A provenly correct translation of *Fickle* into Java

D. Ancona
DISI - Università di Genova
and
C. Anderson
Imperial College - London
and
F. Damiani
DI - Università di Torino
and
S. Drossopoulou
Imperial College - London
and
P. Giannini
DI - Università del Piemonte Orientale
and
E. Zucca
DISI - Università di Genova

We present a translation from *Fickle*, a small object-oriented language allowing objects to change their class at run-time, into Java. The translation is provenly correct, in the sense that it preserves the static and dynamic semantics. Moreover, it is compatible with separate compilation, since the translation of a *Fickle* class does not depend on the implementation of used classes. Based on the formal system, we have developed an implementation.

The translation turned out to be a more subtle problem than we expected. In this paper, we discuss four different possible approaches we considered for the design of the translation and justify our choice, we present formally the translation and the proof of preservation of the static and dynamic semantics, and we discuss the prototype implementation. Moreover, we outline an alternative translation based on generics that avoids most of the casts (but not all) needed in the previous translation.

The language *Fickle* has undergone, and is still undergoing several phases of development. In this paper we are discussing the translation of $Fickle_{\mathrm{II}}$.

Categories and Subject Descriptors: D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*classes and objects*; *inheritance*; *polymorphism*; D.3.4 [**Programming Languages**]: Processors—*preprocessor*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*object-oriented constructs*; *type structure*

General Terms: Theory, Languages

Additional Key Words and Phrases: Type and Effect Systems, Semantics Preserving Translation

---

Work partially supported by IST-2001-33477 DART and MIUR Cofin'04 EOS projects. The founding bodies are not responsible for any use that might be made of the results presented here.

1. INTRODUCTION

Dynamic object re-classification is a programming language feature that allows an object to change its class membership at run-time while retaining its identity. Thus, one can express fundamental change of an object's behavior (*e.g.*, non-empty lists becoming empty, iconified windows getting expanded, *etc.*) through *re-classification*, rather than through replacing objects of the old class by objects of the new class. Lack of re-classification primitives has long been recognized as a practical limitation of object-oriented programming.

$\mathcal{F}ickle_{\mathtt{II}}$ [Drossopoulou et al. 2001; 2002] is a small Java-like language that supports dynamic object re-classification, aiming to demonstrate how object re-classification could extend an imperative, typed, class-based, object-oriented language. Other approaches to the expression of fundamental change of behaviour have been suggested (some of them will be considered in the final section of the paper). $\mathcal{F}ickle_{\mathtt{II}}$ is type-safe, *i.e.*, any type correct program (in terms of the type system) is guaranteed never to attempt to access non-existing fields or methods.

In $\mathcal{F}ickle_{\mathtt{II}}$, there is a *re-classification* operation that changes the class membership of an object while preserving its identity. The biggest challenge is achieving a sound type system in the presence of re-classification. If some object has a field of type c, then soundness requires forbidding re-classification from a subclass of c to a class outside c's subhierarchy, as such re-classification could change the contents of the field to something that does not have its type. Classes for which this kind of re-classification does not happen are said to be "respected" by re-classification and can be safely used as types for fields. In $\mathcal{F}ickle_{\mathtt{II}}$ there is an incomparable set of *root classes*, and re-classification can only occur within a hierarchy rooted at a root class. Subclasses of root classes are called *state classes*. Classes that are neither root nor state classes are the only ones that are respected by re-classification.

Re-classification is traced by *effects*, and methods are annotated with the effects that may be caused by the execution of their body. Effects are sets of root classes, $\{c_1, \ldots, c_n\}$, meaning that there could be a re-classification between two subclasses of a class in the set.

We wanted to study the problem of simulating re-classification in a language without re-classification both theoretically and practically. We proceeded by developing, hand in hand, both a formalization and a prototype implementation of such a simulation. The translation turned out to be a more complex task than we had originally anticipated, and several subtle issues had to be considered. The formalization maps $\mathcal{F}ickle_{\mathtt{II}}$ into $\mathcal{F}ickle^{-}_{\mathtt{II}}$, the re-classification free fragment of $\mathcal{F}ickle_{\mathtt{II}}$. We decided to make the translation as simple as possible, neglecting efficiency in favor of uniformity (*i.e.*, fewer different cases) and simplicity. Our prototype implementation [Anderson 2003] maps $\mathcal{F}ickle^{\mathtt{st}}$, a statement oriented version of $\mathcal{F}ickle_{\mathtt{II}}$, onto Java. Besides the presence of re-classification, $\mathcal{F}ickle^{\mathtt{st}}$ is a subset of Java.

The translation of $\mathcal{F}ickle_{\mathtt{II}}$ into $\mathcal{F}ickle^{-}_{\mathtt{II}}$ is provenly correct: We present here the proofs that it preserves the static and dynamic semantics – *i.e.*, well-formed $\mathcal{F}ickle_{\mathtt{II}}$ programs are translated into well-formed $\mathcal{F}ickle^{-}_{\mathtt{II}}$ programs that behave "in the same way".

The development of the formal system highlighted design errors in our earlier attempts at the translation. For example, we were unable to prove the dynamic

correctness of the translation with respect to field assignment and method call. By inspecting where the proofs had failed, we had an insight into how to repair the translation.

Moreover, the translation is compatible with separate compilation, in the sense that the translation of a $\mathcal{F}ickle_{\mathrm{II}}$ class does not depend on the implementation of any classes *it* uses. Therefore, our translation could form the basis for an extension of a Java compiler; namely any type information needed by the translation can be retrieved from type information stored in binary files, as is done, indeed, by Java compilers for ordinary compilation.

The paper is organized as follows: In Section 2 we introduce $\mathcal{F}ickle_{\mathrm{II}}$ informally in terms of an example. In Section 3 we give a brief formal description of $\mathcal{F}ickle_{\mathrm{II}}$ (syntax, operational semantics, and typing) and state the properties of the type system. In Section 4 we discuss the various design alternatives considered for the translation. In Section 5 we give an informal overview of the translation, while in Section 6 we give the formal description. In Section 7 we state the properties of the translation (preservation of static and dynamic semantics) and illustrate the compatibility of the translation with Java separate compilation. In Section 8 we describe our current implementation of the translation [Anderson 2003]. In Section 9 we discuss how the translation could be enhanced in order to exploit new features of Java 1.5. We conclude by comparing our work with the one of others in Section 10 and then summarizing the relevance of this work and discussing further research directions in Section 11. Some technical definitions are illustrated in Appendix A. Proofs of the main results are given in Appendices B and C. In Appendix D we give the full translation (using generics and wildcards) of the example in used in Section 9.

2. $\mathcal{F}ICKLE_{\mathrm{II}}$, AN INFORMAL OVERVIEW

In this section we introduce $\mathcal{F}ickle_{\mathrm{II}}$ informally using an example. In the example we will use the types `int`, `float`, `void`, `String`, arithmetical expressions, and the `if` without `else` expression that are not present in the formalization of Section 3.1.

The $\mathcal{F}ickle_{\mathrm{II}}$ example in Fig. 1 describes accounts that belong to people, and which may be daily accounts, or savings accounts. It consists of a class `Person`, and a class `Account`, with subclasses `SavingsAccount` and `DailyAccount`. An `Account` belongs to a person (field `owner`), and holds some money (field `amount`). `Account`s implement the method `transact(int x)` that increments or decrements the field `amount` of `x`. If the amount exceeds a threshold (field `sup`), then the account turns into a `SavingsAccount`. Similarly, if the amount falls below a threshold, (field `inf`) then the account turns into a `DailyAccount`. Savings accounts pay interest.[1]

In $\mathcal{F}ickle_{\mathrm{II}}$ class definitions may be preceded by the keyword `state` or `root`. The *state classes* are the classes that may serve as targets of re-classification. Such classes *cannot* be used as types for fields; in our example `DailyAccount` and `SavingsAccount`. The *root classes* define the fields and methods common to their state subclasses; in our example, class `Account` defines the fields `amount` and `owner`,

[1]The example would be expressed more naturally using abstract classes, constructors, and exceptions, but these are not part of $\mathcal{F}ickle_{\mathrm{II}}$.

```
class Person {
  int age;
  String name
}

root class Account {
  int amount;
  Person owner;
  void transact(int x)   {Account} {}
  int interest() {} { 0; }
}

state class DailyAccount extends Account {
  int sup;
  void transact(int x) {Account} {
        amount = amount + x;
        if (amount > sup) {
           this!!SavingsAccount; this.interestRate = 10; this.inf = 200;
        }
   }
 }

state class SavingsAccount extends Account {
  float interestRate;
  int inf;
  void transact(int x) { Account } {
        amount = amount + x;
        if (amount < inf) {
           this!!DailyAccount; this.sup = 1000;
         }
  }
  int interest() { } { interestRate*amount; }
}
```

Fig. 1. Program `Account` - accounts with re-classifications

and the two methods `transact` and `interest`. The subclasses of root classes must be state classes.[2]

A *re-classification expression* has the form `id!!C`, and sets the class of `id` to `C`, where `C` must be a state class with the same root class of the static type of `id`. The re-classification operation preserves the types and the values of the fields defined in the root class, removes the other fields, and adds the fields of `C` that are not defined in the root class, initializing them in the usual way. Re-classifications may be caused by re-classification expressions, *e.g.*, `this!!SavingsAccount` in method `transact` in class `DailyAccount`, or, indirectly, by method calls, like `a.transact(...)`.[3] At the start of method `transact` of class `DailyAccount` the receiver is an object

[2]A root class is the first non-state superclass of a state class. The reason for introducing root classes as a separate kind of class is that in a system with separate compilation and without root classes, it would be impossible to enforce that if a class has a state subclass then all its further subclasses are state classes.

[3]In the example, only `this` is re-classified; note that $\mathcal{F}ickle_{\mathrm{II}}$ also allows re-classification of parameters and local variables.

of (a subclass of) `DailyAccount`, therefore it has the fields `amount`, `owner` and `sup`, while it does not have the fields `interestRate` and `inf`. After execution of `this!!SavingsAccount` the receiver is of class `SavingsAccount`, the fields `amount` and `owner` retain their values, the field `sup` disappears, and the fields `interestRate` and `inf` become available.

Annotations like `{}` and `{Account}` before method bodies are called *effects*. Effects list the root classes of all objects that may be re-classified by invocation of that method.

Methods with the empty effect `{}`, e.g., `interest`, may not cause any re-classification. Methods with non-empty effects, e.g., `transact`, with effect `{Account}`, may re-classify objects of a subclass of their effect; in our case of `Account`.

Consider the following fragment of code:

```
// a is of type DailyAccount and m is of type int
1.   a = new DailyAccount(); a.sup = 1000;
2.   m = a.interest();
3.   a.transact(1500);
4.   m = a.interest();
```

The call in line 2 selects the method `interest` from class `Account`, while the call in line 4 selects the method `interest` from class `SavingsAccount`, since now the object referred to by `a` is of class `SavingsAccount`.

Re-classification removes from the object all fields that are not defined in its root superclass and adds the remaining fields of the target class:

```
// a is of type DailyAccount and m is of type int
1.   a = new DailyAccount(); a.sup = 1000;
2.   m = a.sup;
3.   a.amount = 1500;
4.   a!!SavingsAcccount;
5.   m = a.inf;
6.   a.amount;
```

After line 1 the object denoted by `a` has the field `sup` and `amount` but not `inf` (or `interestRate`), whereas, after line 4 the same object has the field `inf` but not `sup`, and the field amount keeps its value (1500).

Re-classification is transparent to aliasing. For instance, in

```
// a1, a2 are of type DailyAccount
1.   a = new DailyAccount(); a.sup = 1000;
2.   a2 = a1;
3.   a1.transact(1500);
4.   a2.interest();
```

line 3 re-classifies the object, but does not affect the binding. Therefore, the call of method `interest` in line 4 selects the method from `SavingsAccount`. Thus, through aliasing, one re-classification may affect several variables; in the previous example it affects both `a1` and `a2`.

Because the class membership of objects of state class is transient, access to their members is only legal in contexts where it is certain that the object belongs to the particular class. This can be done for "local" entities, i.e., for parameters, the receiver `this`, and for local variables, but it cannot be done for fields, as their lifetime exceeds a method activation. Therefore, we do not allow state classes as the types of fields.

For example, the declaration of field `a` in the following is illegal:

```
class B {
   DailyAccount a;

   int m(){} { a.sup; }
}
```

Indeed, if the declaration of field `a` in class `B` were legal, then, in the following code

```
// b is of type B and acc is of type DailyAccount
1.   acc.transact(1500);
2.   b.m();
```

where `acc` is an alias of `b.a`, (e.g., through execution of `acc = new DailyAccount(); b = new B(); b.a = acc;`), the execution of line 1 would re-classify the object bound to `b.a` to `SavingsAccount`, and the field access `b.a.sup` inside the call of `b.m` in line 2 would raise a `fieldNotFound` error.

Therefore, state classes may not be used as types of fields. However, they may be used as types of `this`, parameters, local variables, or as return types for methods (in our type system we trace the type of `this`, parameters, and local variables).

Consider the following fragment of code that could be contained in a method of class `DailyAccount`.

```
// this is of type DailyAccount
this.sup;               // type correct
this.interestRate;      // type incorrect
this!!SavingsAccount;
this.sup;               // typeincorrect
this.interestRate;      // type correct
```

`this` is of type `DailyAccount` before re-classification and of type `SavingsAccount` after. Similarly for the type of parameters and local variables.
Changes to the type of `this` or a parameter or local variable may be caused either by explicit re-classifications, as before, or by potential, indirect re-classification, due to aliasing, as in the following method:

```
int n(Account a, DailyAccount da) {Account} {
1.  da.sup;              // type correct
2.  a.transact(1500);    // may re-classify a and all its aliases
3.  da.sup;              // type incorrect
4.  da.owner;            // type correct: da is certainly an Account
  }
```

The method call of line 2, if `a.amount` is bigger than 1000 (the value of `a.sup`), re-classifies the object referred to by `a`, whereas it does not if `a.amount` is less than 1000. Since at the time of the call `a` and `da` might be aliases, the possible re-classification of the object referred to by `a` might re-classify also the object referred to by `da`. Therefore, after the call, the only type-safe assumption is that the type of `da` is `Account`. In order to capture such potential re-classifications, each method declares as its effect the set of root classes of objects that may be re-classified through its execution. In our case, `transact` has effect `{Account}`. After the call `a.transact(1500)`, the type of `da` is `Account`, i.e., the application of the effect `{Account}` to the class `DailyAccount`.

A method annotated with effects can be overridden only by methods annotated with the same or fewer effects.[4] By relying on effects annotations, the type and effect system of $\mathcal{F}ickle_{\mathrm{II}}$ ensures that re-classifications will not cause accesses to fields or methods that are not defined for the object.

3. THE LANGUAGE $\mathcal{F}ICKLE_{\mathrm{II}}$, A FORMAL DESCRIPTION

The language $\mathcal{F}ickle_{\mathrm{II}}$, as considered in the present paper, is slightly different from the language introduced by Drossopoulou et al. [2002]. The differences are listed (and motivated) as follows.

—The language considered in this paper is richer: it includes type casts, a test for the `null` value and blocks (with local variables). These additional constructs are introduced by the translation. By adding them we made the target language a subset of the source language. This allowed us to simplify the presentation (*e.g.*, both the source and the target languages use the same type system and operational semantics).

—In the language considered by Drossopoulou et al. [2002] the dynamic semantics is unconventional with respect to the generation of null pointer exceptions (`nullPntrExc`).[5] If e evaluates to `null`, then $e.f = e_1$ and $e.m(e_1, \ldots, e_n)$ produce `nullPntrExc` without even evaluating e_1. This behaviour is not faithful to the semantics of Java that evaluates all the e_i's (which may, in their turn, raise exceptions). The operational semantics considered in this paper conforms to the Java semantics.

3.1 Syntax

The syntax of $\mathcal{F}ickle_{\mathrm{II}}$ is specified in Fig. 2. We use standard extended BNF, where [-] means optional, A^* means zero or more repetitions of A, and A^+ means one or more repetitions of A. We follow the convention that non terminals appear as

[4]This means that adding a new effect in a method of a class c does not require any change to the subclasses of c, but may require some changes to its superclasses, and the classes using them. Note also that effects are explicitly declared by the programmer rather than inferred by the compiler. Even though effects inference could be implemented in practice, more flexibility in method overriding can be achieved by requiring the programmer to annotate methods with more effects than those that would be inferred (similarly to what happens with `throws` clauses for exceptions).

[5]Thanks to an anonymous referee of a previous version of the present paper for pointing this out.

nonTerm and terminals appear as `term`. In the concrete syntax we use separator "`,`" and terminator "`;`" following Java style.

p	::=	$class^*$
$class$	::=	[`root` | `state`] `class` c `extends` c'`{`$field^*$ $meth^*$`}`
$field$	::=	t f
$meth$	::=	t m`(`par^*`)` ϕ $block$
t	::=	`bool` | c
par	::=	t x
ϕ	::=	`{`c^*`}`
$block$	::=	`{`var^* e^+`}`
var	::=	t x
e	::=	id | $sval$ | `isnull`(e) | $e.f$ | $(c)e$ | `new` c |
		id`!!`c | $x = e$ | $e.f = e_1$ | $e.m(e^*)$ |
		`if` e `then` e_1 `else` e_2 | $block$
id	::=	x | `this`
$sval$	::=	`true` | `false` | `null`

Fig. 2. $\mathcal{F}ickle_{\mathrm{II}}$ syntax

Metavariables c, f, m and x range over sets of *class names*, *field names*, *method names* and *variables*, respectively. We assume a distinguished class name `Object` that cannot be used as name of a declared class.

A program is a sequence of class definitions. A class definition may be preceded by the keyword `root` or `state`. As already explained, state classes describe the properties of an object while it satisfies some conditions, whereas root classes abstract over state classes.[6] Any subclass of a state or a root class must be a state class. Objects of a state class c may be re-classified to class c', where c' must be a subclass of the uniquely defined root superclass of c.

A class specifies its superclass and declares a sequence of fields and methods. The type of fields may be either a primitive type or a non-state class; we call such types *non-state types*. Thus, fields may point to objects that *change class*, but these changes do *not affect* their type. In contrast, the type of identifiers (`this`, parameters and local variables) may be a state or root class.

Method declarations have the shape:

$$t\ m\ \ (t_1\ x_1, \ldots, t_q\ x_q)\,\{c_1, \ldots, c_n\}\,block$$

where t is the result type, m the name, $t_1, \ldots, t_q$ are the types of the formal parameters $x_1, \ldots, x_q$, and *block* is the body. The effect consists of root classes c_1, $\ldots$, c_n, with $n \geq 0$.

A block consists of a possibly empty sequence of local variable declarations and a sequence of expressions. Expressions include identifiers (that is, `this`, parameters and local variables), source language values (that is, constants of primitive types

[6]Notice that our proposal is orthogonal to the "abstract superclass rule" discussed by Hürsch [1994]. In fact, root classes are not necessarily abstract classes, and state classes may be superclasses only of other state classes.

and `null`), test for the `null` value, field selection, casting, object creation, reclassification, assignment to a local variable or parameter, assignment to a field, method call, conditional and block.

In object creation `new` c, c may be *any* class, including a state class. Reclassification expressions, $id\,!!\,c$, set the class of id to c – c must be a state or a root class.

We require the inheritance hierarchy to be a tree, root classes to extend only non-root and non-state classes, and state classes to extend either root classes or state classes.

3.2 Operational semantics

We give a structural operational semantics that rewrites pairs of expressions and stores into pairs of either values or *exceptions*, and stores, in the context of a given program p.

The signature of the rewriting relation $\rightsquigarrow$ is:

$$\begin{array}{lcl}
\rightsquigarrow & : & p \longrightarrow e \times store \longrightarrow (\, val \cup exc \,) \times store \\
\\
store & = & (\, \{\texttt{this}\} \longrightarrow addr \,) \cup (\, x \longrightarrow val \,) \cup (\, addr \longrightarrow object \,) \\
val & = & sval \cup addr \\
exc & = & \{\texttt{nullPntrExc}, \texttt{castExc}\} \\
object & = & \{\ [\![f_1 : v_1, \ldots, f_r : v_r]\!]^{\,c} \mid f_1, \ldots, f_r \text{ are field identifiers,} \\
 & & \qquad v_1, \ldots, v_r \in val, \text{ and } c \text{ is a class name } \}
\end{array}$$

Values are the source language values in Section 3.1, or addresses. Addresses may point to objects, but *not* to other addresses, primitive values, or `null`. Thus in $\mathcal{F}ickle_{\mathrm{II}}$, as in Java, pointers are implicit, and there are no pointers to pointers.

Note that in the operational semantics considered by Drossopoulou et al. [2002] we had, in addition to null pointer exception and cast exception, also *stuck error*, which was meant to describe the kind of errors that a non well-typed expression could produce. In particular, access to undefined members of objects, undefined identifiers, etc. The rules for the evaluation of expressions that would produce such error were given. Such rules (and stuck error) are not needed for proof of Theorem 3.1 (type preservation), and are omitted in the present paper.

We denote stores with σ, and addresses with ι.

The store is a partial function with finite domain, which maps `this` to an address, variables to values, and addresses to objects. The store includes both

—the *stack* that maps `this`, local variables, and parameters to values, and

—the *heap* that maps addresses (unique object identifiers) to objects.

An alternative, more elegant solution, would have been to separate stack and heap explicitly. However, we chose to use a semantics which is, with minor differences (no stuck error, additional clauses for blocks and casts, different treatment of null pointer exceptions), that used by Drossopoulou et al. [2002], in order to get from there the type preservation result (Theorem 3.1), which is needed in the proof of adequacy of the translation (Theorem 7.5). This also implies that we chose a big-step semantics. Note that giving a small step operational semantics for the language

requires to extend the language to include the intermediate expressions resulting from the evaluation. Moreover, typing rules must be defined for the extended language. This was done by Damiani et al. [2004], that needed a small step semantics in order to model multi-threading.

To define the operational semantics we need some operations on objects and stores.
For object $o = [[f_1 : v_1, \ldots, f_l : v_l, \ldots, f_r : v_r]]^c$, store σ, value v , address ι, identifier or address z, field identifier f, value or object w, we define:

—*field access* $\quad o(f) = \begin{cases} v_l & \text{if } f = f_l \text{ for some } l \in 1, \ldots, r, \\ \mathcal{U}df & \text{otherwise} \end{cases}$

—*object update* $\quad o[f \mapsto v] = \begin{cases} [[f_1 : v_1, \ldots, f_l : v, \ldots, f_r : v_r]]^c \\ \text{if } f = f_l \text{ for some } l \in 1, \ldots, r, \\ \mathcal{U}df \text{ otherwise} \end{cases}$

—*store update* $\quad \sigma[z \mapsto w](z) = w, \quad \sigma[z \mapsto w](z') = \sigma(z') \text{ if } z' \neq z.$

Also, we follow the convention that $\sigma(\iota)(f) = \mathcal{U}df$ whenever $\sigma(\iota) = \mathcal{U}df$.

Figures 3, 4, 5, and 6 list all the rewrite rules of $\mathcal{F}ickle_{\mathrm{II}}$. We discuss the two most significant rewrite rules of $\mathcal{F}ickle_{\mathrm{II}}$: method call and re-classification.

For method calls, $e.m(e_1, \ldots, e_n)$, we evaluate the receiver e, obtaining an address, say ι. We then evaluate the arguments, $e_1, \ldots, e_n$. We find the appropriate body by looking up m in the class of the object at address ι – we use the term $\mathcal{M}(p, c, m)$ that returns the definition of method m in class c going through the class hierarchy in p, if needed (see Appendix A). We execute the body after substituting `this` with the current object, and assigning to the formal parameters the values of the actual parameters. After the call, we restore the receiver and parameters to the values they had immediately before execution of the body.[7]

For re-classification expressions, $id \,!!\, d$, we take the value of id, which must be the address of an object of some class c. We replace the original object by a new object of class d. We preserve the fields belonging to the root superclass of c and initialize the other fields of d according to their types. The term $\mathcal{R}(p, t)$, defined by

$$\mathcal{R}(p, t) = \begin{cases} c & \text{if } t \text{ is a state class and } c \text{ is the root superclass of } t \\ t & \text{otherwise,} \end{cases}$$

denotes the least superclass of t that is not a state class, if t is a class, and denotes t itself if t is not a class or a non-state class. Moreover, $\mathcal{F}s(p, c)$ denotes the set of all fields (either directly defined or inherited) of class c, and $\mathcal{F}(p, c, f)$ the type of field f in class c (see Appendix A). Note that we do not allow hiding of fields in $\mathcal{F}ickle_{\mathrm{II}}$.[8]

[7] We restore the references, but not the contents: thus, after a method call the side effects caused by execution of the method body survive. Note also that if one of the method parameters was undefined before the call, then it will be undefined after the call as well.

[8] In well-typed programs, $\mathcal{R}(p, c) = \mathcal{R}(p, d)$ always holds, and c and d must be state or root classes. This implies that re-classification depends only on the target class d, not on the class c of the receiver. Therefore, a compiler could fold the type information into the code, by generating specific re-classification code for each state class. The rule for re-classification uses the types of the fields to initialize the fields, similarly to the rule for object creation.

$$\frac{}{v, \sigma \underset{p}{\rightsquigarrow} v, \sigma} \quad (val)$$

$$\frac{e, \sigma \underset{p}{\rightsquigarrow} \iota, \sigma' \quad \sigma'(\iota) = [\![\ldots]\!]^{c'} \quad p \vdash c' \leq c}{(c)e, \sigma \underset{p}{\rightsquigarrow} \iota, \sigma'} \quad (cast)$$

$$\frac{e, \sigma \underset{p}{\rightsquigarrow} \texttt{null}, \sigma'}{(c)e, \sigma \underset{p}{\rightsquigarrow} \texttt{null}, \sigma'} \quad (n\text{-}cast)$$

$$\frac{\sigma(id) \neq \mathcal{U}df}{id, \sigma \underset{p}{\rightsquigarrow} \sigma(id), \sigma} \quad (id)$$

$$\frac{e, \sigma \underset{p}{\rightsquigarrow} \iota, \sigma' \quad \sigma'(\iota)(f) \neq \mathcal{U}df}{e.f, \sigma \underset{p}{\rightsquigarrow} \sigma'(\iota)(f), \sigma'} \quad (field)$$

$$\frac{e, \sigma \underset{p}{\rightsquigarrow} \texttt{null}, \sigma'}{\texttt{isnull}(e), \sigma \underset{p}{\rightsquigarrow} \texttt{true}, \sigma'} \quad (t\text{-}isnull)$$

$$\frac{e, \sigma \underset{p}{\rightsquigarrow} \iota, \sigma' \quad \sigma'(\iota) \neq \mathcal{U}df}{\texttt{isnull}(e), \sigma \underset{p}{\rightsquigarrow} \texttt{false}, \sigma'} \quad (f\text{-}isnull)$$

$$\frac{e, \sigma \underset{p}{\rightsquigarrow} v, \sigma'}{x = e, \sigma \underset{p}{\rightsquigarrow} v, \sigma'[x \mapsto v]} \quad (a\text{-}var)$$

$$\frac{e, \sigma \underset{p}{\rightsquigarrow} \iota, \sigma'' \quad e_1, \sigma'' \underset{p}{\rightsquigarrow} v, \sigma''' \quad \sigma'''(\iota)(f) \neq \mathcal{U}df \quad \sigma' = \sigma'''[\iota \mapsto \sigma'''(\iota)[f \mapsto v]]}{e.f = e_1, \sigma \underset{p}{\rightsquigarrow} v, \sigma'} \quad (a\text{-}field)$$

$$\frac{e, \sigma \underset{p}{\rightsquigarrow} \texttt{true}, \sigma'' \quad e_1, \sigma'' \underset{p}{\rightsquigarrow} v, \sigma'}{\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2, \sigma \underset{p}{\rightsquigarrow} v, \sigma'} \quad (t\text{-}cond)$$

$$\frac{e, \sigma \underset{p}{\rightsquigarrow} \texttt{false}, \sigma'' \quad e_2, \sigma'' \underset{p}{\rightsquigarrow} v, \sigma'}{\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2, \sigma \underset{p}{\rightsquigarrow} v, \sigma'} \quad (f\text{-}cond)$$

$$\frac{v_l \text{ initial for } t_l \ \ (l \in \{1, \ldots, s\}) \quad \sigma_0 = \sigma[x_1 \mapsto v_1, \ldots, x_s \mapsto v_s] \quad e_i, \sigma_{i-1} \underset{p}{\rightsquigarrow} v_i, \sigma_i \ \ (i \in \{1, \ldots, n\})}{\{t_1\, x_1\texttt{;} \ \ldots t_s\, x_s\texttt{;}\ e_1\texttt{;} \ \ldots e_n\texttt{;}\ \}, \sigma \underset{p}{\rightsquigarrow} v_n, \sigma_n[x_1 \mapsto \sigma(x_1), \ldots, x_s \mapsto \sigma(x_s)]} \quad (block)$$

$$\frac{\mathcal{F}s(p, c) = \{f_1, \ldots, f_r\} \quad v_l \text{ initial for } \mathcal{F}(p,c,\, f_l) \quad (l \in \{1, \ldots, r\}) \quad \iota \text{ is new in } \sigma}{\texttt{new } c, \sigma \underset{p}{\rightsquigarrow} \iota, \sigma[\iota \mapsto [\![f_1 : v_1, \ldots, f_r : v_r]\!]^c]} \quad (new)$$

$$\frac{e, \sigma \underset{p}{\rightsquigarrow} \iota, \sigma_0 \quad e_i, \sigma_{i-1} \underset{p}{\rightsquigarrow} v_i, \sigma_i \ \ (i \in \{1, \ldots, n\}) \quad \sigma_n(\iota) = [\![\ldots]\!]^c \quad \mathcal{M}(p, c, m) = t\ m(t_1\ x_1, \ldots, t_n\ x_n)\ \phi\ block \quad \sigma' = \sigma_n[\texttt{this} \mapsto \iota, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \quad block, \sigma' \underset{p}{\rightsquigarrow} v, \sigma''}{e.m(e_1, \ldots, e_n), \sigma \underset{p}{\rightsquigarrow} v, \sigma''[\texttt{this} \mapsto \sigma_n(\texttt{this}), x_1 \mapsto \sigma_n(x_1), \ldots, x_n \mapsto \sigma_n(x_n)]} \quad (meth)$$

Fig. 3. $\mathcal{F}ickle_{\mathrm{II}}$ expression evaluation – without generation and propagation of exceptions

$$\frac{\begin{array}{l}\sigma(id) = \iota \\ \sigma(\iota) = [\![\ldots]\!]^{c} \\ \mathcal{F}s(p, \mathcal{R}(p, c)) = \{f_1, \ldots, f_r\} \\ v_l = \sigma(\iota)(f_l) \quad (l \in \{1, \ldots, r\}) \\ \mathcal{F}s(p, d) \setminus \{f_1, \ldots, f_r\} = \{f_{r+1}, \ldots, f_{r+q}\} \\ v_l \text{ initial for } \mathcal{F}(p,d,f_l) \quad (l \in \{r+1, \ldots, r+q\})\end{array}}{id \,!!\, d, \sigma \rightsquigarrow_p \iota, \sigma[\iota \mapsto [\![f_1 : v_1, \ldots, f_{r+q} : v_{r+q}]\!]^{d}]} \quad (recl)$$

$$\frac{\sigma(id) = \texttt{null}}{id \,!!\, d, \sigma \rightsquigarrow_p \texttt{null}, \sigma} \quad (n\text{-}recl)$$

Fig. 4. $\mathcal{F}ickle_{\mathrm{II}}$ expression evaluation – without generation and propagation of exceptions

$$\frac{\begin{array}{l}e, \sigma \rightsquigarrow_p \iota, \sigma' \\ \sigma'(\iota) = [\![\ldots]\!]^{c'} \\ p \not\vdash c' \leq c\end{array}}{(c)e, \sigma \rightsquigarrow_p \texttt{castExc}, \sigma'} \quad (e\text{-}cast)$$

$$\frac{e, \sigma \rightsquigarrow_p \texttt{null}, \sigma'}{e.f, \sigma \rightsquigarrow_p \texttt{nullPntrExc}, \sigma'} \quad (field\text{-}null)$$

$$\frac{\begin{array}{l}e, \sigma \rightsquigarrow_p \texttt{null}, \sigma'' \\ e_1, \sigma'' \rightsquigarrow_p v, \sigma'\end{array}}{e.f = e_1, \sigma \rightsquigarrow_p \texttt{nullPntrExc}, \sigma'} \quad (a\text{-}field\text{-}null)$$

$$\frac{\begin{array}{l}e, \sigma \rightsquigarrow_p \texttt{null}, \sigma_0 \\ e_i, \sigma_{i-1} \rightsquigarrow_p v_i, \sigma_i \quad (i \in \{1, \ldots, n\})\end{array}}{e.m(e_1, \ldots, e_n), \sigma \rightsquigarrow_p \texttt{nullPntrExc}, \sigma_n} \quad (meth\text{-}null)$$

Fig. 5. $\mathcal{F}ickle_{\mathrm{II}}$ expression evaluation – generation of exceptions

3.3 Typing

3.3.1 *Widening, environments, effects.* The following assertions, defined in Fig. 20 of Appendix A, describe kinds of classes, and the widening relationship between types:

—$p \vdash c \diamond_{ct}$ means that c is any class,

—$p \vdash c \diamond_{rt}$ means that c is a re-classifiable type, *i.e.*, either a root or a state class,

—$p \vdash t \diamond_{ft}$ means that t is a non-state type, *i.e.*, `bool` or a non-`state` class, and

—$p \vdash t \leq t'$ means that type t' widens type t, *i.e.*, t is a subclass of, or identical to, t'.

