}; \emptyset; s \vdash [\overline{v/x}]e{:}t,\ \mathcal{CT}'; \emptyset; s' \vdash [\overline{v'/x}]e{:}t\}$$

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

We now give the definition of adequate relations.

Definition 3.8 (Adequacy). *$\mathbb{R}$ is adequate if and only if:*

$$\begin{array}{l}\forall (s, s', R^{\texttt{val}}, R^{\texttt{cls}}) \in \mathbb{R}.\\ \quad \forall (\mathcal{CT}, \mathcal{CT}') \in CT[R^{\texttt{cls}}].\\ \quad\quad \forall t, \forall (e, e') \in E_t[\mathcal{CT}, \mathcal{CT}', s, s', R^{\texttt{val}}].\\ \quad\quad\quad \forall s_1, w.\\ \quad\quad\quad\quad (\mathcal{CT} \vdash s, e \rightarrow^* s_1, w)\\ \quad\quad\quad\quad \Longrightarrow\ \exists s_1', w', R_1^{\texttt{val}}.\\ \quad\quad\quad\quad\quad (\mathcal{CT}' \vdash s', e' \rightarrow^* s_1', w')\\ \quad\quad\quad\quad\quad \wedge ((w, w') \in V_t[\mathcal{CT}, \mathcal{CT}', s_1, s_1', R_1^{\texttt{val}}])\\ \quad\quad\quad\quad\quad \wedge ((s_1, s_1', R_1^{\texttt{val}}, R^{\texttt{cls}}) \in \mathbb{R})\\ \quad\quad\quad\quad\quad \wedge (R^{\texttt{val}} \subseteq R_1^{\texttt{val}})\end{array}$$

and the reverse.

Adequate relations are sound and complete in the following way.

Theorem 3.9 (Soundness). *If $\mathbb{R}$ is adequate and $(\emptyset, \emptyset, \emptyset, R^{\texttt{cls}}) \in \mathbb{R}$ then $R^{\texttt{cls}} \subseteq (\equiv)$.*

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

Theorem 3.10 (Completeness). *If $R^{\texttt{cls}} \subseteq (\equiv)$ then there exists adequate $\mathbb{R}$ with $(\emptyset, \emptyset, \emptyset, R^{\texttt{cls}}) \in \mathbb{R}$.*

Proof. (Sketch) Let $R^{\texttt{cls}} \subseteq (\equiv)$. Construct $\mathbb{R}$ inductively, starting with $(\emptyset, \emptyset, \emptyset, R^{\texttt{cls}}) \in \mathbb{R}$ as the base case, and adding tuples in $\mathbb{R}$ using Definition 3.8. For $R_1^{\texttt{val}}$ use the extension of $R^{\texttt{val}}$ with the objects that were created during the evaluation. If there is a state of $\mathbb{R}$ that distinguishes the two sides, then it means that there is a context that would invalidate Definition 3.4. Thus the constructed $\mathbb{R}$ is adequate. □

4. Extension of Equivalence to Open Expressions

In this section we give a definition of contextual equivalence between expressions which is closer to the standard one and its con-

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

$$\frac{x{:}t\in\Gamma}{\mathcal{CT};\Gamma;s\vdash x{:}t}\qquad\frac{}{\mathcal{CT};\Gamma;s\vdash \texttt{unit}{:}\texttt{void}}\qquad\frac{c\in\{\texttt{true},\texttt{false}\}}{\mathcal{CT};\Gamma;s\vdash c{:}\texttt{bool}}\qquad\frac{c\in\{\pm1,\pm2,\cdots\}}{\mathcal{CT};\Gamma;s\vdash c{:}\texttt{int}}\qquad\frac{\texttt{class}\ C\ \texttt{extends}\ D\{\cdots\}\in\mathcal{CT}}{\mathcal{CT};\Gamma;s\vdash \texttt{null}{:}C}$$

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

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

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

$\boxed{\mathcal{CT}\vdash t_1<:t_2}$

$$\frac{}{\mathcal{CT}\vdash t<:t}\qquad\frac{\mathcal{CT}\vdash t_1<:t_2\qquad\mathcal{CT}\vdash t_2<:t_3}{\mathcal{CT}\vdash t_1<:t_3}\qquad\frac{\mathcal{CT}.C.\mathsf{super}=D}{\mathcal{CT}\vdash C<:D}$$

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

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

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

$$\frac{\begin{array}{c}\mathcal{K}=C(\overline{t_x\,x})\{\texttt{super}(\overline{e_1});\overline{\texttt{this}.f:=e_2}\}\\ \mathcal{CT}.D.\mathsf{constr.type}=\overline{t_d}{\to}D\\ DontReferToSelf(\overline{e_1})\qquad\mathcal{CT};\overline{x{:}t_x};\emptyset\vdash\overline{e_1{:}t_d}\\ DontReferToSelf(\overline{e_2})\qquad\mathcal{CT};\overline{x{:}t_x};\emptyset\vdash\overline{e_2{:}t_f}\\ \mathcal{CT}\vdash\overline{\mathcal{M}}{:}\mathsf{OK\ in}\ C\end{array}}{\mathcal{CT}\vdash\texttt{class}\ C\ \texttt{extends}\ D\{\overline{t_f\,f};\mathcal{K}\overline{\mathcal{M}}\}{:}\mathsf{OK}}$$

$\boxed{\mathcal{CT}{:}\mathsf{OK}}$

$$\frac{\{\overline{\mathcal{C}}\}\vdash\overline{\mathcal{C}}{:}\mathsf{OK}\qquad wfClassHierarchy(\{\overline{\mathcal{C}}\})}{\{\overline{\mathcal{C}}\}{:}\mathsf{OK}}$$

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

$$\frac{\mathcal{CT}{:}\mathsf{OK}\qquad\mathcal{CT};\emptyset;s\vdash e{:}t}{(\mathcal{CT}\vdash s,e){:}\mathsf{OK}}$$

Figure 3. Typing of FWI Java

nection with ($\equiv$). To do this we need to define a family of relations on open expressions:

Definition 4.1. *A relation R^{exp} on expressions is a set of tuples (Γ,e,e',t), such that for all class tables $\mathcal{CT}$:*

$$\mathcal{CT};\Gamma;\emptyset\vdash e{:}t\iff\mathcal{CT};\Gamma;\emptyset\vdash e'{:}t$$

This definition requires "compile-time" interchangeability of related expressions, in the same sense of Definition 3.2.

