More Dynamic Object Reclassification: $\mathcal{F}ickle_{\text{II}}$

SOPHIA DROSSOPOULOU
Imperial College
FERRUCCIO DAMIANI and MARIANGIOLA DEZANI-CIANCAGLINI
Università di Torino
and
PAOLA GIANNINI
Università del Piemonte Orientale

Reclassification changes the class membership of an object at run-time while retaining its identity. We suggest language features for object reclassification, which extend an imperative, typed, class-based, object-oriented language.

We present our proposal through the language $\mathcal{F}ickle_{\text{II}}$. The imperative features, combined with the requirement for a static and safe type system, provided the main challenges. We develop a type and effect system for $\mathcal{F}ickle_{\text{II}}$ and prove its soundness with respect to the operational semantics. In particular, even though objects may be reclassified across classes with different members, there will never be an attempt to access nonexisting members.

Categories and Subject Descriptors: D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*classes and objects*; *inheritance*; *polymorphism*; 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: Object-oriented languages, type and effect systems

1. INTRODUCTION

In class-based, object-oriented programming, an object's behavior is determined by its class. Case or conditional statements should be avoided when variation

Partially supported by IST-2001-322222 MIKADO, IST-2001-33477 DART, MURST Cofin'00 AITCFA, MURST Cofin'01 COMETA, MURST Cofin'01 NAPOLI Projects, CNR-GNSAGA, and the EPSRC (Grant Ref: GR/L 76709).
Authors' addresses: S. Drossopoulou, Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, U.K.; email: sd@doc.ic.ac.uk; F. Damiani and M. Dezani-Ciancaglini, Dipartimento di Informatica, Università di Torino, Corso Svizzera 185, 10149 Torino, Italy; email: {damiani,dezani}@di.unito.it; P. Giannini, Università del Piemonte Orientale, Corso Borsalino 54, 15100 Alessandria, Italy; email: giannini@di.unito.it.

in behavior can be expressed by using different classes. Thus, students paying reduced and employees paying full conference fees are best described through distinct classes Stdt and Empl with different methods fee().

However, this elegant approach does not scale to the case where objects change classification. For example, how can we represent that mary, who was a Stdt, became an Empl? Usually, class-based programming languages do not provide mechanisms for objects to change their class membership. Two solutions are possible: either replace the original Stdt object by an Empl object, or merge the two classes Stdt and Empl into one, for example, StdtOrEmpl.

Neither solution is satisfactory. The first solution needs to trace and inform all references to mary. The second solution blurs in the same class the differences in behavior that were elegantly expressed through the class system. In fact, Scheer and Pringle [1998] list the lack of reclassification primitives as the first practical limitation of object-oriented programming.

We suggest language features that allow objects to change class membership dynamically, and so, the class of the object pointed at by mary changes from Stdt to Empl. We combine these features with a strong type system.

We take a programming perspective, and base our approach on an imperative, class-based language, where classes are types and subclasses are subtypes,[1] and where methods are defined inside classes and selected depending on the class of the receiver. We achieve dynamic reclassification of objects by explicitly changing the class membership of objects.

We describe our approach through the language $\mathcal{F}ickle_{\mathrm{II}}$: A *reclassification* operation changes the class membership of an object while preserving its identity; it maintains all fields common to the original and the target class and initializes the extra fields. *State* classes are possible targets of reclassifications; in that sense, they represent an object's possible states. *Root* classes are the superclasses of such state classes and declare all the members common to them. Only nonstate classes may appear as types of fields. This limitation is motivated by type soundness, see Example (4) on page 157. $\mathcal{F}ickle_{\mathrm{II}}$ is statically typed, with a type and effect system [Lucassen and Gifford 1988; Talpin and Jouvelot 1992], which determines the reclassification effect of an expression on the receiver and on all other objects. The type system is sound, so that terminating execution of a well-typed expression produces a value of the expected type, or a null-pointer exception, but does *not* get stuck.

$\mathcal{F}ickle_{\mathrm{II}}$ is an extension of $\mathcal{F}ickle$ [Drossopoulou et al. 2001]: $\mathcal{F}ickle$ only allowed the receiver to be of a state class and to be reclassified, whereas $\mathcal{F}ickle_{\mathrm{II}}$ also allows parameters to be of a state class and to be reclassified. Although we do not describe this formally, this extension applies implicitly to local variables as well.

This article is organized as follows: In Section 2, we introduce $\mathcal{F}ickle_{\mathrm{II}}$ informally using an example. In Sections 3, 4, and 5, we outline $\mathcal{F}ickle_{\mathrm{II}}$: the syntax, operational semantics, typing rules, and we state type soundness. In Section 6,

[1]Even though the object-based paradigm may be more fundamental [Abadi and Cardelli 1996] and though classes should not be types, and subclasses should not imply subtypes [Canning et al. 1989], current praxis predominantly uses languages of the opposite philosophy.

```
abstract root class Player extends Object{
  bool brave;

  abstract bool wake() { } ;
  abstract Weapon kissed() {Player};
}

state class Frog extends Player{
  Vocal pouch;

  bool wake() { } {pouch.blow() ; brave}
  Weapon kissed(){Player}{this⇓Prince; sword := new Weapon}
}

state class Prince extends Player{
  Weapon sword;

  bool wake() { } {sword.swing(); brave}
  Weapon kissed(){Player}{sword}
  Frog cursed(){Player}{this⇓Frog; pouch := new Vocal; this}
}

class Princess extends Object{
  bool walk1(Frog mate){Player}{mate.wake(); mate.kissed(); mate.wake()}
  Weapon walk2(Frog mate){Player}{mate⇓Prince; mate.sword := new Weapon}
}
```

Fig. 1. Program P_{pl}—players with reclassification.

we describe design alternatives. In Section 7, we discuss the implementation of $\mathcal{F}ickle_{II}$. In Section 8, we compare our proposal with other approaches. In Section 9, we discuss the limits of our approach and future work. The appendix contains an example, some definitions, and the proof of the type soundness result.

2. AN EXAMPLE

In Figure 1, we give an example inspired by adventure games.[2] We use a syntax similar to Java's, and define a class Player with subclasses Frog and Prince. When woken up, a frog inflates its pouch, while a prince swings his sword. When kissed, a frog turns into a prince; when cursed, a prince turns into a frog.

We have introduced two new kinds of classes: **state** and **root** classes. The state classes are the classes that may serve as targets of reclassifications. Such classes *cannot* be used as types for fields; in our example Frog and Prince. The

[2]A computer science example dealing with linked lists can be found in Appendix A.

root classes define the fields and methods common to their state subclasses; in our example, class Player defines the field brave, and the two abstract methods wake and kissed. The subclasses of root classes must be state classes.[3] A state class c must have a (possibly indirect) root superclass c′; objects of class c may be reclassified to any subclass of c′.

Annotations like { } and {Player} before method bodies are called *effects*. Effects list the root classes of all objects that may be reclassified by invocation of that method.

Methods with the empty effect { }, for example, wake, may not cause any reclassification. Methods with nonempty effects, for example, kissed and cursed with effect {Player}, may reclassify objects of a subclass of their effect; in our case of Player. Such reclassifications may be caused by *reclassification expressions* (e.g., this⇓Prince in method kissed of class Frog, or mate⇓Prince in method walk2 of class Princess), or by further method calls (e.g., mate.kissed () in method walk1 of class Princess).

The method body of kissed in class Frog contains the reclassification expression this⇓Prince. At the beginning of the method, the receiver is an object of class Frog; therefore, it contains the fields brave and pouch, but not the field sword. After execution of this⇓Prince the receiver is of class Prince, and therefore sword is accessible, while pouch is not, and brave retains its value. This mechanism supports the transmission of some information from the object before the reclassification to the object after the reclassification.

Consider the instructions in the method body of walk1 in class Princess:

```
1. mate.wake();        // inflates pouch
2. mate.kissed();                                   (1)
3. mate.wake();        // swings sword
```

Suppose that the parameter mate is bound to a Frog object with field brave containing true. After line 2., the object is reclassified to Prince with the same value for brave. Therefore, the call of wake in line 1. selects the method from Frog, and inflates the pouch, while the call of wake in line 3. selects the method from Prince, and swings the sword.

Reclassification removes from the object all fields that are not defined in its root superclass and adds the remaining fields of the target class. For example, after line 2. in Example (1), the object denoted by mate has a sword but not a pouch. For example, consider the instructions in the method body of walk2 in class Princess:

```
1. mate⇓Prince;
2. mate.sword := new Weapon;                        (2)
```

Let the parameter mate be bound to a Frog object. After line 1., the object is reclassified to Prince. Therefore, in line 2. field sword can be selected.

[3]A root class is the first non-state superclass of a state class. The property that *all* its subclasses are state classes allows for a simpler type system (compare with Drossopoulou et al. [1999a]). 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.

Reclassification is transparent to aliasing. For instance, in Example (3)

```
     Player p1, p2;
1.   p1 := new Prince;
2.   p2 := p1;                                    (3)
3.   p1.cursed();
4.   p2.wake();          // inflates pouch
```

line 3. reclassifies the object, and not the binding. Therefore, the call of method wake in line 4. selects the method from Frog. Thus, through aliasing, one reclassification may affect several variables; in Example (3), the reclassification affects both p1 and p2.

Because the class membership of objects of state class is transient, access to their members (e.g., to sword from class Prince) is only legal in contexts where it is certain that the object belongs to the particular class. This can be done for "local" entities, that is, 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 friend in the following is illegal:

```
class Witch{
   Prince friend;          // illegal!

   Weapon search(){ }{ friend.sword; }
}
```

If, on the other hand, the declaration of field friend in class Witch were legal, then, in the following code

```
...  // w a Witch, p1 a Prince,  w.friend alias of p1:
1.    p1.cursed();                                 (4)
2.    w.search();          // error!
```

where p1 is an alias of w.friend,[4] the execution of line 1. would reclassify the object bound to w.friend to Frog, and the field access w.friend.sword inside the call of w.search() 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, or as return types for methods. In our example, the state class Frog is the parameter type of mate of method walk1 in class Princess, and the return type of method cursed in class Prince.

Before introducing formally the language and its semantics, we present further examples that demonstrate some special features of our typing system.

The class-membership of an object denoted by an identifier (`this` or a parameter) of state class is transient. Thus, the type of an identifier may change

[4]For example, through execution of p1 := **new** Prince ; w := **new** Witch ; w.friend := p1.

within a method body. In Example (5), we add the method freed to the class Frog:

```
Weapon freed(){Player}{
  this.pouch;      // type correct, this is currently a Frog
  this.sword;      // type incorrect, this is currently a Frog
  this⇓Prince;                                                    (5)
  this.pouch;      // type incorrect, this is currently a Prince
  this.sword       /* type correct, this is currently a Prince */ }
```

this has type Frog before the reclassification, and it has type Prince afterwards. Similar changes are possible for the types of the parameters:

```
Weapon meet(Frog x){Player}{
  x.pouch;         // type correct, x is currently a Frog
  x⇓Prince;                                                       (6)
  x.sword          /* type correct, x is currently a Prince */ }
```

Changes to the type of an identifier may be caused either by explicit reclassifications, as in Examples (5) and (6), or by potential, indirect reclassification, due to aliasing, as in method play of Example (7).

```
bool play(Player x1, Frog x2){Player}{
  x2.pouch;        // type correct, x2 is currently a Frog
  x1.kissed();     // reclassifies x1 and all its aliases          (7)
  x2.pouch;        // type incorrect, x2 is currently a Player
  x2.brave         /* type correct, x2 is currently a Player */ }
```

Since at the time of the call x_1 and x_2 might be aliases, the reclassification x_1.kissed() might reclassify the object pointed at by x_2 as well. In order to capture such potential reclassifications, each method declares as its effect the set of root classes of objects that may be reclassified through its execution. In our case, kissed has effect {Player}. After the call x_1.kissed (), the type of x_2 is Player, that is, the application of the effect {Player} to the class Frog.

3. SYNTAX

In Figure 2, we give the syntax of $\mathcal{F}ickle_{\mathrm{II}}$. We use standard extended BNF, where a $[-]$ pair means optional, and A^* means zero or more repetitions of A. We follow the convention that nonterminals appear as *nonTerm*, keywords appear as **keyword**, literals appear as `literal` and identifiers appear as identifier. We omit separators like ";" or "," where they are obvious. Expressions are usually called e, e′, e_1 etc., and values are usually called v, v′, v_1 etc. By id, id′, etc., we denote either this or a parameter name (x, x′, etc.).

A program is a sequence of class definitions. A class definition may be preceded by the keyword **state**, or **root**. State classes describe the properties of an object while it satisfies some conditions; when it no longer satisfies these conditions, it can be explicitly reclassified to another state class. For example, Prince describes princes that have not been cursed; if they are cursed, they are

progr	::=	*class**
class	::=	[**root** | **state**] **class** *c* **extends** *c* { *field* * *meth**}
field	::=	*type f*
meth	::=	*type m* (*par* *) *eff* {*e*}
type	::=	**bool** | *c*
par	::=	*type x*
eff	::=	{*c**}
e	::=	**if** *e* **then** *e* **else** *e* | *var* := *e* | *e* ; *e* | *sVal* | `this` | *var* | **new** *c* | *e.m*(*e**) | *id*⇓*c*
var	::=	*x* | *e.f*
sVal	::=	`true` | `false` | `null`
id	::=	`this` | *x*

with the following conventions

c	::=	c | c′ | c_i | d | ⋯	for class names
f	::=	f | f′ | f_i | ⋯	for field names
m	::=	m | m′ | m_i | ⋯	for method names
x	::=	x | x′ | x_i | ⋯	for parameter names

Fig. 2. Syntax of $\mathcal{F}ickle_{II}$.

reclassified to Frog. Root classes abstract over state classes.[5] Any subclass of a state or a root class must be a state class. Objects of a state class c may be reclassified to class c′, where c′ must be a subclass of the uniquely defined root superclass of c. For example, Player abstracts over Frog and Prince; objects of class Frog may be reclassified to Prince, and vice-versa.

Objects of a nonstate, nonroot class c behave like regular Java objects, that is, they are never reclassified. However, objects pointed at by an identifier id (or field f) declared of type c *may* be reclassified. Namely, if c had two state subclasses d and d′, and id (or f) refers to an object of class d, the object may be reclassified to d′. Our type system ensures that this reclassification will not cause accesses to fields or methods that are not defined for the object.

The type of fields may be either Boolean or a non-state class; we call such types *field types*. Thus, fields may point to objects that *change class*, but these changes do *not affect* their type. In contrast, the type of `this` and parameters may be a state or root class; these variables may also point to objects that *change class*, and these changes *affect* their type.

Objects are created with the expression **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.

Method declarations have the shape:

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

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

where t is the result type, $\mathsf{t}_1, \ldots, \mathsf{t}_q$ are the types of the formal parameters $\mathsf{x}_1, \ldots, \mathsf{x}_q$, and e is the body. The effect consists of root classes $\mathsf{c}_1, \ldots, \mathsf{c}_n$, with $\mathsf{n} \geq 0$.[6]

We require the inheritance hierarchy to be a tree, root classes to extend only nonroot and nonstate classes, and state classes to extend either root classes or state classes. The judgment $\vdash \mathsf{P} \diamond_h$ (the inheritance hierarchy is well formed) asserts that program P satisfies these conditions, and is defined in Appendix B.

Remark 1. Section 2 and Appendix A follow a slightly more liberal syntax, with abstract classes, abstract methods, and the implicit use of `this` to access fields and methods from the current class.