Environments, γ, map parameter names and local variables to types, and the receiver `this` to a class. They have the form $x_1 : t_1, \ldots, x_n : t_n, \texttt{this} : c$. Lookup, $\gamma(id)$, and update, $\gamma[id \mapsto t]$, have the usual meaning, and are defined in Fig. 21 of Appendix A.

An effect, ϕ, is a set $\{\, c_1, \ldots, c_n \,\}$ of root classes; it means that any object of a subclass of c_i (including c_i itself) may be re-classified. The empty effect, $\{\,\}$, guarantees that no object is re-classified. Effects are well-formed, *i.e.*, $p \vdash \{\, c_1, \ldots, c_n \,\} \diamond$, iff $c_1, \ldots, c_n$ are distinct root classes. Thus, $p \vdash \{\, c_1, \ldots, c_n \,\} \diamond$ implies that c_i are not subclasses of each other.

3.3.2 *Typing rules.* The typing rules are given in Fig. 7. We use the look-up functions $\mathcal{F}$ and $\mathcal{M}$, which search for fields and methods through the class hierarchy (see Appendix A).

$$\frac{e, \sigma \overset{}{\underset{p}{\rightsquigarrow}} \mathit{exc}, \sigma'}{\begin{array}{l}(c)e, \sigma \overset{}{\underset{p}{\rightsquigarrow}} \mathit{exc}, \sigma' \\ x = e, \sigma \overset{}{\underset{p}{\rightsquigarrow}} \mathit{exc}, \sigma' \\ e.f, \sigma \overset{}{\underset{p}{\rightsquigarrow}} \mathit{exc}, \sigma' \\ \texttt{isnull}(e), \sigma \overset{}{\underset{p}{\rightsquigarrow}} \mathit{exc}, \sigma' \\ e.m(e_1, \ldots, e_n), \sigma \overset{}{\underset{p}{\rightsquigarrow}} \mathit{exc}, \sigma' \\ e.f = e_1, \sigma \overset{}{\underset{p}{\rightsquigarrow}} \mathit{exc}, \sigma'\end{array}} \qquad \frac{\begin{array}{l}e, \sigma \overset{}{\underset{p}{\rightsquigarrow}} \iota, \sigma'' \\ \sigma''(\iota) \neq \mathit{Udf} \\ e_1, \sigma'' \overset{}{\underset{p}{\rightsquigarrow}} \mathit{exc}, \sigma'\end{array}}{e.f{=}e_1, \sigma \overset{}{\underset{p}{\rightsquigarrow}} \mathit{exc}, \sigma'}$$

$$\frac{\begin{array}{l}e, \sigma \overset{}{\underset{p}{\rightsquigarrow}} \iota, \sigma_0 \\ \sigma_0(\iota) \neq \mathit{Udf} \\ e_i, \sigma_{i-1} \overset{}{\underset{p}{\rightsquigarrow}} v_i, \sigma_i \ \ (i \in \{1, \ldots, q\},\ q < n) \\ e_{q+1}, \sigma_q \overset{}{\underset{p}{\rightsquigarrow}} \mathit{exc}, \sigma_{q+1}\end{array}}{e.m(e_1, \ldots, e_n), \sigma \overset{}{\underset{p}{\rightsquigarrow}} \mathit{exc}, \sigma_{q+1}}$$

$$\frac{\begin{array}{l}e, \sigma \overset{}{\underset{p}{\rightsquigarrow}} \iota, \sigma_0 \\ e_i, \sigma_{i-1} \overset{}{\underset{p}{\rightsquigarrow}} v_i, \sigma_i \ \ (i \in \{1, \ldots, n\}) \\ \sigma_n(\iota) = [\![\ldots]\!]^c \\ \mathcal{M}(p, c, m) = t\ m(t_1\ x_1, \ldots, t_n\ x_n)\ \phi\ \mathit{block} \\ \sigma' = \sigma_n[\texttt{this} \mapsto \iota, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \\ \mathit{block}, \sigma' \overset{}{\underset{p}{\rightsquigarrow}} \mathit{exc}, \sigma''\end{array}}{e.m(e_1, \ldots, e_n), \sigma \overset{}{\underset{p}{\rightsquigarrow}} \mathit{exc}, \sigma''[\texttt{this} \mapsto \sigma_n(\texttt{this}), x_1 \mapsto \sigma_n(x_1), \ldots, x_n \mapsto \sigma_n(x_n)]}$$

$$\frac{\begin{array}{l}e, \sigma \overset{}{\underset{p}{\rightsquigarrow}} \mathit{exc}, \sigma' \\ \quad \text{or} \quad (e, \sigma \overset{}{\underset{p}{\rightsquigarrow}} \texttt{true}, \sigma'' \ \text{ and } \ e_1, \sigma'' \overset{}{\underset{p}{\rightsquigarrow}} \mathit{exc}, \sigma') \\ \quad \text{or} \quad (e, \sigma \overset{}{\underset{p}{\rightsquigarrow}} \texttt{false}, \sigma'' \ \text{ and } \ e_2, \sigma'' \overset{}{\underset{p}{\rightsquigarrow}} \mathit{exc}, \sigma')\end{array}}{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2, \sigma \overset{}{\underset{p}{\rightsquigarrow}} \mathit{exc}, \sigma'}$$

$$\frac{\begin{array}{l}v_l \text{ initial for } t_l \ \ (l \in \{1, \ldots, s\}) \\ \sigma_0 = \sigma[x_1 \mapsto v_1, \ldots, x_s \mapsto v_s] \\ e_i, \sigma_{i-1} \overset{}{\underset{p}{\rightsquigarrow}} v_i, \sigma_i \ \ (i \in \{1, \ldots, q\},\ q < n) \\ e_{q+1}, \sigma_q \overset{}{\underset{p}{\rightsquigarrow}} \mathit{exc}, \sigma_{q+1}\end{array}}{\{t_1\, x_1\texttt{;}\ \ldots t_s\, x_s\texttt{;}\ e_1\texttt{;}\ \ldots e_n\texttt{;}\ \}, \sigma \overset{}{\underset{p}{\rightsquigarrow}} \mathit{exc}, \sigma_{q+1}[x_1 \mapsto \sigma(x_1), \ldots, x_s \mapsto \sigma(x_s)]}$$

Fig. 6. *Fickle*$_{\text{II}}$ expression evaluation – propagation of exceptions

In the rules, $t \sqcup_p t'$ is the least upper bound of t and t' w.r.t. the $\leq$ relation between types in p, and $\gamma \sqcup_p \gamma'$ associates with an identifier the least upper bound in p of its types in γ and γ'. (The formal definitions can be found in Fig. 21 of Appendix A.) Moreover, we define the application of effects to types:

$$\{\, c_1, \ldots, c_n \,\}@_p t \;=\; \begin{cases} c_i & \text{if } \mathcal{R}(p,t) = c_i \text{ for some } i \in 1, \ldots, n \\ t & \text{otherwise} \end{cases}$$

$$\frac{}{p,\gamma \vdash \texttt{true} : \texttt{bool} \parallel \gamma \parallel \{\,\}} \qquad \frac{p \vdash c \diamond_{ct}}{p,\gamma \vdash \texttt{null} : c \parallel \gamma \parallel \{\,\}}$$

$$p,\gamma \vdash \texttt{false} : \texttt{bool} \parallel \gamma \parallel \{\,\}$$

$$p,\gamma \vdash id : \gamma(id) \parallel \gamma \parallel \{\,\}$$

$$p,\gamma \vdash \texttt{new}\ c : c \parallel \gamma \parallel \{\,\}$$

$$\frac{p,\gamma \vdash e : c' \parallel \gamma' \parallel \phi \quad (p \vdash c' \leq c \text{ or } p \vdash c \leq c')}{p,\gamma \vdash (c)e : c \parallel \gamma' \parallel \phi} \qquad \frac{p,\gamma \vdash e : c \parallel \gamma' \parallel \phi \quad \mathcal{F}(p,c,f) = t}{p,\gamma \vdash e.f : t \parallel \gamma' \parallel \phi}$$

$$\frac{p,\gamma \vdash e : c \parallel \gamma_1 \parallel \phi_1 \quad p,\gamma_1 \vdash e_1 : t \parallel \gamma_2 \parallel \phi_2 \quad \mathcal{F}(p,\phi_2@_p c,f) = t' \quad p \vdash t \leq t'}{p,\gamma \vdash e.f = e_1 : t \parallel \gamma_2 \parallel \phi_1 \cup \phi_2} \qquad \frac{p,\gamma \vdash e : t \parallel \gamma' \parallel \phi \quad \gamma'(x) = t' \quad p \vdash t \leq t'}{p,\gamma \vdash x = e : t \parallel \gamma' \parallel \phi}$$

$$\frac{p,\gamma \vdash e : c \parallel \gamma' \parallel \phi}{p,\gamma \vdash \texttt{isnull}(e) : \texttt{bool} \parallel \gamma' \parallel \phi}$$

$$\frac{\begin{array}{l} p,\gamma \vdash e : c \parallel \gamma_0 \parallel \phi_0 \\ p,\gamma_{i-1} \vdash e_i : t_i \parallel \gamma_i \parallel \phi_i \ \ (i \in \{1,\ldots,n\}) \\ \mathcal{M}(p,(\phi_1 \cup \cdots \cup \phi_n)@_p c,m) = t\ m(t'_1\ x_1,\ldots,t'_n\ x_n)\ \phi\ \{\ \ldots\ \} \\ p \vdash (\phi_{i+1} \cup \cdots \cup \phi_n)@_p t_i \leq t'_i \ \ (i \in \{1,\ldots,n\}) \end{array}}{p,\gamma \vdash e.m(e_1,\ldots,e_n) : t \parallel \phi@_p\gamma_n \parallel \phi \cup \phi_0 \cup \cdots \cup \phi_n}$$

$$\frac{p,\gamma \vdash e : \texttt{bool} \parallel \gamma_0 \parallel \phi_0 \quad p,\gamma_0 \vdash e_1 : t_1 \parallel \gamma_1 \parallel \phi_1 \quad p,\gamma_0 \vdash e_2 : t_2 \parallel \gamma_2 \parallel \phi_2}{p,\gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t_1 \sqcup_p t_2 \parallel \gamma_1 \sqcup_p \gamma_2 \parallel \phi_0 \cup \phi_1 \cup \phi_2}$$

$$\frac{\begin{array}{l} \gamma_0 = \gamma[x_1 \mapsto t_1,\ldots,x_s \mapsto t_s] \\ p,\gamma_{i-1} \vdash e_i : t'_i \parallel \gamma_i \parallel \phi_i \ \ (i \in \{1,\ldots,n\}) \end{array}}{p,\gamma \vdash \{t_1\,x_1;\ \ldots t_s\,x_s;\ e_1;\ \ldots e_n;\ \} : t'_n \parallel \gamma_n[x_1 \mapsto \gamma(x_1),\ldots,x_s \mapsto \gamma(x_s)] \parallel \phi_1 \cup \cdots \cup \phi_n}$$

$$\frac{p \vdash c \diamond_{rt} \quad \mathcal{R}(p,c) = \mathcal{R}(p,\gamma(id))}{p,\gamma \vdash id\,!!\,c : c \parallel (\{\,\mathcal{R}(p,c)\,\}@_p\gamma)[id \mapsto c] \parallel \{\,\mathcal{R}(p,c)\,\}}$$

Fig. 7. $\mathcal{F}ickle_{\mathrm{II}}$ – typing rules for expressions

Note that, if

$$p,\gamma \vdash e : t \parallel \gamma' \parallel \phi$$

is a derivable judgment, then the environments γ, and γ' are defined for the same set of identifiers, and any differences in the types associated with an identifier in γ and γ' are due to the effect ϕ. So, if e is a $\mathcal{F}ickle^{-}_{\mathrm{II}}$ expression, then $\gamma = \gamma'$ and $\phi = \{\}$.

3.3.3 *Well-formed Programs.* A program is well formed (written $\vdash p\ \diamond$) if the inheritance hierarchy is well-formed ($\vdash\ p\ \diamond_h$) and all its classes are well-formed ($p \vdash c\ \diamond$): Methods may override superclass methods only if they have the same name, argument, and result type, and their effect is a subset of that of the overridden method. Method bodies must be well formed, return a value appropriate for the method signature, and their effect must be a subset of that in the signature. See Fig. 8, where $\mathcal{C}(p, c)$ returns the definition of class c in program p, and the look-up functions $\mathcal{FD}(p, c, f)$, $\mathcal{MD}(p, c, m)$ search for fields and methods only in class c (see Appendix A).

$$\frac{\begin{array}{l}\mathcal{C}(p,c) = [\texttt{root} \mid \texttt{state}]\ \texttt{class}\ c\ \texttt{extends}\ c'\ \texttt{\{}\ldots\texttt{\}} \\ \forall f:\ \mathcal{FD}(p,c,f) = t_f \implies p \vdash t_f\ \diamond_{ft} \quad \text{and} \quad \mathcal{F}(p,c',f) = \mathcal{U}df \\ \forall m:\ \mathcal{MD}(p,c,m) = t\ m\texttt{(}t_1\ x_1, \ldots, t_n\ x_n\texttt{)}\ \phi\ block \implies \\ \quad p \vdash \phi\ \diamond \\ \quad p, t_1\ x_1, \ldots, t_n\ x_n,\ c\ \texttt{this} \vdash block\ :\ t' \parallel \gamma' \parallel \phi' \\ \quad p \vdash t' \leq t \\ \quad \phi' \subseteq \phi \\ \quad \mathcal{M}(p,c',m) = \mathcal{U}df \text{ or } (\mathcal{M}(p,c',m) = t\ m\texttt{(}t_1\ x_1, \ldots, t_n\ x_n\texttt{)}\ \phi''\ \texttt{\{}\ \ldots\ \texttt{\}} \text{ and } \phi \subseteq \phi'')\end{array}}{p \vdash c\ \diamond}$$

$$\frac{\vdash\ p\ \diamond_h \qquad \forall c:\ \mathcal{C}(p,c) \neq \mathcal{U}df \implies p \vdash c\ \diamond}{\vdash p\ \diamond}$$

Fig. 8. $\mathcal{F}ickle_{\mathrm{II}}$ – rules for well-formed classes and programs

3.3.4 *Type Preservation.* The main property of the type system is type preservation. This result will be needed in the proof of correctness of the translation.

In Fig. 9 we introduce the agreement relations between programs stores and values.

—$p, \sigma \vdash v \lhd t$ means that value v has type t in p, σ. When the value is an address ι, then t must be a class type c, and the store σ must map ι to an object of class c whose fields have the right type. The definition of $p, \sigma \vdash v' \prec t'$ is needed to break the circularity of the definition, since the value of one of the fields of the object could be ι.

—$p, \gamma \vdash \sigma\ \diamond$ means that for all addresses ι, if $\sigma(\iota) = [\![\cdots]\!]^c$ (c is the class of the object to which ι is mapped in σ) then ι has type c in p, σ (see the previous definition). Moreover, the value of the identifiers and of `this` in the store σ agree with their type in the environment γ.

—$p, \phi \vdash \sigma \lhd \sigma'$ means that store σ' is the store that may be obtained from σ after the evaluation of an expression whose effects are ϕ. So the address bound to `this` is not changed, and the only objects that may have been re-classified are the objects whose class is a subclass of one of the classes in ϕ.

$$\frac{v = \texttt{true} \text{ or } v = \texttt{false}}{p, \sigma \vdash v \prec \texttt{bool}} \quad (\texttt{bool} \prec)$$

$$\frac{p \vdash c \diamond_{ct}}{p, \sigma \vdash \texttt{null} \prec c} \quad (\texttt{null} \prec) \qquad \frac{\sigma(\iota) = [\![...]\!]^{c} \quad p \vdash c \leq c'}{p, \sigma \vdash \iota \prec c'} \quad (\iota \prec)$$

$$\frac{p, \sigma \vdash v \prec t \quad v \in sval}{p, \sigma \vdash v \lhd t} \quad (sval \lhd) \qquad \frac{\sigma(\iota) = [\![...]\!]^{c} \quad p, \sigma \vdash \iota \prec c' \quad \forall f \in \mathcal{F}s(p, c) : \; p, \sigma \vdash \sigma(\iota)(f) \prec \mathcal{F}(p, c, f)}{p, \sigma \vdash \iota \lhd c'} \quad (\iota \lhd)$$

$$\frac{\begin{array}{l}\sigma(\texttt{this}) \neq \texttt{null} \text{ and } \sigma(\iota) = [\![\ldots]\!]^{c} \implies p, \sigma \vdash \iota \lhd c \quad (\text{for all addresses } \iota) \\ \gamma(id) \neq \mathcal{U}df \implies p, \sigma \vdash \sigma(id) \lhd \gamma(id) \quad (\text{for all identifiers } id)\end{array}}{p, \gamma \vdash \sigma \diamond} \quad (\diamond)$$

$$\frac{\begin{array}{l}\sigma(\texttt{this}) = \sigma'(\texttt{this}) \\ \sigma(\iota) = [\![...]\!]^{c} \implies \sigma'(\iota) = [\![...]\!]^{c'}, \; \phi@_p c = \phi@_p c'\end{array}}{p, \phi \vdash \sigma \lhd \sigma'} \quad (\sigma \lhd)$$

Fig. 9. Agreement between programs, stores, and values

The following theorem asserts that the evaluation of a well-typed expression in a store that agrees with the typing environment, results either in an exception, or in a value that is of the right type. Moreover, the store resulting from the evaluation agrees with the environment resulting from the typing judgment. That is, types are preserved by reductions. The proof of the theorem is due to Drossopoulou et al. [2002].[9]

THEOREM 3.1. *Let* $p, \gamma \vdash e : t \parallel \gamma' \parallel \phi$. *If* $p, \gamma \vdash \sigma \diamond$ *and* $e, \sigma \overset{}{\underset{p}{\rightsquigarrow}} w, \sigma'$, *then*

—$w = v$, $p, \gamma' \vdash \sigma' \diamond$, $p, \sigma' \vdash v \lhd t$, *and* $p, \phi \vdash \sigma \lhd \sigma'$, *or*

—$w \in \{\texttt{castExc}, \texttt{nullPntrExc}\}$.

4. TRANSLATION OF $\mathcal{FICKLE}_{\texttt{II}}$ INTO $\mathcal{FICKLE}^{-}_{\texttt{II}}$: RATIONALE AND DESIGN ALTERNATIVES

For the design of the translation we had to consider the following issues:

(1) an appropriate encoding for re-classifiable objects,

(2) the relation between the types of a $\mathcal{F}ickle_{\texttt{II}}$ expression and the corresponding translated Java expression,

[9]In its original formulation [Drossopoulou et al. 2002] this result was mislabelled *type soundness*. As pointed out by an anonymous referee of a previous version of the present paper, in today's type terminology, Theorem 3.1 is a *type preservation* theorem, not a *type soundness* theorem. In fact, Theorem 3.1 says that *if* a program produces a result, *then* it is consistent with the program static type. This does not imply soundness, because a program that doesn't produce a result might have gone wrong.

(3) ensuring that a translated Java expression of class type will always denote the object in its most current state,

(4) the fact that a standard Java class c can be extended by a re-classifiable class, possibly after c has been translated (*i.e.*, compiled),

(5) making the translation compatible with separate compilation.

Concerning point 1), the basic idea is to represent each re-classifiable $\mathcal{F}ickle_{\mathrm{II}}$ object o through a pair $<id, imp>$ of Java objects. Roughly speaking, id provides the (immutable) *identity* of o, whereas imp is the *implementor object of the id* object, and provides its (mutable) *behavior*. A re-classification of o changes imp but not id, and method invocations are resolved by imp.

Concerning point 2), our initial idea was that there should be an isomorphism between the types of $\mathcal{F}ickle_{\mathrm{II}}$ expressions and the types of the translated Java expressions. However, as we shall argue in section 4.1, this lead to a complex system, and made the issues around point 3) more difficult. In our current solution all $\mathcal{F}ickle_{\mathrm{II}}$ expressions of class type are translated to Java expressions of the same type, namely `Identity`.

Concerning point 3), originally we were maintaining a chain of objects where each was delegating to the next, more recent implementation object. Thus, every field access or method call needed to follow the chain in order to find the most recent implementation object. However, this solution grew rather complex, and we abandoned it for the pair $<id, imp>$ described above.

Concerning points 4), and 5), we decided to represent *all* objects, even the non-re-classifiable ones, through such pairs $<id, imp>$. Thus, all Fickle classes, even the ones that describe objects that may not be re-classified, are translated in a uniform way. Also, field accesses or method calls need to find the implementation object, and are therefore translated in a uniform way, independently of whether the receiver belongs to a re-classifiable class, or not. Thus, a class may be translated without internal knowledge of the classes it is using.

Lastly, we also had to reconcile the requirements for the production of efficient Java code, the simplicity of the translation, and simplicity of the proofs. We decided to make the translation as simple as possible, neglecting efficiency in favor of uniformity (*i.e.*, fewer different cases) and simplicity.

4.1 Four design alternatives

The design of the translation of *Fickle* is the outcome of several iterations. In this section we outline and compare these.

We first developed *Version_1*, which we implemented through *Carmela*, a Java program mapping $\mathcal{F}ickle^{\mathtt{st}}$ onto Java [Anderson 2001]. $\mathcal{F}ickle^{\mathtt{st}}$ is a statement oriented version of $\mathcal{F}ickle_{\mathrm{II}}$. The development of *Carmela* proved more complex than anticipated, and we thus started a formal treatment, which we continued after the development of the software. This work led to *Version_2* [Ancona et al. 2001]. We had then some further ideas for improvement, which lead to *Version_3* [Ancona et al. 2002]. Finally, we developed *Version_4*, the approach described in this paper. *Version_4* has been implemented through *Isabella* [Anderson 2003], a Java program that maps $\mathcal{F}ickle^{\mathtt{st}}$ onto Java.

All translations are based on the idea of pairs of $<id, imp>$ objects, which represent the identity and the implementor of the corresponding *Fickle* object. Starting from that basic idea, the following questions needed to be assessed:

—Would one translated object play both the role of the identity and implementor?
Originally we allowed translated objects to take on the role of both the identity and implementor. This requires the object to have both `id` and `imp` fields. Later we separated the roles and each object was either an implementor or identity. Hence, each object contained either an `id` or `imp` field.

—Does the translation of a reference to a *Fickle* object refer to the identity or implementor part of the pair?
Having the representation of the *Fickle* object point to the implementor allows for a type preserving translation. However, it also allows references to objects that are "outdated". This reduces the garbage collection possibilities and increases the length of expressions required to reach the "active" implementation object. Referencing the identity simplifies expressions and allows for garbage collection of "outdated" objects.

—How are variables of state class type translated?
In the original definition of *Fickle* only `this` could have state class type. Therefore, the early designs did not cater for that, and only the *Version_4* does.

Figures 10, 11, 12 and 13 show the representation of the *Fickle* objects before and after re-classification in *Version_1*, *Version_2*, *Version_3*, and *Version_4*, respectively. The bold arrows represent references that exist between *Fickle* objects, whereas the normal arrows represent references introduced by the translation. Figures 14 and 15 summarize the differences between the various translations.

4.1.1 *Version_1. Fickle* objects may be represented through one or through a pair of objects, depending on their history. Namely, objects that have not been re-classified are represented by a single object. Implementor and identity objects belong to state subclasses of the root class. This follows because an object that has not yet been re-classified is both the implementor and identity. Both the identity and implementor objects contain a field `id` and `imp`.

Access to members of all classes goes through the indirection of `id` and `imp`. References to *Fickle* objects are represented through references to the implementor object, *e.g.*., the variable `account` in Fig. 10. Therefore, during run-time of the translated program references to "outdated" objects are possible: again in Fig. 10, `account` refers to an outdated object after the re-classification.

The translation preserves types up to roots, *i.e.*, the translation of a *Fickle* expression of type t has type $\mathcal{R}(p, t)$. The translation is optimized in terms of type casts and number of objects created at run-time.

4.1.2 *Version_2. Version_2* is a simplification over *Version_1*, in that *all Fickle* objects are represented through a pair of <`id`, `imp`> objects. In contrast to *Carmela*, implementor objects belong to state classes, and identity objects belong to root classes. As in *Version_1*, identity objects contain a field `id` and implementor objects contain a field `imp`.

Access to members of all classes goes through the indirection of `id` and `imp`. Type casts are required when accessing members because the `imp` field has type

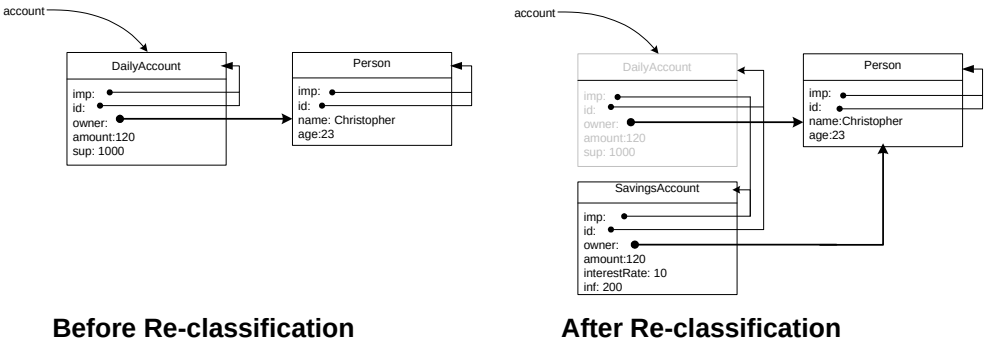

Fig. 10. Objects in *Version_1*

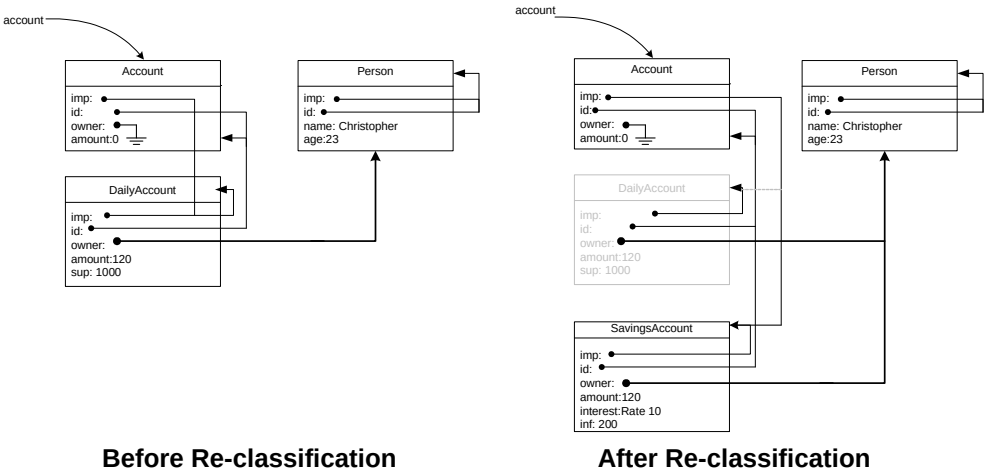

Fig. 11. Objects in *Version_2*

`FickleObject`. `FickleObject` is the superclass of all translated *Fickle* classes. References to *Fickle* objects are represented through references to the identity object. References to "outdated" objects are not possible. This opens more possibilities for garbage collection.

This translation preserves types up to roots. Overall the translation is simpler, but less efficient than *Version_1*.