We also define the necessary class and expression contexts, and the extension of relations on expressions to relations on class tables:

Definition 4.2 (Class and Expression Contexts).

$$\begin{array}{rl}\mathcal{C}[\,]::=&\texttt{class}\ C\ \texttt{extends}\ D\{\\ &\overline{mod\,t_f\,f};\\ &\overline{C(\overline{t\,x})\{\texttt{super}(\overline{E[\,]});\overline{\texttt{this}.f:=E[\,]}\}}\\ &\overline{mod\ t_m\,m(t\,x)\{E[\,]\}}\}\\ E[\,]::=&[\,]\mid E[\,].f\mid E[\,].m(\overline{E[\,]})\mid\texttt{new}\,C(E[\,])\mid(C)E[\,]\\ &\mid E[\,].f:=E[\,]\mid\mathsf{op}(\overline{E[\,]})\mid\texttt{let}\ x=E[\,]\ \texttt{in}\ E[\,]\\ &\mid\texttt{if}\ E[\,]\ \texttt{then}\ E[\,]\ \texttt{else}\ E[\,]\end{array}$$

Definition 4.3. *If R^{exp} is a relation on expressions, then the following is its extension to class tables:*

$$\begin{array}{rl}CT[R^{\text{exp}}]\stackrel{def}{=}\{(\mathcal{CT}[\overline{\mathcal{C}[\overline{e}]}],\mathcal{CT}[\overline{\mathcal{C}[\overline{e'}]}])\mid&(\Gamma,e,e',t)\in R^{\text{exp}},\\ &\mathcal{CT}[\overline{\mathcal{C}[\overline{e}]}]{:}\mathsf{OK},\\ &\mathcal{CT}[\overline{\mathcal{C}[\overline{e'}]}]{:}\mathsf{OK}\}\end{array}$$

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

$$\frac{\mathcal{CT}.C.\mathsf{constr} = C(\overline{t\,x})\{\mathtt{super}(\overline{e_1}); \overline{\mathtt{this}.f := e_2}\} \quad \mathcal{CT}.C.\mathsf{super} = D}{\mathcal{CT} \vdash s, \mathtt{new}\,C(\overline{v}) \to s, \overline{[v/x]}(\mathtt{new}D(\overline{e_1}); \mathtt{this}.\overline{f} := \overline{e_2})_C}$$

$$\frac{\overline{l_c} \notin Dom(s)}{\mathcal{CT} \vdash s, (obj\ D\{\overline{f_d = l_d}\}; \mathtt{this}.\overline{f_c} := \overline{v_c})_C \to s[\overline{l_c = v_c}], obj\ C\{\overline{f_d = l_d}, \overline{f_c = l_c}\}}$$

$$\overline{\mathcal{CT} \vdash s, obj\ C\{\overline{f = l}\}.f_i \to s, s.l_i} \qquad \overline{\mathcal{CT} \vdash s, \mathtt{null}.f_i \to s, \mathtt{nerr}}$$

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

$$\overline{\mathcal{CT} \vdash s, obj\ C\{\overline{f = l}\}.f_i := v \to s[l_i \leftarrow v], \mathtt{unit}} \qquad \overline{\mathcal{CT} \vdash s, \mathtt{null}.f_i := v \to s, \mathtt{nerr}}$$

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

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

Evaluation Contexts

$$\begin{array}{l} E ::= \mathcal{E} \mid E[E] \\ \mathcal{E} ::= [\,] \mid [\,].f \mid [\,].m(\overline{e}) \mid v.m(\overline{v}, [\,], \overline{e}) \mid \mathtt{new}C(\overline{v}, [\,], \overline{e}) \mid (C)[\,] \mid [\,].\overline{f} := e \mid v.f := [\,] \mid \mathtt{let}\ x = [\,]\ \mathtt{in}\ e \\ \quad \mid \mathtt{if}\ [\,]\ \mathtt{then}\ e\ \mathtt{else}\ e \mid \mathtt{op}(\overline{v}, [\,], \overline{e}) \mid ([\,]; \mathtt{this}.\overline{f} := \overline{e})_C \mid (v; \mathtt{this}.\overline{f} := \overline{v}, [\,], \overline{e})_C \end{array}$$

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

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

Figure 4. Small-step Operational Semantics of FWI Java

Contextual equivalence of expressions in FWI Java is defined to be the following relation:

Definition 4.4 (Contextual Equivalence of Expressions ($\equiv_e$)). *($\equiv_e$) is the largest relation between expressions, such that for all $(\mathcal{CT}, \mathcal{CT}') \in CT[\equiv_e]$, expressions e, and types t, such that $\mathcal{CT}; \emptyset; \emptyset \vdash e{:}t$ and $\mathcal{CT}'; \emptyset; \emptyset \vdash e{:}t$, we have:*

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

We now give the connection between ($\equiv$) and ($\equiv_e$):

Theorem 4.5. *$(\Gamma, e, e', t) \in (\equiv_e)$ if and only if $R^{\mathtt{cls}} \in (\equiv)$, and for fresh identifiers $\overline{y}$:*

$$\begin{array}{l} R^{\mathtt{cls}} = \{\mathtt{class}\ C\ \mathtt{extends}\ Object\{\mathtt{public}\ t\,m(t_x\,y)\{\overline{[y/x]}e\}\}, \\ \qquad \mathtt{class}\ C\ \mathtt{extends}\ Object\{\mathtt{public}\ t\,m(t_x\,y)\{\overline{[y/x]}e'\}\})\} \\ \Gamma = \overline{t_x\,x} \end{array}$$

Proof. Appendix. □

In the above theorem we rename the free variables inside the related expressions, in order to avoid erroneous capturing of the special variable `this`.

5. Proof Obligations for Adequacy

One may show the equivalence between two class implementations by constructing a set $\mathbb{R}$ and then prove its adequacy by an induction based on Definition 3.8. The construction of the appropriate $\mathbb{R}$ is not obvious, though. Moreover for each $\mathbb{R}$ one would have to repeat the entire proof of adequacy.

Our goal in this section is to find the sufficient properties of $\mathbb{R}$ that would make the proof of adequacy to go through. These properties will serve as a guide to the construction of $\mathbb{R}$. Furthermore we will show that proving just these conditions on a set is equivalent to doing the inductive proof for that set.

To do this we investigate the class of all inductive proofs based on Definition 3.8 by abstracting over the concrete structure of $\mathbb{R}$ and attempting to prove adequacy. This abstract proof reveals the sub-cases of the induction that don't go through just by using the induction hypothesis, but also require that the states of $\mathbb{R}$ satisfy some extra properties, the *proof obligations* of $\mathbb{R}$.

As we will show in the examples, the individual proof obligations for $\mathbb{R}$ give a guideline on how to construct such an adequate set. Furthermore proving these conditions requires less effort than the full-blown induction.

Conjecture 5.1 (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 3.8 and one for the reverse direction. The induction hypothesis of the former is:

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

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 $e_0, \overline{v}, \overline{v'}$, such that $FV(e_0) \subseteq \{\overline{x}\}$, $Obj(e_0) = \emptyset$, $(\overline{v}, \overline{v'}) \in R^{\mathtt{val}}$, $\mathcal{CT}; \emptyset; s \vdash [\overline{v/x}]e_0 : t$, $\mathcal{CT}'; \emptyset; s' \vdash [\overline{v'/x}]e_0 : t$. We proceed by cases on e_0. We demonstrate here only the case of method invocation:

If $e_0 = e_1.m(\overline{e_2})$ then we have

$$\mathcal{CT} \vdash s, [\overline{v/x}](e_1.m(\overline{e_2})) \rightarrow^{<k+1} s_1, w \tag{1}$$

We have the following cases:

- $w = \varepsilon$ and

$$\mathcal{CT} \vdash s, [\overline{v/x}]([e_1].m(\overline{e_2})) \rightarrow^{<k} s_1, [\overline{v/x}]([\varepsilon].m(\overline{e_2})) \rightarrow s_1, \varepsilon$$

thus:

$$\mathcal{CT} \vdash s, [\overline{v/x}]e_1 \rightarrow^{<k} s_1, \varepsilon \tag{2}$$