4. OPERATIONAL SEMANTICS

We give a structural operational semantics that rewrites pairs of expressions and stores into pairs of values, exceptions, or errors, and stores in the context of a program P. The signature of the rewriting relation $\rightsquigarrow$ is:

$$\rightsquigarrow : progr \rightarrow e \times store \rightarrow (val \cup dev) \times store.$$

The store maps `this` to an address, parameters to values, and addresses to objects. Values are addresses, or the source language values as in Section 3. 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. As we will show, execution of well-typed expressions never produces a stuck error, although it may throw a null pointer exception. We denote stores with σ, addresses with ι, exceptions and errors with dv.

$$\begin{array}{lll} store & = & (\{\texttt{this}\} \rightarrow addr) \cup (x \rightarrow val) \cup (addr \rightarrow object) \\ val & = & sVal \cup addr \\ dev & = & \{\texttt{nullPntrExc}, \texttt{stuckErr}\} \\ object & = & \{\, [[\mathsf{f}_1 : \mathsf{v}_1, \ldots, \mathsf{f}_r : \mathsf{v}_r]]^{\mathsf{c}} \mid \mathsf{f}_1, \ldots, \mathsf{f}_r \text{ are fields identifiers,} \\ & & \qquad\qquad \mathsf{v}_1, \ldots, \mathsf{v}_r \in val, \text{ and } \mathsf{c} \text{ is a class name}\} \end{array}$$

We need some operations on objects and stores. For object $\mathsf{o} = [[\mathsf{f}_1 : \mathsf{v}_1 \cdots \mathsf{f}_l : \mathsf{v}_l \cdots \mathsf{f}_r : \mathsf{v}_r]]^{\mathsf{c}}$, store σ, value v, address ι, identifier or address z, field identifier f, we define:

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

—*object update* $\quad \mathsf{o}[\mathsf{f} \mapsto \mathsf{v}] = [[\mathsf{f}_1 : \mathsf{v}_1 \cdots \mathsf{f}_l : \mathsf{v} \cdots \mathsf{f}_r : \mathsf{v}_r]]^{\mathsf{c}}$,
where $\mathsf{f}_l = \mathsf{f}$ for some $l \in 1, \ldots, r$,

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

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

Figures 3, 4, and 5 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 reclassification.

[6]Extending $\mathcal{F}ickle_{\mathrm{II}}$ to allow methods to have local variables would be straightforward. Their types could be any types, *including* state classes. The typing rules for local variables would be the same as for parameters.

$$\frac{\mathsf{e}, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \texttt{true}\ , \sigma'' \quad \mathsf{e}_1, \sigma'' \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{v}, \sigma'}{\textbf{if}\ \mathsf{e}\ \textbf{then}\ \mathsf{e}_1\ \textbf{else}\ \mathsf{e}_2, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{v}, \sigma'} \qquad \frac{\mathsf{e}, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \texttt{false}\ , \sigma'' \quad \mathsf{e}_2, \sigma'' \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{v}, \sigma'}{\textbf{if}\ \mathsf{e}\ \textbf{then}\ \mathsf{e}_1\ \textbf{else}\ \mathsf{e}_2, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{v}, \sigma'}$$

$$\frac{\sigma(\mathsf{x}) \neq \mathcal{U}df \quad \mathsf{e}, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{v}, \sigma'}{\mathsf{x}{:=}\mathsf{e}, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{v}, \sigma'[\mathsf{x} \mapsto \mathsf{v}]} \qquad \frac{\mathsf{e}, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \iota, \sigma'' \quad \mathsf{e}', \sigma'' \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{v}, \sigma''' \quad \sigma'''(\iota)(\mathsf{f}) \neq \mathcal{U}df \quad \sigma' = \sigma'''[\iota \mapsto \sigma'''(\iota)[\mathsf{f} \mapsto \mathsf{v}]]}{\mathsf{e}.\mathsf{f}\ :=\ \mathsf{e}', \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{v}, \sigma'}$$

$$\frac{\mathsf{e}_1, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{v}', \sigma'' \quad \mathsf{e}_2, \sigma'' \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{v}, \sigma'}{\mathsf{e}_1; \mathsf{e}_2, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{v}, \sigma'} \qquad \frac{}{\mathsf{v}, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{v}, \sigma}$$

$$\frac{\sigma(\mathsf{id}) \neq \mathcal{U}df}{\mathsf{id}, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \sigma(\mathsf{id}), \sigma} \qquad \frac{\mathsf{e}, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \iota, \sigma' \quad \sigma'(\iota)(\mathsf{f}) \neq \mathcal{U}df}{\mathsf{e}.\mathsf{f}, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \sigma'(\iota)(\mathsf{f}), \sigma'}$$

$$\frac{\begin{array}{l}\mathcal{F}s(\mathsf{P}, \mathsf{c}) = \{\mathsf{f}_1, \ldots, \mathsf{f}_\mathsf{r}\} \\ \mathsf{v}_\mathsf{l} \text{ initial for } \mathcal{F}(\mathsf{P}, \mathsf{c}, \mathsf{f}_\mathsf{l}) \quad (\forall \mathsf{l} \in \{1, \ldots, \mathsf{r}\}) \\ \iota \text{ is new in } \sigma\end{array}}{\textbf{new}\ \mathsf{c}, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \iota, \sigma[\iota \mapsto [[\mathsf{f}_1 : \mathsf{v}_1, \ldots, \mathsf{f}_\mathsf{r} : \mathsf{v}_\mathsf{r}]]^{\mathsf{c}}]}$$

$$\frac{\begin{array}{l}\mathsf{e}_0, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \iota, \sigma_0 \\ \mathsf{e}_\mathsf{i}, \sigma_{\mathsf{i}-1} \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{v}_\mathsf{i}, \sigma_\mathsf{i} \quad (\forall \mathsf{i} \in \{1, \ldots, \mathsf{n}\}) \\ \sigma_\mathsf{n}(\iota) = [[\cdots]]^{\mathsf{c}} \\ \mathcal{M}(\mathsf{P}, \mathsf{c}, \mathsf{m}) = \mathsf{t}\ \mathsf{m}(\mathsf{t}_1\ \mathsf{x}_1, \ldots, \mathsf{t}_\mathsf{n}\ \mathsf{x}_\mathsf{n})\ \phi\ \{\mathsf{e}\} \\ \sigma' = \sigma_\mathsf{n}[\texttt{this} \mapsto \iota, \mathsf{x}_1 \mapsto \mathsf{v}_1, \ldots, \mathsf{x}_\mathsf{n} \mapsto \mathsf{v}_\mathsf{n}] \\ \mathsf{e}, \sigma' \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{v}, \sigma''\end{array}}{\mathsf{e}_0.\mathsf{m}(\mathsf{e}_1, \ldots, \mathsf{e}_\mathsf{n}), \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{v}, \sigma''[\texttt{this} \mapsto \sigma_\mathsf{n}(\texttt{this}), \mathsf{x}_1 \mapsto \sigma_\mathsf{n}(\mathsf{x}_1), \ldots, \mathsf{x}_\mathsf{n} \mapsto \sigma_\mathsf{n}(\mathsf{x}_\mathsf{n})]}$$

$$\frac{\begin{array}{l}\sigma(\mathsf{id}) = \iota \\ \sigma(\iota) = [[\cdots]]^{\mathsf{c}} \\ \mathcal{F}s(\mathsf{P}, \mathcal{R}(\mathsf{P}, \mathsf{c})) = \{\mathsf{f}_1, \ldots, \mathsf{f}_\mathsf{r}\} \\ \mathsf{v}_\mathsf{l} = \sigma(\iota)(\mathsf{f}_\mathsf{l}) \quad (\forall \mathsf{l} \in \{1, \ldots, \mathsf{r}\}) \\ \mathcal{F}s(\mathsf{P}, \mathsf{d}) \setminus \{\mathsf{f}_1, \ldots, \mathsf{f}_\mathsf{r}\} = \{\mathsf{f}_{\mathsf{r}+1}, \ldots, \mathsf{f}_{\mathsf{r}+\mathsf{q}}\} \\ \mathsf{v}_\mathsf{l} \text{ initial for } \mathcal{F}(\mathsf{P}, \mathsf{d}, \mathsf{f}_\mathsf{l}) \quad (\forall \mathsf{l} \in \{\mathsf{r}+1, \ldots, \mathsf{r}+\mathsf{q}\})\end{array}}{\mathsf{id}{\Downarrow}\mathsf{d}, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \iota, \sigma[\iota \mapsto [[\mathsf{f}_1 : \mathsf{v}_1, \ldots, \mathsf{f}_{\mathsf{r}+\mathsf{q}} : \mathsf{v}_{\mathsf{r}+\mathsf{q}}]]^{\mathsf{d}}]} \qquad \frac{\mathsf{id}, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \texttt{null}, \sigma'}{\mathsf{id}{\Downarrow}\mathsf{d}, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \texttt{null}, \sigma'}$$

Fig. 3. Execution—without exceptions and errors.

$$\frac{\mathsf{e},\ \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathtt{null},\ \sigma'}{\begin{array}{l}\mathsf{e.f} := \mathsf{e}',\ \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathtt{nullPntrExc},\ \sigma' \\ \mathsf{e.f},\ \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathtt{nullPntrExc},\ \sigma' \\ \mathsf{e.m}(\mathsf{e}_1, \ldots, \mathsf{e}_n),\ \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathtt{nullPntrExc},\ \sigma'\end{array}}$$

$$\frac{\begin{array}{l}\mathsf{e},\ \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{v},\ \sigma' \\ \mathsf{v} \neq \mathtt{true} \quad \text{and} \quad \mathsf{v} \neq \mathtt{false}\end{array}}{\textbf{if}\ \mathsf{e}\ \ \textbf{then}\ \ \mathsf{e}_1\ \ \textbf{else}\ \mathsf{e}_2,\ \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathtt{stuckErr},\ \sigma'} \qquad \frac{\sigma(\mathsf{x}) = \mathtt{true} \ \text{ or } \sigma(\mathsf{x}) = \mathtt{false}}{\mathsf{x}\Downarrow\mathsf{c},\ \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathtt{stuckErr},\ \sigma}$$

$$\frac{\sigma(\mathsf{x}) = \mathcal{U}df}{\begin{array}{l}\mathsf{x},\ \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathtt{stuckErr},\ \sigma \\ \mathsf{x} := \mathsf{e},\ \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathtt{stuckErr},\ \sigma \\ \mathsf{x}\Downarrow\mathsf{c},\ \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathtt{stuckErr},\ \sigma\end{array}} \qquad \frac{\begin{array}{l}\mathsf{e},\ \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{v},\ \sigma' \\ \mathsf{v} \neq \mathtt{null} \\ \mathsf{v} \notin addr\end{array}}{\begin{array}{l}\mathsf{e.f},\ \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathtt{stuckErr},\ \sigma' \\ \mathsf{e.f} := \mathsf{e}',\ \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathtt{stuckErr},\ \sigma'\end{array}}$$

$$\frac{\begin{array}{l}\mathsf{e},\ \sigma \overset{\mathsf{p}}{\rightsquigarrow} \iota,\ \sigma' \\ \sigma'(\iota)(\mathsf{f}) = \mathcal{U}df\end{array}}{\mathsf{e.f},\ \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathtt{stuckErr},\ \sigma'} \qquad \frac{\begin{array}{l}\mathsf{e},\ \sigma \overset{\mathsf{p}}{\rightsquigarrow} \iota,\ \sigma'' \\ \mathsf{e}',\ \sigma'' \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{v},\ \sigma' \\ \sigma'(\iota)(\mathsf{f}) = \mathcal{U}df\end{array}}{\mathsf{e.f} := \mathsf{e}',\ \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathtt{stuckErr},\ \sigma'}$$

$$\frac{\begin{array}{l}\mathsf{e}_0,\ \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{v},\ \sigma_0 \\ \mathsf{v} \neq \mathtt{null} \\ \mathsf{v} \notin addr \text{ or } \sigma_0(\mathsf{v}) = \mathcal{U}df\end{array}}{\mathsf{e}_0.\mathsf{m}(\mathsf{e}_1, \ldots, \mathsf{e}_n),\ \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathtt{stuckErr},\ \sigma_0} \qquad \frac{\begin{array}{l}\mathsf{e}_0,\ \sigma \overset{\mathsf{p}}{\rightsquigarrow} \iota,\ \sigma_0 \\ \mathsf{e}_i,\ \sigma_{i-1} \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{v}_i,\ \sigma_i \quad (\forall \mathsf{i} \in \{1, \ldots, \mathsf{n}\}) \\ \sigma_n(\iota) = [[\cdots]]^{\mathsf{c}} \\ \mathcal{M}(\mathsf{P}, \mathsf{c}, \mathsf{m}) = \mathcal{U}df\end{array}}{\mathsf{e}_0.\mathsf{m}(\mathsf{e}_1, \ldots, \mathsf{e}_n),\ \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathtt{stuckErr},\ \sigma_n}$$

Fig. 4. Execution—Generation of exceptions and errors.

For method calls, $\mathsf{e}_0.\mathsf{m}(\mathsf{e}_1, \ldots, \mathsf{e}_n)$, we evaluate the receiver e_0, obtaining an address, say ι. We then evaluate the arguments, $\mathsf{e}_1, \ldots, \mathsf{e}_n$. We find the appropriate body by looking up m in the class of the object at address ι—we use the function $\mathcal{M}(\mathsf{P}, \mathsf{c}, \mathsf{m})$ that returns the definition of method m in class c going through the class hierarchy, if needed (see Appendix B). 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]

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

$$\frac{\mathsf{e}, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{dv}, \sigma' \quad \text{or} \quad (\mathsf{e}, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathtt{true}\ , \sigma'' \ \text{ and } \ \mathsf{e}_1, \sigma'' \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{dv}, \sigma') \quad \text{or} \quad (\mathsf{e}, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathtt{false}\ , \sigma'' \ \text{ and } \ \mathsf{e}_2, \sigma'' \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{dv}, \sigma')}{\textbf{if } \mathsf{e} \textbf{ then } \mathsf{e}_1 \textbf{ else } \mathsf{e}_2, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{dv}, \sigma'}$$

$$\frac{\mathsf{e}_1, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{dv}, \sigma' \quad \text{or} \quad (\mathsf{e}_1, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{v}, \sigma'' \ \text{ and } \ \mathsf{e}_2, \sigma'' \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{dv}, \sigma')}{\mathsf{e}_1; \mathsf{e}_2, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{dv}, \sigma'}$$

$$\frac{\mathsf{e}, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{dv}, \sigma'}{\begin{array}{l}\mathsf{x} := \mathsf{e}, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{dv}, \sigma' \\ \mathsf{e.f}, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{dv}, \sigma' \\ \mathsf{e.m}(\mathsf{e}_1, \ldots, \mathsf{e}_n), \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{dv}, \sigma' \\ \mathsf{e.f} := \mathsf{e}', \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{dv}, \sigma'\end{array}} \qquad \frac{\begin{array}{c}\mathsf{e}, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \iota, \sigma'' \\ \mathsf{e}', \sigma'' \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{dv}, \sigma'\end{array}}{\mathsf{e.f} := \mathsf{e}', \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{dv}, \sigma'}$$

$$\frac{\begin{array}{l}\mathsf{e}_0, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \iota, \sigma_0 \\ \mathsf{e}_i, \sigma_{i-1} \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{v}_i, \sigma_i \ \ (\forall i \in \{1, \ldots, q\},\ q < n) \\ \mathsf{e}_{q+1}, \sigma_q \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{dv}, \sigma_{q+1}\end{array}}{\mathsf{e}_0.\mathsf{m}(\mathsf{e}_1, \ldots, \mathsf{e}_n), \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{dv}, \sigma_{q+1}}$$

$$\frac{\begin{array}{l}\mathsf{e}_0, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \iota, \sigma_0 \\ \mathsf{e}_i, \sigma_{i-1} \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{v}_i, \sigma_i \ \ (\forall i \in \{1, \ldots, n\}) \\ \sigma_n(\iota) = [[\cdots]]^{\mathsf{c}} \\ \mathcal{M}(\mathsf{P}, \mathsf{c}, \mathsf{m}) = \mathsf{t}\ \mathsf{m}(\mathsf{t}_1\ \mathsf{x}_1, \ldots, \mathsf{t}_n\ \mathsf{x}_n)\ \phi\ \{\mathsf{e}\} \\ \sigma' = \sigma_n[\mathtt{this} \mapsto \iota, \mathsf{x}_1 \mapsto \mathsf{v}_1, \ldots, \mathsf{x}_n \mapsto \mathsf{v}_n] \\ \mathsf{e}, \sigma' \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{dv}, \sigma''\end{array}}{\mathsf{e}_0.\mathsf{m}(\mathsf{e}_1, \ldots, \mathsf{e}_n), \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{dv}, \sigma''[\mathtt{this} \mapsto \sigma_n(\mathtt{this}), \mathsf{x}_1 \mapsto \sigma_n(\mathsf{x}_1), \ldots, \mathsf{x}_n \mapsto \sigma_n(\mathsf{x}_n)]}$$

Fig. 5. Execution—propagation of exceptions and errors.