4.1.3 *Version_3.* In *Version_3* we realized that we could achieve a significant simplification over *Version_2*, by representing identities through objects of the same class, regardless of the root class of the *Fickle* object. Thus, we introduced the class `Identity`, which has the field `imp`, pointing to the implementor of class `FickleObject`. The translated *Fickle* classes contain a field `id`, pointing to objects of type `Identity`.

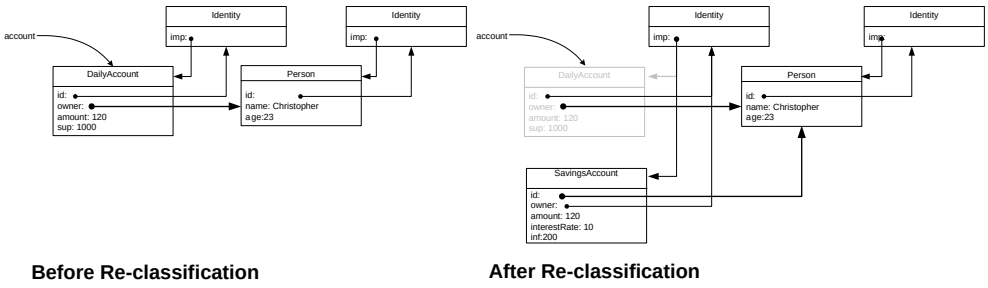

Fig. 12. Objects in *Version_3*

Access to members of all classes goes through the indirection of `id` and `imp`. Type casts are required when accessing members from all classes. References to *Fickle* objects are represented through references to the implementor object, and therefore references to "outdated" objects are possible.

The translation preserves types, *i.e.*, the translation of a *Fickle* expression of type `t` has type `t`. Compared with *Version_2*, *Version_3* requires the same number of objects, but the identity objects are smaller than the corresponding objects of root class type in *Version_2*, as they only contain the field `imp`. Thus, the translation is simpler, and more efficient than *Version_2*.

4.1.4 *Version_4.* Finally, we realized that we could achieve a further simplification, by adopting the identity objects as they are in *Version_3*, and representing references to *Fickle* objects though references to identity objects.

Access to members of all classes goes through the indirection of `imp` only, thus, requiring fewer access than any other translation. Type casts are required for all field accesses and method calls. Access to members of the receiver (`this`) is different because `this` is the only *Fickle* entity that is represented by an implementor rather than an identity object. Because `this` may be re-classified during a method activation, access to its members goes through the indirection of both `id` and `imp`. As in *Version_2*, because references to *Fickle* objects are represented through references to the identity, references to "outdated" objects are not possible, and thus more possibilities for garbage collection are open.

The translation does not preserve types, *i.e.*, the translation of any *Fickle* expression of class type c has type `Identity`.

Comparing *Version_4* with *Version_3*, it requires the same number of objects, but fewer intermediate steps to represent field access and method call, and allows more opportunities to garbage collection. Therefore, *Version_4* combines simplicity with efficiency.

5. TRANSLATION OF $\mathcal{FICKLE}_{\mathrm{II}}$ INTO $\mathcal{FICKLE}_{\mathrm{II}}^{-}$: AN INFORMAL OVERVIEW

In this section we give an informal overview of the translation; we outline the encoding of objects (Section 5.1), and then discuss an example (Section 5.2).

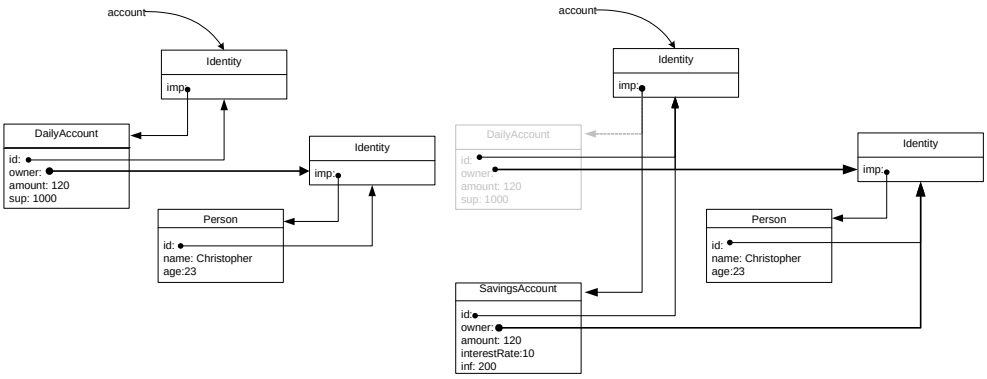

Fig. 13. Objects in *Version_4*

	Version_1	*Version_2*	*Version_3*	*Version_4*
Variables of type c are represented by variables of type:	$\mathcal{R}(p,c)$	$\mathcal{R}(p,c)$	$\mathcal{R}(p,c)$	`Identity`
References represented through reference to:	implementor	identity	implementor	identity
The `id` field has type:	$\mathcal{R}(p,c)$	`FickleObject`	`Identity`	`Identity`
The `imp` field has type:	$\mathcal{R}(p,c)$	`FickleObject`	`FickleObject`	`FickleObject`
Expression of type c translates to expression of type:	$\mathcal{R}(p,c)$	$\mathcal{R}(p,c)$	c	`Identity`
References to outdated objects	Possible	Impossible	Possible	Impossible

Fig. 14. Comparison of translation approaches

$\mathcal{F}ickle_{\mathrm{II}}$	*Version_1*	*Version_2*	*Version_3*	*Version_4*
`A a`	`A a`	`A a`	`A a`	`Identity a`
`P p`	`P p`	`P p`	`P p`	`Identity p`
`a.i()`	`a.i()`	`((A)a.imp).i()`	`((A)((A)a.id.imp)).i()`	`((A)a.imp).i()`
`this.o`	`((SA)this .id.imp).o`	`((SA)((A)this.id) .imp).o`	`((P)((SA)this.id.imp)) .o.id.imp)`	`((SA)this.id .imp).o`
`a1 = a2`	`a1 = a2`	`a1 = a2`	`a1 = (A)(a2.id.imp)`	`a1 = a2`
`SA sa`	`--`	`--`	`--`	`Identity sa`
`sa.o`	`--`	`--`	`--`	`sa.imp.o`

Fig. 15. Differences between the translation of expressions, with `Account,SavingsAccount,Person,Account::interestRate(),Account::owner` represented by `A,SA,P,i(),o` respectively. `a1`,`a2` have type `Account` and `this` has type `SavingsAccount`.

5.1 Encoding of objects

The translation is based on the idea that each object o of a state class c can be encoded in $\mathcal{F}ickle^{-}_{\mathrm{II}}$ by a pair $<id, imp>$ of objects; we call id the *identity object of imp* and imp the *implementor object of id*. Roughly speaking, id provides the identity of o, and imp the behavior of o, so that any re-classification of o changes imp but not id and method invocations are resolved by imp. Hence, two

implementors paired with the same identity represent the same object at different execution stages.

An object o that is not an instance of a state class does not need to be encoded in principle; however, for uniformity, the same kind of encoding described above is adopted also in this case, so that during the execution of a translated program there will be *exactly an identity object for any* $\mathcal{F}ickle_{\mathtt{II}}$ *object*. Note that, while there could be more than one implementor encoding a $\mathcal{F}ickle_{\mathtt{II}}$ object, say $<id, imp>$ and $<id, imp'>$, the converse cannot be true: if $<id, imp>$ and $<id', imp>$ are pairs encoding $\mathcal{F}ickle_{\mathtt{II}}$ objects, then $id = id'$.

Re-classification of objects can be exemplified by the diagram in Fig. 16. As shown there, the identity object is the same during the lifetime of a $\mathcal{F}ickle_{\mathtt{II}}$ object, whereas at different times the implementor object can change; the right implementor can always be recovered by the `imp` field of the identity object (dotted lines in the figure show obsolete values of this field).

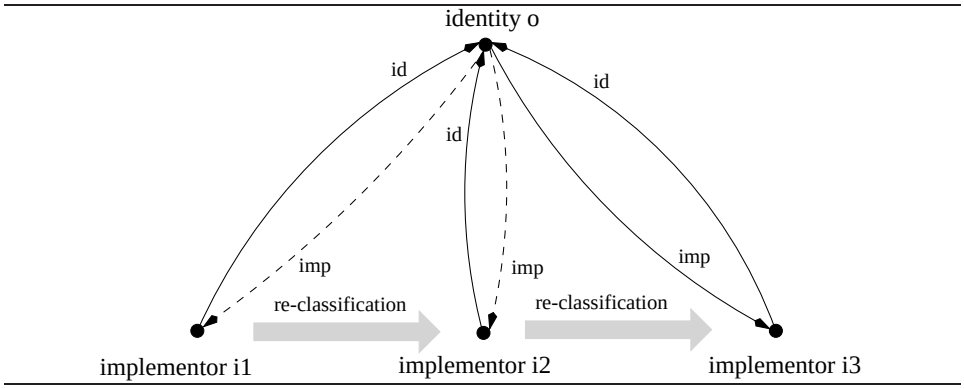

Fig. 16. Re-classification of objects

Classes are translated according to the following two rules:

—each $\mathcal{F}ickle_{\mathtt{II}}$ class (including `Object`) is translated into exactly one $\mathcal{F}ickle^{-}_{\mathtt{II}}$ class (whose instances are implementors);

—the translation preserves the inheritance hierarchy.

We illustrate the above in terms of the classes in Example 5.1.

Example 5.1. The following $\mathcal{F}ickle_{\mathtt{II}}$ program defines the classes `P`, `R`, `S1`, and `S2`.

```
class P extends Object
 { int f1;
   R m1(){}{R s; s = new S1;}
   int m2(S1 x){R}{x!!S2; 1}
   int m3 (S1 x){R} { x.m(this.m2(x));}
   int m4 (S1 x){R} { x.f1 = this.m2(x);}
   int m(int x){}{this.f1 = x; }
}
```

```
root class R extends P { }

state class S1 extends R  {
  int m5(S2 x){R}{this!!S2; this.f2 = x.f2;}
}

state class S2 extends R{
  int f2;
  int m(int x){}{this.f2 = x; }
}
```

After the instruction

```
  s=new S1;
```

in the body of `m1`, the $\mathcal{F}ickle_{\mathrm{II}}$ object referred by `s` is encoded in the translation, as sketched in Fig. 17, by the two $\mathcal{F}ickle^{-}_{\mathrm{II}}$ objects o and $o1$ in which the field `imp` of o points to $o1$ and the field `id` of $o1$ points to o.

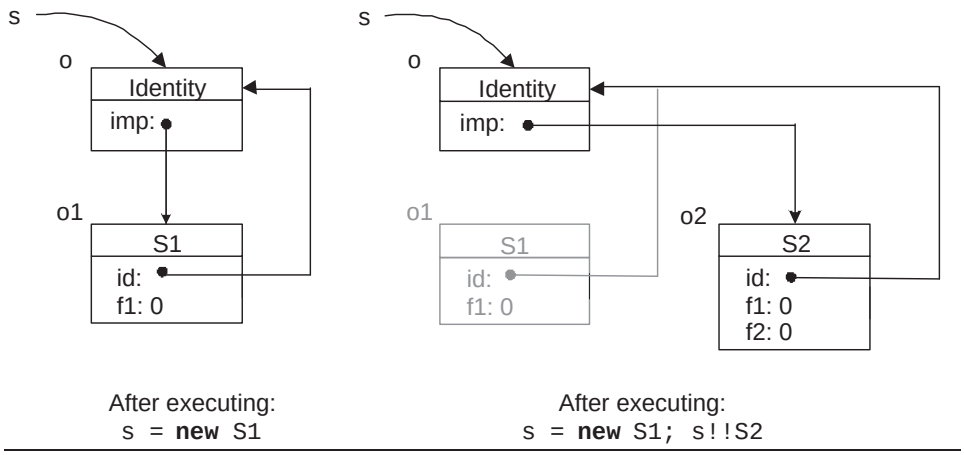

Fig. 17. Encoding of the $\mathcal{F}ickle_{\mathrm{II}}$ object referred by `s`

The object $o1$ has two fields: `id` of type `Identity` (inherited from class `FickleObject`, see Section 6.2), and `f1` (inherited from `P`). The fields `id` and `imp` are used to recover the identity and the implementor of an object, respectively. In this case the field `id` points to the object o of class `Identity` referred by `s`.

Assume now that the object referred by `s` is re-classified to `S2`, *e.g.* through `s!!S2`. Then, a new object, $o2$, of class `S2` is created, and the field `imp` of the identity o points to the new object $o2$.

5.2 An example of translation

In this section we illustrate the translation of the classes in Example 5.1. Below we give the result of the translation of class `P`, together with the definitions[10] of classes `Identity` and `FickleObject` that are added in the translation of any program.

```
class Identity extends Object{FickleObject imp;}
class FickleObject extends Object{Identity id;}
class P extends FickleObject{
 int f1;
 Identity m1(){} {
   Identity s;
   s = {
     Identity theId; S1 theImp; theId=new Identity; theImp=new S1;
     theImp.id=theId; theId.imp=theImp;
     theId;
   }
 }
 int m2(Identity x){} {
   {
     Identity theId; S2 theImp; R theLastImp;
     if (isnull(theId=x)) then null
     else {
       theImp=new S2; theLastImp=(R)theId.imp; theImp.id=theId;
       theId.imp=theImp; theImp.f1=theLastImp.f1;
     }
     theId;
   }
   1;
 }
 int m3(Identity x){} {
   {
     Identity theId; int arg;
     theId=x;
     arg={
       Identity theId1; Identity arg1;
       theId1= this.id;
       arg1=x;
       ((P)(theId1.imp)).m2(arg1);
     }
     ((R) (theId.imp)).m(arg);
   }
 }
 int m4(Identity x){}{
   int rightval;
   rightval = ((P)(this.id.imp)).m2(x);
   ((R) (x.imp)).f1 = rightval;
 }
 int m(int x){this.id.imp.f1=x;}
}
```

The translation maps the $\mathcal{F}ickle_{\mathrm{II}}$ class `P` into the $\mathcal{F}ickle^{-}_{\mathrm{II}}$ class `P`, which extends `FickleObject`, and hence inherits field `Identity id`.

[10] The declarations are just the signatures in some cases.

We now consider the translation of method `m1`. First of all, note that type `R` in the result type of the method and in the local variable declaration is translated to `Identity`. Indeed, a property of the translation is that expressions of a class type are translated to expressions of type `Identity` (see Theorem 7.1).

The translation of the method body also demonstrates the encoding of the creation of a new object of class `S1`: As explained above, two new objects (the identity and the implementor) are created, which point to each other through their fields `imp` and `id`, respectively. Then, the identity object is returned.

The translation of method `m2` demonstrates the encoding of the re-classification of the parameter `x` to `S2`. First of all, it is necessary to check whether `x` is `null`, since in this case the re-classification will have no effect. Otherwise, a new implementor object of class `S2` is created and this implementor object and the identity object are made to point to each other. Moreover, all fields common to the new and old implementor object,[11] are copied from the old (`theLastImp`) to the new (`theImp`) implementor object.

The translation of method `m3` demonstrates the encoding of method calls. Consider the external method call `x.m(...)`. First, the receiver (the variable `x`) is evaluated and assigned to the auxiliary variable `theId`. Second, the argument (the method call `this.m2(x)`) is evaluated. Finally, the current implementor, `theId.imp`, of the receiver is selected and the method `m` is invoked on it. If `x` is `null` then `theId.imp` raises a null pointer exception. Note, that the implementor can be correctly selected only after the evaluation of the argument, because this evaluation could re-classify the receiver object. This is exactly what happens in this case: namely, the receiver is reclassified from `S1` to `S2`, and thus the method `m` from class `S2` has to be executed; if we selected the implementor earlier, then the method `m` from class `P` would be executed instead, which contravenes the $\mathcal{F}ickle_{\mathrm{II}}$ semantics. This is the reason for the introduction of the auxiliary variables `theId` and `arg`. For the same reason, there is a cast to `R` (indeed, since evaluation of the argument could re-classify the receiver, we can only assume that the implementor has type `R`).

The internal method call is translated in the same way. Note that the cast to `P` is necessary because the field `imp` has type `FickleObject`.

The translation of method `m4` demonstrates the encoding of field assignment. The schema is analogous to that for method calls, except that here we have optimized the translation omitting unnecessary blocks and local variables. However, as in method `m3`, it is necessary to select the implementor only after evaluation of the right hand side of the assignment, because this evaluation could also re-classify the object containing the field; otherwise, `1` would be assigned to the field of the old implementor.

The translation of the method `m` has been optimized too, as well as the translation of the classes `R`, `S1` and `S2` below:

```
class R extends P {
}

class S1 extends R  {
```

[11] Common fields are those inherited from their common root superclass.

```
  int m4(Identity x){}{
    {
      Identity theId; S2 theImp; R theLastThis; theId=this.id; theImp=new S2;
      theLastThis=(R)(theId.imp); theImp.id=theId;
      theId.imp=theImp;theImp.f1=theLastThis.f1;
      theId;
    }
    ((S2)this.id.imp).f2= ((S2)x.imp).f2;
  }
}

class S2 extends R{
  int f2;
  int m(int x){}{((S2)this.id.imp).f2 = x}
}
```

6. TRANSLATION OF $\mathcal{FICKLE}_{\mathrm{II}}$ INTO $\mathcal{FICKLE}^{-}_{\mathrm{II}}$: A FORMAL DEFINITION

In this section we give the formal definition of the translation.

6.1 Translation of programs

The translation of a $\mathcal{F}ickle_{\mathrm{II}}$ program p consists of the declaration of the two special classes `FickleObject` and `Identity`, together with the translation of all classes declared in p. Since the translation of expressions depends on their types, the program p is passed as parameter to the translation function for classes.

$$\begin{array}{rl}[\![p]\!]_{prog} \triangleq & \texttt{class Identity extends Object\{FickleObject imp;\}} \\ & \texttt{class FickleObject extends Object\{Identity id;\}} \\ & [\![class_1]\!]_{class}(p) \ldots [\![class_n]\!]_{class}(p), \qquad \text{where } p = class_1 \ldots class_n\end{array}$$

For simplicity, here we are implicitly assuming no name conflicts between the classes and fields declared in p and the names `FickleObject`, `Identity` and `id`;[12] however, such conflicts could be always avoided by a slightly more complex translation where class names and fields are suitably renamed.

6.2 Translation of classes

As previously said, each translated class extends class `FickleObject`. An object o that needs to be re-classified to a state class c (recall that in the translation every class except for `Identity` is subclass of `FickleObject`), and that is encoded by the pair $<id, imp>$, is transformed into $<id, imp'>$, where imp' denotes the new implementor of class c (provided by a proper constructor of c; see definition below). Fields are initialized so that the identity and the new implementor point to each other. We introduce the translation of types.

Definition 6.1. Given a type t and a class c define:

—$theType(t) =$ `Identity` if t is a class, and $theType(t) = t$ otherwise, and
—$theName(c) =$ `FickleObject` if $c =$`Object`, and $theName(c) = c$ otherwise.

Each $\mathcal{F}ickle_{\mathrm{II}}$ class c is translated into a single $\mathcal{F}ickle^{-}_{\mathrm{II}}$ class containing the translation of all field and method declarations of c.

[12] Field `imp` of class `Identity` does not conflict since no translated class extends `Identity`.

The translation of fields and methods is the same for any kind of class. Since the translation of expressions depends on their types, the program p and the class c defining the type of `this` is passed as parameter to the translation function for methods.

$$\begin{array}{l}[\![\texttt{[root | state] class } c \texttt{ extends } c'\texttt{\{}\mathit{field}_1 \cdots \mathit{field}_m\ \mathit{meth}_1 \cdots \mathit{meth}_n\texttt{\}}]\!]_{class}(p) \triangleq \\ \texttt{class } c \texttt{ extends } \mathit{theName}(c')\texttt{\{ } [\![\mathit{field}_1]\!]_{field} \cdots [\![\mathit{field}_m]\!]_{field} \\ \qquad\qquad [\![\mathit{meth}_1]\!]_{meth}(p,c) \cdots [\![\mathit{meth}_n]\!]_{meth}(p,c) \\ \texttt{\}}\end{array}$$

6.3 Translation of field and variable (i.e. parameter or local variable) declarations

$$\begin{array}{l}[\![t\ f]\!]_{field} \triangleq \mathit{theType}(t)\ f \\ [\![t\ x]\!]_{var} \triangleq \mathit{theType}(t)\ x\end{array}$$

6.4 Translation of method declarations

Translating methods consists of translating their bodies. Effects are omitted, and types in the signature are substituted with their translation. Since the translation of expressions depends on their types, the program p and the environment γ defining the type of the parameters and of `this` must be passed as argument to the corresponding translation functions.

$$\begin{array}{l}[\![t\ m(t_1\ x_1, \ldots, t_n x_n)\ \phi\ \mathit{block}]\!]_{meth}(p,c) \triangleq \\ \qquad \mathit{theType}(t)\ m([\![t_1\ x_1]\!]_{var}, \ldots, [\![t_n\ x_n]\!]_{var})\texttt{\{ \}}[\![\mathit{block}]\!]_{expr}(p,\gamma) \\ \text{where } \gamma = t_1\ x_1, \ldots, t_n\ x_n,\ c\ \texttt{this}\end{array}$$

6.5 Translation of expressions

6.5.1 *Values, variables,* `this`*, null test, field selection, and cast.* In our encoding, in order to access the current implementor of an object we have to select the implementor currently pointed to by the identity of the object.

$$\begin{array}{l}[\![\mathit{sval}]\!]_{expr}(p,\gamma) \triangleq \mathit{sval} \\ [\![x]\!]_{expr}(p,\gamma) \triangleq x \\ [\![\texttt{this}]\!]_{expr}(p,\gamma) \triangleq \texttt{this.id} \\ [\![\texttt{isnull}(e)]\!]_{expr}(p,\gamma) \triangleq \texttt{isnull}([\![e]\!]_{expr}(p,\gamma)) \\ [\![e.f]\!]_{expr}(p,\gamma) \triangleq ((\mathit{theName}(c))\ ([\![e]\!]_{expr}(p,\gamma)\texttt{.imp}))\texttt{.}f \\ \text{where } p,\gamma \vdash e : c \parallel \gamma' \parallel \phi.\end{array}$$

Downcasting to c is needed because field `imp` has type `FickleObject`.

$$\begin{array}{l}[\![(c)e]\!]_{expr}(p,\gamma) \triangleq \quad \texttt{\{ Identity } x\texttt{;} \\ \qquad\qquad\quad \texttt{if (isnull(}x = [\![e]\!]_{expr}(p,\gamma)\texttt{))} \\ \qquad\qquad\quad\ \texttt{then null} \\ \qquad\qquad\quad\ \texttt{else } ((\mathit{theName}(c))(x\texttt{.imp}))\texttt{.id} \\ \qquad\qquad \texttt{\}} \\ \text{where } \gamma(x) = \mathcal{U}df\end{array}$$

6.5.2 *Variable assignment, field assignment, and method call.* Field f of the object denoted by the translation of e is accessed through the implementor of its identity. In earlier versions of the translation we naively translated variable

assignment expressions as follows: $[\![e]\!]_{expr}(p,\gamma).imp.f = [\![e_1]\!]_{expr}(p,\gamma)$. When we tried to prove the correctness of the translation we discovered that this was not correct. In particular, the evaluation of the translation of e_1 could re-classify the receiver of the assignment. Therefore, the selection of field *imp* from the translation of e must occur *after* the evaluation of the translation of e_1. We achieved this by introducing auxiliary local variables. The same idea is applied to the translation of method call.

$$[\![x = e]\!]_{expr}(p,\gamma) \triangleq x = [\![e]\!]_{expr}(p,\gamma)$$

$$\begin{array}{l}[\![e.f = e_1]\!]_{expr}(p,\gamma) \triangleq \ \texttt{\{}\ \texttt{Identity}\ x;\ \mathit{theType}(t)\ x_1;\\ \qquad\qquad\qquad\quad x = [\![e]\!]_{expr}(p,\gamma);\\ \qquad\qquad\qquad\quad x_1 = [\![e_1]\!]_{expr}(p,\gamma_1);\\ \qquad\qquad\qquad\quad ((\phi_2 @_p \mathit{theName}(c))(x.\texttt{imp})).f = x_1\\ \qquad\qquad\qquad \texttt{\}}\end{array}$$

where $p,\gamma \vdash e : c \parallel \gamma_1 \parallel \phi_1$ and $p,\gamma_1 \vdash e_1 : t \parallel \gamma_2 \parallel \phi_2$, and $\gamma(x) = \gamma(x_1) = \mathcal{U}df$.

$$\begin{array}{l}[\![e.m(e_1,\ldots,e_n)]\!]_{expr}(p,\gamma) \triangleq \ \texttt{\{}\ \texttt{Identity}\ x;\ \mathit{theType}(t_1)\ x_1;\ldots\mathit{theType}(t_n)\ x_n;\\ \qquad\qquad\qquad\quad x = [\![e]\!]_{expr}(p,\gamma);\\ \qquad\qquad\qquad\quad x_1 = [\![e_1]\!]_{expr}(p,\gamma_0);\\ \qquad\qquad\qquad\quad \ldots\\ \qquad\qquad\qquad\quad x_n = [\![e_n]\!]_{expr}(p,\gamma_{n-1});\\ \qquad\qquad\qquad\quad (((\phi_1 \cup \cdots \cup \phi_n)@_p \mathit{theName}(c))(x.\texttt{imp})).m(x_1,\ldots,x_n)\\ \qquad\qquad\qquad \texttt{\}}\end{array}$$

where $p,\gamma \vdash e : c \parallel \gamma_0 \parallel \phi_0$, for all $i \in \{1,\ldots,n\}$ $p,\gamma_{i-1} \vdash e_i : t_i \parallel \gamma_i \parallel \phi_i$, and $\gamma(x) = \gamma(x_1) = \cdots = \gamma(x_n) = \mathcal{U}df$.

6.5.3 *Object creation and re-classification.* According to the $\mathcal{F}ickle_{\texttt{II}}$ semantics, only the fields of the root superclass are preserved by re-classification.

$$\begin{array}{l}[\![\texttt{new}\ c]\!]_{expr}(p,\gamma) \triangleq \ \texttt{\{}\ \mathit{theName}(c)\ \mathit{theImp};\\ \qquad\qquad\qquad\quad \texttt{Identity}\ \mathit{theId};\\ \qquad\qquad\qquad\quad \mathit{theId} = \texttt{new Identity};\\ \qquad\qquad\qquad\quad \mathit{theImp} = \texttt{new}\ \mathit{theName}(c);\\ \qquad\qquad\qquad\quad \mathit{theImp}.\texttt{id} = \mathit{theId};\\ \qquad\qquad\qquad\quad \mathit{theId}.\texttt{imp} = \mathit{theImp};\\ \qquad\qquad\qquad\quad \mathit{theId}\\ \qquad\qquad\qquad \texttt{\}}\end{array}$$

where $\gamma(\mathit{theImp}) = \mathcal{U}df$, $\gamma(\mathit{theId}) = \mathcal{U}df$

$$[\![\texttt{this!!}c\texttt{;}]\!]_{expr}(p,\gamma) \triangleq \begin{array}{l}\texttt{\{ Identity } \mathit{theId}\texttt{;}\\ \quad \mathit{theName}(c)\ \mathit{theImp}\texttt{;}\\ \quad \mathcal{R}(p,c)\ \mathit{theLastThis}\texttt{;}\\ \quad \mathit{theId} = \texttt{this.id;}\\ \quad \mathit{theImp} = \texttt{new } \mathit{theName}(c)\texttt{;}\\ \quad \mathit{theLastThis} = (\mathcal{R}(p,c))(\mathit{theId}.\texttt{imp})\texttt{;}\\ \quad \mathit{theId}.\texttt{imp} = \mathit{theImp}\texttt{;}\\ \quad \mathit{theImp}.\texttt{id} = \mathit{theId}\texttt{;}\\ \quad \mathit{theImp}.f_1 = \mathit{theLastThis}.f_1\texttt{;}\\ \quad \ldots\\ \quad \mathit{theImp}.f_r = \mathit{theLastThis}.f_r\texttt{;}\\ \quad \mathit{theId}\\ \texttt{\}}\end{array}$$

where $\{f_1,\ldots,f_r\} = \mathcal{F}s(p,\mathcal{R}(p,c))$ and $\gamma(\mathit{theImp}) = \gamma(\mathit{theId}) = \gamma(\mathit{theLastThis}) = \mathcal{U}df$

$$[\![x\texttt{!!}c\texttt{;}]\!]_{expr}(p,\gamma) \triangleq \begin{array}{l}\texttt{\{ Identity } \mathit{theId}\texttt{;}\\ \quad \mathit{theName}(c)\ \mathit{theImp}\texttt{;}\\ \quad \mathcal{R}(p,c)\ \mathit{theLastImp}\texttt{;}\\ \quad \texttt{if (isnull(}\mathit{theId} = x\texttt{))}\\ \quad\ \texttt{then null}\\ \quad\ \texttt{else \{}\ \mathit{theImp} = \texttt{new } \mathit{theName}(c)\texttt{;}\\ \quad\quad\quad \mathit{theLastImp} = (\mathcal{R}(p,c))(\mathit{theId}.\texttt{imp})\texttt{;}\\ \quad\quad\quad \mathit{theId}.\texttt{imp} = \mathit{theImp}\texttt{;}\\ \quad\quad\quad \mathit{theImp}.\texttt{id} = \mathit{theId}\texttt{;}\\ \quad\quad\quad \mathit{theImp}.f_1 = \mathit{theLastImp}.f_1\texttt{;}\\ \quad\quad\quad \ldots\\ \quad\quad\quad \mathit{theImp}.f_r = \mathit{theLastImp}.f_r\\ \quad\quad \texttt{\};}\\ \quad \mathit{theId}\\ \texttt{\}}\end{array}$$

where $\{f_1,\ldots,f_r\} = \mathcal{F}s(p,\mathcal{R}(p,c))$ and $\gamma(\mathit{theImp}) = \gamma(\mathit{theId}) = \gamma(\mathit{theLastImp}) = \mathcal{U}df$

6.5.4 *Conditionals and blocks.*

$$[\![\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2]\!]_{expr}(p,\gamma) \triangleq \texttt{if } [\![e]\!]_{expr}(p,\gamma) \texttt{ then } [\![e_1]\!]_{expr}(p,\gamma_0) \texttt{ else } [\![e_2]\!]_{expr}(p,\gamma_0)$$

where $p,\gamma \vdash e : \texttt{bool} \parallel \gamma_0 \parallel \phi_0$

$$[\![\{t_1\ x_1\texttt{;}\ldots\texttt{;}t_s x_s\texttt{;}\ e_1\texttt{;}\ldots\texttt{;}e_n\}]\!]_{expr}(p,\gamma) \triangleq \begin{array}{l}\texttt{\{}\ [\![t_1\ x_1]\!]_{var}\texttt{;}\\ \quad\ldots\\ \quad [\![t_s\ x_s]\!]_{var}\texttt{;}\\ \quad [\![e_1]\!]_{expr}(p,\gamma_0)\texttt{;}\\ \quad\ldots\\ \quad [\![e_n]\!]_{expr}(p,\gamma_{n-1})\\ \texttt{\}}\end{array}$$

where, $\gamma_0 = \gamma[x_1 \mapsto t_1,\ldots,x_s \mapsto t_s]$ and for all $i \in \{1,\ldots,n\}$ $p,\gamma_{i-1} \vdash e_i : t_i' \parallel \gamma_i \parallel \phi_i$

7. PROPERTIES OF THE TRANSLATION

In this section we formalize and prove the good properties of the translation previously mentioned.