By (2) and $IH(k)$ we get that there exist s_1', $R_1^{\mathtt{val}}$, such that:

$$
\begin{array}{l}
\quad \mathcal{CT} \vdash s', [\overline{v'/x}]e_1 \rightarrow^{*} s_1', \varepsilon \\
\wedge (s_1, s_1', R_1^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R} \\
\wedge R^{\mathtt{val}} \subseteq R_1^{\mathtt{val}} \\
\wedge (\varepsilon, \varepsilon) \in R_1^{\mathtt{val}}
\end{array}
$$

and furthermore:

$$\mathcal{CT} \vdash s', [\overline{v'/x}](e_1.m(\overline{e_2})) \rightarrow^{*} s_1', [\overline{v'/x}]([\varepsilon].m(\overline{e_2})) \rightarrow s_1', \varepsilon$$

- $w = \varepsilon$ and

$$
\begin{array}{l}
\mathcal{CT} \vdash s, [\overline{v/x}]([e_1].m(\overline{e_2})) \\
\quad \rightarrow^{<k} s_1, [\overline{v/x}]([w_1].m(\overline{e_2})) \\
\quad\quad = [\overline{v/x}](w_1.m([e_{21}], \overline{e_2})) \\
\quad \rightarrow^{<k} s_{21}, [\overline{v/x}](w_1.m([w_{21}], \overline{e_2})) \\
\quad\quad = [\overline{v/x}](w_1.m(w_{21}, [e_{22}], \overline{e_2})) \\
\quad \cdots \\
\quad \rightarrow^{<k} s_{2j-1}, [\overline{v/x}](w_1.m(\overline{w_2}, [e_{2j}], \overline{e_2})) \\
\quad \rightarrow^{<k} s_{2j}, [\overline{v/x}](w_1.m(\overline{w_2}, [\varepsilon], \overline{e_2})) \\
\quad \rightarrow s_{2j}, \varepsilon
\end{array}
$$

thus:

$$\mathcal{CT} \vdash s, [\overline{v/x}]e_1 \rightarrow^{<k} s_1, w_1 \tag{3}$$

$$\mathcal{CT} \vdash s_1, [\overline{v/x}]e_{21} \rightarrow^{<k} s_{21}, w_{21} \tag{4}$$

$$\cdots$$

$$\mathcal{CT} \vdash s_{2j-2}, [\overline{v/x}]e_{2j-1} \rightarrow^{<k} s_{2j-1}, w_{2j-1} \tag{5}$$

$$\mathcal{CT} \vdash s_{2j-1}, [\overline{v/x}]e_{2j} \rightarrow^{<k} s_{2j}, \varepsilon \tag{6}$$

By (3)-(6) and the induction hypothesis at k, we get, that there exist s_1', w_1', $R_1^{\mathtt{val}}$, $\overline{s_2'}$, $\overline{w_2'}$, $\overline{R_2^{\mathtt{val}}}$, such that:

$$
\begin{array}{l}
\quad \mathcal{CT} \vdash s', [\overline{v'/x}]e_1 \rightarrow^{*} s_1', w_1' \\
\wedge ((s_1, s_1', R_1^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R}) \\
\wedge (R^{\mathtt{val}} \subseteq R_1^{\mathtt{val}}) \\
\wedge (w_1, w_1') \in R_1^{\mathtt{val}} \\
\\
\quad \mathcal{CT} \vdash s_1', [\overline{v'/x}]e_{21} \rightarrow^{*} s_{21}', w_{21}' \\
\wedge ((s_{21}, s_{21}', R_{21}^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R}) \\
\wedge (R_1^{\mathtt{val}} \subseteq R_{21}^{\mathtt{val}}) \\
\wedge (w_{21}, w_{21}') \in R_{21}^{\mathtt{val}} \\
\quad \cdots \\
\quad \mathcal{CT} \vdash s_{2j-2}', [\overline{v'/x}]e_{2j-1} \rightarrow^{*} s_{2j-1}', w_{2j-1}' \\
\wedge ((s_{2j-1}, s_{2j-1}', R_{2j-1}^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R}) \\
\wedge (R_{2j-2}^{\mathtt{val}} \subseteq R_{2j-1}^{\mathtt{val}}) \\
\wedge (w_{2j-1}, w_{2j-1}') \in R_{2j-1}^{\mathtt{val}} \\
\\
\quad \mathcal{CT} \vdash s_{2j-1}', [\overline{v'/x}]e_{2j} \rightarrow^{*} s_{2j}', \varepsilon \\
\wedge ((s_{2j}, s_{2j}', R_{2j}^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R}) \\
\wedge (R_{2j-1}^{\mathtt{val}} \subseteq R_{2j}^{\mathtt{val}}) \\
\wedge (\varepsilon, \varepsilon) \in R_{2j}^{\mathtt{val}}
\end{array}
$$

Therefore

$$
\begin{array}{l}
\mathcal{CT} \vdash s', [\overline{v'/x}]([e_1].m(\overline{e_2})) \\
\quad \rightarrow^{*} s_{2j}', [\overline{v'/x}](w_1'.m(\overline{w_2'}, [\varepsilon], \overline{e_2})) \\
\quad \rightarrow s_{2j}', \varepsilon
\end{array}
$$

and $((s_{2j}, s_{2j}', R_{2j}^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R})$ and $(R^{\mathtt{val}} \subseteq R_{2j}^{\mathtt{val}})$, as required.

- $w = \mathtt{nerr}$ and

$$
\begin{array}{l}
\mathcal{CT} \vdash s, [\overline{v/x}]([e_1].m(\overline{e_2})) \\
\quad \rightarrow^{<k} s_1, [\overline{v/x}]([\mathtt{null}].m(\overline{e_2})) \\
\quad\quad = [\overline{v/x}](\mathtt{null}_1.m([e_{21}], \overline{e_2})) \\
\quad \rightarrow^{<k} s_{21}, [\overline{v/x}](\mathtt{null}.m([w_{21}], \overline{e_2})) \\
\quad\quad = [\overline{v/x}](\mathtt{null}.m(w_{21}, [e_{22}], \overline{e_2})) \\
\quad \cdots \\
\quad \rightarrow^{<k} s_{2n}, [\overline{v/x}](\mathtt{null}.m(\overline{w_2})) \\
\quad \rightarrow s_{2n}, \mathtt{nerr}
\end{array}
$$

thus:

$$\mathcal{CT} \vdash s, [\overline{v/x}]e_1 \rightarrow^{<k} s_1, \mathtt{null} \tag{7}$$

$$\mathcal{CT} \vdash s_1, [\overline{v/x}]e_{21} \rightarrow^{<k} s_{21}, w_{21} \tag{8}$$

$$\cdots$$

$$\mathcal{CT} \vdash s_{2n-1}, [\overline{v/x}]e_{2n} \rightarrow^{<k} s_{2n}, w_{2n} \tag{9}$$

By (7)-(9) and $IH(k)$ we get that there exist s_1', w_1', $R_1^{\mathtt{val}}$, $\overline{s_2'}$, $\overline{w_2'}$, $\overline{R_2^{\mathtt{val}}}$, such that:

$$
\begin{array}{l}
\quad \mathcal{CT} \vdash s', [\overline{v'/x}]e_1 \rightarrow^{*} s_1', w_1' \\
\wedge ((s_1, s_1', R_1^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R}) \\
\wedge (R^{\mathtt{val}} \subseteq R_1^{\mathtt{val}}) \\
\wedge (w_1, w_1') \in R_1{}^{\mathtt{val}}
\end{array}
$$

$$
\begin{array}{l}
\quad \mathcal{CT} \vdash s_1', [\overline{v'/x}]e_{21} \rightarrow^* s_{21}', w_{21}' \\
\wedge ((s_{21}, s_{21}', R_{21}^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R}) \\
\wedge (R_1^{\mathtt{val}} \subseteq R_{21}^{\mathtt{val}}) \\
\wedge (w_{21}, w_{21}') \in R_{21}^{\mathtt{val}} \\
\quad \ldots \\
\quad \mathcal{CT} \vdash s_{2n-1}', [\overline{v'/x}]e_{2n} \rightarrow^* s_{2n}', w_{2n}' \\
\wedge ((s_{2n}, s_{2n}', R_{2n}^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R}) \\
\wedge (R_{2n-1}^{\mathtt{val}} \subseteq R_{2n}^{\mathtt{val}}) \\
\wedge (w_{2n}, w_{2n}') \in R_{2n}^{\mathtt{val}}
\end{array}
$$

and therefore:

$\mathcal{CT} \vdash s', [\overline{v'/x}](e_1.m(\overline{e_2})) \rightarrow^* s_1', [\overline{v'/x}]([\mathtt{null}].m(\overline{w_2})) \rightarrow s_1', \mathtt{nerr}$ and $((s_{2n}, s_{2n}', R_{2n}^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R})$, $(R^{\mathtt{val}} \subseteq R_{2n}^{\mathtt{val}})$, $(\mathtt{nerr}, \mathtt{nerr}) \in R_{2n}^{\mathtt{val}}$, as required.

- $w \neq \varepsilon$ and

$$
\begin{array}{l}
\mathcal{CT} \vdash s, [\overline{v/x}]([e_1].m(\overline{e_2})) \\
\quad \rightarrow^{<k} s_1, [\overline{v/x}]([w_1].m(\overline{e_2})) = [\overline{v/x}](w_1.m([e_{21}], \overline{e_2})) \\
\quad \rightarrow^{<k} s_{21}, [\overline{v/x}](w_1.m([w_{21}], \overline{e_2})) = [\overline{v/x}](w_1.m(w_{21}, [e_{22}], \overline{e_2})) \\
\quad \ldots \\
\quad \rightarrow^{<k} s_{2n}, (w_1.m(\overline{w_2})) \\
\quad \rightarrow s_{2n}, [\overline{w_2/x}]e_3 \\
\quad \rightarrow^{<k} s_3, w
\end{array}
$$

where $w_1 = obj\, C\, \{\cdots\}$ and $\mathcal{CT}.C.m = \mathtt{public}\ t\, m(\overline{t_x\, x})\{e_3\}$. Thus:

$$\mathcal{CT} \vdash s, [\overline{v/x}]e_1 \rightarrow^{<k} s_1, w_1 \tag{10}$$

$$\mathcal{CT} \vdash s_1, [\overline{v/x}]e_{21} \rightarrow^{<k} s_{21}, w_{21} \tag{11}$$

$$\ldots$$

$$\mathcal{CT} \vdash s_{2n-1}, [\overline{v/x}]e_{2n} \rightarrow^{<k} s_{2n}, w_{2n} \tag{12}$$

$$\mathcal{CT} \vdash s_{2n}, [\overline{w_2/x}]e_3 \rightarrow^{<k} s_3, w \tag{13}$$

By (10)-(12) and the induction hypothesis at k we get:

$$
\begin{array}{l}
\quad \mathcal{CT} \vdash s', [\overline{v'/x}]e_1 \rightarrow^* s_1', w_1' \\
\wedge ((s_1, s_1', R_1^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R}) \\
\wedge (R^{\mathtt{val}} \subseteq R_1^{\mathtt{val}}) \\
\wedge (w_1, w_1{}') \in R_1{}^{\mathtt{val}} \\
\\
\quad \mathcal{CT} \vdash s_1', [\overline{v'/x}]e_{21} \rightarrow^* s_{21}', w_{21}' \\
\wedge ((s_{21}, s_{21}', R_{21}^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R}) \\
\wedge (R_1^{\mathtt{val}} \subseteq R_{21}^{\mathtt{val}}) \\
\wedge (w_{21}, w_{21}') \in R_{21}^{\mathtt{val}} \\
\quad \ldots \\
\quad \mathcal{CT} \vdash s_{2n-1}', [\overline{v'/x}]e_{2n} \rightarrow^* s_{2n}', w_{2n}' \\
\wedge ((s_{2n}, s_{2n}', R_{2n}^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R}) \\
\wedge (R_{2n-1}^{\mathtt{val}} \subseteq R_{2n}^{\mathtt{val}}) \\
\wedge (w_{21}, w_{21}') \in R_{21}^{\mathtt{val}}
\end{array}
$$

To continue our proof we need that $R^{\mathtt{cls}}$ satisfies the following two conditions:

For all $(s, s', R^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R}$, and all $(obj\, C\, \{\cdots\}, obj\, C'\, \{\cdots\}) \in R^{\mathtt{val}}$, we have $C = C'$.

For all $(s, s', R^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R}$, and all $(\mathcal{C}, \mathcal{C}') \in R^{\mathtt{cls}}$, with

$\mathcal{C} = \mathtt{class}\ C\ \mathtt{extends}\ D\{\cdots \overline{\mathtt{public}\ t\, m(\overline{t_x\, x})\{\cdots\}} \cdots\}$

$\mathcal{C} = \mathtt{class}\ C'\ \mathtt{extends}\ D'\{\cdots \overline{\mathtt{public}\ t'\, m'(\overline{t_x'\, x})\{\cdots\}} \cdots\}$

we have $C = C'$, $D = D'$, $\overline{t = t'}$, $\overline{m = m'}$, $\overline{\overline{t_x} = \overline{t_x'}}$,

The second condition on $R^{\mathtt{cls}}$ is implied by Definition 3.2. Assuming these conditions and because $(w_1, w_1') \in R_1^{\mathtt{val}}$ we get that $w_1' = obj\, C\, \{\cdots\}$, and $\mathcal{CT}'.C.m = \mathtt{public}\ t\, m(\overline{t_x\, x})\{e_3'\}$.

It remains to show that for all $(s, s', R^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R}$, all $(obj\, C\, \{\cdots\}, obj\, C\, \{\cdots\}) \in R^{\mathtt{val}}$, and all m with $\mathcal{CT}.C.m = \mathtt{public}\ t\, m(\overline{t_x\, x})\{e_3\}$ and $\mathcal{CT}'.C.m = \mathtt{public}\ t\, m(\overline{t_x\, x})\{e_3'\}$, and for all $\overline{(u, u')} \in V_{\overline{t_x}}[\mathcal{CT}, \mathcal{CT}', s, s', R^{\mathtt{val}}]$, s_1, w we have

$$
\begin{array}{l}
(\mathcal{CT} \vdash s, [\overline{u/x}]e_m \rightarrow^{<k} s_1, w) \\
\quad \Longrightarrow \exists\, s_1', w', R_1^{\mathtt{val}}. \\
\qquad (\mathcal{CT}' \vdash s', [\overline{u'/x}]e_m' \rightarrow^* s_1', w') \\
\qquad \wedge ((w, w') \in V_t[\mathcal{CT}, \mathcal{CT}', s_1, s_1', R_1^{\mathtt{val}}]) \\
\qquad \wedge ((s_1, s_1', R_1^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R}) \\
\qquad \wedge (R^{\mathtt{val}} \subseteq R_1^{\mathtt{val}})
\end{array}
$$

There are two cases:

- Method m is defined in the class-table context (possibly extending a class from $R^{\mathtt{cls}}$ and overriding method m of that class). In this case we have $e_3 = e_3'$ and we can apply immediately the induction hypothesis to get what is required.
- Method m is defined in class D, which is in $R^{\mathtt{cls}}$, $\mathcal{CT} \vdash C <: D$, and m is not overridden between C and D. To prove the conjecture we need that $\mathbb{R}$ will satisfy the following condition:

For all $(s, s', R^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R}$, all $(\mathcal{CT}, \mathcal{CT}') \in CT[R^{\mathtt{cls}}]$, all $(\mathcal{D}, \mathcal{D}') \in R^{\mathtt{cls}}$, with $\mathcal{D}$.classname $= D$, all $(obj\, C\, \{\cdots\}, obj\, C\, \{\cdots\}) \in V_D[\mathcal{CT}, \mathcal{CT}', s, s', R^{\mathtt{val}}]$, all m with $\mathcal{CT}.C.m$.defclass $= D$, and $\mathcal{CT}.C.m = \mathtt{public}\ t\, m(\overline{t_x\, x})\{e_3\}$, $\mathcal{CT}'.C.m = \mathtt{public}\ t\, m(\overline{t_x\, x})\{e_3'\}$, and for all $\overline{(u, u')} \in V_{\overline{t_x}}[\mathcal{CT}, \mathcal{CT}', s, s', R^{\mathtt{val}}]$, s_1, w, we have

$$
\begin{array}{l}
\quad IH(k) \\
\wedge \mathcal{CT} \vdash s, obj\, C\, \{\overline{f = l}\}.m(\overline{u}) \rightarrow^{<k+1} s_1, w \\
\quad \Longrightarrow \exists\, s_1', w', R_1^{\mathtt{val}}. \\
\qquad (\mathcal{CT}' \vdash s', obj\, C\, \{\overline{f' = l'}\}.m(\overline{u'}) \rightarrow^* s_1', w') \\
\qquad \wedge ((w, w') \in V_t[\mathcal{CT}, \mathcal{CT}', s_1, s_1', R_1^{\mathtt{val}}]) \\
\qquad \wedge ((s_1, s_1', R_1^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R}) \\
\qquad \wedge (R^{\mathtt{val}} \subseteq R_1^{\mathtt{val}})
\end{array}
$$

If $\mathbb{R}$ satisfies the above condition then we can show what is required for this case.

We continue similarly for the rest of the cases of e_0. □

In Theorem 5.4 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 5.2 (Inductive Cases).

$$R^{\mathtt{val}}, R^{\mathtt{cls}}, \mathbb{R} \vdash (\mathcal{CT} \vdash s, e{:}t) \sqsubseteq^{<k} (\mathcal{CT}' \vdash s', e'{:}t) \stackrel{def}{=}$$
$$\forall\, s_1, w.$$
$$((e, e') \in E_t[\mathcal{CT}, \mathcal{CT}', s, s', R^{\mathtt{val}}])$$
$$\wedge (\mathcal{CT} \vdash s, e \rightarrow^{<k} s_1, w)$$
$$\implies \exists\, s_1', w', R_1^{\mathtt{val}}.$$
$$(\mathcal{CT}' \vdash s', e' \rightarrow^{*} s_1', w')$$
$$\wedge ((w, w') \in V_t[\mathcal{CT}, \mathcal{CT}', s_1, s_1', R_1^{\mathtt{val}}])$$
$$\wedge ((s_1, s_1', R_1^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R})$$
$$\wedge (R^{\mathtt{val}} \subseteq R_1^{\mathtt{val}})$$

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

Definition 5.3 (Inductive Hypotheses).

$$IH^L_{\mathbb{R}}(k) \stackrel{def}{=}$$
$$\forall (s, s', R^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R}.$$
$$\forall (\mathcal{CT}, \mathcal{CT}') \in CT[R^{\mathtt{cls}}].$$
$$\forall\, t, (e, e') \in E_t[\mathcal{CT}, \mathcal{CT}', s, s', R^{\mathtt{val}}].$$
$$R^{\mathtt{val}}, R^{\mathtt{cls}}, \mathbb{R} \vdash (\mathcal{CT} \vdash s, e{:}t) \sqsubseteq^{<k} (\mathcal{CT}' \vdash s', e'{:}t)$$

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

Our main theorem is the following.

Theorem 5.4 (Adequacy Conditions). *A relation* $\mathbb{R}$ *is adequate if and only if for all states* $(s, s', R^{\mathtt{val}}, R^{\mathtt{cls}})$ *of* $\mathbb{R}$ *the following conditions are satisfied:*

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

$$\mathcal{C} = \mathtt{class}\ C\ \mathtt{extends}\ D\{\overline{\mathtt{public}\, t_1\, f_1}; \overline{\mathtt{private}\, t_2\, f_2};$$
$$\overline{C(\overline{t_3\, x_3})\{\ldots\}},$$
$$\overline{\mathtt{public}\ t_4\, m_4(\overline{t_5\, x_5})\{\cdots\}},$$
$$\overline{\mathtt{private}\ t_6\, m_6(\overline{t_7\, x_7})\{\ldots\}}\}$$

$$\mathcal{C}' = \mathtt{class}\ C'\ \mathtt{extends}\ D'\{\overline{\mathtt{public}\, t_1'\, f_1'}; \overline{\mathtt{private}\, t_2'\, f_2'};$$
$$\overline{C'(\overline{t_3'\, x_3'})\{\ldots\}},$$
$$\overline{\mathtt{public}\ t_4'\, m_4'(\overline{t_5'\, x_5'})\{\cdots\}},$$
$$\overline{\mathtt{private}\ t_6'\, m_6'(\overline{t_7'\, x_7'})\{\ldots\}}\}$$

   we have $C = C'$, $D = D'$, $\overline{t_1 = t_1'}$, $\overline{f_1 = f_1'}$, $\overline{t_3 = t_3'}$, $\overline{t_4 = t_4'}$, $\overline{m_4 = m_4'}$, $\overline{\overline{t_5} = \overline{t_5'}}$.
2. (Related instances.) *For all* $(obj\, C\, \{\cdots\}, obj\, C'\, \{\cdots\}) \in R^{\mathtt{val}}$*,* $C = C'$.
3. (Enough instances.) *For all* $(\mathcal{CT}, \mathcal{CT}') \in CT[R^{\mathtt{cls}}]$*, all* $\mathcal{C} \in \mathcal{CT}$*, with* $\mathcal{C}.\mathit{classname} = C$ *and* $\mathcal{CT}.C.\mathit{constr.type} = \overline{t} \rightarrow C$*, and for all* $\overline{(v, v')} \in V_{\overline{t}}[\mathcal{CT}, \mathcal{CT}', s, s', R^{\mathtt{val}}]$

$$IH^L_{\mathbb{R}}(k) \implies$$
$$R^{\mathtt{val}}, R^{\mathtt{cls}}, \mathbb{R} \vdash (\mathcal{CT} \vdash s, \mathtt{new}\, C(\overline{v}){:}C) \sqsubseteq^{<k+1}$$
$$(\mathcal{CT}' \vdash s', \mathtt{new}\, C(\overline{v'}){:}C)$$
$$IH^R_{\mathbb{R}}(k) \implies$$
$$R^{\mathtt{val}}, R^{\mathtt{cls}}, \mathbb{R} \vdash (\mathcal{CT} \vdash s, \mathtt{new}\, C(\overline{v}){:}C) \sqsupseteq^{<k+1}$$
$$(\mathcal{CT}' \vdash s', \mathtt{new}\, C(\overline{v'}){:}C)$$