For reclassification expressions, $\mathsf{id} \Downarrow \mathsf{d}$, we find the address of id, which points to an object of 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 $\mathcal{R}(\mathsf{P}, \mathsf{t})$, defined by

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

denotes the least superclass of t which is not a state class, if t is a class, and denotes t itself if t is not a class. For example, $\mathcal{R}(\mathsf{P}_{\mathsf{pl}}, \mathsf{Prince}) = \mathsf{Player}$, $\mathcal{R}(\mathsf{P}_{\mathsf{pl}}, \mathsf{Princess}) = \mathsf{Princess}$, and $\mathcal{R}(\mathsf{P}_{\mathsf{pl}}, \mathsf{bool}) = \mathsf{bool}$. Moreover, $\mathcal{F}s(\mathsf{P}, \mathsf{c})$ denotes the set of fields defined in class c, and $\mathcal{F}(\mathsf{P}, \mathsf{c}, \mathsf{f})$ the type of field f in class c (see Appendix B).

For example, for store σ_1, with $\sigma_1(\mathsf{x}_1) = \iota$, and $\sigma_1(\iota) = [[\mathsf{brave} : \texttt{true}, \mathsf{sword} : \iota']]^{\mathsf{Prince}}$, $\sigma_1(\iota') = [[\cdots]]^{\mathsf{Weapon}}$, we have

$$\mathsf{x}_1 \Downarrow \mathsf{Frog}, \sigma_1 \overset{\rightarrow}{\mathsf{p}_{\mathsf{pl}}} \iota, \sigma_2,$$

where $\sigma_2 = \sigma_1[\iota \mapsto [[\mathsf{brave} : \texttt{true}, \mathsf{pouch} : \texttt{null}]]^{\mathsf{Frog}}]$ that is, we obtain an object of class Frog with unmodified field brave.

In well-typed programs, $\mathcal{R}(\mathsf{P}, \mathsf{c}) = \mathcal{R}(\mathsf{P}, \mathsf{d})$ always holds, and c and d must be state or root classes. This implies that reclassification 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 reclassification code for each state class. The rule for reclassification uses the types of the fields to initialize the fields, similarly to the rule for object creation.

5. TYPING

5.1 Widening, Environments, Effects

The following assertions, defined in Figure 10 of Appendix B, describe kinds of classes, and the widening relationship between types:

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

—$\mathsf{P} \vdash \mathsf{c} \diamond_{rt}$ means that c is a reclassifiable type, that is, either a root or a state class,

—$\mathsf{P} \vdash \mathsf{t} \diamond_{ft}$ means that t is a field type, that is, either **bool** or a nonstate class, and

—$\mathsf{P} \vdash \mathsf{t} \leq \mathsf{t}'$ means that type t′ widens type t, that is, t is a subclass of, or identical to, t′.

In our example, $\mathsf{P}_{\mathsf{pl}} \vdash \mathsf{Player} \diamond_{ct}$, $\mathsf{P}_{\mathsf{pl}} \vdash \mathsf{Player} \diamond_{rt}$, $\mathsf{P}_{\mathsf{pl}} \vdash \mathsf{Player} \diamond_{ft}$, $\mathsf{P}_{\mathsf{pl}} \vdash \mathsf{Frog} \diamond_{ct}$, and $\mathsf{P}_{\mathsf{pl}} \vdash \mathsf{Frog} \diamond_{rt}$, but $\mathsf{P}_{\mathsf{pl}} \not\vdash \mathsf{Frog} \diamond_{ft}$.

Environments, Γ, map parameter names to types, and the receiver `this` to a class. They have the form $\{\mathsf{x}_1 : \mathsf{t}_1, \ldots \mathsf{x}_n : \mathsf{t}_n, \texttt{this} : \mathsf{c}\}$. Lookup, $\Gamma(\mathsf{id})$, and update, $\Gamma[\mathsf{id} \mapsto \mathsf{t}]$, have the usual meaning, and are defined in Figure 11 of Appendix B.

An effect, ϕ, is a set $\{\mathsf{c}_1, \ldots, \mathsf{c}_n\}$ of root classes; it means that any object of a state subclass of c_i may be reclassified to any state subclass of c_i. The empty effect, $\{\,\}$, guarantees that no object is reclassified. Effects are well formed, that is, $\mathsf{P} \vdash \{\mathsf{c}_1, \ldots, \mathsf{c}_n\} \diamond$, iff $\mathsf{c}_1, \ldots, \mathsf{c}_n$ are distinct root classes. Thus, $\mathsf{P} \vdash \{\mathsf{c}_1, \ldots, \mathsf{c}_n\} \diamond$ implies that c_i are not subclasses of each other.

5.2 Typing Specialities

Typing an expression e in the context of program P and environment Γ involves three components, namely

$$\mathsf{P}, \Gamma \vdash \mathsf{e} : \mathsf{t} \,[]\, \Gamma' \,[]\, \phi,$$

where t is the type of the value returned by evaluation of e, the environment Γ' contains the type of `this` and of the parameters after evaluation of e, and ϕ conservatively estimates the reclassification effect of the evaluation of e on

objects. (Note that in Drossopoulou et al. [2001], where only `this` could be reclassified, typing had a slightly different format, that is, $\mathsf{P}, \Gamma \vdash \mathsf{e} : \mathsf{t} \,[]\, \mathsf{c} \,[]\, \phi$, where c denoted the type of `this` after execution of e.)

For example, consider environments, Γ_0, Γ_1, with $\Gamma_0(\texttt{this}) = \mathsf{Frog}$, $\Gamma_1(\texttt{this}) = \mathsf{Prince}$. Typing the body of freed in Example (5), after erasing the type incorrect expressions, we have:

$$\begin{aligned}
&\mathsf{P_{pl}}, \Gamma_0 \vdash \texttt{this}.\mathsf{pouch} : \ \mathsf{Vocal} \,[]\, \Gamma_0 \,[]\, \{\,\}\\
&\mathsf{P_{pl}}, \Gamma_0 \vdash \texttt{this}\Downarrow\mathsf{Prince} : \ \mathsf{Prince} \,[]\, \Gamma_1 \,[]\, \{\mathsf{Player}\}\\
&\mathsf{P_{pl}}, \Gamma_1 \vdash \texttt{this}.\mathsf{sword} : \ \mathsf{Weapon} \,[]\, \Gamma_1 \,[]\, \{\,\}.
\end{aligned}$$

Similarly, Example (6) is typed in environments Γ_2, Γ_3, with $\Gamma_2(\mathsf{x}) = \mathsf{Frog}$, $\Gamma_3(\mathsf{x}) = \mathsf{Prince}$:

$$\begin{aligned}
&\mathsf{P_{pl}}, \Gamma_2 \vdash \mathsf{x}.\mathsf{pouch};\ \mathsf{x}\Downarrow\mathsf{Prince} : \ \mathsf{Prince} \,[]\, \Gamma_3 \,[]\, \{\mathsf{Player}\}\\
&\mathsf{P_{pl}}, \Gamma_3 \vdash \mathsf{x}.\mathsf{sword} : \ \mathsf{Weapon} \,[]\, \Gamma_3 \,[]\, \{\,\}.
\end{aligned} \tag{8}$$

Finally, for Example (7) and $\Gamma_4(\mathsf{x}_1) = \Gamma_5(\mathsf{x}_1) = \mathsf{Player}$, $\Gamma_4(\mathsf{x}_2) = \mathsf{Frog}$, $\Gamma_5(\mathsf{x}_2) = \mathsf{Player}$:

$$\begin{aligned}
&\mathsf{P_{pl}}, \Gamma_4 \vdash \mathsf{x}_2.\mathsf{pouch} : \ \mathsf{Vocal} \,[]\, \Gamma_4 \,[]\, \{\,\}\\
&\mathsf{P_{pl}}, \Gamma_4 \vdash \mathsf{x}_1.\mathsf{kissed}() : \ \mathsf{Weapon} \,[]\, \Gamma_5 \,[]\, \{\mathsf{Player}\}\\
&\mathsf{P_{pl}}, \Gamma_5 \vdash \mathsf{x}_2.\mathsf{brave} : \ \mathbf{bool} \,[]\, \Gamma_5 \,[]\, \{\,\}.
\end{aligned}$$

The exact point at which effects modify types is important. In method calls, the evaluation of the arguments may affect the receiver. In Example (9), assuming that method croak takes a Weapon parameter and is only defined in class Frog,

$$\begin{aligned}
&\mathsf{Object\ played(Player\ x_1, Frog\ x_2)\{Player\}}\\
&\quad \{\ \mathsf{x_2.croak(x_1.kissed())} \qquad /* \ \text{type incorrect} \ */\}
\end{aligned} \tag{9}$$

the evaluation of x_1.kissed() may reclassify x_2, so that it is no longer a Frog. Therefore, the effect of the arguments must be taken into account when looking up the method in order to check method calls. In Example (9), type checking requires looking up croak in class Player and results in a type error.

5.3 Typing Rules

The typing rules are given in Figure 6. We use the look-up functions $\mathcal{F}(\mathsf{P}, \mathsf{c}, \mathsf{f})$ and $\mathcal{M}(\mathsf{P}, \mathsf{c}, \mathsf{m})$, defined in Appendix B, which search for fields and methods through the class hierarchy. We follow the convention that rules can be applied only if the types in the conclusion are defined. This is useful in rules (*cond*) and (*id*).

Consider the rule (*seq*) for composition $\mathsf{e}\,;\mathsf{e}'$. The second expression, e', is typed in the environment Γ_0, that is, the environment updated by typing the first expression, e. The effect of the composition is the union of the effects of the components.

$$\frac{\begin{array}{l}\mathsf{P},\Gamma \vdash \mathsf{e} : \ \mathsf{bool} \ [] \ \Gamma_0 \ [] \ \phi \\ \mathsf{P},\Gamma_0 \vdash \mathsf{e}_1 : \ \mathsf{t}_1 \ [] \ \Gamma_1 \ [] \ \phi_1 \\ \mathsf{P},\Gamma_0 \vdash \mathsf{e}_2 : \ \mathsf{t}_2 \ [] \ \Gamma_2 \ [] \ \phi_2\end{array}}{\mathsf{P},\Gamma \vdash \textbf{if } \mathsf{e} \ \textbf{ then } \ \mathsf{e}_1 \ \textbf{ else } \mathsf{e}_2 : \ \mathsf{t}_1 \sqcup_{\mathsf{P}} \mathsf{t}_2 \ [] \ \Gamma_1 \ \sqcup_{\mathsf{P}} \ \Gamma_2 \ [] \ \phi \cup \phi_1 \cup \phi_2} \ (\textit{cond})$$

$$\frac{\begin{array}{l}\mathsf{P},\Gamma \vdash \mathsf{e} : \ \mathsf{c} \ [] \ \Gamma_0 \ [] \ \phi \\ \mathsf{P},\Gamma_0 \vdash \mathsf{e}' : \ \mathsf{t} \ [] \ \Gamma' \ [] \ \phi' \\ \mathcal{F}(\mathsf{P}, \phi'@_{\mathsf{P}}\mathsf{c}, \mathsf{f}) = \mathsf{t}' \\ \mathsf{P} \vdash \mathsf{t} \leq \mathsf{t}'\end{array}}{\mathsf{P},\Gamma \vdash \mathsf{e.f} := \mathsf{e}' : \ \mathsf{t} \ [] \ \Gamma' \ [] \ \phi \cup \phi'} \ (\textit{a-field}) \qquad \frac{\begin{array}{l}\mathsf{P},\Gamma \vdash \mathsf{e} : \ \mathsf{t}' \ [] \ \Gamma' \ [] \ \phi \\ \Gamma'(\mathsf{x}) = \mathsf{t} \\ \mathsf{P} \vdash \mathsf{t}' \leq \mathsf{t}\end{array}}{\mathsf{P},\Gamma \vdash \mathsf{x} := \mathsf{e} : \ \mathsf{t}' \ [] \ \Gamma' \ [] \ \phi} \ (\textit{a-var})$$

$$\frac{\begin{array}{l}\mathsf{P},\Gamma \vdash \mathsf{e} : \ \mathsf{c} \ [] \ \Gamma' \ [] \ \phi \\ \mathcal{F}(\mathsf{P}, \mathsf{c}, \mathsf{f}) = \mathsf{t}\end{array}}{\mathsf{P},\Gamma \vdash \mathsf{e.f} : \ \mathsf{t} \ [] \ \Gamma' \ [] \ \phi} \ (\textit{field}) \qquad \frac{\begin{array}{l}\mathsf{P},\Gamma \vdash \mathsf{e} : \ \mathsf{t} \ [] \ \Gamma_0 \ [] \ \phi \\ \mathsf{P},\Gamma_0 \vdash \mathsf{e}' : \ \mathsf{t}' \ [] \ \Gamma' \ [] \ \phi'\end{array}}{\mathsf{P},\Gamma \vdash \mathsf{e}; \ \mathsf{e}' : \ \mathsf{t}' \ [] \ \Gamma' \ [] \ \phi \cup \phi'} \ (\textit{seq})$$

$$\frac{}{\begin{array}{l}\mathsf{P},\Gamma \vdash \texttt{true} \ : \ \mathsf{bool} \ [] \ \Gamma \ [] \ \{\} \\ \mathsf{P},\Gamma \vdash \texttt{false} \ : \ \mathsf{bool} \ [] \ \Gamma \ [] \ \{\}\end{array}} \ (\textit{bool}) \qquad \frac{\mathsf{P} \vdash \mathsf{c} \ \diamond_{ct}}{\mathsf{P},\Gamma \vdash \texttt{null} : \ \mathsf{c} \ [] \ \Gamma \ [] \ \{\}} \ (\textit{null})$$

$$\frac{}{\mathsf{P},\Gamma \vdash \mathsf{id} : \ \Gamma(\mathsf{id}) \ [] \ \Gamma \ [] \ \{\}} \ (\textit{id}) \qquad \frac{\mathsf{P} \vdash \mathsf{c} \ \diamond_{ct}}{\mathsf{P},\Gamma \vdash \textbf{new } \mathsf{c} : \ \mathsf{c} \ [] \ \Gamma \ [] \ \{\}} \ (\textit{new})$$

$$\frac{\begin{array}{l}\mathsf{P},\Gamma \vdash \mathsf{e}_0 : \ \mathsf{c} \ [] \ \Gamma_0 \ [] \ \phi_0 \\ \mathsf{P},\Gamma_{i-1} \vdash \mathsf{e}_i : \ \mathsf{t}'_i \ [] \ \Gamma_i \ [] \ \phi_i \ \ (\forall i \in \{1,\ldots,n\}) \\ \mathcal{M}(\mathsf{P}, (\phi_1 \cup \cdots \cup \phi_n)@_{\mathsf{P}}\mathsf{c}, \mathsf{m}) \ = \mathsf{t} \ \mathsf{m}(\mathsf{t}_1 \ \mathsf{x}_1, \ldots, \mathsf{t}_n \ \mathsf{x}_n) \ \phi \ \{\cdots\} \\ \mathsf{P} \vdash (\phi_{i+1} \cup \cdots \cup \phi_n)@_{\mathsf{P}}\mathsf{t}'_i \leq \mathsf{t}_i \ \ (\forall i \in \{1,\ldots,n\})\end{array}}{\mathsf{P},\Gamma \vdash \mathsf{e}_0.\mathsf{m}(\mathsf{e}_1,\ldots,\mathsf{e}_n) : \ \mathsf{t} \ [] \ \phi@_{\mathsf{P}}\Gamma_n \ [] \ \phi \cup \phi_0 \cup \cdots \cup \phi_n} \ (\textit{meth})$$

$$\frac{\begin{array}{l}\mathsf{P} \vdash \mathsf{c} \ \diamond_{rt} \\ \mathcal{R}(\mathsf{P}, \mathsf{c}) = \mathcal{R}(\mathsf{P}, \Gamma(\mathsf{id}))\end{array}}{\mathsf{P},\Gamma \vdash \mathsf{id}\Downarrow\mathsf{c} : \ \mathsf{c} \ [] \ (\{\mathcal{R}(\mathsf{P}, \mathsf{c})\}@_{\mathsf{P}}\Gamma)[\mathsf{id} \mapsto \mathsf{c}] \ [] \ \{\mathcal{R}(\mathsf{P}, \mathsf{c})\}} \ (\textit{recl})$$

Fig. 6. Typing rules for expressions.