7.1 Preservation of static semantics

For any environment γ, its translation $[\![\gamma]\!]$ is defined by

$$[\![\gamma]\!] = \{theType(t)\ x \mid \gamma(x) = t\} \cup \{theName(\gamma(\texttt{this}))\ \texttt{this}\}.$$

THEOREM 7.1. *Let p be a program s.t. $\vdash p\ \diamond$, γ, and γ' environments s.t. $p \vdash \gamma(\texttt{this})\ \diamond_{ct}$, e an expression, t a type and ϕ an effect.*
If $p, \gamma\ \vdash\ e\ :\ t\ \|\ \gamma'\ \|\ \phi$, then $[\![p]\!]_{prog}, [\![\gamma]\!]\ \vdash\ [\![e]\!]_{expr}(p, \gamma)\ :\ theType(t)\ \|\ [\![\gamma]\!]\ \|\ \{\,\}$.

PROOF. See Appendix B. □

Note that, in the typing judgement $[\![p]\!]_{prog}, [\![\gamma]\!]\ \vdash\ [\![e]\!]_{expr}(p, \gamma)\ :\ theType(t)\ \|\ [\![\gamma]\!]\ \|\ \{\,\}$ in the statement of the previous theorem, the environment on the right is $[\![\gamma]\!]$ (and not $[\![\gamma']\!]$). This is due to the fact that in the translated expression, $[\![e]\!]_{expr}(p, \gamma)$, there is no re-classification.

THEOREM 7.2. *Let p be a program s.t. $\vdash p\ \diamond$, and c a class name. If $p \vdash c\ \diamond$, then $[\![p]\!]_{prog} \vdash c\ \diamond$.*

PROOF. See Appendix B. □

THEOREM 7.3. *Let p be a program. If $\vdash p\ \diamond$, then $\vdash [\![p]\!]_{prog}\ \diamond$.*

PROOF. See Appendix B. □

7.2 Preservation of dynamic semantics

In this section we show that the dynamic semantics of expressions is preserved by the translation.

We introduce a relation between stores $p, \gamma \vdash \sigma \approx \sigma'$ that expresses the fact that store σ' contains the "translation" of the objects in store σ. More precisely, an object o of class c in σ is translated in σ' into an object of class `Identity` whose `imp` field points to an implementor object o' that is an instance of the translation of the class c.

Values of identifiers in σ are preserved in σ', except for `this`, whose value in σ' is (the address of) an implementor object whose `id` field points to the address that is the value of `this` in σ. The store σ and σ' (except for `this`), are assumed to agree with the environments γ and $[\![\gamma]\!]$, that is, they contain values that agree, w.r.t. typing, with their definitions (see Fig. 9 for the formal definition of $p, \gamma \vdash \sigma\ \diamond$).

Regarding the agreement of `this` in store σ', observe that the store σ'' resulting from the evaluation of the translation of the re-classification of `this`, from class c to class d, is such that $\sigma''(\texttt{this}) = \iota$ and $\sigma''(\iota) = \texttt{[[id}: \iota' \cdots \texttt{]]}^c$, (the translation of the expression does not contain re-classifications, whereas the original expression did) and $\sigma''(\sigma''(\iota'))(\texttt{imp}) = \texttt{[[id}: \iota' \cdots \texttt{]]}^d$. So $\sigma''(\texttt{this})$ does not agree with d but agrees with c that is the type of `this` before re-classification.

Definition 7.4. Let p be a program, γ an environment, σ and σ' stores such that $p, \gamma \vdash \sigma\ \diamond$ and $[\![p]\!], [\![\gamma]\!][\texttt{this} \mapsto c] \vdash \sigma'\ \diamond$, for some c. We say that σ' *is a translation of* σ, and write $p, \gamma \vdash \sigma \approx \sigma'$, if

(1) $\sigma(\texttt{this}) = \sigma'(\sigma'(\texttt{this}))(\texttt{id})$
(2) for all x, $\gamma(x) \neq \mathcal{U}df$ implies $\sigma(x) = \sigma'(x)$, and
(3) for all ι, if $\sigma(\iota) = \texttt{[[}f_1 : v_1, \ldots, f_n : v_n\texttt{]]}^c$ then

—$\sigma'(\iota) = [[\texttt{imp} : \iota']]^{\texttt{Id}}$,
—$\sigma'(\iota') = [[\texttt{id} : \iota, f_1 : v_1, \ldots, f_n : v_n]]^{theName(c)}$.

For every store σ, let $addr(\sigma)$ denote the set of the addresses that occur in σ. Note that, according to Definition 7.4, we have that $p, \gamma \vdash \sigma \approx \sigma'$ implies $addr(\sigma) \subseteq addr(\sigma')$. We can now state the theorem that asserts that our translation is adequate.

THEOREM 7.5. *Let e be an expression such that: $p, \gamma \vdash e : t \parallel \gamma' \parallel \phi$, and σ, and σ_1, be stores such that $p, \gamma \vdash \sigma \approx \sigma_1$. Then*

(1) *For all w and σ', if $e, \sigma \underset{p}{\rightsquigarrow} w, \sigma'$ and $(addr(\sigma') - addr(\sigma)) \cap addr(\sigma_1) = \emptyset$, then there is σ'_1 such that $[\![e]\!], \sigma_1 \underset{[\![p]\!]}{\rightsquigarrow} w, \sigma'_1$ and*
—either $w \in val$ and $p, \gamma' \vdash \sigma' \approx \sigma'_1$,
—or $w \in \{\texttt{castExc}, \texttt{nullPntrExc}\}$.

(2) *For all w and σ'_1, if $[\![e]\!], \sigma_1 \underset{[\![p]\!]}{\rightsquigarrow} w, \sigma'_1$, then there is σ' such that $e, \sigma \underset{p}{\rightsquigarrow} w, \sigma'$ and*
—either $w \in val$ and $p, \gamma' \vdash \sigma' \approx \sigma'_1$,
—or $w \in \{\texttt{castExc}, \texttt{nullPntrExc}\}$.

PROOF. See Appendix C. □

Note that the condition

$$(addr(\sigma') - addr(\sigma)) \cap addr(\sigma_1) = \emptyset \tag{1}$$

in point (1) of the statement of the previous theorem is not restrictive. In fact, if $e, \sigma \underset{p}{\rightsquigarrow} w, \sigma'''$ for some store σ''', then there exists a store σ' such that $e, \sigma \underset{p}{\rightsquigarrow} w, \sigma'$ and σ' satisfies condition (1).

Remark 7.6. From the preservation of dynamic semantics we derive that, the casts introduced by the translation are all safe, since if the original program did not rise a cast exception also its translation does not raise a cast exception. Indeed, the casts are needed to obtain the preservation of static semantics, that is to convince the type checker that the field `imp` of `Identity` objects has the right type. Note that, during the lifetime of an `Identity` object such type may change, so the field `imp` cannot be given a specific re-classifiable type. As we will see, also with a translation in which the field `imp` has a generic type we cannot avoid some cast (see Section 9), since a generic type must be instantiated to a specific (re-classifiable) type upon creation.

8. *ISABELLA*, A PROTOTYPE IMPLEMENTATION

Our prototype implementation, *Isabella* [Anderson 2003], follows *Version_4* to map *Fickle*$^{\texttt{st}}$ onto Java. It is an extension of *Carmela* [Anderson 2001], which followed *Version_1*. *Isabella* is written in Java and follows a design based on the Sun Java compiler (version 1.4). *Isabella* consists of a type checker and code generator, both implemented using the visitor pattern [Gamma et al. 1995] as in the Java compiler. The whole compiler consists of approximately 6000 lines of code and can be found at `http://www.macs.hw.ac.uk/DART/software/isabella/index.html`. *Isabella* extends *Fickle*$^{\texttt{st}}$ in order to make testing easier, the extensions include:

—Output via `System.out`

—Integers and booleans with relevant operations such as `++,--` etc.

One of the challenges in *Isabella* was representing the blocks of code that arise in the translation *Version_4*. Noting that Java does not allow blocks as expressions, any entity in the source that requires a block of code in the translation must be represented as a flattened statement and a fresh local variable to contain the result. The variable can then be used where the value of the block is required.

For example, consider the translation of `a = new DailyAccount (); a.transact ( 1200 );a.interest();`. For expression, `a = new DailyAccount ()` we first translate `new DailyAccount` to get:

```
DailyAccount s1;
Identity s2;
s2= new Identity();
s1= new DailyAccount();
s1.id = s2;
s2.imp = s1;
```

Note the use of new temporary variables `s1` and `s2`, with the result being in variable `s2`. The assignment to variable `a` is represented as `a = s2`. For method call `a.transact(1200)` we have:

```
Identity s3 = a;
int s5 = 1200;
((Account)(s3.imp)).transact (s5);
```

Note the use of temporary variables `s3` for the receiver and `s5` for the argument. As method `transact` is `void` we have no temporary variable for the return value. Finally for method call `a.interest();` we have:

```
Identity s6 = a;int s7;
s7 = ((Account)(s6.imp)).interest ();
```

As with the previous method call we have a temporary variable for the receiver and in this case a temporary variable for the result `s7`.

Thus, the code produced by *Isabella* is identical to that produced by the translation given in this paper, apart from erasure of blocks and the extra features mentioned above.

9. TRANSLATING $\mathcal{FICKLE}^{\mathtt{ST}}$ INTO JAVA 1.5

As seen in Section 8, the prototype *Isabella* translates $\mathcal{F}ickle^{\mathtt{st}}$ into Sun Java 1.4 by following the formal translation of $\mathcal{F}ickle_{\mathtt{II}}$ into $\mathcal{F}ickle^{-}_{\mathtt{II}}$ defined in Section 6. However, during the review process of this article, the new version 1.5 of the Sun Java Compiler was released, including the new interesting features of *generics* and *wildcards* [Joy et al. 2005].

This section informally proposes a new translation schema which is still mainly based on *Version_4* from Section 6, but exploits the expressive power of generics and wildcards, and uses Java 1.5 as target language. This new approach has two main advantages: first, it is possible to minimize the insertion of cast operators needed for

ensuring the type safety of the code generated by the translation. Second, method overloading is supported, since the subtyping relation is fully preserved by the new translation.

These advantages come at the cost of an increased complexity of the translation scheme; for reasons of space and time limits, no formal definition is provided here, but only the basic ideas are outlined by means of examples. We leave to future work the full formalization, which would include the extension of $\mathcal{F}ickle^{-}_{\mathtt{II}}$ (and, thus, of $\mathcal{F}ickle_{\mathtt{II}}$ as well) with generics, and the adaptation of the proofs presented here.

Remark. Even though the translation scheme outlined here represents a significant improvement, the two advantages mentioned above are lost at the bytecode level, because of the limitation of the JVM implementation of generic classes which relies on type erasure.[13]

Despite this, the translation scheme presented here is more appealing than the one on which the prototype *Isabella* is based; clearly, the only source of problem is the JVM limitation, which we hope will be eventually overcome in some future release.

9.1 Definition of classes `Identity` and `FickleObject`

The main drawback of the translation *Version_4* is that any field access or method invocation requires a type cast in order to ensure the type correctness of the generated code. Consider, for instance, the classes in Example 5.1 and the following code fragment:

```
P p; ...  p.f1=1;
```
(2)

According to *Version_4*, the code in (2) is translated into

```
Identity p;
...
((P) p.imp).f1=1;
```

The type cast is needed since the type of imp is `FickleObject`. Because of type preservation, we know that `p.imp` will always contain objects of type `P`; therefore, the down cast will never throw an exception. However, the cast does affect the performance of field access in the translated code. The same issue arrises with all the translations in Section 4, except for *Version_1*. In *Version_1* although the casts are avoided the fields `imp` and `id` are redefined in each of the translated classes. This results in each instance of a class with n ancestors having $n+1$ `imp` and $n+1$ `id` fields.

By replacing the class `Identity` with a generic class we can assign to field imp the most specific correct type:

```
class Identity<X>{
    X imp;
```

[13]Basically, at the bytecode level each parameterized type `C<T1,...,Tn>` is translated into the corresponding raw type `C`, and, consequently, appropriate type casts must be inserted by the compiler.

```
    Identity(X imp){
        this.imp=imp;
    }
}
```

With this new definition of `Identity` we could translate code fragment (2) as follows:

```
Identity<P> p;
...
p.imp.f1=1;
```

We see that the cast is no longer required because `p.imp` has type `P`.

Let us now consider how the translation of `P` needs to be modified w.r.t. *Version_4*.

First, the translated class must have a field `id` of type `Identity<P>` (recall that `id` is needed for retrieving the identity of `this`). Furthermore, we would like to avoid duplication of `id` in the descendant classes; therefore, in the translation, field `id` is declared once at the root of the class hierarchy, *i.e.*, in class `FickleObject` (the translation of `Object`), and each class is parametric in the type of field `id`:

```
class FickleObject<X extends FickleObject<?>>{
    Identity<X> id;
}

class P<X extends P<?>> extends FickleObject<X>{
...
}
```

We see that `P` inherits the field `id` of type `X` from `FickleObject`, where `X` is a type variable with upper bound `P<?>`. The type bounds in `FickleObject` and `P` exploit wildcards [Joy et al. 2005]; if `C` is a generic class, then `C<?>` corresponds to the existential type $\exists$ `X`$\leq$`Object.C<X>`. In Section 9.3 we shall see why this upper bound is essential for ensuring the type safety of the translation.

Note that the following alternative translation of class `P`:

```
class P extends FickleObject<P>{
...
}
```

would prevent the translation of the descendent classes of `P` inheriting `id` with the most specific type. The two classes `FickleObject` and `P` serve as class generators. The "fix-points" of these generators can be obtained by declaring two other classes:

```
class FickleObjectFix extends FickleObject<FickleObjectFix>{}

class PFix extends P<PFix>{}
```

The sole purpose of fixed point classes is to allow object creation without having to resort to the use of raw types. Raw types allow the use of a name of a generic type declaration without any accompanying actual type parameters; for instance, `new P()` returns a new instance of the class obtained from `P` by erasing the parameter

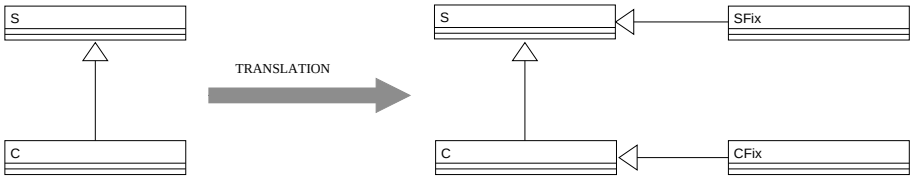

Fig. 18. Mapping of the inheritance hierarchy

`X`. However, the use of raw types is allowed only as a concession to compatibility of legacy code; therefore, is strongly discouraged since it is possible that they will be dropped from future versions of the Java programming language.

Generalizing the discussion above, the new proposed translation has to adhere to the following pattern: each class `C`, with direct superclass `S`, is translated into two classes named `C` and `CFix` having the following shape:

```
class C<X extends C<?>> extends S<X>{ // class generator
// the translation of the body of the original C is inserted here
...
}
class CFix extends C<CFix>{// fix-point of C
// the body is empty
}
```

Figure 18 shows the inheritance hierarchy produced by our translation.

9.2 Translation of reference types

Another important consequence of the use of generic classes is that the subtyping relation can be fully preserved by the translation.

For *Version_4* it is possible to prove that if two expressions have types t_1 and t_2 respectively, and $t_1 \leq t_2$, then the two translated expressions have types t'_1 and t'_2, respectively, such that $t'_1 \leq t'_2$. This follows directly from Theorem 7.1 and Lemma B.8 since all reference types are flattened to `Identity`. However, because of the flattening, subtyping is obviously not preserved in the opposite direction. Therefore, *Version_4* does not support method overloading as in Java.

By using a generic version of the class `Identity`, we avoid the type flattening and make it possible to preserve subtyping. However, our tentative translation of `P` to `Identity<P>` does not work for several reasons.

First, in the translated program class `P` is a raw type; something we want to avoid. Clearly, we cannot translate `P` into `Identity<PFix>`, because, as shown in Figure 18, the classes `CFix` and `SFix` are unrelated; even though in the source program `C` is a (direct) subclass of `S`. Therefore, the use of `Identity<PFix>` would not preserve subtyping. However, in the translation, `C` is a (direct) subclass of `S`; therefore, `C<?>` is a subtype of `S<?>`. This makes the translation of `P` into `Identity<P<?>>` a possible candidate.

The difference between type `P<?>` and `PFix` is minimal; in the translated code all implementors of `P` are instances of `PFix`. Both specify the same set of accessible class members with the same types except for field `id` which has type `Identity<PFix>` in `PFix` and type `Identity<?>` in `P<?>`. In fact, type `P<?>` is

the most general non-raw type containing all the accessible members of `P`; despite its generality it fits our purposes. We do not require a type more specific than `P<?>` because: (a) in each class generator the only member depending on the type parameter is the field `id`, inherited from `FickleObject`; (b) in the translation, the field `imp` is never directly used as a target to access the id field; (c) if `p` has type `Identity<P<?>>`, then `p.imp` has type`P<?>`.

Finally, note that if `C` is subtype of `S`, then `Identity<C<?>>` is not a subtype of `Identity<S<?>>`. By exploiting wildcards we see that `Identity<? extends C<?>>` is a subtype of `Identity<? extends S<?>>`. Therefore, the correct translation of code fragement (1) is:

```
Identity<? extends P<?>> p;
...
p.imp.f1=1;
```

Note that the inferred type for `p.imp` is `P<?>`; therefore, `p.imp.f1` has, as expected, type `int`.

Summarizing, a reference type `C` is translated[14] to `Identity<? extends C<?>>`, and it is not difficult to prove the following property:
`C1` $\leq$ `C2` iff `Identity<? extends C1<?>>` $\leq$ `Identity<? extends C2<?>>`.

9.3 Translation of expressions

So far, we have outlined the translation of class declarations (except for the bodies) and of reference types. The translation of class bodies (that is, field and method declarations) is immediate once one has specified how types and expressions are translated. Therefore, it remains to show the translation of expressions. Because of space limits, we do not provide a general definition, but only show the behavior of the translation on some examples. We only focus on those kinds of expression whose translation is different from *Version_4*.

The whole translation of Example 5.1 has been tested and can be found in Appendix D.

Member access and assignment. As already shown, the translation is the same as in *Version_4*, except for the avoidance of type casts.

Instance creation. Let us consider the translation of `S1 s1=new S1`:

```
Identity<? extends S1<?>> s1;
S1Fix temp=new S1Fix();
temp.id=new Identity<S1Fix>(temp);
s1=temp.id;
```

Note that the inferred type of `temp.id` is `Identity<S1Fix>`; by definition, `S1Fix` is a subtype of `S1<S1Fix>` which, in turn, is a subtype of `S1<?>`. Therefore, `Identity<S1Fix>` is a subtype of `Identity<? extends S1<?>>`, and the last assignment is statically correct.

[14]This translation is only applied to types used in field and local variable declarations, and in method headers. We refer to Section 9.3 for the translation of types used in cast expressions.

Type cast. Let us consider the translation of `R r=(R) p` where we assume that `p` has type `P`:

```
Identity<? extends R<?>> r = ((p==null)?null:((R<?>) p.imp).id);
```

As happens in *Version_4*, if the object is null then the cast succeeds and null is returned. Otherwise, the cast must be performed on the implementor and, if successful, the identity is returned.

One could be tempted to give the simpler translation:

```
Identity<? extends R<?>> r = (Identity<? extends R<?>>) p;
```

which works only in principle. Because the implementation of generics is based on type erasure, the cast above would be unchecked. That is, the cast could succeed when it should not. To see why, observe that `Identity<? extends P<?>>` is not a subtype of `Identity<? extends R<?>>`, but both erase to `Identity`. If we consider the implementor field, `imp`, as in our proposed translation of casts, we can elide the problem. For example, an expression e of type `C` will translate to an expression e' of type `C'`, where the `imp` field of e' has type `C<?>`. Recall that, the erasure of type `C<?>` always preserves subtyping, *i.e.*, `C1` $\leq$ `C2` iff `C1<?>` $\leq$ `C2<?>`. Hence, the cast to `R<?>` instead of `Identity<? extends R<?>>` is equivalent to casting to the corresponding raw type `R` which is always checked in Java.

We now give some explanation of why the proposed translation is type safe. If we assume that `p` has type `P` in the source code, then `p` has type `Identity<? extends P<?>>` in the translated code. Therefore, `p.imp` has type `P<?>` which is a supertype of `R<?>`; this means that the cast expression is statically correct. Finally, note that `id` is accessed only after the cast has been performed; therefore, the assigned expression has type `Identity<? extends R<?>>`, which is the type of `r`. This follows because, according to the translation, the upper bound of the type parameter in `R` is `R<?>`.

Object re-classification, `this` *and variables.* Let us assume that `s1` has type `S1` in the following code fragment:

```
s1!!S2;
s1.f2=2;
```

According to the new translation, the assignment expression after the re-classification of `s1` is translated into `s1.imp.f2=2`. Since in the code generated by the translation the type of `s1` becomes `Identity<? extends<S1<?>>`, it follows that `s1.imp` has type `<S1<?>>` which implies that the expression is ill-typed.

A possible solution to this problem consists in mimicking the approach followed in the type system. Indeed, according to the typing rules of $\mathcal{F}ickle_{\mathrm{II}}$, after the re-classification the type of `s1` in the type environment becomes `S2`. Since the static type of `s1` cannot be changed, what the translation can do is to "replace" the old variable `s1` with a new variable, say `s1_S2`, declared with the proper new type (in this case, the translation of `S2`, that is, `Identity<? extends S2<?>>`), and referencing the same object as `s1`.

The translation has to keep track of this name change, in order to properly translate the next occurrences of `s1` into `s1_S2`, as happens with the assignment immediately following the re-classification:

```
Identity<? extends S2<?>> s1_S2=null;
if(s1!=null){
    S1<?> oldImp=s1.imp;
    S2Fix temp=new S2Fix();
    temp.f1=oldImp.f1;
    temp.id=(Identity<S2Fix>) (Object) s1;
    temp.id.imp=temp;
    s1_S2=temp.id;
}
s1_S2.imp.f2=2;  // must use s1_S2 and not s1!!!
```

After re-classification, `s1` and `s1_S2` contain the same object; what changes is their static type. Since `temp.id` and `s1` have type `Identity<S2Fix>` and `Identity<? extends S1<?>>` respectively, and the former type is not a supertype of the latter, a type cast is needed to make the assignment statically correct. However, since the types of `temp.id` and `s1` are unrelated, the expression `(Identity<S2Fix>) s1` would not be type correct. The problem can be avoided by first inserting a cast to `Object` (this is always allowed and type safe). Note that the second cast to `Identity<S2Fix>` is unchecked because of type erasure (recall the comments in the previous paragraph). Type preservation ensures that `temp.id` is never used in an unsafe way.

Finally, note that for the same reason explained above, the expression `this.id`, used in *Version_4* for accessing members of `this`, has to be replaced with a corresponding variable whose name and type depends on the context (for instance, see the translation of method `m5` in Appendix D).

10. RELATED WORK

Several other approaches to the expression of fundamental change of behaviour have been suggested. *Predicate classes* [Chambers 1993; Ernst et al. 1998] support a form of dynamic classification of objects based on their run-time value: Code is broken down on a per-function basis, while *Fickle* follows the mainstream, where code is broken down on a per-class basis. Similarly, for single method dispatch, Tailvasaari [1993] considers classes with "modes" representing different states, *e.g.*, opened vs. iconified window. *Wide classes* [Serrano 1999] are the nearest to our approach; they allow an object to be temporarily "widened" or "shrunk", to a subclass or a superclass. However they differ from *Fickle*, by dropping the requirement for a strong type system, and requiring run-time tests for the presence of fields. Finally, the language GILGUL [Costanza 2001] is an extension of Java that allows for dynamical object replacement through *implementation-only classes*, *i.e.*, classes that cannot be used as types. Objects belonging to a Java class can be replaced only by instances of the same class or of any subclass, while objects belonging to an implementation-only class can be replaced also by instances of any class having the same least non-implementation-only superclass. Like the other approaches we discussed, GILGUL is not strongly typed, and a run-time exception is raised when a forbidden object replacement is attempted.

Promoting innovation by extending a popular existing language, and defining the new language features by translation into the old, is a widely-used technique that

has been proved successful in many cases. For the Java case, a seminal work has been that on Pizza [Odersky and Wadler 1997], the first proposal for an extension of Java with generics (and also higher-order functions and algebraic data types), hence we report here some general ideas from this paper. The authors first identify as essential goals a Java extension should meet the fact that new code should compile into the Java Virtual Machine, and that existing code compiled from Java should smoothly inter-operate with new code. However, since JVM and Java are tightly coupled, a language that compiles into JVM can lose little in efficiency and gain much in clarity by translating into Java as an intermediate stage. When an extension is directly implemented by translation, inter-operability amounts to say that the translation is the identity on old code. This paper also introduces the *heterogeneous* and *homogeneous* terminology for translation of generics. A heterogeneous translation produces a specialised copy of code for each instantiation, yielding code that runs faster, whereas a homogeneous translation uses a single copy of the code with a universal representation, yielding more compact code. The homogeneous translation is at the basis of GJ [Bracha et al. 1998] and of the new version 1.5 of the Sun Java Compiler.

Also implementation of Jam [Ancona et al. 2003], a Java extension supporting *mixin classes* (parametric heir classes) is done by translation into Java source code. The current prototype translates each mixin instantiation into a Java class obtained, roughly, by extending the parent by a copy of the definitions contained in the mixin, that is, adopts a heterogeneous translation, which favors running time of the generated code, penalizing size and modularity. In particular, the approach is not compatible with separate compilation, since for translating a mixin instantiation the mixin source code is needed. Alternative solutions could be homogeneous translation or direct generation of bytecode, as done in Jiazzi, a system for constructing and linking components in Java [McDirmid et al. 2001].

Translations into plain Java have also been adopted for explaining the semantics of *inner classes* [Igarashi and Pierce 2002] and for extending Java with *parasitic methods* [Boyland and Castagna 1997], an encapsulated form of multimethods. The MultiJava project [Clifton et al. 2006], an extension that adds open classes and symmetric multiple dispatch, adopts compilation into Java bytecode. The compilation schema has been carefully designed to overcome a major obstacle to adding symmetric multimethods to an existing statically-typed programming language, that is, their modularity problem (solved by employing asymmetric multiple dispatch by Boyland and Castagna [1997]). In MultiJava, each class can be compiled separately. Moreover, MultiJava retains backward-compatibility and interoperability with existing Java source and bytecode.

Summarizing the work above, we can say that the issues that any Java extension should deal with are compilation in either source or bytecode, preserving separate compilation, and inter-operability with Java code. In addition, extensions that allow to write parametric code in place of many instances must choose a heterogeneous or homogeneous approach: however, this is not the case of our extension that goes in the direction of a more flexible run-time behaviour. The Fickle compilation schema presented in this paper chooses the approach of translation into Java source code, which, as mentioned above, is the most appropriate for a prototype mainly aiming at showing a clear and simple semantics of the extension. Gaining

in efficiency via direct generation of Java bytecode could be investigated, but we do not think it would cause major changes in the translation. An important result is that our translation allows separate compilation, since a Fickle class can be compiled in a context where only type information on other Fickle classes is available. On the contrary, inter-operability with old code is an important issue for further work. Currently, translation of plain Java classes is not the identity, hence we cannot compile separately Fickle code in a context where Java bytecode produced by standard compilation is available, since Fickle code does not manipulate objects as standard Java objects but via identity and implementor fields.

11. CONCLUSIONS AND FURTHER WORK

We have defined a translation from $\mathcal{F}ickle_{\mathtt{II}}$ into $\mathcal{F}ickle^{-}_{\mathtt{II}}$ (the subset of $\mathcal{F}ickle_{\mathtt{II}}$ without re-classification), and have proven that this translation is well-behaved in the sense that it preserves static and dynamic semantics. The translation described in this paper is the basis of *Isabella*, an implementation that maps $\mathcal{F}ickle^{\mathtt{st}}$ onto Java.

We believe that the work presented in this paper is significant for (at least) two complementary reasons. On the one hand, it is a worked example of a formal description of an implementation technique, based on a few clean ideas and supported by a complete correctness proof. Though much other work would be necessary in order to have a real implementation (see discussion below), the formalization given here provides a solid starting point where many subtle problems have been already solved in a simplified context, and the proofs give us confidence in the correctness. On the other hand, the work we carried out can also be considered a nice example of how formalization can help in the development of an implementation. Let us illustrate this point more in detail.

This work started when we decided to implement $\mathcal{F}ickle_{\mathtt{II}}$ through a translation into Java. In the beginning, we expected the translation to be straightforward, and the formal work and the various prototype implementations mentioned in Sect. 4 take place in parallel. However, as soon as we started to reason about correctness, we realized that the translation had to be based on a simple formal correspondence between $\mathcal{F}ickle_{\mathtt{II}}$ objects and $\mathcal{F}ickle^{-}_{\mathtt{II}}$ objects, and also between $\mathcal{F}ickle_{\mathtt{II}}$ heaps and $\mathcal{F}ickle^{-}_{\mathtt{II}}$ heaps. These invariants were necessary for the formulation of the theorems, but also expressed concepts that arose naturally as properties of the translation. Through the investigation of these invariants, we discovered several alternative designs, as discussed in Section 4.1, and we were able to compare them on a formal basis. Therefore, we conclude that the formalization of the translation and its properties was indispensable, and that any such translation tasks should not be attempted without.

Moreover, formal reasoning allowed us to discover design errors at a very preliminary stage, *e.g.*, the translation of field assignment and method call (see Section 6.5.2).