4. (Related public fields.) *For all* $(\mathcal{CT}, \mathcal{CT}') \in CT[R^{\mathtt{cls}}]$*,* $(obj\, C\, \{\overline{f = l}\}, obj\, C\, \{\overline{f' = l'}\}) \in R^{\mathtt{val}}$*,* $\mathtt{public}\, t_i\, f_i \in \mathcal{CT}.C.\mathit{fields}$*, we have*

$$(s.l_i, s'.l_i') \in V_{t_i}[\mathcal{CT}, \mathcal{CT}', s, s', R^{\mathtt{val}}]$$

5. (Related updates.) *For all* $(\mathcal{CT}, \mathcal{CT}') \in CT[R^{\mathtt{cls}}]$*,* $(obj\, C\, \{\overline{f = l}\}, obj\, C\, \{\overline{f' = l'}\}) \in R^{\mathtt{val}}$*,* $\mathtt{public}\, t_i\, f_i \in \mathcal{CT}.C.\mathit{fields}$*, and* $(v, v') \in V_{t_i}[\mathcal{CT}, \mathcal{CT}', s, s', R^{\mathtt{val}}]$*, we have*

$$(s[l_i \leftarrow v], s'[l_i' \leftarrow v'], R^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R}$$

6. (Related public methods.) *For all* $(\mathcal{CT}, \mathcal{CT}') \in CT[R^{\mathtt{cls}}]$*,* $(\mathcal{D}, \mathcal{D}') \in R^{\mathtt{cls}}$*, with* $\mathcal{D}.\mathit{classname} = D$*, all* $(obj\, C\, \{\cdots\}, obj\, C\, \{\cdots\}) \in V_D[\mathcal{CT}, \mathcal{CT}', s, s', R^{\mathtt{val}}]$*, all* m *with* $\mathcal{CT}.C.m.\mathit{defclass} = D$*, and* $\mathtt{public}\, t_m\, m \in \mathcal{CT}.C.\mathit{methods}$*, with* $\mathcal{CT}.C.m = \mathtt{public}\, t\, m(\overline{t_x\, x})\{e_3\}$*,* $\mathcal{CT}'.C.m = \mathtt{public}\, t\, m(\overline{t_x\, x})\{e_3'\}$*, and for all* $\overline{(v, v')} \in V_{\overline{t_x}}[\mathcal{CT}, \mathcal{CT}', s, s', R^{\mathtt{val}}]$*, we have*

$$IH^L_{\mathbb{R}}(k) \implies$$
$$R^{\mathtt{val}}, R^{\mathtt{cls}}, \mathbb{R} \vdash (\mathcal{CT} \vdash s, obj\, C\, \{\overline{f = l}\}.m(\overline{v}){:}t_m) \sqsubseteq^{<k+1}$$
$$(\mathcal{CT}' \vdash s', obj\, C\, \{\overline{f' = l'}\}.m(\overline{v'}){:}t_m)$$
$$IH^R_{\mathbb{R}}(k) \implies$$
$$R^{\mathtt{val}}, R^{\mathtt{cls}}, \mathbb{R} \vdash (\mathcal{CT} \vdash s, obj\, C\, \{\overline{f = l}\}.m(\overline{v}){:}t_m) \sqsupseteq^{<k+1}$$
$$(\mathcal{CT}' \vdash s', obj\, C\, \{\overline{f' = l'}\}.m(\overline{v'}){:}t_m)$$

Proof. Immediate by the proof analysis of Conjecture 5.1. □

The first condition of Theorem 5.4 requires that related classes have the same public interface. The second condition requires that related objects instantiate related classes. The third condition requires that $\mathbb{R}$ relates all possible objects that can be created by the constructors. Conditions 4 and 5 test that related public fields can be assigned all the possible well-typed related values, and only those.

The last condition tests the behavior of related public methods. It considers only the methods that are defined in a class D taken from $R^{\mathtt{cls}}$, but the instances on which these methods are invoked are all the possible instances of subclasses of D.

Most of the above conditions contain a 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 examples that follow, 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.

6. Examples

6.1 Cells

Here we give two implementations of a Cell class. The first is the usual one, while the other is somewhat more complicated by using two, instead of one, 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.

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

$\mathcal{C}' =$ class Cell extends Object{
  private Object c1, c2;
  private int n;
  Cell(Object o){
    this.c1 := o;
    this.c2 := o;
    this.n := 0}
  public void set(Object o){
    this.c1 := o;
    this.c2 := o}
  public Object 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:

$$\begin{array}{l}\mathbb{R} = \{(s, s', R^{\mathtt{val}}, R^{\mathtt{cls}}) \mid \exists \mathcal{CT}, \overline{f_1, k_1, k_1', v_1, v_1', D},\\ \quad \overline{f_2, k_2, k_2', v_2, v_2', l_c, l_{c1}, l_{c2}, l_n, u, u', m, C} :\\ \qquad s = [\overline{k_1 = v_1}][\overline{k_2 = v_2}][\overline{l_c = u}]\\ \qquad s' = [\overline{k_1' = v_1'}][\overline{k_2' = v_2'}][\overline{l_{c1} = u', l_{c2} = u', l_n = m}]\\ \qquad R^{\mathtt{cls}} = \{\overline{(\mathcal{C}, \mathcal{C}')}\}\\ \qquad R^{\mathtt{val}} = \{\overline{(obj\, D\{\overline{f_1 = k_1}\}, obj\, D\{\overline{f_1 = k_1'}\})}\\ \qquad\quad \overline{(obj\, C\{\overline{f_2 = k_2}, c = l_c\},}\\ \qquad\qquad \overline{obj\, C\{\overline{f_2 = k_2'}, c1 = l_{c1}, c2 = l_{c2}, n = l_n\})}\\ \qquad \mathcal{CT} \vdash \overline{C <: \mathtt{Cell}}\\ \qquad \mathcal{CT} \vdash \overline{D \not<: \mathtt{Cell}}\\ \qquad \overline{(v_1, v_1')}, \overline{(v_2, v_2')}, \overline{(u, u')} \in R^{\mathtt{val}}\}\end{array}$$

We choose this particular $\mathbb{R}$ by inspecting the conditions of Theorem 5.4. Condition 1 is obviously satisfied by $\mathcal{C}$ and $\mathcal{C}'$. To satisfy conditions 2 and 3 we add in $R^{\mathtt{val}}$ the instances of any class C, sub-type of Cell, and any class D, not a sub-type of Cell. These are the classes of any possible class table. Furthermore we construct the stores s and s' in the appropriate way to keep the values of the fields of these instances. The values stored in $\overline{k_1}$ and $\overline{k_1'}$ are related in $R^{\mathtt{val}}$, since these are fields of identical classes in the class table; similarly for the values stored in $\overline{k_2}$ and $\overline{k_2'}$. From this we conclude that conditions 4 and 5 are satisfied.

We also require that the values stored in the private fields of Cell to be related in $R^{\mathtt{val}}$. 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 $(s, s', R^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R}$, 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 $o = obj\, C\{\overline{f_{2i} = k_{2i}}, c = l_{ci}\}$, $o' = obj\, C\{\overline{f_{2i} = k_{2i}'}, c1 = l_{c1i}, c2 = l_{c2i}, n = l_{ni}\}$, with $(o, o') \in V_{\mathtt{Cell}}[\mathcal{CT}, \mathcal{CT}', s, s', R^{\mathtt{val}}]$ we have that $\mathcal{CT} \vdash C <: \mathtt{Cell}$, $\mathcal{CT}' \vdash C <: \mathtt{Cell}$.