Consider now the rule (*cond*) for conditionals. With $\mathsf{t} \sqcup_{\mathsf{P}} \mathsf{t}'$, we denote the *least upper bound of* t *and* t′ *in* P with respect to $\leq$, when it exists.[8] See Figure 11 of Appendix B for a formal definition. With $\Gamma \sqcup_{\mathsf{P}} \Gamma'$, we denote the extension of the above operation to environments, defined as follows:

$$\Gamma \sqcup_{\mathsf{P}} \Gamma' \ = \ \{\mathsf{id} : (\mathsf{t} \sqcup_{\mathsf{P}} \mathsf{t}') \mid \Gamma(\mathsf{id}) = \mathsf{t} \text{ and } \Gamma'(\mathsf{id}) = \mathsf{t}'\}.$$

Least upper bounds are used in rule (*cond*) to determine a conservative approximation of the type of the conditional expression. The two branches may cause

[8]Note that for any class c the least upper bound $\mathsf{c} \sqcup_{\mathsf{P}}$ **bool** does not exist.

different reclassifications for `this` and the parameters. So, after the evaluation we can only assert that `this` and the parameters belong to the least upper bound of their relative classes in Γ_1 and Γ_2. Proposition 1(2), which appears at the end of the section, assures that for this rule $\Gamma_1 \sqcup_P \Gamma_2$ is defined. On the other hand, the least upper bound of the types of the branches, $t_1 \sqcup_P t_2$, may not be defined, in which case the rule cannot be applied.

Consider now the typing of assignments, that is, rules (*a-field*) and (*a-var*). Evaluation of the right-hand side may modify the type of the left-hand side. In particular, in (*a-var*) evaluation of e can modify the type of x. This is taken into account by looking up x in the environment Γ'. Also, in rule (*a-field*) evaluation of e' may modify the class of the object e. For this purpose, we define the application of effects to types:

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

For example, $\{\mathsf{Player}\}@_{P_{pl}}\mathsf{Frog} = \mathsf{Player}$, and $\{\mathsf{Player}\}@_{P_{pl}}\mathsf{Princess} = \mathsf{Princess}$. By applying ϕ' to c before looking up f, we provide for the case where evaluation of e' might reclassify e and remove f in the process. Note that the field type cannot be changed since it cannot be a state class.

Consider now (*recl*): $id \Downarrow c$ is type correct if c, the target of the reclassification, is a state or root class, and if c and the class of id before the reclassification (the class $\Gamma(id)$) are subclasses of the same root class. A reclassification updates the environment by changing the class of the identifier id. Moreover, since there could be aliasing with identifiers of state classes that are subclasses of the root class of id, the static type of all such variables is set to the root class. For this reason, we define the application of effects to environments:

$$\phi@_P\Gamma \;=\; \{id : \phi@_P t \mid \Gamma(id) = t\}.$$

For example, for an environment Γ_1, with $\Gamma_1(x_1) = \Gamma_1(x_2) = \mathsf{Frog}$, $\Gamma_1(x_3) = \mathsf{Prince}$, we have $\{\mathsf{Player}\}@_{P_{pl}}\Gamma_1 = \Gamma_2$, where $\Gamma_2(x_1) = \Gamma_2(x_2) = \Gamma_2(x_3) = \mathsf{Player}$. Therefore, the following typing judgment can be derived:

$$P_{pl}, \Gamma_1 \vdash x_2 \Downarrow \mathsf{Prince} : \; \mathsf{Prince} \,[]\, \Gamma_3 \,[]\, \{\mathsf{Player}\}, \tag{10}$$

where $\Gamma_3(x_1) = \Gamma_3(x_3) = \mathsf{Player}$, but $\Gamma_3(x_2) = \mathsf{Prince}$.

Consider rule (*meth*) for method calls, $e_0.m(e_1, \ldots, e_n)$. The evaluation of the arguments $e_{i+1}, \ldots, e_n$ may modify the types of the arguments $e_1, \ldots, e_i$ and of the object e_0, as shown in Example (9). This could happen if a superclass of the original type of e_j ($1 \le j \le i$) is among the effects of $e_{i+1}, \ldots, e_n$. (Existence of such a class implies uniqueness, since effects are sets of root classes.) The definition of m has to be found in the new class of the object e_0, and the types of the formal parameters must be compared with the new types of $e_1, \ldots, e_{n-1}$. In (*meth*), we look up the definition of m in the class obtained by applying the effect of the arguments to the class of the receiver (c.f. Example (9)) and we compare the types of formal and actual parameters by keeping into account the effects of the actual parameters.

$$\frac{\begin{array}{l}\mathcal{C}(\mathsf{P},\mathsf{c}) = [\mathbf{root} \mid \mathbf{state}]\ \mathbf{class}\ \mathsf{c}\ \mathbf{extends}\ \ \mathsf{c}'\ \{\cdots\}\\ \forall \mathsf{f}:\ \mathcal{FD}(\mathsf{P},\mathsf{c},\mathsf{f}) = \mathsf{t}_0 \Rightarrow \mathsf{P} \vdash \mathsf{t}_0\ \diamond_{ft} \quad \text{and} \quad \mathcal{F}(\mathsf{P},\mathsf{c}',\mathsf{f}) = \mathcal{U}df\\ \forall \mathsf{m}:\ \mathcal{MD}(\mathsf{P},\mathsf{c},\mathsf{m}) = \mathsf{t}\ \mathsf{m}(\mathsf{t}_1\ \mathsf{x}_1,\ldots,\mathsf{t}_n\ \mathsf{x}_n)\ \phi\ \{\mathsf{e}\} \Rightarrow\\ \quad \mathsf{P} \vdash \phi\ \diamond\\ \quad \mathsf{P}, \{\mathsf{x}_1 : \mathsf{t}_1,\ldots,\mathsf{x}_n : \mathsf{t}_n, \mathtt{this} : \mathsf{c}\} \vdash \mathsf{e} :\ \mathsf{t}'\ []\ \Gamma'\ []\ \phi'\\ \quad \mathsf{P} \vdash \mathsf{t}' \leq \mathsf{t}\\ \quad \phi' \subseteq \phi\\ \quad \mathcal{M}(\mathsf{P},\mathsf{c}',\mathsf{m}) = \mathcal{U}df \text{ or } (\mathcal{M}(\mathsf{P},\mathsf{c}',\mathsf{m}) = \mathsf{t}\ \mathsf{m}(\mathsf{t}_1\ \mathsf{x}_1,\ldots,\mathsf{t}_n\ \mathsf{x}_n)\ \phi''\ \{\cdots\} \quad \text{and} \quad \phi \subseteq \phi'')\end{array}}{\mathsf{P} \vdash \mathsf{c}\ \diamond} \ (wfc)$$

$$\frac{\vdash\ \mathsf{P}\ \diamond_h \qquad \forall \mathsf{c}:\ \ \mathcal{C}(\mathsf{P},\mathsf{c}) \neq \mathcal{U}df\ \Rightarrow\ \mathsf{P} \vdash \mathsf{c}\ \diamond}{\vdash \mathsf{P}\ \diamond} \ (wfp)$$

Fig. 7. Rules for well-formed classes and programs.

Overall, in our typing rules, we applied the effects to types and environments only when this was necessary to guarantee soundness, since we wanted to preserve as much typing information as possible.

In Proposition 1, we state that if $\mathsf{P}, \Gamma \vdash \mathsf{e} :\ \mathsf{t}\ []\ \Gamma'\ []\ \phi$ can be derived, then the environments Γ and Γ' are defined for the same set of identifiers, and any differences in Γ and Γ' are due to the effect ϕ:

PROPOSITION 1. *Let* $\mathsf{P}, \Gamma \vdash \mathsf{e} :\ \mathsf{t}\ []\ \Gamma'\ []\ \phi$. *Then, for all* id :

(1) $\Gamma(\mathsf{id}) \neq \mathcal{U}df$ *if and only if* $\Gamma'(\mathsf{id}) \neq \mathcal{U}df$,

(2) $\Gamma(\mathsf{id}) \neq \mathcal{U}df$ *implies* $\phi@_{\mathsf{P}}\Gamma(\mathsf{id}) = \phi@_{\mathsf{P}}\Gamma'(\mathsf{id})$.

For example, in the typing judgment (8), for $\Gamma = \Gamma_2$ and $\Gamma' = \Gamma_3$ we have $\Gamma_2(\mathsf{x}) = \mathsf{Frog}$, $\Gamma_3(\mathsf{x}) = \mathsf{Prince}$, $\phi = \{\mathsf{Player}\}$ and $\phi@_{\mathsf{P}}\Gamma_2(\mathsf{x}) = \phi@_{\mathsf{P}}\Gamma_3(\mathsf{x}) = \mathsf{Player}$.

Looking at rule (*cond*), Proposition 1(2) ensures that if $\Gamma(\mathsf{id})$ is defined, then $\Gamma_1(\mathsf{id}) \sqcup_{\mathsf{P}} \Gamma_2(\mathsf{id})$ is defined as well, for all id.

In general, $\mathsf{P}, \Gamma \vdash \mathsf{e} :\ \mathsf{t}\ []\ \Gamma'\ []\ \phi$ does *not* imply $\Gamma' = \phi@_{\mathsf{P}}\Gamma$. A counterexample is the typing judgment (10).

5.4 Well-Formed Programs

A program is well formed (i.e., $\vdash \mathsf{P}\ \diamond$) if the inheritance hierarchy is well formed (i.e., $\vdash\ \mathsf{P}\ \diamond_h$) and all its classes are well formed (i.e., $\mathsf{P} \vdash \mathsf{c}\ \diamond$). Fields may not redefine fields from superclasses, and methods may redefine superclass methods only if they have the same name, arguments, and result type, and their effect is a subset of that of the overridden method.[9] Method bodies must be well formed, must return a value appropriate for the method signature, and their effect must be a subset of that in the signature. See Figure 7, where

[9]Thus, in contrast to Java and C++, $\mathcal{F}ickle_{\mathrm{II}}$ does not allow field shadowing, nor method overloading. These features can be included into $\mathcal{F}ickle_{\mathrm{II}}$ adopting the approach from Drossopoulou et al. [1999b]. However, this would complicate the presentation unnecessarily.

$\mathcal{C}(\mathsf{P}, \mathsf{c})$ returns the definition of class c in program P, and the look-up functions $\mathcal{FD}(\mathsf{P}, \mathsf{c}, \mathsf{f})$, $\mathcal{MD}(\mathsf{P}, \mathsf{c}, \mathsf{m})$, defined in Appendix B, search for fields and methods only in class c.

5.5 Soundness

The judgment $\mathsf{P}, \sigma \vdash \mathsf{v} \lhd \mathsf{t}$, guarantees that value v conforms to type t. In particular, it requires that an address ι points to an object of class c, a subclass of t, that the object contains all fields required in the description of c, and that the fields contain values that conform to their type in c. The judgment $\mathsf{P}, \Gamma \vdash \sigma \diamond$ guarantees that all object fields contain values that conform to their types in the class of the objects, and that all parameters and the receiver are mapped to values which conform to their types in Γ. Formal definitions can be found in Figure 12 of Appendix B.

The type system is sound in the sense that a converging well-typed expression returns a value that agrees with the expression's type, or `nullPntrExc`; but is *never* stuck.

THEOREM 1 (TYPE SOUNDNESS). *For a well-formed program* P*, environment* Γ*, and expression* e*, such that*

$$\mathsf{P}, \Gamma \vdash \mathsf{e} : \mathsf{t} \,[]\, \Gamma' \,[]\, \phi$$

if $\mathsf{P}, \Gamma \vdash \sigma \diamond$*, and* e, σ *converges, then*

—$\mathsf{e}, \sigma \gamma_{\mathsf{p}}^{*} \mathsf{v}, \sigma'$, $\mathsf{P}, \sigma' \vdash \mathsf{v} \lhd \mathsf{t}$, $\mathsf{P}, \Gamma' \vdash \sigma' \diamond$,
or

—$\mathsf{e}, \sigma \gamma_{\mathsf{p}}^{*} \texttt{nullPntrExc}, \sigma'$.

Remark 2. As far as divergent expressions go, the theorem does not say anything. However, the operational semantics forces convergence for standard typing errors or access to members undefined for an object, see Figure 4. Therefore, Theorem 1 suffices to ensure that execution of well-typed expression never accesses nonexisting identifiers, fields or methods, and is never stuck.

6. DESIGN ALTERNATIVES

Our aim was to develop language features supporting reclassification of objects in an imperative setting, which also allows aliasing. The operational semantics was straightforward, the design of a strong type system less so. The main challenges were:

(*c1*) The type of id inside method bodies containing reclassifications—c.f. Examples (5)–(6), Section 2.

(*c2*) Reclassification of an aliased object may remove members, which the object may need in another context—c.f. Example (4), Section 2.

(*c3*) The possibility that an object uses members removed by a reclassification earlier in the call stack—c.f. Example (9), Section 5.2.

We have considered, and experimented with, several ideas:

For (*c1*). The type of `this` and the parameters may change after a reclassification; we express this through the second component of our typing scheme.

For (*c2*). We considered several solutions, and chose (f):

(a) Check the existence of members at run-time, as in Serrano [1999]. This is not type safe.

(b) An object should have all members for all possible state subclasses of its root superclass, as in Ernst et al. [1998]. Although type safe, this does not allow compact representations as required in Serrano [1999], and does not express our intention of exclusive cases.

(c) Require all state subclasses of a root class to have exactly the same members, and differ in the method bodies only. However, this requirement is too strong, for example, does not hold for empty and nonempty lists.

(d) In $\mathcal{F}ickle$–99, we allow state class fields, but avoid the aliasing introduced through line 2. in Example (3) of Section 2, through the type system. Types are either nonstate, nonroot classes, or sets of state classes. Accessing a member of an expression is only legal if all state classes of the type of the expression define this member. A set of state classes is a subtype of another set, only if they are identical.

(e) In $\mathcal{F}ickle$, we forbid the use of state classes as types, except for the type of `this`. Thus state classes may have different members, but all state subclasses of the same root class offer the same interface to all their clients.

(f) In $\mathcal{F}ickle_{\mathrm{II}}$ only fields cannot have state classes as types.

For (*c3*).

(a)–(c) With any of the approaches described in (*c2*)(a), (*c2*)(b), or (*c2*)(c), the problem would not arise; but we have rejected these solutions in (*c2*).

(d) In $\mathcal{F}ickle$–99, we "lock" an object of a state class when it starts executing a method, and "unlock" when it finishes. Attempting to reclassify a locked object throws an exception; for example, Example (9) could throw such an exception. This is too restrictive, and has the drawback that it allows run-time errors.

(e)–(f) In $\mathcal{F}ickle_{\mathrm{II}}$, the type system ensures against the problem; the effects from any called methods are applied to the type of id; therefore, after a call that may modify an object referred by id, the type of id will be the root superclass, and so, access to state class members will be type incorrect. Similarly in $\mathcal{F}ickle$, but only when id is the receiver `this`.

7. IMPLEMENTATION

A prototype implementation of $\mathcal{F}ickle$ through a Java translator has already been developed [Jarman 2000; Anderson 2001; Ancona et al. 2001]. Type correct $\mathcal{F}ickle$ programs are mapped into equivalent Java programs, where root classes are represented by wrapper classes, containing a field value, which points to

an object of one of its state subclasses. Method calls are forwarded from the wrapper object to its value field, and reclassifications are implemented by overwriting the value field. We are currently working on a translator for $\mathcal{F}ickle_{II}$ [Ancona et al. 2002], allowing for parameters of state class.

In a production compiler, one can avoid the wrapper object and the indirection in the method dispatch, provided that the maximal size of state subclasses of any given root is known:[10] Notably, the constraint that the target and source of reclassification have a common root superclass allows the standard, efficient implementation of method call, where we look up through an offset into the method dispatch table of the receiver. The fact that sources and targets of reclassifications have the same maximal size allows us to implement reclassification through simple in-place overwriting of the source object.

Alternatively, the use of object tables [Goldberg and Robson 1983], where object references are represented through pointers to a table of pointers to objects, would allow a more direct implementation of object reclassification, and would dispense with the restriction of explicit double indirection. Such an approach is taken in some Java implementations and in the implementation of GILGUL [Costanza 2001].

We are currently working on an implementation of $\mathcal{F}ickle_{II}$ using an extended JVM [Shuttlewood 2002]. We extend the JVM by one additional instruction, which represents reclassification. We use object tables, but we also plan to use object fragmentation, avoiding the indirection of object tables in the general case, and using indirection only when required by the size of the reclassified objects.

8. RELATED WORK

Most foundational work on the semantics of object-oriented programming languages is based on functional object-based languages.