Except for the prototype implementations, we are interested in investigating the possibility of implementing an extension of Java with re-classification. From this point of view, our translation is a good basis since it exhibits the following additional properties:

—It is fully compatible with Java separate compilation, since each $\mathcal{F}ickle_{\mathtt{II}}$ class can

be translated without having other class bodies. Therefore, it would be sufficient to have the other classes in binary form, as done by current Java compilers.

—The dependencies across classes are exactly those of standard Java compilation, in the sense that a $\mathcal{F}ickle_{\mathrm{II}}$ class can be translated only if the type information from all the ancestor and all used classes is available.

Further work includes the extension of $\mathcal{F}ickle_{\mathrm{II}}$ onto the full Java language. On the one hand, such an extension should take into account other Java features (like overloading) that, though in principle orthogonal to re-classification, should be carefully analyzed in order to be sure that the interaction behaves correctly. On the other hand, as mentioned above, an extended compiler should be able to work even in a context where only binary files are available, while our prototype implementation works on source files. Finally, the issue of inter-operability with code produced by standard Java compilation should be considered.

A. DEFINITIONS OF LOOKUP, SUBTYPES, ACYCLIC PROGRAMS, AND AGREEMENTS

Fig. 19 defines the judgment $\vdash\ p\ \diamond_u$, which guarantees that a program has unique definitions. In the judgment *defs* is defined by

$$defs ::= (\ field \mid meth\ )^*$$

The first requirement says that there should be no more than one class definition for any identifier c – note that it implicitly guarantees $c' = c''$ and that the class bodies are identical. The second requirement says that there should be no more than one field definition in c for any identifier f – note that it implicitly guarantees $t = t'$. The third requirement says that there should be a unique method definition in c for any identifier m – note that it implicitly guarantees $t = t'$, $t_1{=}t'_1$, …, $t_q{=}t_{q'}$, $x_1{=}x'_1$, …, $x_q{=}x_{q'}$, $\phi{=}\phi'$, and $block = block'$.

$$
\begin{array}{ll}
\forall c: & p = p_1\ [\texttt{root}\ \mid\ \texttt{state}]\ \texttt{class}\ c\ \texttt{extends}\ c'\texttt{\{}\ \ldots\ \texttt{\}}\ p_2,\\
& \quad p = p_3\ [\texttt{root}\ \mid\ \texttt{state}]\ \texttt{class}\ c\ \texttt{extends}\ c''\texttt{\{}\ \ldots\ \texttt{\}}\ p_4\\
& \qquad \Longrightarrow\ \ p_1 = p_3,\ p_2 = p_4\\
\forall f: & p = p_1\ [\texttt{root}\ \mid\ \texttt{state}]\ \texttt{class}\ c\ \texttt{extends}\ c'\texttt{\{}\ defs_1\ t\ f\ defs_2\ \texttt{\}}\ p_2,\\
& \quad p = p_1\ [\texttt{root}\ \mid\ \texttt{state}]\ \texttt{class}\ c\ \texttt{extends}\ c'\texttt{\{}\ defs_3\ t'\ f\ defs_4\ \texttt{\}}\ p_2\\
& \qquad \Longrightarrow\ \ defs_1 = defs_3,\ defs_2 = defs_4;\\
\forall m: & p = p_1\ [\texttt{root}\ \mid\ \texttt{state}]\ \texttt{class}\ c\ \texttt{extends}\ c'\texttt{\{}\ defs_1\ t\ m\ (t_1\ x_1, \ldots, t_q\ x_q)\ \phi\ block\ defs_2\ \texttt{\}}\ p_2,\\
& \quad p = p_1\ [\texttt{root}\ \mid\ \texttt{state}]\ \texttt{class}\ c\ \texttt{extends}\ c'\texttt{\{}\ defs_3\ t'\ m\ (t'_1\ x'_1\ \ldots, t'_n\ x'_n\ )\ \phi'\ \ block'\ \ defs_4\ \texttt{\}}\ p_2\\
& \qquad \Longrightarrow\ \ defs_1 = defs_3,\ defs_2 = defs_4\\
\hline
\vdash\ p\ \diamond_u &
\end{array}
$$

Fig. 19. Programs with unique definitions

For program p with $\vdash\ p\ \diamond_u$, class name $c \neq \texttt{Object}$, and qualifier $qual = \texttt{root}$, or $qual = \texttt{state}$, or $qual = \epsilon$, we define the lookup of the class declaration for c:

$$\mathcal{C}(p, c)\ =\ \begin{cases} qual\ \texttt{class}\ c\ \texttt{extends}\ c'\texttt{\{}defs\texttt{\}} & if\ p = p'\ qual\ \texttt{class}\ c\ \texttt{extends}\ c'\texttt{\{}cBody\texttt{\}}\ p'',\\ \mathcal{U}df & otherwise \end{cases}$$

The assertion $p \vdash c \sqsubseteq c'$, defined in Fig. 20, means that the class c is a subclass of c'. The class hierarchy in a program p is well-formed, *i.e.*, $\vdash\ p\ \diamond_h$, if the subclass relationship is acyclic, root classes extend only non-root and non-state classes, and state classes extend either root classes or state classes. Notice that $\vdash\ p\ \diamond_u$ whenever $\vdash\ p\ \diamond_h$.

It is straightforward to prove the following properties of programs with well-formed inheritance hierarchies: Two types that are in the subclass relationship are classes, the relation $\sqsubseteq$ is reflexive, transitive and antisymmetric, and the subclass hierarchy forms a tree with `Object` at its root.

The following judgments, also defined in Fig. 20, distinguish the kinds of classes: $p \vdash c\ \diamond_{ct}$ means that c is any class, $p \vdash c\ \diamond_{rt}$ means that c is a re-classifiable type *i.e.*, either a root or a state class. The judgment $p \vdash t\ \diamond_{ft}$ means that t is a non-state type, *i.e.*, either a primitive type or a non-state class.

Widening, the extension of the subclass relationship to types, is expressed by the assertion $p \vdash t \leq t'$, and is also defined by the rules in Fig. 20.

Environment lookup and update, and the least upper bound operation on types and environments are defined in Fig. 21.

For program p with $\vdash\ p\ \diamond_h$, class name c such that

$$\mathcal{C}(p, c)\ =\ [\texttt{root}\ \mid\ \texttt{state}]\ \texttt{class}\ c\ \texttt{extends}\ c'\{\mathit{defs}\},$$

field name f and method name m we define:

$$\begin{array}{lcl}
\mathcal{FD}(p, c, f) & = & \begin{cases} t & \mathit{if\ defs} = \ldots\ t\ f\ \ldots \\ \mathcal{U}\mathit{df} & \mathit{otherwise} \end{cases} \\
\mathcal{F}(p, c, f) & = & \begin{cases} \mathcal{FD}(p, c, f) & \mathit{if}\ \ \mathcal{FD}(p, c, f) \neq \mathcal{U}\mathit{df}, \\ \mathcal{F}(p, c', f) & \mathit{otherwise} \end{cases} \\
\mathcal{F}(p, \texttt{Object}, f) & = & \mathcal{U}\mathit{df} \\
\mathcal{F}s(p, c) & = & \{f \mid \mathcal{F}(p, c, f) \neq \mathcal{U}\mathit{df}\} \\
\mathcal{MD}(p, c, m) & = & \begin{cases} t\ m(t_1\ x_1, \ldots, t_n\ x_n)\ \phi\ \mathit{body} & \mathit{if\ defs} = \ldots\ t\ m\ (t_1\ x_1 \ldots t_n\ x_n)\phi\ \mathit{body} \ldots \\ \mathcal{U}\mathit{df} & \mathit{otherwise} \end{cases} \\
\mathcal{M}(p, c, m) & = & \begin{cases} \mathcal{MD}(p, c, m) & \mathit{if}\ \ \mathcal{MD}(p, c, m) \neq \mathcal{U}\mathit{df}, \\ \mathcal{M}(p, c', m) & \mathit{otherwise} \end{cases} \\
\mathcal{M}(p, \texttt{Object}, m) & = & \mathcal{U}\mathit{df}
\end{array}$$

B. PROOF OF PRESERVATION OF STATIC SEMANTICS

We write $\gamma \subseteq \gamma'$ if for all $\mathit{id}\ \gamma(\mathit{id}) \neq \mathcal{U}\mathit{df} \Rightarrow \gamma(\mathit{id}) = \gamma'(\mathit{id})$.

LEMMA B.1. *Let p be a program, γ, γ' two environments s.t. $\gamma \subseteq \gamma'$, e an expression, and t a type.*
If $p, \gamma\ \vdash\ e\ :\ t \parallel \gamma \parallel \{\ \}$, then $p, \gamma'\ \vdash\ e\ :\ t \parallel \gamma' \parallel \{\ \}$.

PROOF. First, by induction on the typing rules for expressions the following claim can be proved:

$$(*)\quad \text{if}\ \ p, \gamma\ \vdash\ e\ :\ t \parallel \gamma' \parallel \{\ \},\ \text{then}\ \ \gamma = \gamma'.$$

Then, the lemma is proved by induction on the typing rules using claim $(*)$. □

$$\frac{\vdash\ p\ \diamond_u}{p \vdash \mathit{Object} \sqsubseteq \mathit{Object}} \qquad \frac{\vdash\ p\ \diamond_u \quad p = \ldots[\texttt{root}\ \mid\ \texttt{state}\ ]\ \texttt{class}\ c\ \texttt{extends}\ c'\texttt{\{ ... \}}\ldots}{p \vdash c \sqsubseteq c \quad p \vdash c \sqsubseteq c'} \qquad \frac{p \vdash c \sqsubseteq c' \quad p \vdash c' \sqsubseteq c''}{p \vdash c \sqsubseteq c''}$$

$$\frac{\begin{array}{l}\forall c, c' : \\ \quad p \vdash c \sqsubseteq c' \ \text{ and } \ p \vdash c' \sqsubseteq c \implies c = c' \\ \quad \mathcal{C}(p,c) = \texttt{class}\ c\ \texttt{extends}\ c'\ \texttt{\{...\}} \implies \mathcal{C}(p,c') = \texttt{class}\ c' \ldots \\ \quad \mathcal{C}(p,c) = \texttt{root class}\ c\ \texttt{extends}\ c'\ \texttt{\{...\}} \implies \mathcal{C}(p,c') = \texttt{class}\ c' \ldots \\ \quad \mathcal{C}(p,c) = \texttt{state class}\ c\ \texttt{extends}\ c'\ \texttt{\{...\}} \implies \\ \qquad ((\mathcal{C}(p,c') = \texttt{root class}\ c' \ldots) \ \text{ or } \ (\mathcal{C}(p,c') = \texttt{state class}\ c' \ldots))\end{array}}{\vdash\ p\ \diamond_h}$$

$$\frac{\vdash\ p\ \diamond_h \quad \mathcal{C}(p,c) = \ \texttt{class}\ c\ldots}{p \vdash c\ \diamond_{ft} \quad p \vdash c\ \diamond_{ct}} \qquad \frac{\vdash\ p\ \diamond_h \quad \mathcal{C}(p,c) = \ \texttt{root class}\ c\ldots}{p \vdash c\ \diamond_{ft} \quad p \vdash c\ \diamond_{rt} \quad p \vdash c\ \diamond_{ct}} \qquad \frac{\vdash\ p\ \diamond_h \quad \mathcal{C}(p,c) = \ \texttt{state class}\ c\ldots}{p \vdash c\ \diamond_{rt} \quad p \vdash c\ \diamond_{ct}}$$

$$\frac{}{p \vdash \mathit{bool}\ \diamond_{ft}} \qquad \frac{}{p \vdash \mathit{bool} \leq \mathit{bool}} \qquad \frac{p \vdash c \sqsubseteq c'}{p \vdash c \leq c'}$$

Fig. 20. Subclasses, well-formed inheritance hierarchy, subtypes

$$\frac{\gamma = x_1 : t_1, \ldots, x_n : t_n, \texttt{this} : c}{\gamma(\texttt{id}) = \begin{cases} t_i & \text{if } \texttt{id} = x_i \\ c & \text{if } \texttt{id} = \texttt{this} \\ \mathcal{U}\mathit{df} & \text{otherwise} \end{cases}} \qquad \gamma[\texttt{id} \mapsto t](\texttt{id}') = \begin{cases} t & \text{if } \texttt{id}' = \texttt{id} \\ \gamma(\texttt{id}') & \text{otherwise} \end{cases}$$

$$t_1 \sqcup_p t_2 = \begin{cases} t & \text{if } p \vdash t_1 \leq t \quad p \vdash t_2 \leq t \quad \forall t'.(p \vdash t_1 \leq t' \text{ and } p \vdash t_2 \leq t') \Rightarrow p \vdash t \leq t' \\ \mathcal{U}\mathit{df} & \text{otherwise} \end{cases}$$

$$\gamma \sqcup_p \gamma' \ = \ \{id : (t \sqcup_p t') \ \mid \gamma(id) = t \text{ and } \gamma'(id) = t'\}$$

Fig. 21. Environment lookup and update, lub on types and environments

LEMMA B.2. *Let p be a program, γ, γ' two environments, e an expression, t a type, and ϕ an effect.*
If $p, \gamma \ \vdash \ e \ : \ t \ \| \ \gamma' \ \| \ \phi$ is provable, then $[\![\gamma]\!] = [\![\gamma']\!]$.

PROOF. By induction on the typing rules for expressions. □

LEMMA B.3. *Let p be a program s.t. $\vdash p \ \diamond$, c, c' two class names s.t. $p \vdash c \leq c'$, f a field name, m a method name, t, $t_1, \ldots, t_n$ types ($n \geq 0$), $x_1, \ldots, x_n$ variables, and ϕ an effect. Then*

(1) $\mathcal{F}(p, c', f) = t \Rightarrow \mathcal{F}(p, c, f) = t$

(2) $\mathcal{M}(p, c', m) = t\ m(t_1\ x_1, \ldots, t_n\ x_n)\ \phi\ \{\ \ldots\ \} \Rightarrow$
$\mathcal{M}(p, c, m) = t\ m(t_1\ x_1, \ldots, t_n\ x_n)\ \phi'\ \{\ \ldots\ \}$ *for some effect* ϕ'.

PROOF. By induction on the height of the inheritance tree. □

LEMMA B.4. *Let* p *be a program s.t.* $\vdash p\ \diamond$, c *a class name,* f *a field name,* m *a method name,* t, $t_1, \ldots, t_n$ *types* ($n \geq 0$), $x_1, \ldots, x_n$ *variables, and* ϕ *an effect. Then*

(1) $\mathcal{F}(p, c, f) = \mathcal{U}df \Rightarrow \mathcal{F}([\![p]\!]_{prog}, theName(c), f) = \mathcal{U}df$;
(2) $\mathcal{F}(p, c, f) = t \Rightarrow \mathcal{F}([\![p]\!]_{prog}, theName(c), f) = theType(t)$;
(3) $\mathcal{M}(p, c, m) = \mathcal{U}df \Rightarrow \mathcal{M}([\![p]\!]_{prog}, theName(c), m) = \mathcal{U}df$;
(4) $\mathcal{M}(p, c, m) = t\ m(t_1\ x_1, \ldots, t_n\ x_n)\ \phi\ \{\ \ldots\ \} \Rightarrow$
$\mathcal{M}([\![p]\!]_{prog}, theName(c), m) = theType(t)\ m(theType(t_1)\ x_1, \ldots, theType(t_n)\ x_n)\ \{\ \}\ \{\ \ldots\ \}$.

PROOF. By induction on the height of the inheritance tree. □

LEMMA B.5. *Let* p *be a program s.t.* $\vdash p\ \diamond$, *and* c *a class name s.t.* $p \vdash c\ \diamond_{ct}$. *Then* $\mathcal{F}([\![p]\!]_{prog}, theName(c), \texttt{id}) = \texttt{Identity}$.

PROOF. By induction on the height of the inheritance tree. □

LEMMA B.6. *Let* p *be a program s.t.* $\vdash p\ \diamond$, *and* c *a class name s.t.* $p \vdash c\ \diamond_{ct}$. *Then* $[\![p]\!]_{prog} \vdash theName(c) \leq \texttt{FickleObject}$.

PROOF. By induction on the height of the inheritance tree. □

LEMMA B.7. *Let* p *be a program.*
If $\vdash p\ \diamond$, *then* $\vdash\ [\![p]\!]_{prog}\ \diamond_h$.

PROOF. By induction on the height of the inheritance tree. □

LEMMA B.8. *Let* p *be a program, and* t, t' *two types.*
Then $p \vdash t \leq t'$ *implies* $[\![p]\!]_{prog} \vdash theType(t) \leq theType(t')$.

PROOF. Trivial. □

LEMMA B.9. *Let* p *be a program s.t.* $\vdash p\ \diamond$. *Then* $[\![p]\!]_{prog} \vdash \texttt{Identity}\ \diamond_{ct}$.

PROOF. Trivial. □

LEMMA B.10. *Let* p *be a program s.t.* $\vdash p\ \diamond$. *Then*
$\mathcal{F}([\![p]\!]_{prog}, \texttt{Identity}, \texttt{imp}) = \texttt{FickleObject}$.

PROOF. Trivial. □

LEMMA B.11. *Let* p *be a program s.t.* $\vdash p\ \diamond$, t *a type, and* ϕ *an effect.*
Then $p \vdash t \leq \phi@_p t$.

PROOF. Trivial. □

LEMMA B.12. *Let* p *be a program s.t.* $\vdash p\ \diamond$, *and* t, t' *two types.*
Then $theType(t \sqcup_p t') = theType(t) \sqcup_{[\![p]\!]_{prog}} theType(t')$.

PROOF. Trivial. □

Proof of Theorem 7.1

PROOF. The proof proceeds by induction on the typing rules (or, equivalently, on the structure of expressions) and by case analysis on the kinds of expressions and relies on the fact that the generation (or inversion) lemma for the typing relation, see Pierce [2002], is trivial since there is a one-to-one relation between typing rules and kinds of expressions.

Cases $e \equiv \texttt{true}, \texttt{false}$ are trivial.

Case $e \equiv id$.
If $id \equiv x$, then $[\![e]\!]_{expr}(p,\gamma) \equiv x$ and by hypothesis $\gamma(x) = t$, therefore $[\![\gamma]\!](x) = \mathit{theType}(t)$ and we can conclude by applying the suitable typing rule.
If $id \equiv \texttt{this}$, then $[\![e]\!]_{expr}(p,\gamma) \equiv \texttt{this.id}$ and by hypothesis $\gamma(\texttt{this}) = t$, with $p \vdash t \diamond_{ct}$ and $t \neq \texttt{Object}$; therefore $\mathit{theType}(t) = \texttt{Identity}$, $\mathit{theName}(t) = t$, $[\![\gamma]\!](\texttt{this}) = t$, and Lemma B.5 is applicable hence $\mathcal{F}([\![p]\!]_{prog}, t, \texttt{id}) = \texttt{Identity}$; we conclude by applying the suitable typing rules.

Case $e \equiv \texttt{null}$.
Then $[\![e]\!]_{expr}(p,\gamma) \equiv \texttt{null}$ and by hypothesis $p \vdash t \diamond_{ct}$, therefore $\mathit{theType}(t) = \texttt{Identity}$ and by Lemma B.9 $[\![p]\!]_{prog} \vdash \texttt{Identity} \diamond_{ct}$, hence we can conclude by applying the suitable typing rule.

Case $e \equiv \texttt{isnull}(e')$.
Then $[\![e]\!]_{expr}(p,\gamma) \equiv \texttt{isnull}([\![e']\!]_{expr}(p,\gamma))$ and by hypothesis $p, \gamma \vdash e' : c \parallel \gamma' \parallel \phi$, therefore by inductive hypothesis $[\![p]\!]_{prog}, [\![\gamma]\!] \vdash [\![e']\!]_{expr}(p,\gamma) : \texttt{Identity} \parallel [\![\gamma]\!] \parallel \texttt{\{ \}}$, hence we can conclude by applying the suitable typing rule.

Case $e \equiv \texttt{new}\ c$.
Then $[\![e]\!]_{expr}(p,\gamma) \equiv$
$\{\ \mathit{theName}(c)\ \mathit{theImp};$
$\texttt{Identity}\ \mathit{theId};$
$\mathit{theId} = \texttt{new Identity};$
$\mathit{theImp} = \texttt{new}\ \mathit{theName}(c);$
$\mathit{theImp}.\texttt{id} = \mathit{theId};$
$\mathit{theId}.\texttt{imp} = \mathit{theImp};$
theId
$\}$
where *theImp* and *theId* are chosen s.t. $\gamma(\mathit{theImp}) = \mathcal{U}df$, $\gamma(\mathit{theId}) = \mathcal{U}df$. By hypothesis $p \vdash t \diamond_{ct}$, therefore $\mathit{theType}(t) = \texttt{Identity}$; furthermore by Lemmas B.5, B.10 and B.6 it is possible to apply the suitable typing rules to derive the conclusion.

Case $e \equiv (c)\, e'$.
Then $[\![e]\!]_{expr}(p,\gamma) \equiv$
$\{\ \texttt{Identity}\ x;$
$\texttt{if}\ (\texttt{isnull}(x = [\![e']\!]_{expr}(p,\gamma)))$
$\texttt{then null}$
$\texttt{else}\ ((\mathit{theName}(c))(x.\texttt{imp})).\texttt{id}$
$\}$
where x is chosen s.t. $\gamma(x) = \mathcal{U}df$. By hypothesis $p, \gamma \vdash e' : c' \parallel \gamma' \parallel \phi$ and $t = c$, therefore $\mathit{theType}(t) = \texttt{Identity}$ and by inductive hypothesis
$[\![p]\!]_{prog}, [\![\gamma]\!] \vdash [\![e']\!]_{expr}(p,\gamma) : \texttt{Identity} \parallel [\![\gamma]\!] \parallel \texttt{\{ \}}$ and by Lemma B.1,
$[\![p]\!]_{prog}, [\![\gamma_0]\!] \vdash [\![e']\!]_{expr}(p,\gamma) : \texttt{Identity} \parallel [\![\gamma_0]\!] \parallel \texttt{\{ \}}$ where $\gamma_0 = [\![\gamma]\!][x \mapsto \texttt{Identity}]$; finally, by Lemmas B.5, B.9, B.10 and B.6, it is possible to apply the suitable typing rules in order to conclude.

Case $e \equiv e'.f$.
By hypothesis $p, \gamma \vdash e' : c \parallel \gamma' \parallel \phi$, then the translation is well-defined:
$[\![e]\!]_{expr}(p,\gamma) \equiv ((\mathit{theName}(c))\ ([\![e']\!]_{expr}(p,\gamma).\texttt{imp})).f$
By inductive hypothesis $[\![p]\!]_{prog}, [\![\gamma]\!] \vdash [\![e']\!]_{expr}(p,\gamma) : \texttt{Identity} \parallel [\![\gamma]\!] \parallel \texttt{\{ \}}$, furthermore, by hypothesis $\mathcal{F}(p,c,f) = t$, therefore by Lemma B.4, $\mathcal{F}([\![p]\!]_{prog}, \mathit{theName}(c), f) = \mathit{theType}(t)$; finally, by Lemmas B.10 and B.6, it is possible to apply the suitable typing rules in order to conclude.

Case $e \equiv e_1.f{=}e_2$.
By hypothesis $p, \gamma \vdash e_1 : c \parallel \gamma_1 \parallel \phi_1$ and $p, \gamma_1 \vdash e_2 : t \parallel \gamma_2 \parallel \phi_2$, then the translation is well-defined:

$$
\begin{array}{l}
[\![e]\!]_{expr}(p,\gamma) \equiv \{\ \texttt{Identity}\ x_1; \\
\qquad \mathit{theType}(t)\ x_2; \\
\qquad x_1 = [\![e_1]\!]_{expr}(p,\gamma); \\
\qquad x_2 = [\![e_2]\!]_{expr}(p,\gamma_1); \\
\qquad ((\phi_2 @_p \mathit{theName}(c))(x_1.\texttt{imp})).f = x_2 \\
\quad \}
\end{array}
$$

with x_1 and x_2 chosen s.t. $\gamma(x_1) = \gamma(x_2) = \mathcal{U}df$. By inductive hypothesis
$[\![p]\!]_{prog}, [\![\gamma]\!] \vdash [\![e_1]\!]_{expr}(p,\gamma) : \texttt{Identity} \parallel [\![\gamma]\!] \parallel \texttt{\{ \}}$ and
$[\![p]\!]_{prog}, [\![\gamma_1]\!] \vdash [\![e_2]\!]_{expr}(p,\gamma_1) : \mathit{theType}(t) \parallel [\![\gamma_1]\!] \parallel \texttt{\{ \}}$
furthermore by Lemma B.2, $[\![\gamma]\!] = [\![\gamma_1]\!]$, and by Lemma B.1,
$[\![p]\!]_{prog}, \gamma_0 \vdash [\![e_1]\!]_{expr}(p,\gamma) : \texttt{Identity} \parallel \gamma_0 \parallel \texttt{\{ \}}$ and
$[\![p]\!]_{prog}, \gamma_0 \vdash [\![e_2]\!]_{expr}(p,\gamma_1) : \mathit{theType}(t) \parallel \gamma_0 \parallel \texttt{\{ \}}$
with $\gamma_0 = [\![\gamma]\!][x_1 \mapsto \texttt{Identity}, x_2 \mapsto \mathit{theType}(t)]$.
Again, by hypothesis $\mathcal{F}(p, \phi_2 @_p c, f) = t'$ and $p \vdash t \leq t'$, and by definition $\mathit{theName}(\phi_2 @_p c) = \phi_2 @_p \mathit{theName}(c)$, therefore by Lemmas B.4 and B.8, $\mathcal{F}([\![p]\!]_{prog}, \phi_2 @_p \mathit{theName}(c), f) = \mathit{theType}(t')$ and $[\![p]\!]_{prog} \vdash \mathit{theType}(t) \leq \mathit{theType}(t')$. Finally, by Lemmas B.10 and B.6, it is possible to apply the suitable typing rules in order to conclude.

Case $e \equiv x = e_1$.
Then $[\![e]\!]_{expr}(p,\gamma) \equiv x = [\![e_1]\!]_{expr}(p,\gamma)$
By hypothesis $p, \gamma \vdash e_1 : t \parallel \gamma' \parallel \phi$, $\gamma'(x) = t'$ and $p \vdash t \leq t'$, therefore by inductive hypothesis and Lemmas B.2 and B.8, $[\![p]\!]_{prog}, [\![\gamma]\!] \vdash [\![e_1]\!]_{expr}(p,\gamma) : \mathit{theType}(t) \parallel [\![\gamma]\!] \parallel \texttt{\{ \}}$, $[\![\gamma]\!] = [\![\gamma']\!]$ hence $[\![\gamma]\!](x) = \mathit{theType}(t')$, and $\vdash \mathit{theType}(t) \leq \mathit{theType}(t')$. Finally, it is possible to apply the suitable typing rule in order to conclude.

Case $e \equiv e_0.m(e_1, \ldots, e_n)$.
By hypothesis, $p, \gamma \vdash e_0 : c \parallel \gamma_0 \parallel \phi_0$ and $p, \gamma_{i-1} \vdash e_i : t_i \parallel \gamma_i \parallel \phi_i$ for all $i = 1, \ldots, n$, then the translation is well-defined:

$$
\begin{array}{l}
[\![e]\!]_{expr}(p,\gamma) \equiv \{\ \texttt{Identity}\ x; \\
\qquad \mathit{theType}(t_1)\ x_1; \\
\qquad \ldots \\
\qquad \mathit{theType}(t_n)\ x_n; \\
\qquad x = [\![e_0]\!]_{expr}(p,\gamma); \\
\qquad x_1 = [\![e_1]\!]_{expr}(p,\gamma_0); \\
\qquad \ldots \\
\qquad x_n = [\![e_n]\!]_{expr}(p,\gamma_{n-1}); \\
\qquad (((\phi_1 \cup \cdots \cup \phi_n) @_p \mathit{theName}(c))(x.\texttt{imp})).m(x_1, \ldots, x_n) \\
\quad \}
\end{array}
$$

with $x, x_1, \ldots, x_n$ chosen s.t. $\gamma(x) = \gamma(x_1) = \cdots = \gamma(x_n) = \mathcal{U}df$. By Lemma B.2,

$[\![\gamma]\!] = [\![\gamma_0]\!] = \cdots = [\![\gamma_n]\!]$, and by inductive hypothesis and Lemma B.1,
$[\![p]\!]_{prog}, \gamma' \vdash [\![e_0]\!]_{expr}(p,\gamma)$: `Identity` $\| \gamma' \|$ `{ }` and
$[\![p]\!]_{prog}, \gamma' \vdash [\![e_i]\!]_{expr}(p,\gamma_{i-1})$: $theType(t_i) \| \gamma' \|$ `{ }` for all $i = 1,\ldots,n$, with
$\gamma' = [\![\gamma]\!][x \mapsto$ `Identity`$, x_1 \mapsto theType(t_1), \ldots, x_n \mapsto theType(t_n)]$.
Again by hypothesis, $\mathcal{M}(p, (\phi_1 \cup \cdots \cup \phi_n)@_p c, m) = t\ m(t'_1\ x_1, \ldots, t'_n\ x_n)\ \phi$ `{ ... }`
and $p \vdash (\phi_{i+1} \cup \cdots \cup \phi_n)@_p t_i \leq t'_i$ for all $i = 1,\ldots,n$, and by definition $theName((\phi_1 \cup \cdots \cup \phi_n)@_p c) = (\phi_1 \cup \cdots \cup \phi_n)@_p theName(c)$ and `{ }`$@_{[\![p]\!]_{prog}} theType(t_i) = theType(t_i)$ for all $i = 1,\ldots,n$, therefore by Lemmas B.4, B.8 and B.11 and transitivity of the subtyping relation,
$\mathcal{M}([\![p]\!]_{prog}, (\phi_1 \cup \cdots \cup \phi_n)@_p theName(c), m) =$
$\quad theType(t)\ m(theType(t'_1)\ x_1, \ldots, theType(t'_n)\ x_n)$ `{ } { ... }` and
$[\![p]\!]_{prog} \vdash$ `{ }`$@_{[\![p]\!]_{prog}} theType(t_i) \leq theType(t'_i)$ for all $i = 1,\ldots,n$. Finally, by Lemmas B.10 and B.6 it is possible to apply the suitable typing rules in order to conclude.