We now consider any public $t_x \to t\, m \in \mathcal{CT}.C.\mathtt{methods}$. These are the methods get and set.

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

$$\begin{array}{l}\mathcal{CT} \vdash s, o.\mathtt{get}() \to^* s, u_i\\ \mathcal{CT}' \vdash s', o'.\mathtt{get}() \to^* s'[l_{ni} \leftarrow m{+}1], u_i'\end{array}$$

where $s'.l_{ni} = m$. Moreover:

$$(u_i, u_i') \in V_{\mathtt{Object}}[\mathcal{CT}, \mathcal{CT}', s, s'[l_{ni} \leftarrow m{+}1], R^{\mathtt{val}}]$$

and $(s, s'[l_{ni} \leftarrow m{+}1], R^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R}$.

Similarly for the case of set we have $\overline{t_x} \to t = \mathtt{Object} \to \mathtt{void}$. Let $(u, u') \in V_{\mathtt{Object}}[\mathcal{CT}, \mathcal{CT}', s, s', R^{\mathtt{val}}]$. We have:

$$\begin{array}{l}\mathcal{CT} \vdash s, o.\mathtt{set}(u) \to^* s[l_{ci} \leftarrow u], \mathtt{unit}\\ \mathcal{CT}' \vdash s', o'.\mathtt{set}(u') \to^* s'[l_{c1\,i} \leftarrow u'][l_{c2i} \leftarrow u'], \mathtt{unit}\end{array}$$

and $(s[l_{ci} \leftarrow u], s'[l_{c1\,i} \leftarrow u'][l_{c2i} \leftarrow u'], R^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R}$.

6.2 The Observer Pattern

Here we will prove the equivalence of the two implementations of the Observer pattern, shown in Figure 1. For the purposes of this example, we assume that the language is extended in the usual way with vectors.

As before we construct a set $\mathbb{R}$ that will satisfy the adequacy conditions of Theorem 5.4. This set is shown in Figure 5, where $\mathcal{C}$ and $\mathcal{C}'$ are the two related definitions of the Observable class. As in the previous example, it is easy to see that $\mathbb{R}$ satisfies conditions 1-5 of Theorem 5.4.

It remains to show condition 6 for the public methods add and notifyAll. Consider an arbitrary tuple $(s, s', R^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R}$, and an arbitrary pair of related class tables $(\mathcal{CT}, \mathcal{CT}') \in CT[R^{\mathtt{cls}}]$. For any

$$\begin{array}{l}o = obj\, C\{\overline{f_{2i} = k_{2i}}, \mathtt{ovect}_1 = l_{1i}, \cdots, \mathtt{ovect}_{10} = l_{10i},\\ \qquad\qquad \mathtt{count} = l_{ci}\},\\ o' = obj\, C\{\overline{f_{2i} = k_{2i}}, \mathtt{o} = l_{1i}', \mathtt{next} = l_{nxt1i}', \mathtt{count} = l_{ci}'\}\end{array}$$

with $(o, o') \in V_{\mathtt{Observable}}[\mathcal{CT}, \mathcal{CT}', s, s', R^{\mathtt{val}}]$ we have that $\mathcal{CT} \vdash C <: \mathtt{Observable}$, $\mathcal{CT}' \vdash C <: \mathtt{Observable}$.

We need to show that the applications of $o.\mathtt{add}(u)$ and $o'.\mathtt{add}(u')$, where $(u, u') \in V_{\mathtt{Observer}}[\mathcal{CT}, \mathcal{CT}', s, s', R^{\mathtt{val}}]$, both terminate, and the final states are described in $\mathbb{R}$. We do this by two easy inductions on n_i, one for each implementation of add.

Next we need to show condition 6 for invocations of the applyAll method. We do this in two steps: first we show that the invoking applyAll on both sides causes the invocation of the notify method on the objects $u_{1i}, \ldots, u_{ni}$ on the left-hand side, and on the objects $u_{1i}', \ldots, u_{ni}'$ on the right-hand side. We do this by an induction on the first argument of the notify method for the left-hand side, and an induction on the size of the linked list for the right-hand side.

Then we need to show that each invocation of notify on the related objects will not break condition 6; i.e. for all $(s, s', R^{\mathtt{val}}, R^{\mathtt{cls}}) \in \mathbb{R}$, all $(\mathcal{CT}, \mathcal{CT}') \in CT[R^{\mathtt{cls}}]$, and all $(v, v') \in V_{\mathtt{Observer}}[\mathcal{CT}, \mathcal{CT}', s, s', R^{\mathtt{val}}]$, we have:

$$\begin{array}{l}IH^L_{\mathbb{R}}(k) \implies\\ \quad R^{\mathtt{val}}, R^{\mathtt{cls}}, \mathbb{R} \vdash (\mathcal{CT} \vdash s, u_{ji}.\mathtt{notify}(v){:}\mathtt{void}) \sqsubseteq^{<k}\\ \qquad\qquad (\mathcal{CT}' \vdash s', u_{ji}'.\mathtt{notify}(v'){:}\mathtt{void})\end{array}$$

and the reverse. Here we don't specify the concrete definition of the Observer class. Instead we rely on the quantification over all related well-typed class tables to consider all class definitions of Observable that have the right notify method. Furthermore, we know that the computation $u_{ji}.\mathtt{notify}(o)$ will terminate in k steps, since it is a sub-computation of the invocation $o_i.\mathtt{notifyAll}(o)$ which terminates in $k + 1$ steps. Therefore we can apply the induction hypothesis $IH^L_{\mathbb{R}}(k)$ and conclude that the above formula is true. This concludes the proof of equivalence of the two implementations of Observable.

7. Related Work

Banerjee and Naumann in [2] present a method for reasoning about the equivalence of programs in a subset of Java, similar to FWI Java. 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 they don't show that there is a simulation relation for any two equivalent and confined class-tables, but also because the technique seems helpful

$$
\begin{array}{l}
\mathbb{R} = \{(s, s', R^{\mathtt{val}}, R^{\mathtt{cls}}) \mid \exists \mathcal{CT}, \overline{f_1, k_1, k_1', v_1, v_1', D}, \overline{C, f_2, k_2, k_2', v_2, v_2', l_1, \cdots, l_{10}, u_1, \cdots, u_{10},} \\
\qquad\qquad \overline{l_c, l_c', n, l_1', \cdots, l_{n+1}', u_1', \cdots, u_{n+1}', l_{nxt1}', \cdots, l_{nxtn+1}', o_1', \cdots, o_{n+1}'} : \\
\quad s = [\overline{k_1 = v_1}][\overline{k_2 = v_2}][l_1 = u_1, \cdots, l_{10} = u_{10}, l_c = n] \\
\quad s = [k_1' = v_1'][\overline{k_2' = v_2'}][l_1' = u_1', \cdots, l_{n+1}' = u_{n+1}', l_{nxt1}' = o_1', \cdots, l_{nxtn+1}' = o_{n+1}', l_c' = n] \\
\quad R^{\mathtt{cls}} = \{(\mathcal{C}, \mathcal{C}')\} \\
\quad R^{\mathtt{val}} = \{\overline{(obj\, D\{f_1 = k_1\}, obj\, D\{f_1 = k_1'\})}, \\
\qquad\qquad \overline{(obj\, C\{\overline{f_2 = k_2}, \mathtt{ovect}_1 = l_1, \cdots, \mathtt{ovect}_{10} = l_{10}, \mathtt{count} = l_c\}, \\
\qquad\qquad\quad \overline{obj\, C\{\overline{f_2 = k_2}, \mathtt{o} = l_1', \mathtt{next} = l_{nxt1}', \mathtt{count} = l_c'\}})}\} \\
\quad \mathcal{CT} \vdash \overline{C <: \mathtt{Observable}} \\
\quad \mathcal{CT} \vdash \overline{D \not<: \mathtt{Observable}} \\
\quad \overline{o_j = obj\, \mathtt{Observable}\{o = l_{j+1}', next = l_{nxtj+1}', count = l_{cj}'\}, 1 \leq j \leq n} \\
\quad \overline{(v_1, v_1'), (v_2, v_2'), (u_1, u_1'), \cdots (u_n, u_n')} \in R^{\mathtt{val}} \\
\quad \overline{u_{n+1} = \ldots = u_{10} = \mathtt{null}} \\
\quad \overline{u_{n+1}' = o_{n+1} = \mathtt{null}}\}
\end{array}
$$