In Abadi and Cardelli [1996], method overriding models field update and delegation. In Fisher et al. [1994], method extension represents class inheritance, while Bono et al. [1999], Di Gianantonio et al. [1998], Rémy [1995], Fisher and Mitchell [1995], and Riecke and Stone [1998] enhance the above representation by introducing a limited form of method subtyping. These calculi deal with questions of width-subtyping over deep-subtyping, the use of MyType, method extension and overriding; they were primarily developed as a means of understanding inheritance and delegation.

Object extension in these calculi can be seen as the reclassification of an object of class c to an object of a subclass of c. Unrestricted subtyping followed by object expansion might cause messageNotUnderstood errors, and so type soundness is recovered by imposing certain restrictions on the use of subtyping [Fisher and Mitchell 1995; Rémy 1995; Di Gianantonio et al. 1998; Ghelli and Palmerini 1999; Bono et al. 1999] with the consequence that an object cannot be promoted to a superclass and then to the original subclass.

[10]Restrictions on possible subclasses can be found in several systems, for example, in [Chambers and Leavens 1995].

For databases, Bertino and Guerrini [1995] suggest multiple most specific classes, thus in a way allowing multiple inheritance, while Ghelli and Palmerini [1999] allow objects to accumulate different roles in a functional setting. They model nonexclusive roles (for example, female and professor), whereas we model objects changing mutually exclusive classes (for example, opened window versus iconified window).

Refinement types in functional languages distinguish cases through subtypes, see Freeman and Pfenning [1991]. The main questions in Freeman and Pfenning [1991] are type inference, and establishing that functions are well defined, in the sense that they cover all possible cases. Side-effects are not considered; therefore, problems like aliasing that are central to our development do not arise.

Predicate classes [Chambers 1993] support a form of dynamic classification of objects based on their run-time value: Code is broken down on a per-function basis, while $\mathcal{F}ickle_{\mathrm{II}}$ follows the mainstream, where code is broken down on a per-class basis. Also, in Chambers [1993], the term *reclassification* denotes changes in attribute values that imply changes in predicates when calculated next. Thus, reclassification in Chambers [1993] is implicit and lazy, whereas in $\mathcal{F}ickle_{\mathrm{II}}$ reclassification is explicit and eager. Predicate dispatching [Ernst et al. 1998] suggests multimethod dispatch depending on predicates on the receiver and argument. Different methods may dispatch depending on different predicates, for example, insert may depend on whether the list is a priority or a last-in–first-out list, whereas print may depend on whether the list is empty or not. This is not possible in $\mathcal{F}ickle_{\mathrm{II}}$, unless extended with multiple inheritance. Finally, Chambers [1993] and Ernst et al. [1998] raise the question of disjointness and completeness of predicates (unambiguous and complete).

Similarly, for single method dispatch, in Tailvasaari [1993] classes have "modes" representing different states, for example, opened vs. iconified window. Wide classes from 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 $\mathcal{F}ickle_{\mathrm{II}}$, by dropping the requirement for a strong type system, and requiring run-time tests for the presence of fields. (The primary aim of wide classes was better memory use in the presence of changes of object structures.)

The language GILGUL [Costanza 2001] is an extension of Java that allows for dynamical object replacement. GILGUL supports implementation-only classes, that is, 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 nonimplementation-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.

For concurrent objects, Ravara and Vasconcelos [2000] give behavioral types that guarantee that every message has a chance of being received if it requires a method that may be enabled at some point in the future.

Our work is also related to Strom and Yellin [1993] and DeLine and Fähndrich [2001], who use the type system to track state changes. In Strom

and Yellin [1993], the language NIL attaches states to objects along with their types and does not allow aliasing of objects. Building on the Capability Calculus [Walker et al. 2001], the language Vault [DeLine and Fähndrich 2001] tracks states of objects in the presence of aliasing.

We now consider related features in some of the more widely used programming languages: Modula-3 [Cardelli et al. 1989] offers a limited, "one-off" possibility of reclassification, since the method suite of an object can be determined at object creation time. Self [Agesen et al. 1992] allows dynamic inheritance among prototypes in order to change state dynamically. However, Self is only dynamically typed [Agesen et al. 1995]. In BETA [Kristensen et al. 1987], nested patterns can be used to model dynamic state changes, at the price of dynamic type checking. The "become:" primitive of Smalltalk [Goldberg and Robson 1983] has also been used to get the effect of dynamic reclassification. Again, Smalltalk is dynamically typed.

$\mathcal{F}ickle_{II}$ succeeds two earlier proposals: $\mathcal{F}ickle$–99 [Drossopoulou et al. 1999a] and $\mathcal{F}ickle$ [Drossopoulou et al. 2001], which addressed the same requirements. $\mathcal{F}ickle$ improves $\mathcal{F}ickle$–99 in at least two respects: First, $\mathcal{F}ickle$ allows all reclassifications of state class objects, whereas $\mathcal{F}ickle$–99 prevents reclassification of objects while they are executing a method; it achieves this either through run-time locks or through an effect system. Second, $\mathcal{F}ickle$ has one type hierarchy both for reclassifiable and nonreclassifiable objects, whereas $\mathcal{F}ickle$–99 distinguishes two kinds of types. $\mathcal{F}ickle_{II}$ further improves $\mathcal{F}ickle$: it allows parameter and result types to be state classes, and has a more elegant typing system.

9. CONCLUSIONS

$\mathcal{F}ickle_{II}$ is the outcome of several designs and successive improvements. We are now satisfied that the suggested approach is useful and usable. In the process, we also developed an interesting typing scheme, where typing an expression affects the environment in which the following or enclosing expressions are typed.

Reclassifiable objects support the differentiation of the behavior of an object according to its class, thus keeping more in the object-oriented style, and the expression of special cases through subclasses, thus also more in style with pattern matching as in the functional programming paradigm.

Reclassifiable objects support carrying over practices from functional programming into imperative, object oriented languages. Less use of the special value null is made in programs adopting reclassification. (See the example in Appendix A.)

Having state classes as types of parameters considerably extends the range of type correct programs. Type casts, which become necessary when accessing fields in state classes, can be avoided through the introduction of further methods. (Again, see Appendix A.) Thus, the type system encourages breaking down code into several methods. We believe that the restrictions imposed from the type system do not seriously constrain the applicability, and have shown this through larger cases studies in Drossopoulou [2002]. When we developed the case studies we were surprised at the possibilities opened up by reclassifiable

objects. The extent of their use, and the appropriate idioms, can only be determined after we have had more experience in using $\mathcal{F}ickle_{\mathrm{II}}$.

We discuss most of the above points in some detail in Appendix A, and in more detail in Drossopoulou [2002], where we explore the use of object reclassification through the study of three examples, describing normal and privileged accounts, empty or nonempty lists, and adventure games. That study differs from our earlier one [Jarman and Drossopoulou 2000], which follows $\mathcal{F}ickle$, as it makes ample use of the extra expressive power offered in $\mathcal{F}ickle_{\mathrm{II}}$.

Most weaknesses of the current design are due to the fact that the effects system is too crude:

—Reclassification of any object of a certain class affects the types of all variables of subclasses of the object's root class.

—Writing all the effects of a method may become cumbersome. This, however could be addressed through some naming conventions, or through tools.

—A more important weakness is the fact that methods are required to list the effects of execution of their bodies, and thus methods higher up in the calling chain accumulate the effects of all transitively called methods, c.f. Example 1.1 in Drossopoulou [2002]. Therefore, the effects of a method may end up mentioning classes that are not visible, and irrelevant to the class where the method appears. Although a similar situation happens with the listing of all exceptions in the throws clause of method headers, exceptions may be eliminated from the throws clause when caught in the method body. Unfortunately, no such opportunity is available in $\mathcal{F}ickle_{\mathrm{II}}$.

—Also, the restriction on fields not to be of a state class forbids, for example, the expression of lists of frogs even in the context where we know that none of these will be kissed.

—The current system does not work for threads, given that the effect calculated for an invocation that spawns a child thread can in reality happen later, after the moment the types of identifiers have changed in the main thread.

Further work includes the incorporation of $\mathcal{F}ickle_{\mathrm{II}}$ into a full language, the refinement of the effect system (e.g., through data-flow analysis techniques), the incorporation of myType, multiple inheritance, the distinction of subclassing from subtyping, the modelling of irreversible reclassifications (e.g., pupa to butterfly). Moreover, we are generalizing our type and effect system in two different directions: to deal with a limited form of concurrency, and to allow fields to have state classes, for example, by distinguishing between reclassifiable and nonreclassifiable types and allowing fields of state classes but nonreclassifiable types [Fidgett 2002].

APPENDIXES

A LINKED LISTS

In this appendix, we demonstrate the use of reclassifiable objects through the example of lists of integers: Lists may be empty or nonempty, and support the insertion of removal of integers (see Figure 8). The method **void** insertFront(**int**)

```
class ListException  extends Exception{
}

abstract root class List extends Object{
   abstract void insertFront(int i){List};
   abstract int getFront(){List};
// auxiliary methods
   abstract void copyTo(NonEmptyList x){List};
   // if receiver is NonEmplyList, copies fields of receiver onto x,
   // if receiver is EmptyList, re-classifies x to EmptyList
}

state class EmptyList extends List{
   void insertFront(int i){List}{
      this⇓NonEmptyList; content := i; next := new EmptyList; }
   int getFront() { } {throw new ListException; }
// auxiliary methods
   void copyTo(NonEmptyList x){List}{x⇓EmptyList; }
}

state class NonEmptyList extends List{
   int content;
   List next;

   void insertFront(int i) { } {
      // create second as a copy of the receiver
      NonEmptyList second := new NonEmptyList;
      copyTo(second);
      // modify the receiver
      content := i; next := second; }
   int getFront(){List}{
      int result := content;
      // copy the next element on the receiver
      next.copyTo(this); return result; }
// auxiliary methods
   void copyTo(NonEmptyList x) { } {
      // copy the fields of the receiver onto x
      x.content := content; x.next := next; }
}
```

Fig. 8. Program P_{list}—lists with reclassifications.

inserts an integer at the front of the list, while the method **int** getFront() removes the element from the front of the list and returns it.

The "traditional" approach represents lists through header objects that contain a reference to a link. If the reference is `null`, then the list is empty. If the reference is not `null`, then the list is not empty, and the link pointed at represents the first entry of the list. Each link contains an integer and a reference to the next link. If that reference is `null`, then the link is the last in the list. Thus, the distinction between the cases (empty and nonempty list, and also last and nonlast link) is reflected by a particular field containing the special value `null` or not. The variation in behavior is not reflected in the class hierarchy; instead, it is buried inside the method bodies.

The $\mathcal{F}ickle_{\mathrm{II}}$ approach seeks to express this distinction directly in the class hierarchy. In Drossopoulou [2000], we explore two different implementations of lists: The first represents lists through a Header object (which may be reclassified to EmptyHeader or NonEmptyHeader), and a sequence of Links. The second represents lists without separate header objects. In this article, we discuss the second approach, because it is more interesting in terms of the use of reclassifiable objects.

Our representation reflects the distinction between empty and nonempty lists through the state subclasses EmptyList and NonEmptyList of the root class List. A NonEmptyList contains an integer, and a reference to the next list. An EmptyList contains no fields.

Thus, our list representation is akin to the list representation in functional programming with pattern matching, were we would have defined

$$\mathsf{data\ intlist} = \mathsf{empty} \mid \mathsf{nonempty}(\mathsf{int},\ \mathsf{intlist})$$

The functional programming representation, like ours, allows a List object to signify the beginning or any "intermediate" element in the list.

In that sense, reclassification allows us to maintain practices from functional programming in imperative, object-oriented languages. This realization came to us as a surprise, when we were writing the case studies. The similarities between the functional programming representation and the $\mathcal{F}ickle_{\mathrm{II}}$ representation end in that $\mathcal{F}ickle_{\mathrm{II}}$ was designed in the context of languages that allow aliasing. For example, if list1 and list2 point to the same empty list, then the expression list1.insertFront(7) affects both list1 and list2. For this reason, it is necessary to reclassify the object pointed at by list1, while in functional programming it would be sufficient to create a new value with the type constructor nonempty.

On the other hand, the traditional approach makes heavy use of the special value `null`: it represents an empty list, and also the last link of a sequence. Here we do not make use of `null` at all, as we are using state classes instead.[11] Furthermore, the code from above does not expect to encounter the value `null`.

[11]Of course, it would have been possible to avoid `null` in the traditional approach as well. Namely, `null` is a value that belongs to all reference types; this can also be represented by adding a NullClass as a subclass to any class Class. The extra ingredient of $\mathcal{F}ickle_{\mathrm{II}}$ comes from combining that possibility with the reclassification of objects.

The behavior of the method getFront depends on the state class membership of `this.next`. If it is an EmptyList, then `this` is reclassified to EmptyList, whereas if it is a NonEmptyList, then the fields from `this.next` have to be copied onto `this`. We achieve that by a form of double dispatch, that is, by delegating to the method copyTo. This also avoids type casts.[12] The example also demonstrates the importance of allowing state classes as types of parameters. In the method copyTo in class NonEmptyList, the fields content and next of x can be accessed without type casts, whereas in $\mathcal{F}ickle$, where parameters could not have had state classes as types, type casts would be needed, c.f. Section 2.2.1 in Jarman and Drossopoulou [2000].

In general, through the introduction of further methods one can avoid the type casts necessitated by the access of members of state classes. Therefore, the type system encourages breaking down code into several methods.

Programming linked lists as in this section is significantly different than the "traditional" approach, also in that insertions and removals are applied to the links themselves rather than to the links preceding them. For example, removing the last element of the list[13] is more directly expressed with the current approach, as it only requires finding and reclassifying the last component of the list, whereas, in the traditional approach, it is less direct, as it requires finding the link immediately preceding the last link (i.e., the x, such that x.next.next == `null`), and changing its next field.

The choice of linked lists as an example to demonstrate $\mathcal{F}ickle_{II}$ is, in some ways, debatable: Such basic data structures have been programmed in certain ways for many years now, and one might expect to find them expressed in the old, familiar patterns. Also, such structures are usually found in libraries, and modern programmers will typically not need to write linked list implementations. More importantly, in linked lists, there is a single reference to each of the links, and so the full power of reclassification, which is transparent to aliasing, does not come to light in the case of lists.

Nevertheless, we believe that this example is illuminating, because it shows how familiar data structures can be programmed in significantly different ways than usual.

---

[12]If instead, we chose to achieve the different behavior directly within the function getFront, then we would have needed type casts. The corresponding code is:

```
int getFront(){List}{
  int result := content;
  if (this.next.isEmpty()){
    this⇓EmptyList; }
   else {
    this.content := ((NonEmptyList)this.next).content;
    this.next := ((NonEmptyList)this.next).next; }
  return result; }
```

[13]This is shown in Section 2.2.2 of Jarman and Drossopoulou [2000].

$$\forall \mathsf{c}: \quad \mathsf{P} = \mathsf{P}_1\ [\mathbf{root} \mid \mathbf{state}]\ \mathbf{class}\ \mathsf{c}\ \mathbf{extends}\ \mathsf{c}'\{\cdots\}\ \mathsf{P}_2,\ \mathsf{P} = \mathsf{P}_3\ [\mathbf{root} \mid \mathbf{state}]\ \mathbf{class}\ \mathsf{c}\ \mathbf{extends}\ \mathsf{c}''\{\cdots\}\ \mathsf{P}_4 \Rightarrow \mathsf{P}_1 = \mathsf{P}_3,\ \mathsf{P}_2 = \mathsf{P}_4$$

$$\forall \mathsf{f}: \quad \mathsf{P} = \mathsf{P}_1\ [\mathbf{root} \mid \mathbf{state}]\ \mathbf{class}\ \mathsf{c}\ \mathbf{extends}\ \mathsf{c}'\{\mathit{defs}_1\, \mathsf{t}\ \mathsf{f}\ \mathit{defs}_2\}\ \mathsf{P}_2,\ \mathsf{P} = \mathsf{P}_1\ [\mathbf{root} \mid \mathbf{state}]\ \mathbf{class}\ \mathsf{c}\ \mathbf{extends}\ \mathsf{c}'\{\mathit{defs}_3\ \mathsf{t}'\ \mathsf{f}\ \mathit{defs}_4\}\ \mathsf{P}_2 \Rightarrow \mathit{defs}_1 = \mathit{defs}_3,\ \mathit{defs}_2 = \mathit{defs}_4;$$

$$\forall \mathsf{m}: \quad \mathsf{P} = \mathsf{P}_1\ [\mathbf{root} \mid \mathbf{state}]\ \mathbf{class}\ \mathsf{c}\ \mathbf{extends}\ \mathsf{c}'\{\mathit{defs}_1\ \mathsf{t}\ \mathsf{m}(\mathsf{t}_1\mathsf{x}_1, \ldots \mathsf{t}_\mathsf{q}\ \mathsf{x}_\mathsf{q})\phi\ \{\mathsf{e}\}\ \mathit{defs}_2\}\ \mathsf{P}_2,\ \mathsf{P} = \mathsf{P}_1\ [\mathbf{root} \mid \mathbf{state}]\ \mathbf{class}\ \mathsf{c}\ \mathbf{extends}\ \mathsf{c}'\{\mathit{defs}_3\ \mathsf{t}'\ \mathsf{m}(\mathsf{t}'_1\ \mathsf{x}'_1 \cdots \mathsf{t}'_\mathsf{n}\ \mathsf{x}'_\mathsf{n})\phi'\{\mathsf{e}'\}\ \mathit{defs}_4\}\ \mathsf{P}_2 \Rightarrow \mathit{defs}_1 = \mathit{defs}_3,\ \mathit{defs}_2 = \mathit{defs}_4$$

$$\vdash \mathsf{P}\ \diamond_u$$

Fig. 9. Programs with unique definitions.