Case $e \equiv id!!c$.
Let us consider the case $id \equiv x$ (the case $id \equiv$ `this` is analogous); the translation is defined by

```
[[e]]_expr(p,γ) ≡ { Identity theId;
                    theName(c) theImp;
                    R(p,c) theLastImp;
                    if (isnull(theId = x)) then null
                     else { theImp = new theName(c);
                            theLastImp = (R(p,c))(theId.imp);
                            theId.imp = theImp;
                            theImp.id = theId;
                            theImp.f_1 = theLastImp.f_1;
                            ...
                            theImp.f_r = theLastImp.f_r
                          };
                    theId
                  }
```

with *theId*, *theImp* and *theLastImp* chosen s.t. $\gamma(theImp) = \gamma(theId) = \gamma(theLastImp) = \mathcal{U}df$ and $\{f_1,\ldots,f_r\} = \mathcal{F}s(p, \mathcal{R}(p,c))$.
By hypothesis, $p \vdash c \diamond_{rt} \mathcal{R}(p,c) = \mathcal{R}(p,\gamma(x))$. By definition, $f_i \in \mathcal{F}s(p,\mathcal{R}(p,c)) \Rightarrow \exists t\ \mathcal{F}(p,\mathcal{R}(p,c),f_i) = t$ for all $i = 1,\ldots,r$; furthermore, since by definition $p \vdash c \leq \mathcal{R}(p,c)$, by Lemma B.3, $\mathcal{F}(p,c,f_i) = t$ for all $i = 1,\ldots,r$. Since by definition, $theName(\mathcal{R}(p,c)) = \mathcal{R}(p,c)$, by Lemma B.4, $\mathcal{F}([\![p]\!]_{prog}, \mathcal{R}(p,c), f_i) = \mathcal{F}([\![p]\!]_{prog}, theName(c), f_i) = theType(t)$ for all $i = 1,\ldots,r$. Finally, by Lemmas B.5, B.9, B.10 and B.6 the suitable typing rules can be applied in order to conclude.

Case $e \equiv$ `if` e_0 `then` e_1 `else` e_2.
By hypothesis,
$p, \gamma \vdash e_0$: `bool` $\| \gamma_0 \| \phi_0$
$p, \gamma_0 \vdash e_1 : t_1 \| \gamma_1 \| \phi_1$
$p, \gamma_0 \vdash e_2 : t_2 \| \gamma_2 \| \phi_2$
Then the translation is well-defined:
$[\![e]\!]_{expr}(p,\gamma) \equiv$ `if` $[\![e_0]\!]_{expr}(p,\gamma)$ `then` $[\![e_1]\!]_{expr}(p,\gamma_0)$ `else` $[\![e_2]\!]_{expr}(p,\gamma_0)$
By Lemma B.2, $[\![\gamma]\!] = [\![\gamma_0]\!] = [\![\gamma_1]\!] = [\![\gamma_2]\!]$ and by inductive hypothesis,

$[\![p]\!]_{prog}, [\![\gamma]\!] \vdash [\![e_0]\!]_{expr}(p,\gamma) : \texttt{bool} \parallel [\![\gamma]\!] \parallel \{\,\}$
$[\![p]\!]_{prog}, [\![\gamma]\!] \vdash [\![e_1]\!]_{expr}(p,\gamma_0) : \mathit{theType}(t_1) \parallel [\![\gamma]\!] \parallel \{\,\}$
$[\![p]\!]_{prog}, [\![\gamma]\!] \vdash [\![e_2]\!]_{expr}(p,\gamma_0) : \mathit{theType}(t_2) \parallel [\![\gamma]\!] \parallel \{\,\}$
Finally, it is possible to conclude by applying the suitable typing rule and Lemma B.12.

Case $e \equiv \{t_1\ x_1;\ \ldots t_s\ x_s;\ e_1;\ \ldots e_n\}$.
By hypothesis,
$\gamma_0 = \gamma[x_1 \mapsto t_1, \ldots, x_s \mapsto t_s]$
$p, \gamma_{i-1} \vdash e_i : t'_i \parallel \gamma_i \parallel \phi_i$ for all $i = 1, \ldots, n$
Then the translation is well-defined:
$[\![e]\!]_{expr}(p,\gamma) \equiv \{[\![t_1\ x_1]\!]_{var}; \ldots [\![t_s\ x_s]\!]_{var};\ [\![e_1]\!]_{expr}(p,\gamma_0); \ldots [\![e_n]\!]_{expr}(p,\gamma_{n-1})\}$
By Lemma B.2, $[\![\gamma_0]\!] = \cdots [\![\gamma_n]\!]$ and by translation of variable declarations and by inductive hypothesis,
$[\![p]\!]_{prog}, [\![\gamma_0]\!] \vdash [\![e_i]\!]_{expr}(p,\gamma_{i-1}) : \mathit{theType}(t'_i) \parallel [\![\gamma_0]\!] \parallel \{\,\}$ for all $i = 1, \ldots, n$
where, by definition, $[\![\gamma_0]\!] = [\![\gamma]\!][x_1 \mapsto \mathit{theType}(t_1), \ldots, x_s \mapsto \mathit{theType}(t_s)]$. Finally, it is possible to conclude by applying the suitable typing rule. □

Proof of Theorem 7.2

Proof. Let us take an arbitrary class named c and defined in p (recall that in the translation of programs – see comments in Section 6.1 – we assume no name conflicts, so that $c \neq \texttt{Identity}, \texttt{FickleObject}$ and $f \neq \texttt{id}$ for any field f declared in p); by translation of class and field declarations, for any type t'_f, and field name $f \neq \texttt{id}$, $\mathcal{FD}([\![p]\!]_{prog}, c, f) = t'_f$ implies $\mathcal{FD}(p, c, f) = t_f$, for a type t_f s.t. $\mathit{theType}(t_f) = t'_f$. By definition of $\mathit{theType}$, $[\![p]\!]_{prog} \vdash t'_f \diamond_{ft}$; furthermore, by hypothesis, $\mathcal{F}(p, c', f) = \mathcal{U}df$, therefore, by Lemma B.4, $\mathcal{F}([\![p]\!]_{prog}, \mathit{theName}(c'), f) = \mathcal{U}df$.

By translation of class and method declarations, for all types $t'_0, \ldots, t'_n (n \geq 0)$, method name m, variables $x_1, \ldots, x_n$, effect ϕ_0, and block block',
$\mathcal{MD}([\![p]\!]_{prog}, c, m) = t'_0\ m(t'_1\ x_1, \ldots, t'_n\ x_n)\ \phi_0\ \mathit{block}' \Rightarrow$
$\phi_0 = \{\,\}$ and $\mathcal{MD}(p, c, m) = t\ m(t_1\ x_1, \ldots, t_n\ x_n)\ \phi\ \mathit{block}$,
for types $t, t_1, \ldots, t_n$, s.t. $\mathit{theType}(t) = t'_0$, $\mathit{theType}(t_i) = t'_i$ for $i = 1, \ldots, n$, an effect ϕ, and a block block s.t. $[\![\mathit{block}]\!]_{expr}(p,\gamma) = \mathit{block}'$, with $\gamma = t_1\ x_1, \ldots, t_n\ x_n,\ c\ \texttt{this}$.

Therefore $p \vdash \phi_0 \diamond$ trivially holds, $[\![p]\!]_{prog}, [\![\gamma]\!] \vdash \mathit{block}' : \mathit{theType}(t') \parallel [\![\gamma]\!] \parallel \{\,\}$ is provable by hypothesis and Theorem 7.1, $[\![p]\!]_{prog} \vdash \mathit{theType}(t') \leq t'_0$ holds by hypothesis and Lemma B.8 and trivially $\{\,\} \subseteq \phi_0$, therefore we can conclude by applying the typing rule for class declarations. □

Proof of Theorem 7.3

Proof. By Lemma B.7, $\vdash [\![p]\!]_{prog} \diamond_h$; by definition of `Identity` and `FickleObject`, and by the typing rules, $[\![p]\!]_{prog} \vdash \texttt{Identity} \diamond$ and $[\![p]\!]_{prog} \vdash \texttt{FickleObject} \diamond$. Moreover, every class $c \neq \texttt{Identity}, \texttt{FickleObject}$ defined in $[\![p]\!]_{prog}$ is defined also in p (by definition of translation of programs and classes), and by Theorem 7.2 we have $[\![p]\!]_{prog} \vdash c \diamond$. Hence we can conclude the result from the typing rule for programs. □

C. PROOF OF PRESERVATION OF DYNAMIC SEMANTICS

Lemma C.1. *Let p be a program s.t. $\vdash p \diamond$, c, c' two class names. If $p \vdash c \leq c'$, then $p \vdash \mathit{theName}(c) \leq \mathit{theName}(c')$.*

PROOF. If both c and c' are different from `Object` then it is obvious since $\mathit{theName}(c) = c$ and $\mathit{theName}(c') = c'$. If either one is equal to `Object` then since $p \vdash c \leq c'$ it must be $c' = \texttt{Object}$. Therefore, $\mathit{theName}(c') = \texttt{FickleObject}$ and for all classes d we have that $p \vdash d \leq \texttt{FickleObject}$. □

LEMMA C.2. *Let t be a type. If v is the initial value of type t then v is also the initial value of type $\mathit{theType}(t)$.*

PROOF. If t is a class type then $v = \texttt{null}$ which is initial also for `Identity`, otherwise t is a primitive type and $\mathit{theType}(t) = t$. □

LEMMA C.3. *Let e be an expression such that: $p, \gamma \vdash e : t \parallel \gamma' \parallel \phi$, and σ be a store such that $[\![p]\!], [\![\gamma]\!][\texttt{this} \mapsto c] \vdash \sigma \diamond$, for some c. If*

$$[\![e]\!], \sigma \underset{[\![p]\!]}{\rightsquigarrow} w, \sigma'$$

then for all x, $\gamma(x) = \mathit{Udf}$ implies $\sigma'(x) = \sigma(x)$.

PROOF. By induction on the depth of the derivation tree of $[\![e]\!], \sigma \underset{[\![p]\!]}{\rightsquigarrow} w, \sigma'$. For the translation of the expressions corresponding to values, variables, `this`, null test, and field selection $[\![e]\!]$ does not contain assignments, therefore for all x, $\sigma(x) = \sigma'(x)$.

For the translation of assignment, note that for $x = e'$ to be well-typed, it must be $\gamma(x) \neq \mathcal{U}\mathit{df}$. So since $[\![e]\!] \stackrel{\Delta}{=} x = [\![e']\!]$, the result is by induction hypothesis on $[\![e']\!]$.

For all the other expressions note that the translation of the expression is a block in which the local variables y are such that $\gamma(y) = \mathcal{U}\mathit{df}$. From the operational semantics of blocks we have that for all the local variables y, $\sigma'(y) = \sigma(y)$ (for all the local variables the value before execution of the block is restored). So the result is by induction hypotheses on the subexpressions. □

Proof of Theorem 7.5

Proof of (1): if $e, \sigma \underset{p}{\rightsquigarrow} w, \sigma'$ and $(\mathit{addr}(\sigma') - \mathit{addr}(\sigma)) \cap \mathit{addr}(\sigma_1) = \emptyset$, then there is σ_1' such that $[\![e]\!], \sigma_1 \underset{[\![p]\!]}{\rightsquigarrow} w, \sigma_1'$. By induction on the depth of the derivation tree of the judgement: $e, \sigma \underset{p}{\rightsquigarrow} w, \sigma'$. Note that, since e is well-typed from Theorem 3.1 either w is a value or $w \in \{\texttt{castExc}, \texttt{nullPntrExc}\}$, and the rules that may have been used are the rules in Fig. 3, Fig. 4, Fig. 5, and Fig. 6.

We will consider a subset of the rules applied. Namely: (*id*), (*new*), (*recl*), (*a-field*), (*meth*), (*cast*), (*n-cast*), (*a-field-null*), and (*e-cast*). The other cases are similar, and for the propagation rules the result is derived directly from the inductive hypothesis.

Consider rule (*id*): $id, \sigma \underset{p}{\rightsquigarrow} \sigma(id), \sigma$. There are two cases: $\texttt{id} = x$, and $id = \texttt{this}$. In the first case since $[\![x]\!]_{expr}(p, \gamma) \stackrel{\Delta}{=} x$ we derive that $[\![x]\!], \sigma_1 \underset{[\![p]\!]}{\rightsquigarrow} \sigma_1(x), \sigma_1$. Moreover, $p, \gamma \vdash \sigma \approx \sigma_1$ implies that $\sigma(x) = \sigma_1(x)$. In the second case $[\![\texttt{this}]\!]_{expr}(p, \gamma) \stackrel{\Delta}{=}$ `this.id` again from $p, \gamma \vdash \sigma \approx \sigma_1$ we have that $\sigma(\texttt{this}) = \sigma_1(\sigma_1(\texttt{this}))(\texttt{id})$. Applying rule (*field*) we have that $[\![\texttt{this.id}]\!], \sigma_1 \underset{[\![p]\!]}{\rightsquigarrow} \sigma_1(\sigma_1(\texttt{this}))(\texttt{id}), \sigma_1$ which proves the result.

Consider rule (*new*):

$$\texttt{new } c, \sigma \underset{p}{\rightsquigarrow} \iota, \sigma[\iota \mapsto [\![f_1 : v_1, \ldots, f_r : v_r]\!]^{\,c}]$$

where ι is new in σ and in σ_1, and v_l is initial for $\mathcal{F}(p, c, f_l)$, $1 \leq l \leq r$. Assume that $c \neq \texttt{Object}$, so $\mathit{theName}(c) = c$. (The proof for the case $c = \texttt{Object}$ is simpler since $\mathcal{F}s(p, c) = \emptyset$.) From the definition of the translation $[\![\texttt{new } c]\!](p, \gamma) \stackrel{\Delta}{=} \{c\ \mathit{theImp}; \texttt{Identity}\ \mathit{theId}; e_1; e_2; e_3; e_4; \mathit{theId}\}$ where

—e_1 is $\mathit{theId} = \texttt{new Identity}$,

—e_2 is $\mathit{theImp} = \texttt{new } c$,

—e_3 is $\mathit{theImp}.\texttt{id} = \mathit{theId}$, and

—e_4 is $\mathit{theId}.\texttt{imp} = \mathit{theImp}$.

and $\gamma(\mathit{theId}) = \gamma(\mathit{theImp}) = \mathit{Udf}$.
Let $\sigma_1'' = \sigma_1[\mathit{theImp} \mapsto \texttt{null}, \mathit{theId} \mapsto \texttt{null}]$. From the definition of the operational semantics of blocks, and the fact that `null` is the initial value for an object, we have that

—$e_1, \sigma_1'' \underset{[\![p]\!]}{\rightsquigarrow} \iota, \sigma_2$ and
$\sigma_2 = \sigma_1''[\iota \mapsto [\![\texttt{imp} : \texttt{null}]\!]^{\texttt{Id}}, \mathit{theId} \mapsto \iota]$

—$e_2, \sigma_2 \underset{[\![p]\!]}{\rightsquigarrow} \iota, \sigma_3$ where $\iota' \neq \iota$ is new in σ_1 (and, therefore, also in σ), and, since c extends `FickleObject`,
$\sigma_3 = \sigma_2[\iota' \mapsto [\![\texttt{id} : \texttt{null}, f_1 : v_1', \ldots, f_r : v_r']\!]^{\,c}, \mathit{theImp} \mapsto \iota']$ and v_l' is initial for $\mathit{theType}(\mathcal{F}(p, c, f_l))$, $1 \leq l \leq r$,

—$e_3; e_4, \sigma_3 \underset{[\![p]\!]}{\rightsquigarrow} \iota, \sigma_1'''$, where σ_1''' is
$\sigma_1[\iota' \mapsto [\![\texttt{id} : \iota, f_1 : v_1', \ldots, f_r : v_r']\!]^{\,c}, \mathit{theImp} \mapsto \iota', \iota \mapsto [\![\texttt{imp} : \iota']\!]^{\texttt{Id}}, \mathit{theId} \mapsto \iota]$.

Let $\sigma_1' = \sigma_1'''[\mathit{theImp} \mapsto \sigma_1(\mathit{theImp}), \mathit{theId} \mapsto \sigma_1(\mathit{theId})]$

$$[\![\texttt{new } c]\!], \sigma_1 \underset{[\![p]\!]}{\rightsquigarrow} \iota, \sigma_1'$$

Let $\sigma' = \sigma[\iota \mapsto [\![f_1 : v_1, \ldots, f_r : v_r]\!]^{\,c}]$. For all x, $\sigma'(x) = \sigma(x)$, and $\sigma_1'(x) = \sigma_1(x)$ (by definition of σ_1'). Therefore, since $p, \gamma \vdash \sigma \approx \sigma_1$ clauses 1. and 2. of Definition 7.4 hold for σ' and σ_1'. Moreover, for all ι'', $\iota'' \neq \iota$, $\sigma'(\iota'') = \sigma(\iota'')$ and $\iota'' \neq \iota'$, implies also $\sigma_1'(\iota'') = \sigma_1(\iota'')$. So we have to consider only ι.

—$\sigma'(\iota) = [\![f_1 : v_1, \ldots, f_r : v_r]\!]^{\,c}$,

—$\sigma_1'(\iota) = [\![\texttt{imp} : \iota']\!]^{\texttt{Id}}$, $\sigma_1'(\iota') = [\![\texttt{id} : \iota, f_1 : v_1', \ldots, f_r : v_r']\!]^{\,c}$

Since we assumed that $\sigma(\iota') = \mathit{Udf}$, and from Lemma C.2, we derive that $v_i' = v_i$, $1 \leq i \leq r$, then also clause 3. of Definition 7.4 is verified, and we conclude that $p, \gamma' \vdash \sigma' \approx \sigma_1'$.

Consider rule (*recl*), and assume that the expression is $\texttt{this}\,!!\,d$, therefore:

$$\frac{\begin{array}{l}\sigma(\texttt{this}) = \iota \\ \sigma(\iota) = [\![\ldots]\!]^{\,c} \\ \mathcal{F}s(p, \mathcal{R}(p,c)) = \{f_1, \ldots, f_r\} \\ v_l = \sigma(\iota)(f_l) \;\; (l \in \{1, \ldots, r\}) \\ \mathcal{F}s(p,d) \setminus \{f_1, \ldots, f_r\} = \{f_{r+1}, \ldots, f_{r+q}\} \\ v_l \text{ initial for } \mathcal{F}(p,d,f_l) \quad (l \in \{r+1, \ldots, r+q\})\end{array}}{\texttt{this}\,!!\,d, \sigma \underset{p}{\rightsquigarrow} \iota, \sigma[\iota \mapsto [\![f_1 : v_1, \ldots, f_{r+q} : v_{r+q}]\!]^{\,d}]} \tag{3}$$

Since the expression is well-typed, and `Object` is not a root or state class, we have that $d \neq \texttt{Object}$, and so $\mathit{theName}(d) = d$. From the definition of the translation we have:

$$[\![\texttt{this}\,!!\,d]\!](p,\gamma) \triangleq \{d\ \mathit{theImp}; \texttt{Identity}\ \mathit{theId}; \mathcal{R}(p,d)\ \mathit{theLastThis}; e_1; e_2; e_3; e_4; e'_1; \cdots; e'_r; \mathit{theId}\}$$

where

—e_1 is $\mathit{theId} = \texttt{this.id}$,
—e_2 is $\mathit{theLastThis} = (\mathcal{R}(p,d))\mathit{theId}.\texttt{imp}$,
—e_3 is $\mathit{theImp} = \texttt{new}\ d$,
—e_4 is $\mathit{theImp}.\texttt{id} = \mathit{theId}$,
—e_5 is $\mathit{theId}.\texttt{imp} = \mathit{theImp}$,
—e'_i is $\mathit{theImp}.f_i = \mathit{theLastThis}.f_i$ for $1 \leq i \leq r$.

and $\gamma(\mathit{theId}) = \gamma(\mathit{theLastThis}) = \gamma(\mathit{theImp}) = \mathit{Udf}$. From the fact that $p, \gamma \vdash \sigma \approx \sigma_1$, and rule (3) we have that:

$$\begin{array}{l}\sigma_1(\texttt{this}) = \iota' \quad \sigma_1(\iota') = [\![\texttt{id} : \iota, \ldots]\!]^{\,c'} \quad \sigma_1(\iota) = [\![\texttt{imp} : \iota'']\!]^{\,\texttt{Id}} \\ \sigma_1(\iota'') = [\![\texttt{id} : \iota, f_1 : v_1, \ldots, f_r : v_r, f_{r+1} : v'_1, \ldots, f_{r+p} : v'_p]\!]^{\,c}\end{array} \tag{4}$$

and $v'_i = \sigma(\texttt{this})(f_{r+i})$ for $1 \leq i \leq p$.

Let $\sigma''_1 = \sigma_1[\mathit{theImp} \mapsto \texttt{null}, \mathit{theId} \mapsto \texttt{null}, \mathit{theLastThis} \mapsto \texttt{null}]$.From the definition of the operational semantics of blocks, the fact that `null` is the initial value for an object, we have that:

—$e_1, \sigma''_1 \underset{[\![p]\!]}{\rightsquigarrow} \iota, \sigma'''_1$ where $\sigma'''_1 = \sigma''_1[\mathit{theId} \mapsto \iota]$
—$e_2, \sigma'''_1 \underset{[\![p]\!]}{\rightsquigarrow} \iota'', \sigma_2$ where $\sigma_2 = \sigma'''_1[\mathit{theLastThis} \mapsto \iota'']$ since $\mathit{theId}.\texttt{imp}, \sigma'''_1 \underset{[\![p]\!]}{\rightsquigarrow} \iota'', \sigma'''_1$ and from the hypothesis that $\texttt{this}\,!!\,d$ is well-typed we have that $c \leq \mathcal{R}(p,d)$, so also $(\mathcal{R}(p,d))\mathit{theId}.\texttt{imp}, \sigma'''_1 \underset{[\![p]\!]}{\rightsquigarrow} \iota'', \sigma'''_1$ (no cast exception may occur).
—$e_3, \sigma_2 \underset{[\![p]\!]}{\rightsquigarrow} \iota''', \sigma_3$ where ι''' is new in σ_1 (and, therefore, also in σ), and, since d extends `FickleObject`, $\sigma_3 = \sigma_2[\iota''' \mapsto [\![\texttt{id} : \texttt{null}, f_1 : v''_1, \ldots, f_r : v''_r, f_{r+1} : v''_{r+1}, \ldots, f_{r+q} : v''_{r+q}]\!]^{\,d}, \mathit{theImp} \mapsto \iota''']$ where v''_l is initial for $\mathit{theType}(\mathcal{F}(p,c,f_l))$, $l \in \{1, \ldots, r+q\}$,
—$e_4; e_5, \sigma_3 \underset{[\![p]\!]}{\rightsquigarrow} \iota, \sigma_4$, where σ_4 is $\sigma_3[\iota \mapsto [\![\texttt{imp} : \iota''']\!]^{\,\texttt{Id}}, \iota''' \mapsto [\![\texttt{id} : \iota, f_1 : v''_1, \ldots, f_r : v''_r, f_{r+1} : v''_{r+1}, \ldots, f_{r+q} : v''_{r+q}]\!]^{\,d}]$, and
—$e'_1; \cdots; e'_r, \sigma_4 \underset{[\![p]\!]}{\rightsquigarrow} \iota, \sigma'''_1$ where from (4) $\sigma'''_1 = \sigma_4[\iota''' \mapsto [\![\texttt{id} : \iota, f_1 : v_1, \ldots, f_r : v_r, f_{r+1} : v''_{r+1}, \ldots, f_{r+q} : v''_{r+q}]\!]^{\,d}]$.

Let $\sigma_1' = \sigma_1'''[\mathit{theImp} \mapsto \sigma_1(\mathit{theImp}), \mathit{theId} \mapsto \sigma_1(\mathit{theId}), \mathit{theLastThis} \mapsto \sigma_1(\mathit{theLastThis})]$.

$$[\![\texttt{this} \,!!\, d]\!], \sigma_1 \;\overset{}{\rightsquigarrow}_{[\![p]\!]}\; \iota, \sigma_1'$$

Let $\sigma' = \sigma[\iota \mapsto [\![[f_1 : v_1, \ldots, f_{r+q} : v_{r+q}]\!]^{\,c}]$. We have to show that, $p, \gamma' \vdash \sigma' \approx \sigma_1'$. First notice that from Theorem 3.1, $p, \gamma' \vdash \sigma' \diamond$, and since $[\![p]\!], [\![\gamma]\!][\texttt{this} \mapsto c'] \vdash \sigma_1 \diamond$, see (4), also $[\![p]\!], [\![\gamma']\!][\texttt{this} \mapsto c'] \vdash \sigma_1' \diamond$. So we have to show that the three clauses of Definition 7.4 are satisfied. From the definition σ' and σ_1', we have that for all x, $\sigma'(x) = \sigma(x)$, and $\sigma_1'(x) = \sigma_1(x)$. Therefore, clause 1. of Definition 7.4 holds for σ' and σ_1', and since $\sigma_1'(\texttt{this})(\texttt{id}) = \sigma_1(\texttt{this})(\texttt{id})$, then also clause 2. holds. To conclude the proof, we have to verify clause 3. for the address ι.

—$\sigma'(\iota) = [\![[f_1 : v_1, \ldots, f_{r+q} : v_{r+q}]\!]^{\,c}$, $\sigma_1'(\iota) = [\![[\texttt{imp} : \iota'']\!]^{\,\texttt{Id}}$,
—$\sigma_1'(\iota'') = [\![[\texttt{id} : \iota, f_1 : v_1, \ldots, f_r : v_r, f_{r+1} : v''_{r+1}, \ldots, f_{r+q} : v''_{r+q}]\!]^{\,d}$

where for $1 \leq i \leq q$, v_{r+i} is an initial value for $\mathcal{F}(p, c, f_{r+i})$ and v''_{r+i} is an initial value for $\mathit{theType}(\mathcal{F}(p, c, f_{r+i}))$. Therefore, from Lemma C.2, $v_{r+i} = v''_{r+i}$, and clause 3. holds. This concludes the proof that $p, \gamma' \vdash \sigma' \approx \sigma_1'$.
The proof for the case in which the rule (*recl*) is applied to $x \,!!\, d$ is similar.

Consider rule (*n-recl*). In this case the expression must be $x \,!!\, d$, since $p, \gamma \vdash \sigma \diamond$, see Fig. 9, implies that $\sigma(\texttt{this}) \neq \texttt{null}$. So $\sigma(x) = \texttt{null}$ and

$$x \,!!\, d, \sigma \;\overset{}{\rightsquigarrow}_{p}\; \texttt{null}, \sigma.$$

From the definition of the translation (as for the case (*recl*), we have that $\mathit{theName}(d) = d$)

$$[\![x \,!!\, d]\!](p, \gamma) \triangleq \{d\ \mathit{theImp}; \texttt{Identity}\ \mathit{theId}; \mathcal{R}(p, d)\ \mathit{theLastImp}; \texttt{if}\ e_1\ \texttt{then null else}\ e_2;\ \mathit{theId}\}$$

where

—e_1 is $\texttt{isnull}(\mathit{theId} = x)$,
—e_2 is

$$\begin{array}{l}\{\mathit{theImp} = \texttt{new}\ d; \mathit{theLastImp} = (\mathcal{R}(p, d))\mathit{theId}.\texttt{imp}; \\ \mathit{theImp}.\texttt{id} = \mathit{theId}; \mathit{theId}.\texttt{imp} = \mathit{theImp}; \\ e_1'; \ldots e_r' \\ \}\end{array}$$

where e_i' is $\mathit{theImp}.f_i = \mathit{theLastImp}.f_i$ for $1 \leq i \leq r$.

and $\gamma(\mathit{theId}) = \gamma(\mathit{theLastImp}) = \gamma(\mathit{theImp}) = \mathit{Udf}$.

Let $\sigma_1'' = \sigma_1[\mathit{theImp} \mapsto \texttt{null}, \mathit{theId} \mapsto \texttt{null}, \mathit{theLastImp} \mapsto \texttt{null}]$. Observe that, from $p, \gamma \vdash \sigma \approx \sigma_1$ we derive that $\sigma_1(x) = \texttt{null}$, and so also $\sigma_1''(x) = \texttt{null}$. From the definition of the operational semantics of blocks, and the fact that `null` is the initial value for an object, we have that:

$$e_1, \sigma_1'' \;\overset{}{\rightsquigarrow}_{[\![p]\!]}\; \texttt{true}, \sigma_1''$$

Therefore,

—$\texttt{if}\ e_1\ \texttt{then null else}\ e_2;\ \mathit{theId}, \sigma_1'' \;\overset{}{\rightsquigarrow}_{[\![p]\!]}\; \texttt{null}, \sigma_1''$, and

—$[\![x \,!!\, d]\!], \sigma_1 \overset{\rightarrow}{\underset{[\![p]\!]}{\sim}} \texttt{null}, \sigma_1$ since
$\sigma_1' = \sigma_1''[\mathit{theImp} \mapsto \sigma_1(\mathit{theImp}), \mathit{theId} \mapsto \sigma_1(\mathit{theId}), \mathit{theLastImp} \mapsto \sigma_1(\mathit{theLastImp})] = \sigma_1$, implies $\sigma_1' = \sigma_1$.

Consider rule (*a-field*):

$$\frac{\begin{array}{l}\alpha.\ e, \sigma \underset{p}{\rightsquigarrow} \iota, \sigma'' \\ \beta.\ e', \sigma'' \underset{p}{\rightsquigarrow} v, \sigma''' \\ \chi.\ \sigma'''(\iota)(f) \neq \mathcal{U}df \quad \sigma' = \sigma'''[\iota \mapsto \sigma'''(\iota)[f \mapsto v]]\end{array}}{e.f = e', \sigma \underset{p}{\rightsquigarrow} v, \sigma'} \tag{5}$$

From the fact that the expression is well-typed, we have

$$\frac{\begin{array}{l}a.\ p, \gamma \vdash e : c \parallel \gamma_0 \parallel \phi_0 \\ b.\ p, \gamma_0 \vdash e' : t \parallel \gamma' \parallel \phi' \\ c.\ \mathcal{F}(p, \phi' @_p c, f) = t' \quad p \vdash t \leq t' \quad \phi = \phi \cup \phi'\end{array}}{p, \gamma \vdash e.f = e' : t \parallel \gamma' \parallel \phi} \tag{6}$$

Note that, since $\mathcal{F}s(p, \texttt{Object}) = \emptyset$, $c \neq \texttt{Object}$, and therefore $\mathit{theName}(c) = c$. Consider the translation of the expression, from (6) we have

$$[\![e.f = e']\!](p, \gamma) \triangleq \{\ \texttt{Identity}\ x; \mathit{theType}(c)\ x_1; e_1; e_2; e_3\ \}$$

where

—e_1 is $x = [\![e]\!]_{expr}(p, \gamma)$
—e_2 is $x_1 = [\![e']\!]_{expr}(p, \gamma)$
—e_3 is $((\phi' @_p c)(x.\texttt{imp})).f = x_1$

and $\gamma(x) = \gamma(x_1) = \mathcal{U}df$. Since γ, γ_0 and γ' are defined on the same set of identifiers, we have that also $\gamma_0(x_1) = \gamma_0(x) = \mathcal{U}df$ and $\gamma'(x_1) = \gamma'(x) = \mathcal{U}df$.