Figure 5. Set $\mathbb{R}$ for the Observer Pattern Equivalence

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. We were able to show contextually equivalent all of their examples, and others not expressible in their language, that make use of different private interfaces of the related classes (e.g. our Observer pattern example).

Another technique for reasoning about contextual equivalence in a class-based language is the one from Jeffrey and Rathke in [10]. 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 using an inductive proof as the one in Section 5. The difference with our work is that 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 FWI Java where the interaction of classes with their context is more complex, and equivalent classes communicate with the context through *related* messages and through the store.

8. Conclusions and Future Work

We have presented a sound an complete method for reasoning about contextual equivalence in a subset of Java. This method successfully deals with inheritance, public and private interfaces of classes, and imperative fields. We were able to use this method to prove equivalences in examples with different local store behavior, and higher-order features like the invocations of callbacks.

We have seen that our method can deal with the null-pointer and cast exceptions of Java. We would like to investigate further on this direction and see if our technique can prove contextual equivalence in a language with more advanced control effects (e.g. [5]).

In the future we would also like to see whether our method can benefit from ideas in closely related areas, like Separation Logic [18].

References

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

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

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

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

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

[6] 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.

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

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

[9] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[10] 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.

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

[12] 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.

[13] 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.

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

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

A. Appendix

Here we show the proof of Theorem 4.5.

Definition A.1. *$\mathcal{CT} \preccurlyeq \mathcal{CT}_1 \cup \mathcal{CT}_2$ is the ternary relation defined by the smallest congruence $\mathcal{CT} \preccurlyeq \mathcal{CT}_1$ containing*

$$(e, (\texttt{new}X())).m(\overline{x}))$$

and the smallest set $\mathcal{CT}_2$ containing the class definition

$$\mathcal{X} = \texttt{class}\ X\ \texttt{extends}\ \texttt{Object}\{\texttt{public}\ t\, m(\overline{t_x\, y})\{\overline{[y/x]}e\}\}$$

for all e, x, t, t_x, and for some X, y, m, such that:

$$\mathcal{CT}; \overline{x{:}t_x}; \emptyset \vdash e{:}t$$
$$\mathcal{X} \notin \mathcal{CT}$$
$$\mathcal{CT}{:}\mathsf{OK}$$

Lemma A.2. *If $\mathcal{CT} \preccurlyeq \mathcal{CT}_1 \cup \mathcal{CT}_2$, then $\mathcal{CT}_1 \cup \mathcal{CT}_2{:}\mathsf{OK}$, and for all stores s, expressions e, and types t such that $\mathcal{CT}; \emptyset; s \vdash e{:}t$ we have $\mathcal{CT}_1 \cup \mathcal{CT}_2; \emptyset; s \vdash e{:}t$.*

Proof. By induction on the structure of $\mathcal{CT}$, $\mathcal{CT}_1$, $\mathcal{CT}_2$. □

Lemma A.3. *If $\mathcal{CT} \preccurlyeq \mathcal{CT}_1 \cup \mathcal{CT}_2$, then for all stores s, expressions e, and types t, such that: $\mathcal{CT}; \emptyset; s \vdash e{:}t$, we have*

$$\mathcal{CT} \vdash s, e\downarrow \iff \mathcal{CT}_1 \cup \mathcal{CT}_2 \vdash s, e\downarrow$$

Proof. As in [20]. □

The proof of Theorem 4.5 follows:

Proof. The forward direction is done by inspection of the definitions of $(\equiv_e)$ and $(\equiv)$, and because $CT[\equiv] \subseteq CT[\equiv_e]$.

For the reverse direction let

$$R^{\texttt{cls}} = \{(\texttt{class}\ X\ \texttt{extends}\ \texttt{Object}\{\texttt{public}\ t\, m(\overline{t_x\, y})\{\overline{[y/x]}e\}\},\ \texttt{class}\ X\ \texttt{extends}\ \texttt{Object}\{\texttt{public}\ t\, m(\overline{t_x\, y})\{\overline{[y/x]}e'\}\})\}$$

and $R^{\texttt{cls}} \subseteq (\equiv)$.

Also let $R^{\texttt{exp}} = \{(\Gamma, e, e', t)\}$, where $\Gamma = \overline{x{:}t_x}$. Take any $(\mathcal{CT}, \mathcal{CT}') \in CT[R^{\texttt{exp}}]$, e_0, and t, such that X is not a class name in $\mathcal{CT}$, $\mathcal{CT}; \emptyset; \emptyset \vdash e_0{:}t$, and $\mathcal{CT} \vdash \emptyset, e_0\downarrow$.

Construct $\mathcal{CT}_1$, $\mathcal{CT}_2$, $\mathcal{CT}_1'$, and $\mathcal{CT}_2'$ such that:

$$\mathcal{CT} \preccurlyeq \mathcal{CT}_1 \cup \mathcal{CT}_2$$
$$\mathcal{CT}' \preccurlyeq \mathcal{CT}_1' \cup \mathcal{CT}_2'$$

$$\mathcal{CT}_2 = \{\texttt{class}\ X\ \texttt{extends}\ \texttt{Object}\{\texttt{public}\ t\, m(\overline{t_x\, y})\{\overline{[y/x]}e\}\}\}$$

$$\mathcal{CT}_2' = \{\texttt{class}\ X\ \texttt{extends}\ \texttt{Object}\{\texttt{public}\ t\, m(\overline{t_x\, y})\{\overline{[y/x]}e'\}\}\}$$

We have:

$$\begin{array}{lll} & \mathcal{CT} \vdash \emptyset, e_0\downarrow & \\ \Leftrightarrow & \mathcal{CT}_1 \cup \mathcal{CT}_2 \vdash \emptyset, e_0\downarrow & \text{by Lemma A.3} \\ \Leftrightarrow & \mathcal{CT}_1' \cup \mathcal{CT}_2' \vdash \emptyset, e_0\downarrow & (CT_1 \cup CT_2, CT_1' \cup CT_2') \in CT[\equiv] \\ \Leftrightarrow & \mathcal{CT}' \vdash \emptyset, e_0\downarrow & \text{by Lemma A.3} \end{array}$$

□

$\mathcal{C}$.classname	Returns the class name defined by the class definition $\mathcal{C}$.
$\mathcal{CT}.C$.super	Returns the name of the class that C extends.
$\mathcal{CT}.C$.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 6. Meta-Functions