B DEFINITIONS OF LOOKUP, SUBTYPES, ACYCLIC PROGRAMS, AND AGREEMENTS

Figure 9 defines the judgment $\vdash \mathsf{P}\ \diamond_u$, which guarantees that a program has unique definitions. The first requirement says that there should be no more than one class definition for any identifier c—note that it implicitly guarantees $\mathsf{c}' = \mathsf{c}''$ 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 $\mathsf{t} = \mathsf{t}'$. The third requirement says that there should be a unique method definition in c for any identifier m—note that it implicitly guarantees $\mathsf{t} = \mathsf{t}', \mathsf{t}_1 = \mathsf{t}'_1, \ldots, \mathsf{t}_\mathsf{q} = \mathsf{t}_{\mathsf{q}'}, \mathsf{x}_1 = \mathsf{x}'_1, \ldots, \mathsf{x}_\mathsf{q} = \mathsf{x}_{\mathsf{q}'}, \phi = \phi'$, and $\mathsf{e} = \mathsf{e}'$.

For program P with $\vdash \mathsf{P}\ \diamond_u$, identifier $\mathsf{c} \neq \mathsf{Object}$, and qualifier $\mathit{qual} = \mathbf{root}$, or $\mathit{qual} = \mathbf{state}$, or $\mathit{qual} = \epsilon$, we define the lookup of the class declaration for c:

$$\mathcal{C}(\mathsf{P}, \mathsf{c}) = \begin{cases} \mathit{qual}\ \mathbf{class}\ \mathsf{c}\ \mathbf{extends}\ \mathsf{c}'\{\mathsf{cBody}\} & \mathit{if}\ \ \mathsf{P} = \mathsf{P}'\ \mathit{qual}\ \mathbf{class}\ \mathsf{c}\ \mathbf{extends}\ \mathsf{c}'\{\mathsf{cBody}\}\ \mathsf{P}'', \\ \mathcal{U}\mathit{df} & \mathit{otherwise} \end{cases}$$

The assertion $\mathsf{P} \vdash \mathsf{c} \sqsubseteq \mathsf{c}'$, defined in Figure 10, means that the class c is a subclass of c′. The class hierarchy in a program P is well formed, that is, $\vdash\ \mathsf{P}\ \diamond_h$, if the subclass relationship is acyclic, root classes extend only nonroot and nonstate classes, and state classes extend either root classes or state classes. Notice that $\vdash \mathsf{P}\ \diamond_u$ whenever $\vdash\ \mathsf{P}\ \diamond_h$.

It is straightforward to state and 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 Figure 10, distinguish the kinds of classes: $\mathsf{P} \vdash \mathsf{c}\ \diamond_{ct}$ means that c is any class, $\mathsf{P} \vdash \mathsf{c}\ \diamond_{rt}$ means that c is a reclassifiable type that is, either a root or a state class. The judgment $\mathsf{P} \vdash \mathsf{t}\ \diamond_{ft}$ means that t is a field type, that is, either **bool** or a nonstate class.

Widening, the extension of the subclass relationship to types, is expressed by the assertion $\mathsf{P} \vdash \mathsf{t} \leq \mathsf{t}'$, and is also defined by the rules in Figure 10.

$$\frac{\vdash \mathsf{P} \diamond_u}{\mathsf{P} \vdash \mathsf{Object} \sqsubseteq \mathsf{Object}} \qquad \frac{\vdash \mathsf{P} \diamond_u \quad \mathsf{P} = \cdots[\mathbf{root} \mid \mathbf{state}\ ]\ \mathbf{class}\ \mathsf{c}\ \mathbf{extends}\ \mathsf{c}'\{\ \cdots\ \}\cdots}{\begin{array}{c}\mathsf{P} \vdash \mathsf{c} \sqsubseteq \mathsf{c} \\ \mathsf{P} \vdash \mathsf{c} \sqsubseteq \mathsf{c}'\end{array}} \qquad \frac{\mathsf{P} \vdash \mathsf{c} \sqsubseteq \mathsf{c}' \quad \mathsf{P} \vdash \mathsf{c}' \sqsubseteq \mathsf{c}''}{\mathsf{P} \vdash \mathsf{c} \sqsubseteq \mathsf{c}''}$$

$$\frac{\begin{array}{l}\forall \mathsf{c}, \mathsf{c}' : \\ \quad \mathsf{P} \vdash \mathsf{c} \sqsubseteq \mathsf{c}' \quad \text{and} \quad \mathsf{P} \vdash \mathsf{c}' \sqsubseteq \mathsf{c} \ \Rightarrow\ \mathsf{c} = \mathsf{c}' \\ \quad \mathcal{C}(\mathsf{P}, \mathsf{c}) = \mathbf{class}\ \mathsf{c}\ \mathbf{extends}\ \mathsf{c}'\ \{\cdots\} \ \Rightarrow\ \mathcal{C}(\mathsf{P}, \mathsf{c}') = \mathbf{class}\ \mathsf{c}' \cdots \\ \quad \mathcal{C}(\mathsf{P}, \mathsf{c}) = \mathbf{root\ class}\ \mathsf{c}\ \mathbf{extends}\ \mathsf{c}'\ \{\cdots\} \ \Rightarrow\ \mathcal{C}(\mathsf{P}, \mathsf{c}') = \mathbf{class}\ \mathsf{c}' \cdots \\ \quad \mathcal{C}(\mathsf{P}, \mathsf{c}) = \mathbf{state\ class}\ \mathsf{c}\ \mathbf{extends}\ \mathsf{c}'\ \{\cdots\} \ \Rightarrow \\ \qquad ((\mathcal{C}(\mathsf{P}, \mathsf{c}') = \mathbf{root\ class}\ \mathsf{c}' \cdots) \quad \text{or} \quad (\mathcal{C}(\mathsf{P}, \mathsf{c}') = \mathbf{state\ class}\ \mathsf{c}' \cdots))\end{array}}{\vdash\ \mathsf{P} \diamond_h}$$

$$\frac{\vdash\ \mathsf{P} \diamond_h \quad \mathcal{C}(\mathsf{P}, \mathsf{c}) = \mathbf{class}\ \mathsf{c} \cdots}{\begin{array}{c}\mathsf{P} \vdash \mathsf{c} \diamond_{ft} \\ \mathsf{P} \vdash \mathsf{c} \diamond_{ct}\end{array}} \qquad \frac{\vdash\ \mathsf{P} \diamond_h \quad \mathcal{C}(\mathsf{P}, \mathsf{c}) = \mathbf{root\ class}\ \mathsf{c} \cdots}{\begin{array}{c}\mathsf{P} \vdash \mathsf{c} \diamond_{ft} \\ \mathsf{P} \vdash \mathsf{c} \diamond_{rt} \\ \mathsf{P} \vdash \mathsf{c} \diamond_{ct}\end{array}} \qquad \frac{\vdash\ \mathsf{P} \diamond_h \quad \mathcal{C}(\mathsf{P}, \mathsf{c}) = \mathbf{state\ class}\ \mathsf{c} \cdots}{\begin{array}{c}\mathsf{P} \vdash \mathsf{c} \diamond_{rt} \\ \mathsf{P} \vdash \mathsf{c} \diamond_{ct}\end{array}}$$

$$\frac{}{\mathsf{P} \vdash \mathsf{bool} \diamond_{ft}} \qquad \frac{}{\mathsf{P} \vdash \mathsf{bool} \leq \mathsf{bool}} \qquad \frac{\mathsf{P} \vdash \mathsf{c} \sqsubseteq \mathsf{c}'}{\mathsf{P} \vdash \mathsf{c} \leq \mathsf{c}'}$$

Fig. 10. Subclasses, well-formed inheritance hierarchy, subtypes.

Environment lookup and update, and the least upper-bound operation on types and environments are defined in Figure 11.

For program P with $\vdash\ \mathsf{P} \diamond_h$, identifier c such that

$$\mathcal{C}(\mathsf{P}, \mathsf{c}) \ =\ [\mathbf{root} \mid \mathbf{state}]\ \mathbf{class}\ \mathsf{c}\ \mathbf{extends}\ \mathsf{c}'\{\mathsf{cBody}\},$$

and identifiers f and m we define:

$$\begin{aligned}
\mathcal{FD}(\mathsf{P}, \mathsf{c}, \mathsf{f}) &= \begin{cases} \mathsf{t} & \textit{if}\ \ \mathsf{cBody} = \cdots\ \mathsf{t}\ \mathsf{f}\ \cdots \\ \mathcal{U}df & \textit{otherwise} \end{cases} \\
\mathcal{F}(\mathsf{P}, \mathsf{c}, \mathsf{f}) &= \begin{cases} \mathcal{FD}(\mathsf{P}, \mathsf{c}, \mathsf{f}) & \textit{if}\ \mathcal{FD}(\mathsf{P}, \mathsf{c}, \mathsf{f}) \neq \mathcal{U}df, \\ \mathcal{F}(\mathsf{P}, \mathsf{c}', \mathsf{f}) & \textit{otherwise} \end{cases} \\
\mathcal{F}(\mathsf{P}, \mathsf{Object}, \mathsf{f}) &= \mathcal{U}df \\
\mathcal{F}s(\mathsf{P}, \mathsf{c}) &= \{\mathsf{f} \mid \mathcal{F}(\mathsf{P}, \mathsf{c}, \mathsf{f}) \neq \mathcal{U}df\} \\
\mathcal{MD}(\mathsf{P}, \mathsf{c}, \mathsf{m}) &= \begin{cases} \mathsf{t}\ \mathsf{m}(\mathsf{t}_1\ \mathsf{x}_1, \ldots, \mathsf{t}_n\ \mathsf{x}_n)\,\phi\,\{\mathsf{e}\} & \textit{if}\ \ \mathsf{cBody} \\ & \quad = \cdots \mathsf{t}\ \mathsf{m}\,(\mathsf{t}_1\ \mathsf{x}_1 \cdots \mathsf{t}_n\ \mathsf{x}_n)\phi\{\mathsf{e}\} \cdots \\ \mathcal{U}df & \textit{otherwise} \end{cases} \\
\mathcal{M}(\mathsf{P}, \mathsf{c}, \mathsf{m}) &= \begin{cases} \mathcal{MD}(\mathsf{P}, \mathsf{c}, \mathsf{m}) & \textit{if}\ \mathcal{MD}(\mathsf{P}, \mathsf{c}, \mathsf{m}) \neq \mathcal{U}df, \\ \mathcal{M}(\mathsf{P}, \mathsf{c}', \mathsf{m}) & \textit{otherwise} \end{cases} \\
\mathcal{M}(\mathsf{P}, \mathsf{Object}, \mathsf{m}) &= \mathcal{U}df
\end{aligned}$$

Figure 12 introduces agreement notions between programs, stores, and values as informally introduced in Section 5.5. The judgment $\mathsf{P}, \sigma \vdash \mathsf{v} \prec \mathsf{t}$ is instrumental to the definition of $\mathsf{P}, \sigma \vdash \mathsf{v} \lhd \mathsf{t}$: it avoids the use of coinduction.

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

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

$$\Gamma \sqcup_P \Gamma' = \{id : (t \sqcup_P t') \mid \Gamma(id) = t \text{ and } \Gamma'(id) = t'\}$$

Fig. 11. Environment lookup and update, lub on types and environments.

$$\frac{v = \texttt{true or } v = \texttt{false}}{P, \sigma \vdash v \prec \textbf{bool}} \ (\textbf{bool} \prec) \qquad \frac{P \vdash t \diamond_{ct}}{P, \sigma \vdash \texttt{null} \prec t} \ (\texttt{null} \prec) \qquad \frac{\sigma(\iota) = [[\cdots]]^c \quad P \vdash c \leq t}{P, \sigma \vdash \iota \prec t} \ (\iota \prec)$$

$$\frac{P, \sigma \vdash v \prec t \quad v \in sVal}{P, \ \sigma \vdash v \lhd t} \ (sVal \lhd) \qquad \frac{\begin{array}{l}\sigma(\iota) = [[\cdots]]^c \quad P, \sigma \vdash \iota \prec t \\ \forall f \in \mathcal{F}s(P, c) : \ P, \sigma \vdash \sigma(\iota)(f) \prec \mathcal{F}(P, c, f)\end{array}}{P, \ \sigma \vdash \iota \lhd t} \ (\iota \lhd)$$

$$\frac{\begin{array}{l}\sigma(\texttt{this}) = \sigma'(\texttt{this}) \\ \sigma(\iota) = [[\cdots]]^c \Rightarrow \sigma'(\iota) = [[\cdots]]^{c'}, \ \phi @_P c = \phi @_P c'\end{array}}{P, \phi \vdash \sigma \lhd \sigma'} \ (\sigma \lhd)$$

$$\frac{\begin{array}{l}\sigma(\iota) = [[\cdots]]^c \Rightarrow P, \ \sigma \vdash \iota \lhd c \quad \text{(for all addresses } \iota) \\ \Gamma(id) \neq \mathcal{U}df \Rightarrow P, \ \sigma \vdash \sigma(id) \lhd \Gamma(id) \quad \text{(for all identifiers id)}\end{array}}{P, \Gamma \vdash \sigma \diamond} \ (\diamond)$$

Fig. 12. Agreement between programs, stores, and values.

The judgment $P, \phi \vdash \sigma \lhd \sigma'$ guarantees that the differences from σ to σ' are "small"; in particular, only objects of a state subclass of a class in ϕ may be reclassified.

C PROOF OF THE TYPE SOUNDNESS THEOREM

We start with Propositions 2 and 3. Proposition 2 states that in well-formed programs, any subclass c' of a class c inherits all the fields of c, and also method definitions are inherited provided that they are not overridden along intermediate classes.

PROPOSITION 2. *For program* P*, identifiers* c, c', f, m*, expression* e*, types* t, t', t_i, t_i', $(i \in 1, \ldots, n)$*, effects* ϕ, ϕ'*, with* $\vdash P \diamond$:

(1) $\mathcal{F}(P, c', f) \neq \mathcal{U}df, \ P \vdash c \sqsubseteq c' \ \Rightarrow \ \mathcal{F}(P, c, f) = \mathcal{F}(P, c', f)$.

(2) $\mathcal{M}(P, c, m) \neq \mathcal{U}df \ \Rightarrow \ \exists c'' : \ P \vdash c \sqsubseteq c'', \ \mathcal{M}(P, c, m) = \mathcal{MD}(P, c'', m)$.