Let $\sigma_1'' = \sigma_1[x \mapsto \texttt{null}, x_1 \mapsto v']$, where v' is an initial value of type $\mathit{theType}(t)$. From $p, \gamma \vdash \sigma \approx \sigma_1$ we have that $p, \gamma \vdash \sigma \approx \sigma_1''$ (x and x_1 are not defined in γ). Since (6).a, and (5).α, we can apply the inductive hypothesis to e, and derive that $[\![e]\!], \sigma_1'' \overset{\rightarrow}{\underset{[\![p]\!]}{\sim}} \iota, \sigma_1'''$ where $p, \gamma_0 \vdash \sigma'' \approx \sigma_1'''$. Therefore, $e_1, \sigma_1'' \overset{\rightarrow}{\underset{[\![p]\!]}{\sim}} \iota, \sigma_2$, where $\sigma_2 = \sigma_1'''[x \mapsto \iota]$.
From $p, \gamma_0 \vdash \sigma'' \approx \sigma_1'''$, as before, we can derive that also $p, \gamma_0 \vdash \sigma'' \approx \sigma_2$ (x is not defined in γ_0). Since (6).b, and (5).β, we can apply the inductive hypothesis to e', and derive that $[\![e']\!], \sigma_2 \overset{\rightarrow}{\underset{[\![p]\!]}{\sim}} v, \sigma_2'$ where $p, \gamma' \vdash \sigma''' \approx \sigma_2'$ and from Lemma C.3, $\sigma_2'(x) = \sigma_2(x) = \iota$. Therefore, $e_2, \sigma_2 \overset{\rightarrow}{\underset{[\![p]\!]}{\sim}} v, \sigma_3$, where $\sigma_3 = \sigma_2'[x_1 \mapsto v]$, $p, \gamma' \vdash \sigma''' \approx \sigma_3$, and $\sigma_3(x) = \sigma_2(x) = \iota$.
Now, to show that e_3 is evaluated to v , not only we have to show that the object refereed to by $x.\texttt{imp}$ in σ_3 has field f, which can be derived from $p, \gamma' \vdash \sigma''' \approx \sigma_3$ and (5).χ, but also that $\sigma_3(\iota)(\texttt{imp}) = [[\cdots]]^d$ where $d \leq \phi' @_p c$ (so the evaluation of the cast expression does not produce a cast exception).
From (6).a, (5).α and Theorem 3.1, we have that $p, \sigma'' \vdash \iota \lhd c$, therefore $\sigma''(\iota) = [[\cdots]]^{c'}$ where $c' \leq c$. From (6).b, (5).β and Theorem 3.1, $\sigma'''(\iota) = [[\cdots]]^d$ where $\phi' @_p c' = \phi' @_p d$. So $\phi' @_p d \leq \phi' @_p c$, and since $d \leq \phi' @_p d$, and $p, \gamma' \vdash \sigma''' \approx \sigma_3$ we have that $\sigma_3(\iota)(\texttt{imp}) = [[\cdots]]^d$ where $d \leq \phi' @_p c$. Therefore $e_3, \sigma_3 \overset{\rightarrow}{\underset{[\![p]\!]}{\sim}} v, \sigma_4$ where $\sigma_4 = \sigma_3[\iota' \mapsto \sigma_3(\iota)[f \mapsto v]]$ for $\iota' = \sigma_3(\iota)(\texttt{imp})$.

Let $\sigma'_1 = \sigma_4[x \mapsto \sigma_1(x), x_1 \mapsto \sigma_1(x_1)]$, and $\sigma' = \sigma'''[\iota \mapsto \sigma'''(\iota)[f \mapsto v]]$,

$$[\![e.f = e']\!], \sigma_1 \ \underset{[\![p]\!]}{\rightsquigarrow} \ v, \sigma'_1$$

From $p, \gamma' \vdash \sigma''' \approx \sigma_3$ we can show that $p, \gamma' \vdash \sigma' \approx \sigma_4$. In particular, clauses (1) and (2) of Definition 7.4 are trivially verified. For clause (3) observe that for all $\bar{\iota}$ such that $\bar{\iota} \neq \iota$ if $\sigma'(\bar{\iota}) = [[f_1 : v_1, \ldots, f_n : v_n]]^{\,d}$, then $\sigma_4(\bar{\iota}) = [[\texttt{imp} : \bar{\iota'}]]^{\texttt{Id}}$ where $\sigma_4(\bar{\iota'})(\texttt{id}) = \bar{\iota} \neq \iota$. Therefore, $\sigma_4(\bar{\iota'}) = \sigma_3(\bar{\iota'})$, and since $\sigma'(\bar{\iota}) = \sigma'''(\bar{\iota})$ the result derives from $p, \gamma' \vdash \sigma''' \approx \sigma_3$. For the object ι the result is obvious. This concludes the proof of the assignment to a field.

Consider rule (*meth*): We will consider methods with a single parameter, $e.m(e')$. So

$$\frac{\begin{array}{l}\alpha.\ e, \sigma \underset{p}{\rightsquigarrow} \iota, \sigma'' \\ \beta.\ e', \sigma'' \underset{p}{\rightsquigarrow} v', \sigma''' \\ \chi.\ \sigma'''(\iota) = [[\cdots]]^{\,c'} \quad \mathcal{MD}(p, c', m) = t\ m(t_1\ y_1)\ \phi\ block \\ \alpha'.\ \bar{\sigma} = \sigma'''[\texttt{this} \mapsto \iota, y_1 \mapsto v'] \\ \beta'.\ block, \bar{\sigma} \underset{p}{\rightsquigarrow} v, \bar{\sigma}' \\ \chi'.\ \sigma' = \bar{\sigma}'[\texttt{this} \mapsto \sigma'''(\texttt{this}), y_1 \mapsto \sigma'''(y_1)]\end{array}}{e.m(e'), \sigma \underset{p}{\rightsquigarrow} v, \sigma'} \tag{7}$$

From the fact that the expression is well-typed, we have

$$\frac{\begin{array}{l}a.\ p, \gamma \ \vdash\ e\ :\ c \parallel \gamma_0 \parallel \phi_0 \\ b.\ p, \gamma_0 \ \vdash\ e'\ :\ t_1 \parallel \gamma_1 \parallel \phi_1 \\ c.\ \mathcal{MD}(p, \phi_1 @_p c, m) = t\ m(t'_1\ y_1)\ \phi'\ block \\ d.\ p \vdash t_1 \leq t'_1 \quad \phi = \phi_0 \cup \phi_1 \cup \phi' \quad \gamma' = \phi' @_p \gamma_1\end{array}}{p, \gamma \ \vdash\ e.m(e')\ :\ t \parallel \gamma' \parallel \phi} \tag{8}$$

Note that, since $\mathcal{MD}(p, \phi_1 @_p c, m) \neq \mathcal{U}df$, and $p \vdash c \leq \phi_1 @_p c$, we have that $c \neq \texttt{Object}$. Therefore $\mathit{theName}(c) = c$. Consider the translation of the expression, from (8) we have

$$[\![e.m(e')]\!]_{expr}(p, \gamma) \triangleq \{\texttt{Identity}\ x; \mathit{theType}(t_1)\ x_1; e_1; e_2; e_3\ \}$$

where

—e_1 is $x = [\![e]\!]_{expr}(p, \gamma)$
—e_2 is $x_1 = [\![e']\!]_{expr}(p, \gamma_0)$
—e_3 is $((\phi_1 @_p c)(x.\texttt{imp})).m(x_1)$

and $\gamma(x_1) = \gamma(x) = \mathcal{U}df$. Since γ, γ_0 and γ' are defined on the same set of identifiers, we have that also $\gamma_0(x_1) = \gamma_0(x) = \mathcal{U}df$ and $\gamma'(x_1) = \gamma'(x) = \mathcal{U}df$.

Let $\sigma''_1 = \sigma_1[x \mapsto \texttt{null}, x_1 \mapsto v']$, where v' is an initial value of type $\mathit{theType}(t)$. From $p, \gamma \vdash \sigma \approx \sigma_1$ we have that $p, \gamma \vdash \sigma \approx \sigma''_1$ (x_1 and x are not defined in γ). Since (8).a, and (7).α, we can apply the inductive hypothesis to e, and derive that $[\![e]\!], \sigma''_1 \ \underset{[\![p]\!]}{\rightsquigarrow} \ \iota, \sigma'''_1$ where $p, \gamma_0 \vdash \sigma'' \approx \sigma'''_1$. Therefore, $e_1, \sigma''_1 \ \underset{[\![p]\!]}{\rightsquigarrow} \ \iota, \sigma_2$, where $\sigma_2 = \sigma'''_1[x \mapsto \iota]$.

From $p, \gamma_0 \vdash \sigma'' \approx \sigma'''_1$, as before, we can derive that also $p, \gamma_0 \vdash \sigma'' \approx \sigma_2$ (x is not defined in γ_0). Since (8).b, and (7).β, we can apply the inductive hypothesis to e', and derive that $[\![e']\!], \sigma_2 \ \underset{[\![p]\!]}{\rightsquigarrow} \ \iota, \sigma'_2$ where $p, \gamma' \vdash \sigma''' \approx \sigma'_2$ and from Lemma C.3, $\sigma'_2(x_1) = \sigma_2(x_1) = \iota$. Therefore, $e_2, \sigma_2 \ \underset{[\![p]\!]}{\rightsquigarrow} \ v, \sigma_3$, where $\sigma_3 = \sigma'_2[x_1 \mapsto v]$,

$p, \gamma' \vdash \sigma''' \approx \sigma_3$, and $\sigma_3(x) = \sigma_2(x) = \iota$.
Now we have to evaluate $[\![e_3]\!]$ in the store σ_3. From the rule for method call (*meth*): we evaluate first $(\phi_1 @_p c)(x.\texttt{imp})$ in the store σ_3. Since $p, \gamma' \vdash \sigma''' \approx \sigma_2'$, $\sigma_2''(\iota) = \sigma_3(\iota)$, and

—$\sigma_3(x) = \iota$,
—$\sigma_3(\iota) = [[\texttt{imp} : \iota' \cdots]]^{\texttt{Identity}}$, and
—$\sigma_3(\iota') = [[\texttt{id} : \iota \cdots]]^{c'}$.

Moreover, from (8).a, (7).α and Theorem 3.1 we have that $c' \leq c$. Again from (8).b, (7).β and Theorem 3.1 (in particular $p, \phi \vdash \sigma'' \lhd \sigma'''$) we have that $\phi_1 @_p c' \leq \phi_1 @_p c$. Therefore, the cast succeeds and we have that:

$$(\phi_1 @_p c)(x.\texttt{imp}), \sigma_3 \ \overset{\rightarrow}{\underset{[\![p]\!]}{\sim}} \ \iota', \sigma_3 \text{ and } x_1, \sigma_3 \ \overset{\rightarrow}{\underset{[\![p]\!]}{\sim}} \ v', \sigma_3.$$

From (7).χ, $\mathcal{MD}(p, c', m) = t\ m(t_1\ y_1)\ \phi\ block$. Since the program is well formed from Figure 8, we have that

$$(*) \quad p, \gamma'' \ \vdash \ block \ : \ t' \ [\![\ \gamma''' \ [\![\ \phi''$$

where $\gamma'' = t_1\ y_1, c'\ \texttt{this}, t' \leq t$, and $\phi'' \subseteq \phi'$. From the definition of the translation of methods we have that

$$[\![t\ m(t_1\ y_1)\ \phi\ block]\!]_{meth}(p, c) \triangleq \\ theType(t)\ m([\![t_1\ y_1]\!]_{var})\{\ \}[\![block]\!]_{expr}(p, \gamma'').$$

Consider the store $\sigma_4 = \sigma_3[\texttt{this} \mapsto \iota', y_1 \mapsto v']$. It is immediate to see that $p, \gamma'' \vdash \bar{\sigma} \approx \sigma_4$. Applying the inductive hypothesis to (7).β', $(*)$, we get:

$$[\![block]\!]_{expr}(p, \gamma''), \sigma_4 \ \overset{\rightarrow}{\underset{[\![p]\!]}{\sim}} \ v, \sigma_5$$

where $p, \gamma''' \vdash \bar{\sigma}' \approx \sigma_5$ and, since $\gamma''(x) = \gamma''(x_1) = \mathcal{U}df$, from Lemma C.3 and definition of σ_4 we have that $\sigma_4(x) = \sigma_3(x) = \sigma_5(x)$ and $\sigma_4(x_1) = \sigma_3(x_1) = \sigma_5(x_1)$. Let $\sigma_6 = \sigma_5[\texttt{this} \mapsto \sigma_3(\texttt{this}), y_1 \mapsto \sigma_3(y_1)]$ we have that

$$[\![e_4]\!], \sigma_3 \ \overset{\rightarrow}{\underset{[\![p]\!]}{\sim}} \ v, \sigma_6.$$

Let σ' be defined in (7).χ'. Since $p, \gamma' \vdash \sigma''' \approx \sigma_3$ we have that $p, \gamma' \vdash \sigma' \approx \sigma_6$. Let $\sigma_1' = \sigma_6[x \mapsto \sigma_1(x), x_1 \mapsto \sigma_1(x_1)]$, it is immediate to show that $p, \gamma' \vdash \sigma' \approx \sigma_1'$. This concludes the proof of the case of method call.

Consider rule (*cast*):

$$\frac{\begin{array}{l} e, \sigma \overset{\rightarrow}{\underset{p}{\sim}} \ \iota, \sigma' \\ \sigma'(\iota) = [[\ldots]]^{c'} \\ p \vdash c' \leq c \end{array}}{(c)e, \sigma \overset{\rightarrow}{\underset{p}{\sim}} \ \iota, \sigma'}$$

Since the expression is well-typed we have that

$$\frac{\begin{array}{l} a.\ p, \gamma \ \vdash \ e \ : \ c' \ [\![\ \gamma' \ [\![\ \phi \\ b.\ (p \vdash c' \leq c \text{ or } p \vdash c \leq c') \end{array}}{p, \gamma \ \vdash \ (c)e \ : \ c \ [\![\ \gamma' \ [\![\ \phi} \tag{9}$$

Let $[\![(c)e]\!]_{expr}(p, \gamma) \triangleq \{\ \texttt{Identity}\ x;\ \texttt{if}\ e_1\ \texttt{then null else}\ e_2\ \}$, where

—e_1 is $\texttt{isnull}(x = [\![e]\!]_{expr}(p,\gamma))$, and

—e_2 is $(\mathit{theName}(c)x.\texttt{imp}).\texttt{id}$

Let $\sigma_1'' = \sigma_1[x \mapsto \texttt{null}]$. From $p,\gamma \vdash \sigma \approx \sigma_1$, we have $p,\gamma \vdash \sigma \approx \sigma_1''$. From (9).a and $e,\sigma \overset{\rightarrow}{\underset{p}{}} \iota,\sigma'$ we can apply the inductive hypothesis to e, and derive that $[\![e]\!],\sigma_1'' \overset{\rightarrow}{\underset{[\![p]\!]}{}} \iota,\sigma_1'''$ and $p,\gamma' \vdash \sigma' \approx \sigma_1'''$. So $\sigma_1'''(\iota) = [[\texttt{imp} : \iota']]^{\texttt{Id}}$, and $\sigma_1'''(\iota') = [[\ldots,\texttt{id} : \iota]]^{\mathit{theName}(c')}$. Therefore, $[\![e_1]\!],\sigma_1'' \overset{\rightarrow}{\underset{[\![p]\!]}{}} \texttt{false},\sigma_2$, where $\sigma_2 = \sigma_1'''[x \mapsto \iota]$, and the expression e_2 is evaluated.
Since $\sigma_2(x)(\texttt{imp}) = \iota'$, $\sigma_2(\iota')(\texttt{id}) = \iota$, (9).b, and Lemma C.1, so $p \vdash \mathit{theName}(c') \leq \mathit{theName}(c)$, we have that $[\![e_2]\!],\sigma_2 \overset{\rightarrow}{\underset{[\![p]\!]}{}} \iota,\sigma_2$.
Let $\sigma_1' = \sigma_2[x \mapsto \sigma_1(x)]$, we have that $[\![(c)e]\!],\sigma_1 \overset{\rightarrow}{\underset{[\![p]\!]}{}} \iota,\sigma_1'$, and from $p,\gamma' \vdash \sigma' \approx \sigma_1'''$, we get that $p,\gamma' \vdash \sigma' \approx \sigma_1'$.

Consider rule (*n-cast*), that is assume that $e,\sigma \overset{\rightarrow}{\underset{p}{}} \texttt{null},\sigma'$. From rule (*n-cast*), $(c)e,\sigma \overset{\rightarrow}{\underset{p}{}} \texttt{null},\sigma'$. Consider $[\![(c)e]\!]$ as for rule (*cast*). We can apply the inductive hypothesis to e, and derive that $[\![e]\!],\sigma_1'' \overset{\rightarrow}{\underset{[\![p]\!]}{}} \texttt{null},\sigma_1'''$ and $p,\gamma' \vdash \sigma' \approx \sigma_1'''$. Therefore, $[\![e_1]\!],\sigma_1'' \overset{\rightarrow}{\underset{[\![p]\!]}{}} \texttt{true},\sigma_2$, where $\sigma_2 = \sigma_1'''[x \mapsto \texttt{null}]$, and `null` is evaluated. Let $\sigma_1' = \sigma_1''[x \mapsto \sigma_1(x)]$, we have that $[\![(c)e]\!],\sigma_1 \overset{\rightarrow}{\underset{[\![p]\!]}{}} \texttt{null},\sigma_1'$. From $p,\gamma' \vdash \sigma' \approx \sigma_1'''$, we get that $p,\gamma' \vdash \sigma' \approx \sigma_1'$.

Let us now assume that the result of the evaluation is an exception. We will consider `nullPntrExc` raised during the evaluation of a field update and `castExc` raised by a cast expression.

Consider rule (*a-field-null*):

$$\frac{\begin{array}{l}\alpha.\ e,\sigma \overset{\rightarrow}{\underset{p}{}} \texttt{null},\sigma'' \\ \beta.\ e',\sigma'' \overset{\rightarrow}{\underset{p}{}} v,\sigma'\end{array}}{e.f = e',\sigma \overset{\rightarrow}{\underset{p}{}} \texttt{nullPntrExc},\sigma'} \tag{10}$$

Let the translation of the expression as for the case of rule (*a-field*). Let $\sigma_1'' = \sigma_1[x \mapsto \texttt{null}, x_1 \mapsto v']$, where v' is an initial value of type $\mathit{theType}(t)$. From $p,\gamma \vdash \sigma \approx \sigma_1$ we have that $p,\gamma \vdash \sigma \approx \sigma_1''$. Since the expression is well typed $p,\gamma \vdash e : c \parallel \gamma_0 \parallel \phi_0$ for some c, γ_0, and ϕ_0. So, from $e,\sigma \overset{\rightarrow}{\underset{p}{}} \texttt{null},\sigma''$, we can apply the inductive hypothesis to e, and derive that $[\![e]\!],\sigma_1'' \overset{\rightarrow}{\underset{[\![p]\!]}{}} \texttt{null},\sigma_1'''$ where $p,\gamma_0 \vdash \sigma'' \approx \sigma_1'''$. Therefore, $e_1,\sigma_1'' \overset{\rightarrow}{\underset{[\![p]\!]}{}} \texttt{null},\sigma_2$, where $\sigma_2 = \sigma_1'''[x \mapsto \texttt{null}]$.
From $p,\gamma_0 \vdash \sigma'' \approx \sigma_1'''$, as before, we can derive that also $p,\gamma_0 \vdash \sigma'' \approx \sigma_2$ (x is not defined in γ_0). Since (6).b and (10).β, we can apply the inductive hypothesis to e', and derive that $[\![e']\!],\sigma_2 \overset{\rightarrow}{\underset{[\![p]\!]}{}} v,\sigma_2'$ where $p,\gamma' \vdash \sigma' \approx \sigma_2'$ and from Lemma C.3, $\sigma_2'(x) = \sigma_2(x) = \texttt{null}$. Therefore, $e_2,\sigma_2 \overset{\rightarrow}{\underset{[\![p]\!]}{}} v,\sigma_3$, where $\sigma_3 = \sigma_2'[x_2 \mapsto v]$, $p,\gamma' \vdash \sigma' \approx \sigma_3$, and $\sigma_3(x) = \sigma_2(x) = \texttt{null}$.
Since $\sigma_3(x) = \texttt{null}$, we have that $e_3,\sigma_3 \overset{\rightarrow}{\underset{[\![p]\!]}{}} \texttt{nullPntrExc},\sigma_3$ (we apply the rule that generate the exception to $x.\texttt{imp}$ and then the rules for propagation of exceptions of Fig. 6).
Let $\sigma_1' = \sigma_3[x \mapsto \sigma_1(x), x_1 \mapsto \sigma_1(x_1)]$, then $[\![e.f = e']\!],\sigma_1 \overset{\rightarrow}{\underset{[\![p]\!]}{}} \texttt{nullPntrExc},\sigma_1'$.

Consider rule (*e-cast*):

$$\frac{\begin{array}{l} e,\sigma \overset{\rightarrow}{\underset{p}{\rightsquigarrow}} \iota,\sigma' \\ \sigma'(\iota) = \texttt{[[}\ldots\texttt{]]}^{\,c'} \\ p \not\Vdash c' \leq c \end{array}}{(c)e,\sigma \overset{\rightarrow}{\underset{p}{\rightsquigarrow}} \texttt{castExc},\sigma'}$$

Let $[\![(c)e]\!]_{expr}(p,\gamma)$, and $\sigma_1'' = \sigma_1[x \mapsto \texttt{null}]$ be as for the case of rule (*cast*). From the inductive hypothesis applied to e,
$[\![e]\!],\sigma_1'' \overset{\rightarrow}{\underset{[\![p]\!]}{\rightsquigarrow}} \iota,\sigma_1'''$ and $p,\gamma' \vdash \sigma' \approx \sigma_1'''$. So $\sigma_1'''(\iota) = \texttt{[[imp : } \iota'\texttt{]]}^{\texttt{Id}}$, and $\sigma_1'''(\iota') = \texttt{[[}\ldots\texttt{]]}^{theName(c')}$. Therefore, $[\![e_1]\!],\sigma_1'' \overset{\rightarrow}{\underset{[\![p]\!]}{\rightsquigarrow}} \texttt{false},\sigma_2$, where $\sigma_2 = \sigma_1'''[x \mapsto \iota]$, and the expression e_2 is evaluated.
Since $\sigma_2(x)(\texttt{imp}) = \iota'$, $\sigma_2(\iota') = \texttt{[[}\ldots\texttt{]]}^{theName(c')}$, and $p \not\Vdash c' \leq c$, from Lemma C.1 we derive that $p \not\Vdash theName(c') \leq theName(c)$. Therefore, $[\![(c)x.\texttt{imp}]\!],\sigma_2 \overset{\rightarrow}{\underset{[\![p]\!]}{\rightsquigarrow}} \texttt{castExc},\sigma_2$. Applying the rules for propagation of exceptions we get that $[\![e_2]\!],\sigma_2 \overset{\rightarrow}{\underset{[\![p]\!]}{\rightsquigarrow}} \texttt{castExc},\sigma_2$. Let $\sigma_1' = \sigma_2[x \mapsto \sigma_1(x)]$, we have that $[\![(c)e]\!],\sigma_1 \overset{\rightarrow}{\underset{[\![p]\!]}{\rightsquigarrow}} \texttt{castExc},\sigma_1'$. Note that the cast in the translation is essential to raise the exception, that otherwise would not occur since every `Identity` object has the field `imp` and every `FickleObject` has the field `id`.

Proof of (2): if $[\![e]\!],\sigma_1 \overset{\rightarrow}{\underset{[\![p]\!]}{\rightsquigarrow}} w,\sigma_1'$, *then there is* σ' *such that* $e,\sigma \overset{\rightarrow}{\underset{p}{\rightsquigarrow}} w,\sigma'$. The result is proved by induction on the depth of the derivation tree of $[\![e]\!],\sigma_1 \overset{\rightarrow}{\underset{[\![p]\!]}{\rightsquigarrow}} w,\sigma_1'$. Note that, since e is well-typed, from Theorem 7.1 we have that also $[\![e]\!]$ is well-typed, and from Theorem 3.1, w is either a value or $w \in \{\texttt{castExc},\texttt{nullPntrExc}\}$. We will consider the translation of the following expressions: identifiers, object creation, re-classification, field update, and cast. The other cases are similar.

Consider $[\![id]\!]$. There are two cases: $id = x$, or $id = \texttt{this}$. In the first case $[\![id]\!]_{expr}(p,\gamma) \triangleq id$, and the only rule applicable is $[\![id]\!],\sigma_1 \overset{\rightarrow}{\underset{[\![p]\!]}{\rightsquigarrow}} \sigma_1(id),\sigma_1$. From $p,\gamma \vdash \sigma \approx \sigma_1$ we have that $\sigma(id) = \sigma_1(id)$. Therefore also $id,\sigma \overset{\rightarrow}{\underset{p}{\rightsquigarrow}} \sigma(id),\sigma$.
In the second case $[\![\texttt{this}]\!]_{expr}(p,\gamma) \triangleq \texttt{this.id}$ and, from $p,\gamma \vdash \sigma \approx \sigma_1$ we have that $\sigma_1(\sigma_1(\texttt{this}))(\texttt{id}) \neq \mathcal{U}df$ and $\sigma_1(\sigma_1(\texttt{this}))(\texttt{id}) \neq \sigma(\texttt{this})$. Therefore w is a value. In particular,

$$[\![\texttt{this}]\!],\sigma_1 \overset{\rightarrow}{\underset{[\![p]\!]}{\rightsquigarrow}} \sigma_1(\sigma_1(\texttt{this}))(\texttt{id}),\sigma_1$$

applying rule (*field*). Since $\texttt{this},\sigma \overset{\rightarrow}{\underset{p}{\rightsquigarrow}} \sigma(\texttt{this}),\sigma$, we derive the result.

Consider $[\![\texttt{new } c]\!] \triangleq \{theName(c)\ theImp; \texttt{Identity}\ theId; e_1; e_2; e_3; e_4; theId\}$ where

—e_1 is $theId = \texttt{new Identity}$,
—e_2 is $theImp = \texttt{new}\ theName(c)$,
—e_3 is $theImp.\texttt{id} = theId$, and
—e_4 is $theId.\texttt{imp} = theImp$.

and $\gamma(theId) = \gamma(theImp) = Udf$. Assume that $c \neq \texttt{Object}$, so $theName(c) = c$. (The case $c = \texttt{Object}$ is simpler since $\mathcal{F}s(p,\texttt{Object}) = \emptyset$.) From the definition of the operational semantics, and the fact that: $c \leq \texttt{FickleObject}$ has the field `id` and

an object of class `Identity` has the field `imp` (so the evaluation of the expressions cannot produce an exception), as for the case of the corresponding "only if" proof, we have that:

$$[\![\texttt{new } c]\!], \sigma_1 \ \overset{\rightarrow}{\underset{[\![p]\!]}{\rightsquigarrow}} \ \iota, \sigma_1'$$

where

—$e_1; e_2; e_3; e_4; \mathit{theId}, \sigma_1[\mathit{theImp} \mapsto \texttt{null}, \mathit{theId} \mapsto \texttt{null}] \underset{p}{\rightsquigarrow} \iota, \sigma_1''$,
—$\sigma_1'' = \sigma_1[\iota' \mapsto [[\texttt{id} : \iota, f_1 : v_1', \ldots, f_r : v_r']]^{c}, \mathit{theImp} \mapsto \iota', \iota \mapsto [[\texttt{imp} : \iota']]^{\texttt{Id}}, \mathit{theId} \mapsto \iota]$, and
—$\sigma_1' = \sigma_1'''[\mathit{theImp} \mapsto \sigma_1(\mathit{theImp}), \mathit{theId} \mapsto \sigma_1(\mathit{theId})]$.

Since ι and ι' are new in σ_1, they are new also in σ. So we have

$$\texttt{new } c, \sigma \underset{p}{\rightsquigarrow} \iota, \sigma'$$

where $\sigma' = \sigma[\iota \mapsto [[f_1 : v_1, \ldots, f_r : v_r]]^{c}]$, and v_l is initial for $\mathcal{F}(p, c, f_l)$, $1 \leq l \leq r$. As for the case of the corresponding "only if" proof, using Lemma C.2 we can show that $p, \gamma' \vdash \sigma' \approx \sigma_1'$.

Consider $[\![x \,!!\, d]\!] \stackrel{\Delta}{=}$

$\{\mathit{theName}(d)\ \mathit{theImp}; \texttt{Identity}\ \mathit{theId}; \mathcal{R}(p, d)\ \mathit{theLastImp}; \texttt{if}\ e_1\ \texttt{then null else}\ e_2;\ \mathit{theId}\}$

where

—e_1 is $\texttt{isnull}(\mathit{theId} = x)$,
—e_2 is

$\{\mathit{theImp} = \texttt{new}\ \mathit{theName}(d); \mathit{theLastImp} = (\mathcal{R}(p, d))\mathit{theId}.\texttt{imp};$
$\mathit{theImp}.\texttt{id} = \mathit{theId}; \mathit{theId}.\texttt{imp} = \mathit{theImp};$
$e_1'; \ldots e_r'$
$\}$

—e_i' is $\mathit{theImp}.f_i = \mathit{theLastImp}.f_i$ for $1 \leq i \leq r$, and

$\gamma(\mathit{theId}) = \gamma(\mathit{theLastImp}) = \gamma(\mathit{theImp}) = \mathit{Udf}$.

If $\sigma_1(x) = \texttt{null}$, then $[\![x \,!!\, d]\!], \sigma_1 \ \overset{\rightarrow}{\underset{[\![p]\!]}{\rightsquigarrow}} \ \texttt{null}, \sigma_1$ since

$e_1, \sigma_1[\mathit{theImp}, \mathit{theId}, \mathit{theLastImp} \mapsto \texttt{null}] \ \overset{\rightarrow}{\underset{[\![p]\!]}{\rightsquigarrow}} \ \texttt{true}, \sigma_1[\mathit{theImp}, \mathit{theId}, \mathit{theLastImp} \mapsto \texttt{null}]$

Since the expression is well-typed $d \neq \texttt{Object}$, and $\mathit{theName}(d) = d$. Applying rule (*n-recl*) also $x \,!!\, d, \sigma \underset{p}{\rightsquigarrow} \texttt{null}, \sigma$ and the result holds.