(3) $\mathcal{M}(\mathsf{P}, \mathsf{c}, \mathsf{m}) = \mathsf{t}\ \mathsf{m}(\mathsf{t}_1\ \mathsf{x}_1, \ldots, \mathsf{t}_n\ \mathsf{x}_n)\ \phi\ \{\mathsf{e}\},\ \mathsf{P} \vdash \mathsf{c} \sqsubseteq \mathsf{c} \Rightarrow$
$\exists \mathsf{e}', \phi' : \mathcal{M}(\mathsf{P}, \mathsf{c}', \mathsf{m}) = \mathsf{t}\ \mathsf{m}(\mathsf{t}_1\ \mathsf{x}_1, \ldots, \mathsf{t}_n\ \mathsf{x}_n)\ \phi'\ \{\ \mathsf{e}'\ \},\ \phi' \subseteq \phi.$

(4) $\mathcal{M}(\mathsf{P}, \mathsf{c}, \mathsf{m}) = \mathsf{t}\ \mathsf{m}(\mathsf{t}_1\ \mathsf{x}_1, \ldots, \mathsf{t}_n\ \mathsf{x}_n)\ \phi\ \{\mathsf{e}\},\ \mathsf{P} \vdash \mathsf{c}' \sqsubseteq \mathsf{c},$
$\mathcal{M}(\mathsf{P}, \mathsf{c}', \mathsf{m}) = \mathsf{t}'\ \mathsf{m}(\mathsf{t}_1\ \mathsf{x}_1, \ldots, \mathsf{t}_n\ \mathsf{x}_n)\ \phi'\ \{\ \mathsf{e}'\ \}$
$\Rightarrow\ \mathsf{t} = \mathsf{t}',\ \mathsf{t}_i = \mathsf{t}'_i$ *for* $\mathsf{i} \in 1, \ldots, \mathsf{n},\ \phi' \subseteq \phi.$

Proposition 3 states some properties of agreement that easily follow from the definitions in Figure 12.

PROPOSITION 3. *For programs* P, *environments* Γ, *states* σ, σ', σ'', *values* v, *types* t, t', *effects* ϕ, ϕ', *identifiers* id:

(1) $\mathsf{P}, \sigma \vdash \mathsf{v} \prec \mathsf{t},\ \mathsf{P} \vdash \mathsf{t} \leq \mathsf{t}' \Rightarrow \mathsf{P}, \sigma \vdash \mathsf{v} \prec \mathsf{t}'.$

(2) $\mathsf{P}, \sigma \vdash \mathsf{v} \lhd \mathsf{t},\ \mathsf{P} \vdash \mathsf{t} \leq \mathsf{t}' \Rightarrow \mathsf{P}, \sigma \vdash \mathsf{v} \lhd \mathsf{t}.$

(3) $\mathsf{P}, \phi \vdash \sigma \lhd \sigma',\ \mathsf{P}, \phi' \vdash \sigma' \lhd \sigma'' \Rightarrow \mathsf{P}, \phi \cup \phi' \vdash \sigma \lhd \sigma''.$

(4) $\mathsf{P}, \phi \vdash \sigma \lhd \sigma',\ \phi \subseteq \phi' \Rightarrow \mathsf{P}, \phi' \vdash \sigma \lhd \sigma'.$

(5) $\mathsf{P},\ \Gamma\ \vdash \sigma \diamond, \mathsf{P} \vdash \Gamma(\mathsf{id}) \leq \mathsf{t}' \Rightarrow \mathsf{P}, \Gamma[\mathsf{id} \mapsto \mathsf{t}'] \vdash \sigma \diamond.$

(6) $\phi \subseteq \phi',\ \mathsf{P} \vdash \mathsf{t} \leq \mathsf{t}' \Rightarrow \mathsf{P} \vdash \phi @_{\mathsf{P}} \mathsf{t} \leq \phi' @_{\mathsf{P}} \mathsf{t}'.$

Theorem 1 is a direct consequence of the following lemma that, asserts that, if the evaluation of a well-typed expression terminates, either it produces a value or a null-pointer exception (no stuck error). Moreover, if the evaluation produces a value, then such a value agrees with the type of the expression, the store produced agrees with the original store with respect to the final typing environment, and the evaluation only modifies the identifiers that are explicitly mentioned in the typing environment. If the evaluation produces an exception, then the store produced agrees with the original store with respect to an environment which is weaker than the final typing environment. This weakening is due to the propagation of exceptions in the case of sequences of expressions, see Remark 3.

LEMMA 1. *For well-formed program* P, *environments* Γ, Γ', *expression* e, *effect* ϕ, *and type* t, *such that*

$$\mathsf{P}, \Gamma \vdash \mathsf{e} :\ \mathsf{t}\ []\ \Gamma'\ []\ \phi$$

if $\mathsf{P}, \Gamma \vdash \sigma \diamond$, *and* e, σ *converges, then*

—*either* $\mathsf{e}, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{v}, \sigma'$, *where*
(1) $\mathsf{P},\ \sigma' \vdash \mathsf{v} \lhd\ \mathsf{t},$
(2) $\mathsf{P}, \phi \vdash \sigma \lhd \sigma',$
(3) $\mathsf{P}, \Gamma' \vdash \sigma' \diamond,$
(4) *for all* id, $\Gamma'(\mathsf{id}) = \mathcal{U}df$ *implies* $\sigma(\mathsf{id}) = \sigma'(\mathsf{id}).$

—*or* $\mathsf{e}, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{dv}, \sigma'$, *where*
(1) $\mathsf{dv} = \texttt{nullPntrExc},$
(2) $\mathsf{P}, \phi \vdash \sigma \lhd \sigma',$
(3) $\mathsf{P}, \phi @_{\mathsf{P}} \Gamma' \vdash \sigma' \diamond,$
(4) *for all* id, $\Gamma'(\mathsf{id}) = \mathcal{U}df$ *implies* $\sigma(\mathsf{id}) = \sigma'(\mathsf{id}).$

PROOF. Since e, σ converges there exists a value or a deviation r and a store σ' such that $\mathsf{e}, \sigma \overset{*}{\underset{p}{\rightarrow}} \mathsf{r}, \sigma'$. We prove the theorem by induction on the application of the rules that define the operational semantics.

We first consider the significant rules of Figure 3.

—*CONDITIONAL*. In this case e is **if** e' **then** e_1 **else** e_2. Since

$$\mathsf{P}, \Gamma \vdash \mathsf{e} : \mathsf{t} \,[]\, \Gamma' \,[]\, \phi$$

from (*cond*) we get that

$$\frac{\begin{array}{l}(\alpha)\ \ \mathsf{P}, \Gamma \vdash \mathsf{e}' : \mathsf{bool} \,[]\, \Gamma_0 \,[]\, \phi' \\ (\beta)\ \ \mathsf{P}, \Gamma_0 \vdash \mathsf{e}_1 : \mathsf{t}_1 \,[]\, \Gamma_1 \,[]\, \phi_1 \\ \qquad\ \ \mathsf{P}, \Gamma_0 \vdash \mathsf{e}_2 : \mathsf{t}_2 \,[]\, \Gamma_2 \,[]\, \phi_2\end{array}}{\mathsf{P}, \Gamma \vdash \textbf{if } \mathsf{e}' \textbf{ then } \mathsf{e}_1 \textbf{ else } \mathsf{e}_2 : \mathsf{t} \,[]\, \Gamma' \,[]\, \phi}$$

for $\mathsf{t} = \mathsf{t}_1 \sqcup_{\mathsf{P}} \mathsf{t}_2$, $\Gamma' = \Gamma_1 \sqcup_{\mathsf{P}} \Gamma_2$, and $\phi = \phi' \cup \phi_1 \cup \phi_2$. From the operational semantics, we have

$$\frac{\begin{array}{l}(\gamma)\ \ \mathsf{e}', \sigma \overset{*}{\underset{p}{\rightarrow}} \texttt{true}\ , \sigma'' \\ (\delta)\ \ \mathsf{e}_1, \sigma'' \overset{*}{\underset{p}{\rightarrow}} \mathsf{v}, \sigma'\end{array}}{\textbf{if } \mathsf{e}' \textbf{ then } \mathsf{e}_1 \textbf{ else } \mathsf{e}_2, \sigma \overset{*}{\underset{p}{\rightarrow}} \mathsf{v}, \sigma'} \quad \text{or} \quad \frac{\begin{array}{l}\mathsf{e}', \sigma \overset{*}{\underset{p}{\rightarrow}} \texttt{false}\ , \sigma'' \\ \mathsf{e}_2, \sigma'' \overset{*}{\underset{p}{\rightarrow}} \mathsf{v}, \sigma'\end{array}}{\textbf{if } \mathsf{e}' \textbf{ then } \mathsf{e}_1 \textbf{ else } \mathsf{e}_2, \sigma \overset{*}{\underset{p}{\rightarrow}} \mathsf{v}, \sigma'}$$

Assume that the first rule was used, the other case being similar. By inductive hypothesis on (α), $\mathsf{P}, \Gamma \vdash \sigma \diamond$, and (γ), we get that

$(2')$ $\mathsf{P}, \phi' \vdash \sigma \lhd \sigma''$,

$(3')$ $\mathsf{P}, \Gamma_0 \vdash \sigma'' \diamond$,

$(4')$ for all id, $\Gamma_0(\mathsf{id}) = \mathit{Udf}$ implies $\sigma(\mathsf{id}) = \sigma''(\mathsf{id})$.

By inductive hypothesis on (β), $(3')$, and (δ), we derive

$(1'')$ $\mathsf{P}, \sigma' \vdash \mathsf{v} \lhd \mathsf{t}_1$,

$(2'')$ $\mathsf{P}, \phi_1 \vdash \sigma'' \lhd \sigma'$,

$(3'')$ $\mathsf{P}, \Gamma_1 \vdash \sigma' \diamond$,

$(4'')$ for all id, $\Gamma_1(\mathsf{id}) = \mathit{Udf}$ implies $\sigma''(\mathsf{id}) = \sigma'(\mathsf{id})$.

From $(1'')$, $\mathsf{P} \vdash \mathsf{t}_1 \leq \mathsf{t}$, and Proposition 3(2), we derive statement (1). From the fact that $\phi = \phi' \cup \phi_1 \cup \phi_2$, and from $(2')$, $(2'')$, and Proposition 3(3) and 3(4), we derive statement (2). From $\Gamma = \Gamma_1 \sqcup_{\mathsf{P}} \Gamma_2$, we obtain $\mathsf{P} \vdash \Gamma_1(\mathsf{id}) \leq \Gamma(\mathsf{id})$ for all id such that $\Gamma(\mathsf{id}) \neq \mathit{Udf}$. From that, $(3'')$, and Proposition 3(5), we derive statement (3). Lastly, (4) follows from $(4')$ and $(4'')$ by Proposition 1(1).

—*ASSIGNMENT TO IDENTIFIERS*. This case is similar and simpler than the following one. Notice that `this` cannot be on the left hand side of an assignment; therefore, the location to which `this` is bound before and after the execution of the assignment does not change. Moreover, for an assignment to be well typed, the identifier must be defined in the environment.

—*ASSIGNMENT TO FIELDS*. In this case, e is $\mathsf{e}'.f := \mathsf{e}''$. Since e is well typed, from (*a-field*), we have

$$\frac{\begin{array}{ll}(\alpha) & \mathsf{P}, \Gamma \vdash \mathsf{e}' : \mathsf{c}\ []\ \Gamma_0\ []\ \phi' \\ (\beta) & \mathsf{P}, \Gamma_0 \vdash \mathsf{e}'' : \mathsf{t}\ []\ \Gamma'\ []\ \phi'' \\ (\gamma) & \mathcal{F}(\mathsf{P}, \phi''@_{\mathsf{P}}\mathsf{c}, \mathsf{f}) = \mathsf{t}' \\ (\delta) & \mathsf{P} \vdash \mathsf{t} \leq \mathsf{t}'\end{array}}{\mathsf{P}, \Gamma \vdash \mathsf{e}'.\mathsf{f} := \mathsf{e}'' : \mathsf{t}\ []\ \Gamma'\ []\ \phi}$$

where $\phi = \phi' \cup \phi''$. From the operational semantics rules, we get that

$$\frac{\begin{array}{ll}(\mu) & \mathsf{e}', \sigma \curvearrowright_{\mathsf{p}} \iota, \sigma'' \\ (\nu) & \mathsf{e}'', \sigma'' \curvearrowright_{\mathsf{p}} \mathsf{v}, \sigma''' \\ & \sigma'''(\iota)(\mathsf{f}) \neq \mathcal{U}df \\ (\pi) & \sigma' = \sigma'''[\iota \mapsto \sigma'''(\iota)[\mathsf{f} \mapsto \mathsf{v}]]\end{array}}{\mathsf{e}'.\mathsf{f} := \mathsf{e}'', \sigma \curvearrowright_{\mathsf{p}} \mathsf{v}, \sigma'}$$

By inductive hypothesis on (α), $\mathsf{P}, \Gamma \vdash \sigma\,\diamond$, (μ) we get

$(1')$ $\mathsf{P},\ \sigma'' \vdash \iota \lhd\ \mathsf{c}$,

$(2')$ $\mathsf{P}, \phi' \vdash\ \sigma \lhd\ \sigma''$,

$(3')$ $\mathsf{P}, \Gamma_0 \vdash\ \sigma''\,\diamond$,

$(4')$ for all id, $\Gamma_0(\mathsf{id}) = \mathcal{U}df$ implies $\sigma(\mathsf{id}) = \sigma''(\mathsf{id})$.

By inductive hypothesis on (β), $(3')$, (ν) we get

$(1'')$ $\mathsf{P},\ \sigma''' \vdash \mathsf{v} \lhd \mathsf{t}$,

$(2'')$ $\mathsf{P}, \phi'' \vdash\ \sigma'' \lhd\ \sigma'''$,

$(3'')$ $\mathsf{P}, \Gamma' \vdash\ \sigma'''\,\diamond$,

$(4'')$ for all id, $\Gamma'(\mathsf{id}) = \mathcal{U}df$, implies $\sigma''(\mathsf{id}) = \sigma'''(\mathsf{id})$.

We first prove (3), that is, $\mathsf{P}, \Gamma' \vdash \sigma'\,\diamond$. By the definition given in rule $(\diamond)$ of Figure 12, we have to prove that:

(5) $\sigma'(\iota') = [[\cdots]]^{\mathsf{c}'} \Rightarrow \mathsf{P},\ \sigma' \vdash \iota' \lhd\ \mathsf{c}'$ for all addresses ι',

(6) $\Gamma'(\mathsf{id}) \neq \mathcal{U}df \Rightarrow \mathsf{P},\ \sigma' \vdash \sigma'(\mathsf{id}) \lhd\ \Gamma'(\mathsf{id})$ for all identifiers id.

Notice that σ' is obtained from σ''' by modifying only one field of the object bound to address ι, see (π). Therefore, the classes of all objects do not change, that is,

(7) $\sigma'''(\iota') = [[\cdots]]^{\mathsf{c}'} \Leftrightarrow\ \sigma'(\iota') = [[\cdots]]^{\mathsf{c}'}$.

From $(3'')$ by rule $(\diamond)$ of Figure 12 we have:

$(5')$ $\sigma'''(\iota') = [[\cdots]]^{\mathsf{c}'} \Rightarrow \mathsf{P},\ \sigma''' \vdash \iota' \lhd\ \mathsf{c}'$ for all addresses ι', and

$(6')$ $\Gamma'(\mathsf{id}) \neq \mathcal{U}df \Rightarrow \mathsf{P},\ \sigma''' \vdash \sigma'''(\mathsf{id}) \lhd\ \Gamma'(\mathsf{id})$ for all identifiers id.

When $\iota' \neq \iota$ we get (5) from $(5')$ and (π) since the fields of the object at address ι' in σ' is the same as in σ'''.

If $\iota' = \iota$ from $(1')$ and rules $(\iota \lhd)$ and $(\iota \prec)$ of Figure 12, we obtain that $\sigma''(\iota) = [[\cdots]]^{\mathsf{d}}$, for some d such that $\mathsf{P} \vdash \mathsf{d} \sqsubseteq \mathsf{c}$. From (7), $(2'')$ and rule $(\sigma \lhd)$ of Figure 12, we derive that $\sigma'(\iota) = [[\cdots]]^{\mathsf{d}'}$, for some d′ such that $\phi''@_{\mathsf{P}}\mathsf{d} = \phi''@_{\mathsf{P}}\mathsf{d}'$. Therefore, by Proposition 3(6), we get $\mathsf{P} \vdash \mathsf{d}' \sqsubseteq \phi''@_{\mathsf{P}}\mathsf{d}'$, $\mathsf{P} \vdash \phi''@_{\mathsf{P}}\mathsf{d} \sqsubseteq \phi''@_{\mathsf{P}}\mathsf{c}$, and these imply $\mathsf{P} \vdash \mathsf{d}' \sqsubseteq \phi''@_{\mathsf{P}}\mathsf{c}$. This, together with Proposition 2(1), and (γ), gives $\mathcal{F}(\mathsf{P},\mathsf{d}', \mathsf{f}) = \mathsf{t}'$. Notice that $\mathsf{v} = \sigma'(\iota)(\mathsf{f})$, so $(1'')$,

rule ($\iota \lhd$) of Figure 12, (δ) and Proposition 3(1) imply $\mathsf{P}, \sigma''' \vdash \sigma'(\iota)(\mathsf{f}) \prec \mathsf{t}'$. Using (7), if v is an address, we obtain $\mathsf{P}, \sigma' \vdash \sigma'(\iota)(\mathsf{f}) \prec \mathsf{t}'$, and therefore $\mathsf{P},\ \sigma' \vdash \iota \lhd \mathsf{d}'$. This concludes the proof of (5).