Consider now the case $\sigma_1(x) \neq \texttt{null}$. From $p, \gamma' \vdash \sigma \approx \sigma_1$ we have $\sigma(x) \neq \texttt{null}$. Moreover, since the expression is well-typed $\gamma(x) = c$, so from $p, \gamma \vdash \sigma \diamond$, $\sigma(x) = \iota$ and $\sigma(\iota) = [[\cdots]]^{c'}$ for some $c' \leq c$. Again from $p, \gamma' \vdash \sigma \approx \sigma_1$, we derive that

$$\sigma_1(x) = \iota \qquad \sigma_1(\iota) = [[\texttt{imp} : \iota']]^{\texttt{Id}} \qquad \sigma_1(\iota') = [[\texttt{id} : \iota, \cdots]]^{c'} \tag{11}$$

From the definition of the operational semantics of blocks, and the fact that `null` is the initial value for an object, we have that:

—$e_1, \sigma_1[\mathit{theImp}, \mathit{theId}, \mathit{theLastImp} \mapsto \texttt{null}] \ \overset{\rightarrow}{\underset{[\![p]\!]}{\rightsquigarrow}} \ \texttt{false}, \sigma_1''$ where
$\sigma_1'' = \sigma_1[\mathit{theImp}, \mathit{theLastImp} \mapsto \texttt{null}, \mathit{theId} \mapsto \iota]$,

—$theImp = \texttt{new}\ d, \sigma_1'' \underset{[\![p]\!]}{\rightsquigarrow} \iota'', \sigma_2$ where ι'' is new and $\sigma_2 = \sigma_1''[theImp \mapsto \iota'', \iota'' \mapsto [\![\texttt{id} : \texttt{null}, f_1 : v_1'', \ldots, f_r : v_r'', f_{r+1} : v_{r+1}'', \ldots, f_{r+q} : v_{r+q}'']\!]^d, theImp \mapsto \iota''']$ where v_l'' is initial for $theType(\mathcal{F}(p, d, f_l))$, $l \in \{1, \ldots, r+q\}$,

—from equation (11), the fact that the expression is well typed, therefore $\mathcal{R}(p, d) = \mathcal{R}(p, c')$, we have that

$$theLastImp = (\mathcal{R}(p, d))theId.\texttt{imp}, \sigma_2 \underset{[\![p]\!]}{\rightsquigarrow} \iota', \sigma_3$$

where $\sigma_3 = \sigma_2[theLastImp \mapsto \iota']$(neither the cast operation, nor the access to field `imp` produce an exception)

—$theImp.\texttt{id} = theId; theId.\texttt{imp} = theImp;, \sigma_3 \underset{[\![p]\!]}{\rightsquigarrow} \iota'', \sigma_4$ where $\sigma_4 = \sigma_3[\iota \mapsto [\![\texttt{imp} : \iota'']\!]^{\texttt{Id}}, \iota'' \mapsto [\![\texttt{id} : \iota'', f_1 : v_1'', \ldots, f_r : v_r'', f_{r+1} : v_{r+1}'', \ldots, f_{r+q} : v_{r+q}'']\!]^d, theImp \mapsto \iota''']$, and finally

—$e_1'; \ldots e_r', \sigma_4 \underset{[\![p]\!]}{\rightsquigarrow} \iota'', \sigma_1''$ where $\sigma_1'' = \sigma_4[\iota'' \mapsto [\![\texttt{id} : \iota'', f_1 : v_1, \ldots, f_r : v_r, f_{r+1} : v_{r+1}'', \ldots, f_{r+q} : v_{r+q}'']\!]^d]$ where $\{f_1, \ldots, f_r\}$ are the fields of the class $\mathcal{R}(p, d)$ and $v_i = \sigma_1(x)(f_i)$ for $1 \leq i \leq r$.

Let $\sigma_1' = \sigma_1''[theImp \mapsto \sigma_1(theImp), theId \mapsto \sigma_1(theId), theLastThis \mapsto \sigma_1(theLastThis)]$. We derive that

$$[\![x\,!!\,d]\!], \sigma_1 \underset{[\![p]\!]}{\rightsquigarrow} \iota, \sigma_1'.$$

On the other hand, since $\sigma(x) = \iota$, $\sigma(\iota) = [\![\cdots]\!]^{c'}$, and $\{f_1, \ldots, f_r\}$ are the fields of the class $\mathcal{R}(p, d)$ from rule (*recl*) we have that

$$x\,!!\,d, \sigma \underset{p}{\rightsquigarrow} \iota, \sigma'$$

where $\sigma' = \sigma[\iota \mapsto [\![f_1 : v_1', \ldots, f_r : v_r', f_{r+1} : v_{r+1}', \ldots, f_{r+q} : v_{r+q}']\!]^d$, $v_i' = \sigma(x)(f_i)$ for $1 \leq i \leq r$, and v_i' is the initial value for $\mathcal{F}(p, d, f_i)$, $r \leq i \leq r+q$.

To show that $p, \gamma' \vdash \sigma' \approx \sigma_1'$ we have to prove that the three clauses of Definition 7.4 are satisfied for σ' and σ_1'. From the definition of σ' and σ_1' and the fact that $p, \gamma \vdash \sigma \approx \sigma_1$ we have that for all x', $\sigma'(x') = \sigma_1'(x')$. Therefore, clause 1. of Definition 7.4 holds, and since the only field `id` that is updated in σ_1' is the one of a newly created location also $\sigma_1'(\texttt{this})(\texttt{id}) = \sigma_1(\texttt{this})(\texttt{id})$ and clause 2. holds. To conclude the proof, we have to verify clause 3. for the address ι. Let

—$\sigma'(\iota) = [\![[\![f_1 : v_1', \ldots, f_r : v_r']\!]^c, \sigma_1'(\iota) = [\![\texttt{imp} : \iota''']\!]^{\texttt{Id}}$,

—$\sigma_1'(\iota''') = [\![\texttt{id} : \iota, f_1 : v_1, \ldots, f_r : v_r, f_{r+1} : v_{r+1}'', \ldots, f_{r+q} : v_{r+q}'']\!]^d$

From $p, \gamma \vdash \sigma \approx \sigma_1$ for all i, $1 \leq i \leq r$, $v_i = v_i'$. Moreover, for all i, $r \leq i \leq r+q$, v_i' is an initial value for $\mathcal{F}(p, d, f_{r+i})$ and v_{r+i}'' is an initial value for $theType(\mathcal{F}(p, d, f_{r+i}))$. Therefore, from Lemma C.2, $v_{r+i} = v_{r+i}''$, and clause 3. holds. This concludes the proof that $p, \gamma' \vdash \sigma' \approx \sigma_1'$.
The case in which the expression is $[\![\texttt{this}\,!!\,d]\!]$ is similar.

Consider $[\![e.f = e']\!]$ =$\{$ `Identity` $x; theType(c)\ x_1;\ e_1; e_2; e_3$ $\}$ where

—e_1 is $x = [\![e]\!]_{expr}(p, \gamma)$

—e_2 is $x_1 = [\![e']\!]_{expr}(p, \gamma_0)$

—e_3 is $((\phi'@_p c)(x.\texttt{imp})).f = x_1$

and $\gamma(x) = \gamma(x_1) = \mathcal{U}df$. From the fact that the expression is well-typed, we have

$$\frac{\begin{array}{l} a.\ p,\gamma \vdash e : c \parallel \gamma_0 \parallel \phi_0 \\ b.\ p,\gamma_0 \vdash e' : t \parallel \gamma' \parallel \phi' \\ c.\ \mathcal{F}(p,\phi'@_p c, f) = t' \quad p \vdash t \leq t' \quad \phi = \phi \cup \phi' \end{array}}{p,\gamma \vdash e.f = e' : t \parallel \gamma' \parallel \phi} \tag{12}$$

Note that, since $\mathcal{F}s(p,\texttt{Object}) = \emptyset$, $c \neq \texttt{Object}$, and therefore $\mathit{theName}(c) = c$. Let $\sigma_1'' = \sigma_1[x \mapsto \texttt{null}, x_1 \mapsto v']$, where v' is an initial value of type $\mathit{theType}(t)$. Since $p,\gamma \vdash \sigma \approx \sigma_1$ and x and x_1 are not defined in γ also $p,\gamma \vdash \sigma \approx \sigma_1''$. From $[\![e.f = e']\!], \sigma_1 \overset{}{\underset{[\![p]\!]}{\rightsquigarrow}} w, \sigma_1'$ we derive that:

$$e_1; e_2; e_3, \sigma_1'' \overset{}{\underset{[\![p]\!]}{\rightsquigarrow}} w, \sigma_1''' \tag{13}$$

where $\sigma_1' = \sigma_1'''[x \mapsto \sigma_1(x), x_1 \mapsto \sigma_1(x_1)]$. Therefore,

(1) $e_1, \sigma_1'' \overset{}{\underset{[\![p]\!]}{\rightsquigarrow}} w, \sigma_1'''$, with $w \in \{\texttt{castExc}, \texttt{nullPntrExc}\}$, or
(2) $e_1, \sigma_1'' \overset{}{\underset{[\![p]\!]}{\rightsquigarrow}} v, \sigma_2$ for some v and σ_2.

(1) For the first case observe that, using the propagation rules for exceptions

$$[\![e.f = e']\!], \sigma_1 \overset{}{\underset{[\![p]\!]}{\rightsquigarrow}} w, \sigma_1'$$

Moreover, since an assignment does not produce an exception it must be the case that

$$[\![e]\!], \sigma_1'' \overset{}{\underset{[\![p]\!]}{\rightsquigarrow}} w, \sigma_1'''$$

Since e is well-typed, then $p,\gamma \vdash e : c \parallel \gamma_0 \parallel \phi_0$ for some c, γ_0, and ϕ_0. So, from $p,\gamma \vdash \sigma \approx \sigma_1''$ we can apply the inductive hypothesis to e and derive that: $e, \sigma \overset{}{\underset{p}{\rightsquigarrow}} w, \sigma'$. From the propagation rules for exceptions we derive

$$e.f = e', \sigma \overset{}{\underset{p}{\rightsquigarrow}} w, \sigma'$$

that proves the result.

(2) Assume that the evaluation of e_1 produces v. Therefore

$$[\![e]\!], \sigma_1'' \overset{}{\underset{[\![p]\!]}{\rightsquigarrow}} v, \sigma_2'' \tag{14}$$

where $\sigma_2 = \sigma_2''[x \mapsto v]$. From $p,\gamma \vdash \sigma \approx \sigma_1$ we have that $p,\gamma \vdash \sigma \approx \sigma_1''$ (x and x_1 are not defined in γ). From (12).a, and (14), we can apply the inductive hypothesis to e, and derive that $e, \sigma \overset{}{\underset{p}{\rightsquigarrow}} v, \sigma''$ where $p,\gamma_0 \vdash \sigma'' \approx \sigma_2''$ and since x is not defined in γ_0 also $p,\gamma_0 \vdash \sigma'' \approx \sigma_2$. From (13) we derive that
—$e_2, \sigma_2 \overset{}{\underset{[\![p]\!]}{\rightsquigarrow}} w, \sigma_1'''$, with $w \in \{\texttt{castExc}, \texttt{nullPntrExc}\}$, or
—$e_2, \sigma_2 \overset{}{\underset{[\![p]\!]}{\rightsquigarrow}} v', \sigma_3$ for some σ_3 and v'.
In the first case, as for the corresponding case of the evaluation of e_1,

$$[\![e.f = e']\!], \sigma_1 \overset{}{\underset{[\![p]\!]}{\rightsquigarrow}} w, \sigma_1'.$$

From the inductive hypothesis on e' and the propagation rules for exceptions $e.f = e', \sigma \overset{}{\underset{p}{\rightsquigarrow}} w, \sigma'$.
In the second case (the evaluation of e_2 produces v') we have that

$$[\![e']\!], \sigma_2 \overset{}{\underset{[\![p]\!]}{\rightsquigarrow}} v', \sigma_3'' \tag{15}$$

where $\sigma_3 = \sigma_3''[x_1 \mapsto v']$. (Note that since $\gamma_0(x) = \mathcal{U}df$ we also have that $\sigma_3(x) = \sigma_2(x) = v$.) From (12).b, (15), and $p, \gamma_0 \vdash \sigma'' \approx \sigma_2$ we can apply the inductive hypothesis to e' and derive that $e', \sigma'' \underset{p}{\rightsquigarrow} v', \sigma'''$ where $p, \gamma' \vdash \sigma''' \approx \sigma_3''$, and since x_1 is not defined in γ' then also $p, \gamma' \vdash \sigma''' \approx \sigma_3$.

From (12).a, and Theorem 3.1, $p, \sigma'' \vdash v \lhd c$. Therefore, either $v = \texttt{null}$, or $v = \iota$ and $\sigma''(\iota) = \texttt{[[}\cdots\texttt{]]}^{c'}$ where $c' \leq c$.

Assume that $v = \iota$. From $p, \gamma_0 \vdash \sigma'' \approx \sigma_2$ also $\sigma_2(\iota) = \texttt{[[imp : } \iota' \texttt{]]}^{\texttt{Id}}$, $\sigma_2(\iota') = \texttt{[[id : } \iota \cdots \texttt{]]}^{c'}$, and $\sigma_2(x_1) = \iota$. Moreover, (12).b, and Theorem 3.1, implies that $p, \phi' \vdash \sigma'' \lhd \sigma'''$ and $p, \gamma' \vdash \sigma''' \diamond$. So $\sigma'''(\iota) = \texttt{[[}\cdots\texttt{]]}^{c''}$, and $\phi' @_p c' = \phi' @_p c''$. So (12).c, $\phi' @_p c' \leq \phi' @_p c$ implies that field f is defined for $\sigma'''(\iota)$. From $p, \gamma' \vdash \sigma''' \approx \sigma_3$ we have that $\sigma_3(\iota) = \texttt{[[imp : } \iota'' \texttt{]]}^{\texttt{Id}}$, $\sigma_2(\iota'') = \texttt{[[id : } \iota \cdots \texttt{]]}^{c''}$ (for some ι'') and field f is defined for $\sigma_3(\iota')$. Therefore,

$$e_3, \sigma_3 \underset{[\![p]\!]}{\rightsquigarrow} v', \sigma_3[\iota'' \mapsto \sigma_3(\iota''[f \mapsto v'])]$$

and $[\![e.f = e']\!], \sigma_1 \underset{[\![p]\!]}{\rightsquigarrow} v', \sigma_1'$ where $\sigma_1' = \sigma_3[x_1 \mapsto \sigma_1(x_1), x_2 \mapsto \sigma_1(x_2), \iota'' \mapsto \sigma_3(\iota''[f \mapsto v'])]$.
On the other hand,

$$\frac{\begin{array}{l} e, \sigma \underset{p}{\rightsquigarrow} \iota, \sigma'' \\ e', \sigma'' \underset{p}{\rightsquigarrow} v', \sigma''' \\ \sigma'''(\iota)(f) \neq \mathcal{U}df \quad \sigma' = \sigma'''[\iota \mapsto \sigma'''(\iota)[f \mapsto v]] \end{array}}{e.f = e', \sigma \underset{p}{\rightsquigarrow} v', \sigma'}$$

From $p, \gamma' \vdash \sigma''' \approx \sigma_3$ it is easy to show that $p, \gamma' \vdash \sigma' \approx \sigma_1'$.

Assume that $v = \texttt{null}$. Therefore,

$$[\![e]\!], \sigma_1'' \underset{[\![p]\!]}{\rightsquigarrow} \texttt{null}, \sigma_2'' \tag{16}$$

where $\sigma_2 = \sigma_2''[x \mapsto \texttt{null}]$. From $p, \gamma \vdash \sigma \approx \sigma_1$ we have that $p, \gamma \vdash \sigma \approx \sigma_1''$ (x and x_1 are not defined in γ). From (12).a, and (16), we can apply the inductive hypothesis to e, and derive that $e, \sigma \underset{p}{\rightsquigarrow} \texttt{null}, \sigma''$ where $p, \gamma_0 \vdash \sigma'' \approx \sigma_2''$ and since x is not defined in γ_0 also $p, \gamma_0 \vdash \sigma' \approx \sigma''$.
Since $\sigma_3(x) = \sigma_2(x) = \texttt{null}$ we have that the evaluation of $x.\texttt{imp}$ generate `nullPntrExc`. Therefore,

$$e_3, \sigma_3 \underset{[\![p]\!]}{\rightsquigarrow} \texttt{nullPntrExc}, \sigma_3$$

By the propagation rules for exception

$$[\![e.f = e']\!], \sigma_1 \underset{[\![p]\!]}{\rightsquigarrow} \texttt{nullPntrExc}, \sigma_3'$$

where $\sigma_3' = \sigma_3[x \mapsto \sigma_1(x), x_1 \mapsto \sigma_1(x_1)]$.
On the other hand, applying rule (*a-field-null*),

$$\frac{\begin{array}{l} e, \sigma \underset{p}{\rightsquigarrow} \texttt{null}, \sigma'' \\ e', \sigma'' \underset{p}{\rightsquigarrow} v', \sigma''' \end{array}}{e.f = e', \sigma \underset{p}{\rightsquigarrow} \texttt{nullPntrExc}, \sigma'''}.$$

This concludes the proof of the case of the assignment to a field.

The proof for $e.m(e')$ is similar.

Consider $[\![(c)e]\!]_{expr} \triangleq \{\ \mathit{theType}(t)\ x;\ \texttt{if}\ e_1\ \texttt{then null else}\ e_2\ \}$, where

—e_1 is $\texttt{isnull}(x = [\![e]\!]_{expr}(p,\gamma))$, and
—e_2 is $(\mathit{theName}(c)x.\texttt{imp}).\texttt{id}$

Since the expression is well-typed

$$\frac{\begin{array}{l} a.\ p,\gamma\ \vdash\ e\ :\ c'\ [\![\ \gamma'\ [\![\ \phi \\ b.\ (p \vdash c' \leq c \text{ or } p \vdash c \leq c') \end{array}}{p,\gamma\ \vdash\ (c)e\ :\ c\ [\![\ \gamma'\ [\![\ \phi} \tag{17}$$

holds. From $[\![(c)e]\!]_{expr}, \sigma_1 \overset{}{\rightsquigarrow}_{[\![p]\!]} w, \sigma_1'$ and the definition of the operational semantics of blocks we have that

$$\texttt{if}\ e_1\ \texttt{then null else}\ e_2, \sigma_1'' \overset{}{\rightsquigarrow}_{[\![p]\!]} w, \sigma_1''' \tag{18}$$

where $\sigma_1'' = \sigma_1[x \mapsto \texttt{null}]$ and $\sigma_1' = \sigma_1'''[x \mapsto \sigma_1(x)]$. Therefore,

(1) $e_1, \sigma_1'' \overset{}{\rightsquigarrow}_{[\![p]\!]} w, \sigma_1'''$, with $w \in \{\texttt{castExc}, \texttt{nullPntrExc}\}$, or
(2) $e_1, \sigma_1'' \overset{}{\rightsquigarrow}_{[\![p]\!]} \texttt{true}, \sigma_1'''$ or
(3) $e_1, \sigma_1'' \overset{}{\rightsquigarrow}_{[\![p]\!]} \texttt{false}, \sigma_1'''$.

(1) For the first case observe that it must be that,

$$[\![e]\!], \sigma_1'' \overset{}{\rightsquigarrow}_{[\![p]\!]} w, \sigma_1'''$$

Using (17).a, and $p,\gamma \vdash \sigma \approx \sigma_1''$ we can apply the inductive hypothesis to e and derive that: $e, \sigma \overset{}{\rightsquigarrow}_{p} w, \sigma'$ and $p,\gamma \vdash \sigma \approx \sigma'$. From the propagation rules for exceptions we derive that

$$[\![(c)e]\!]_{expr}, \sigma_1 \overset{}{\rightsquigarrow}_{[\![p]\!]} w, \sigma_1'$$

and also

$$(c)e, \sigma \overset{}{\rightsquigarrow}_{p} w, \sigma'$$

which prove the result.

(2) Assume that the evaluation of e_1 produces $\texttt{false}$. Therefore

$$[\![e]\!], \sigma_1'' \overset{}{\rightsquigarrow}_{[\![p]\!]} v, \sigma_2 \tag{19}$$

where $\sigma_1''' = \sigma_2[x \mapsto v]$ and $v \neq \texttt{null}$. From $p,\gamma \vdash \sigma \approx \sigma_1$ we have that $p,\gamma \vdash \sigma \approx \sigma_1''$ (x is not defined in γ). From (17).a, and (19), we can apply the inductive hypothesis to e, and derive that $e, \sigma \overset{}{\rightsquigarrow}_{p} v, \sigma'$ where $p,\gamma' \vdash \sigma' \approx \sigma_2$ and since x is not defined in γ' also $p,\gamma' \vdash \sigma' \approx \sigma_1'''$.
Observe that, (17).a, and Theorem 3.1, implies that $p,\sigma' \vdash v \lhd c'$, therefore $v = \iota$ and $\sigma'(\iota) = [\![\cdots]\!]^{c''}$ where $c'' \leq c'$. From $p,\gamma' \vdash \sigma' \approx \sigma_1'''$ also $\sigma_1'''(\iota) = [\![\texttt{imp} : \iota']\!]^{\texttt{Id}}$, $\sigma_1'''(\iota') = [\![\texttt{id} : \iota \cdots]\!]^{\mathit{theName}(c'')}$, and $\sigma_1'''(x) = \iota$. For the evaluation of e_2 there are two cases:
—$e_2, \sigma_1''' \overset{}{\rightsquigarrow}_{[\![p]\!]} \texttt{castExc}, \sigma_1'''$ if $c' \not\leq c$, or

—$e_2, \sigma_1''' \underset{[\![p]\!]}{\rightsquigarrow} \iota, \sigma_1'''$ if $c' \leq c$.
For the first case we have that from the propagation rules for exceptions

$$[\![(c)e]\!], \sigma_1 \underset{[\![p]\!]}{\rightsquigarrow} \texttt{castExc}, \sigma_1'$$

and from the application of the rule (*e-cast*) also $(c)e, \sigma \underset{p}{\rightsquigarrow} \texttt{castExc}, \sigma'$.
For the second case

$$[\![(c)e]\!], \sigma_1 \underset{[\![p]\!]}{\rightsquigarrow} \iota, \sigma_1'$$

where $\sigma_1' = \sigma_1'''[x \mapsto \sigma_1(x)]$. On the other hand we can apply rule (*cast*) and get $(c)e, \sigma \underset{p}{\rightsquigarrow} \iota, \sigma'$. From $p, \gamma' \vdash \sigma' \approx \sigma_1'''$ the definition of σ_1', and the fact that $\gamma(x) = \mathcal{U}df$ we have $p, \gamma' \vdash \sigma' \approx \sigma_1'$.

(3) If $e_1, \sigma_1'' \underset{[\![p]\!]}{\rightsquigarrow} \texttt{true}, \sigma_1'''$ then

$$[\![e]\!], \sigma_1'' \underset{[\![p]\!]}{\rightsquigarrow} \texttt{null}, \sigma_1'''$$

Applying (*f-cond*) we have

$$\texttt{if } e_1 \texttt{ then null else } e_2, \sigma_1'' \underset{p}{\rightsquigarrow} \texttt{null}, \sigma_1'''$$

and finally

$$[\![(c)e]\!], \sigma_1 \underset{p}{\rightsquigarrow} \texttt{null}, \sigma_1'$$

where $\sigma_1' = \sigma_1'''[x \mapsto \sigma_1(x)]$. From (17).a, we can apply the inductive hypothesis to e, and derive that $e, \sigma \underset{p}{\rightsquigarrow} \texttt{null}, \sigma'$ where $p, \gamma' \vdash \sigma' \approx \sigma_1'''$. Therefore, applying rule (*n-cast*) we have

$$(c)e, \sigma \underset{p}{\rightsquigarrow} \texttt{null}, \sigma'$$

From $p, \gamma' \vdash \sigma' \approx \sigma_1'''$ the definition of σ_1', and the fact that $\gamma(x) = \mathcal{U}df$ we have $p, \gamma' \vdash \sigma' \approx \sigma_1'$. This concludes the proof.

□

D. TRANSLATION OF EXAMPLE 5.1 IN JAVA 1.5

```
class Identity<X>{
    X imp;
    Identity(X imp){
        this.imp=imp;
    }
}

class FickleObject<X extends FickleObject<?>>{
    Identity<X> id;
}


class P<X extends P<?>> extends FickleObject<X>{
    int f1;
    Identity<? extends R<?>> m1(){
        Identity<? extends R<?>> s;
        S1Fix temp=new S1Fix();
        temp.id=new Identity<S1Fix>(temp);
```

```
        s=temp.id;
        return s;
    }
    int m2(Identity<? extends S1<?>> x){
        Identity<? extends S2<?>> x_S2;
        if(x!=null){
            S1<?> oldImp=x.imp;
            S2Fix temp=new S2Fix();
            temp.f1=oldImp.f1;
            temp.id=(Identity<S2Fix>) (Object) x;
            temp.id.imp=temp;
            x_S2=temp.id;
        }
        return 1;
    }
    int m3(Identity<? extends S1<?>> x){
        int temp=this.id.imp.m2(x);
        return x.imp.m(temp);
    }
    int m4 (Identity<? extends S1<X>> x){
        int temp=this.id.imp.m2(x);
        return x.imp.f1=temp;
    }
    int m(int x){
        return this.id.imp.f1=x;
    }
}

class R<X extends R<?>> extends P<X>{
}

class S1<X extends S1<?>> extends R<X>{
    int m5(Identity<? extends S2<?>> x){
        Identity<? extends S1<?>> this_id_S1=this.id;
        Identity<? extends S2<?>> this_id_S2;
        S1<?> oldImp=this_id_S1.imp;
        S2Fix temp=new S2Fix();
        temp.f1=oldImp.f1;
        temp.id=(Identity<S2Fix>) (Object) this_id_S1;
        temp.id.imp=temp;
        this_id_S2=temp.id;
        return this_id_S2.imp.f2=x.imp.f2;
    }
}

class S2<X extends S2<?>> extends R<X>{
    int f2;
    int m(int x){
        return this.id.imp.f2 = x;
    }
}
```

```
class FickleObjectFix extends FickleObject<FickleObjectFix>{}

class PFix extends P<PFix>{}

class RFix extends R<RFix>{}

class S1Fix extends S1<S1Fix>{}

class S2Fix extends S2<S2Fix>{}
```

Acknowledgments

We thank the anonymous referees for insightful and constructive comments, which greatly improved the submitted version.

REFERENCES

Ancona, D., Anderson, C., Damiani, F., Drossopoulou, S., Giannini, P., and Zucca, E. 2001. An effective translation of Fickle into Java (extended abstract). In *Italian Conf. on Theoretical Computer Science 2001*. Number 2202 in Lecture Notes in Computer Science. Springer, 215–234.

Ancona, D., Anderson, C., Damiani, F., Drossopoulou, S., Giannini, P., and Zucca, E. 2002. A type preserving translation of Fickle into Java. In *TOSCA'01*. ENTCS, vol. 62. Elsevier.

Ancona, D., Lagorio, G., and Zucca, E. 2003. Jam–designing a Java extension with mixins. *ACM Transactions on Programming Languages and Systems 25,* 5 (September), 641–712.

Anderson, C. 2001. Implementing Fickle, Imperial College, final year thesis.

Anderson, C. 2003. Isabella-Fickle translator. Available at `http://www.macs.hw.ac.uk/DART/software/isabella/index.html`.

Boyland, J. and Castagna, G. 1997. Parasitic methods: An implementation of multi-methods for Java. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1997*. ACM Press, 66–76.

Bracha, G., Odersky, M., Stoutmire, D., and Wadler, P. 1998. Making the future safe for the past: Adding genericity to the Java programming language. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1998*. ACM Press, 183–200.

Chambers, C. 1993. Predicate Classes. In *ECOOP'93 - European Conference on Object-Oriented Programming*. Number 707 in Lecture Notes in Computer Science. Springer, 268–296.

Clifton, C., Millstein, T., Leavens, G. T., and Chambers, C. 2006. MultiJava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems 28,* 3, 517–575.

Costanza, P. 2001. Dynamic object replacement and implementation-only classes. In *WCOP'01 (at ECOOP'01)*. Available from http://www.cs.uni-bonn.de/∼costanza/implementationonly.pdf.

Damiani, F., Dezani-Ciancaglini, M., and Giannini, P. 2004. Re-classification and multithreading: Fickle$_{MT}$. In *OOPS track at SAC'04*. Vol. 2. ACM Press, 1297–1304.

Drossopoulou, S., Damiani, F., Dezani-Ciancaglini, M., and Giannini, P. 2001. Fickle: Dynamic object re-classification. In *ECOOP'01 - European Conference on Object-Oriented Programming*. Number 2072 in Lecture Notes in Computer Science. Springer, 130–149.

Drossopoulou, S., Damiani, F., Dezani-Ciancaglini, M., and Giannini, P. 2002. More dynamic object re-classification: Fickle$_{II}$. *Transactions On Programming Languages and Systems 24,* 2, 153–191.

Ernst, M. D., Kaplan, C., and Chambers, C. 1998. Predicate dispatching: A unified theory of dispatch. In *ECOOP'98 - European Conference on Object-Oriented Programming*. Number 1445 in Lecture Notes in Computer Science. Springer, 186–211.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley.

Hürsch, W. 1994. Should superclasses be abstract? In *ECOOP'94 - European Conference on Object-Oriented Programming*. Number 821 in Lecture Notes in Computer Science. Springer, 12–31.

Igarashi, A. and Pierce, B. C. 2002. On inner classes. *Information and Computation 177,* 1 (Aug.), 56–89.

Joy, B., Gosling, J., Steele, G., and Bracha, G. 2005. *The Java Language Specification (third edition)*. Addison-Wesley.

McDirmid, S., M.Flatt, and Hsieh, W. 2001. Jiazzi: New age components for old fashioned Java. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2001)*. ACM Press. SIGPLAN Notices.

Odersky, M. and Wadler, P. 1997. Pizza into Java: Translating theory into practice. In *ACM Symp. on Principles of Programming Languages 1997*. ACM Press.

Pierce, B. C. 2002. *Types and Programming Languages*. The MIT Press.

Serrano, M. 1999. Wide classes. In *ECOOP'99 - European Conference on Object-Oriented Programming*. Number 1628 in Lecture Notes in Computer Science. Springer, 391–415.

Taivalsaari, A. 1993. Object oriented programming with modes. *Journal of Object Oriented Programming 6,* 3, 27–32.