To prove (6), consider an id such that $\Gamma'(\mathsf{id}) \neq \mathit{Udf}$. By ($\pi$), $\sigma'(\mathsf{id}) = \sigma'''(\mathsf{id})$. Therefore, from condition (6′), we get $\mathsf{P}, \sigma' \vdash \sigma'(\mathsf{id}) \prec \Gamma'(\mathsf{id})$. If $\Gamma'(\mathsf{id}) = \mathbf{bool}$, we are done. Otherwise, let $\sigma'(\sigma'(\mathsf{id})) = [[\cdots]]^{\mathsf{c}'}$. Rule ($\iota \prec$) of Figure 12 implies that $\mathsf{P} \vdash \mathsf{c}' \sqsubseteq \Gamma'(\mathsf{id})$. From (5) and Proposition 3(2), we can conclude that $\mathsf{P},\ \sigma' \vdash \sigma'(\mathsf{id}) \lhd \Gamma'(\mathsf{id})$, that is, we proved (6). This concludes also the proof of (3).

We get (1) from (1″) using (5) and (7) when v is an address.

From (2′), (2″) and Proposition 3(3), we derive that $\mathsf{P}, \phi' \cup \phi'' \vdash \sigma \lhd \sigma'''$, and from the construction of σ' out of σ''', that is, (π), we also obtain that $\mathsf{P}, \{\} \vdash \sigma''' \lhd \sigma'$. The two last two statements together with Proposition 3(3) give statement (2).

Lastly, (4) follows from (4′) and (4″) by (π) and Proposition 1(1).

—*METHOD CALL*. We prove the result for methods with just one parameter. (The proof for more than one parameter would be similar.) Let $\mathsf{e} = \mathsf{e}_0.\mathsf{m}(\mathsf{e}_1)$. From (*meth*), we have

$$\frac{\begin{array}{ll}(\alpha) & \mathsf{P}, \Gamma \vdash \mathsf{e}_0 :\ \mathsf{c}\ []\ \Gamma_0\ []\ \phi_0 \\ (\beta) & \mathsf{P}, \Gamma_0 \vdash \mathsf{e}_1 :\ \mathsf{t}'_1\ []\ \Gamma_1\ []\ \phi_1 \\ (\gamma) & \mathcal{M}(\mathsf{P}, \phi_1 @_{\mathsf{P}} \mathsf{c}, \mathsf{m}) = \mathsf{t}\ \mathsf{m}(\mathsf{t}_1\ \mathsf{x}_1)\ \phi'\ \{\mathsf{e}'\} \\ (\delta) & \mathsf{P} \vdash \mathsf{t}'_1 \leq \mathsf{t}_1\end{array}}{\mathsf{P}, \Gamma \vdash \mathsf{e}_0.\mathsf{m}(\mathsf{e}_1) :\ \mathsf{t}\ []\ \Gamma'\ []\ \phi}$$

where $\Gamma' = \phi' @_{\mathsf{P}} \Gamma_1$, and $\phi = \phi' \cup \phi_0 \cup \phi_1$.

The rule for the operational semantics is

$$\frac{\begin{array}{ll}(\mu) & \mathsf{e}_0,\ \sigma \overset{\cdot}{\underset{\mathsf{p}}{\frown}} \iota,\ \sigma_0 \\ (\nu) & \mathsf{e}_1,\ \sigma_0 \overset{\cdot}{\underset{\mathsf{p}}{\frown}} \mathsf{v}_1,\ \sigma_1 \\ (\pi) & \sigma_1(\iota) = [[\cdots]]^{\mathsf{c}'} \\ (\rho) & \mathcal{M}(\mathsf{P}, \mathsf{c}', \mathsf{m}) = \mathsf{t}''\ \mathsf{m}(\mathsf{t}''_1\ \mathsf{x}_1)\ \phi''\ \{\mathsf{e}''\} \\ (\xi) & \sigma'' = \sigma_1[\mathtt{this} \mapsto \iota, \mathsf{x}_1 \mapsto \mathsf{v}_1] \\ (\zeta) & \mathsf{e}'',\ \sigma'' \overset{\cdot}{\underset{\mathsf{p}}{\frown}} \mathsf{v},\ \sigma'''\end{array}}{\mathsf{e}_0.\mathsf{m}(\mathsf{e}_1),\ \sigma \overset{\cdot}{\underset{\mathsf{p}}{\frown}} \mathsf{v},\ \sigma'}$$

where (χ) $\sigma' = \sigma'''[\mathtt{this} \mapsto \sigma_1(\mathtt{this}), \mathsf{x}_1 \mapsto \sigma_1(\mathsf{x}_1)]$.

Applying the inductive hypothesis to (α), $\mathsf{P}, \Gamma \vdash \sigma \diamond$, and ($\mu$), we derive

(1′) $\mathsf{P},\ \sigma_0 \vdash \iota \lhd \mathsf{c}$,

(2′) $\mathsf{P}, \phi_0 \vdash \sigma \lhd \sigma_0$,

(3′) $\mathsf{P}, \Gamma_0 \vdash \sigma_0 \diamond$,

(4′) for all id, $\Gamma_0(\mathsf{id}) = \mathit{Udf}$ implies $\sigma(\mathsf{id}) = \sigma_0(\mathsf{id})$.

Applying the inductive hypothesis to (β), (3′), and (ν), we derive

(1″) $\mathsf{P},\ \sigma_1 \vdash \mathsf{v}_1 \lhd \mathsf{t}'_1$,

(2″) $\mathsf{P}, \phi_1 \vdash \sigma_0 \lhd \sigma_1$,

(3″) $\mathsf{P}, \Gamma_1 \vdash \sigma_1 \diamond$,

(4″) for all id, $\Gamma_1(\mathsf{id}) = \mathit{Udf}$ implies $\sigma_0(\mathsf{id}) = \sigma_1(\mathsf{id})$.

From $(3'')$ and (ξ), we know that all the objects have fields consistent with their class in the state σ'', that is,

(5) $\sigma''(\iota') = [[\cdots]]^{\mathsf{d}'} \Rightarrow \mathsf{P}''$, $\sigma' \vdash \iota' \lhd \mathsf{d}'$ for all addresses ι'.

From $(1')$, $(2'')$, and (π), we obtain by rules $(\iota \lhd)$, $(\iota \prec)$ and $(\sigma \lhd)$ of Figure 12 that $\mathsf{P} \vdash \phi_1@_{\mathsf{P}}\mathsf{c}' \sqsubseteq \phi_1@_{\mathsf{P}}\mathsf{c}$. Therefore, by Proposition 3(6), we also get $\mathsf{P} \vdash \mathsf{c}' \sqsubseteq \phi_1@_{\mathsf{P}}\mathsf{c}$. From (γ), (ρ), and Proposition 2(4), we have that $\mathsf{t}'' = \mathsf{t}$, $\mathsf{t}''_1 = \mathsf{t}_1$, and $\phi'' \subseteq \phi'$. From the fact that the program is well formed, and Proposition 2(2), we obtain that there exists a c'', with $\mathsf{P} \vdash \mathsf{c}' \sqsubseteq \mathsf{c}''$, and $\mathcal{M}(\mathsf{P}, \mathsf{c}', \mathsf{m}) = \mathcal{MD}(\mathsf{P}, \mathsf{c}'', \mathsf{m})$. Now, applying the requirements for well-formed programs from Figure 7, we get that for some t', Γ''', ϕ''':

(6) $\mathsf{P}, \Gamma'' \vdash \mathsf{e}'' : \mathsf{t}' \,[]\, \Gamma''' \,[]\, \phi'''$,

where $\Gamma'' = \{\mathtt{this} : \mathsf{c}'', \mathsf{x}_1 : \mathsf{t}_1\}$, and $\mathsf{P} \vdash \mathsf{t}' \leq \mathsf{t}$, $\phi''' \subseteq \phi''$. From $(1'')$, Proposition 3(2), and (δ) we have that P, $\sigma_1 \vdash \mathsf{v}_1 \lhd \mathsf{t}_1$. By (ξ) this implies

(7) P, $\sigma'' \vdash \sigma''(\mathsf{x}_1) \lhd \Gamma''(\mathsf{x}_1)$.

Because $\mathsf{P} \vdash \mathsf{c}' \sqsubseteq \mathsf{c}''$, we also have using (5) and (ξ)

(8) P, $\sigma'' \vdash \sigma''(\mathsf{this}) \lhd \Gamma''(\mathsf{this})$.

So from $(3'')$, (ξ), (5), (7), (8) and rule $(\diamond)$ of Figure 12, we get

(9) $\mathsf{P}, \Gamma'' \vdash \sigma'' \diamond$.

We now apply the inductive hypothesis to (6), (9) and (ζ). Thus, we derive that

$(1''')$ P, $\sigma''' \vdash \mathsf{v} \lhd \mathsf{t}'$,

$(2''')$ $\mathsf{P}, \phi''' \vdash \sigma'' \lhd \sigma'''$,

$(3''')$ $\mathsf{P}, \Gamma''' \vdash \sigma''' \diamond$,

$(4''')$ for all id, $\Gamma'''(\mathsf{id}) = \mathcal{U}df$ implies $\sigma''(\mathsf{id}) = \sigma'''(\mathsf{id})$.

Because of $(1''')$, the construction of σ' from σ''', $\mathsf{P} \vdash \mathsf{t}' \leq \mathsf{t}$, and Proposition 3(2), we derive statement (1).

From $(2')$, $(2'')$, and Proposition 3(3), we have $\mathsf{P}, \phi_0 \cup \phi_1 \vdash \sigma \lhd \sigma_1$. Because of Proposition 3(4), we also have

(10) $\mathsf{P}, \phi \vdash \sigma \lhd \sigma_1$.

From $(2''')$, and Proposition 3(4), since $\phi''' \subseteq \phi'' \subseteq \phi' \subseteq \phi$, we obtain that

(11) $\mathsf{P}, \phi \vdash \sigma'' \lhd \sigma'''$.

Now we show (2), that is, $\mathsf{P}, \phi \vdash \sigma \lhd \sigma'$. First, $\sigma(\mathtt{this}) = \sigma_0(\mathtt{this})$ by $(2')$, $\sigma_0(\mathtt{this}) = \sigma_1(\mathtt{this})$ by $(2'')$ and $\sigma'(\mathtt{this}) = \sigma_1(\mathtt{this})$ by construction of σ'. So $\sigma(\mathtt{this}) = \sigma'(\mathtt{this})$. To show the second condition of rule $(\sigma \lhd)$ of Figure 12, let ι' be any address, and let class d be the class of ι' in σ, that is, $\sigma(\iota') = [[\cdots]]^{\mathsf{d}}$. Because of (10), there exists a class d' with $\sigma_1(\iota') = [[\cdots]]^{\mathsf{d}'}$, and $\phi@_{\mathsf{P}}\mathsf{d} = \phi@_{\mathsf{P}}\mathsf{d}'$. Also, by (ξ), we obtain that $\sigma''(\iota') = \sigma_1(\iota')$. Furthermore, because of (11), there exists a class d'' with $\sigma'''(\iota') = [[\cdots]]^{\mathsf{d}''}$, where $\phi@_{\mathsf{P}}\mathsf{d}' = \phi@_{\mathsf{P}}\mathsf{d}''$. By (χ), we also have $\sigma'(\iota') = \sigma'''(\iota')$. This concludes the proof of statement (2).

To prove (3), that is, $\mathsf{P},\Gamma' \vdash \sigma' \diamond$, by rule $(\diamond)$ of Figure 12, we have to show that:

(12) $\sigma'(\iota') = [[\cdots]]^{\mathsf{c}'''} \Rightarrow \mathsf{P},\ \sigma' \vdash \iota' \lhd \mathsf{c}'''$ for all addresses ι', and

(13) $\Gamma'(\mathsf{id}) \neq \mathcal{U}df \Rightarrow \mathsf{P},\ \sigma' \vdash \sigma'(\mathsf{id}) \lhd \Gamma'(\mathsf{id})$ for all identifiers id.

From (3‴), we get (12) since (χ) implies $\sigma'(\iota') = \sigma'''(\iota')$ for all addresses ι'. Notice that (3″) implies

(14) $\mathsf{P},\ \sigma_1 \vdash \sigma_1(\mathsf{id}) \lhd \Gamma_1(\mathsf{id})$.

To prove (13) consider first $\mathsf{id} \neq \mathsf{x}_1$ and $\mathsf{id} \neq \mathtt{this}$. By (ξ), $\sigma''(\mathsf{id}) = \sigma_1(\mathsf{id})$, and by (χ), $\sigma'(\mathsf{id}) = \sigma'''(\mathsf{id})$. By (4‴) and Proposition 1(1) applied to (6) $\sigma''(\mathsf{id}) = \sigma'''(\mathsf{id})$. This shows $\sigma'(\mathsf{id}) = \sigma_1(\mathsf{id})$. So from (14) and Proposition 3(1), $\mathsf{P},\sigma' \vdash \sigma'(\mathsf{id}) \prec \phi'@_{\mathsf{P}}\Gamma_1(\mathsf{id}) = \Gamma'(\mathsf{id})$ since $\mathsf{P} \vdash \Gamma_1(\mathsf{id}) \leq \phi'@_{\mathsf{P}}\Gamma_1(\mathsf{id})$. If $\Gamma'(\mathsf{id}) = \mathbf{bool}$, we are done. Otherwise, let $\sigma'(\sigma'(\mathsf{id})) = [[\cdots]]^{\mathsf{c}'}$. Rule $(\iota \prec)$ of Figure 12 and (3″), imply $\mathsf{P} \vdash \mathsf{c}' \sqsubseteq \Gamma'(\mathsf{id})$. Condition (12) and Proposition 3(2), allow us to conclude that $\mathsf{P},\ \sigma' \vdash \sigma'(\mathsf{id}) \lhd \Gamma'(\mathsf{id})$.

Let now $\mathsf{id} = \mathtt{this}$ or $\mathsf{id} = \mathsf{x}_1$. If $\Gamma_1(\mathsf{id}) = \mathbf{bool}$, then $\Gamma'(\mathsf{id}) = \mathbf{bool}$, and from $\sigma'(\mathsf{id}) = \sigma_1(\mathsf{id})$ and (14), we have that $\mathsf{P},\ \sigma' \vdash \sigma'(\mathsf{id}) \lhd \Gamma'(\mathsf{id})$.

If instead $\mathsf{P} \vdash \Gamma_1(\mathsf{id}) \diamond_{ct}$, let $\sigma_1(\mathsf{id}) = \iota''$, and $\sigma_1(\iota'') = [[\cdots]]^{\mathsf{c}_1}$. By (ξ), we get $\sigma''(\iota'') = \sigma_1(\iota'')$. Let $\sigma'''(\iota'') = [[\cdots]]^{\mathsf{d}_1}$. From (2‴), we get $\phi'''@_{\mathsf{P}}\mathsf{c}_1 = \phi'''@_{\mathsf{P}}\mathsf{d}_1$. Therefore from (14), $\phi''' \subseteq \phi'$, Proposition 3(1), and 3(6), we have that $\mathsf{P},\sigma' \vdash \iota'' \prec \phi'@_{\mathsf{P}}\Gamma_1(\mathsf{id})$. Since from (χ) we have $\sigma'(\iota'') = \sigma'''(\iota'')$, then, from rule $(\iota \prec)$ we get $\mathsf{P} \vdash \mathsf{d}_1 \sqsubseteq \Gamma'(\mathsf{id})$. Condition (12) and Proposition 3(2) allow us to get $\mathsf{P},\ \sigma' \vdash \sigma'(\mathsf{id}) \lhd \Gamma'(\mathsf{id})$. This concludes the proof of (13) and so also of (3).

Last, (4) follows from (4′), (4″) and (4‴) by Proposition 1(1).

—*RECLASSIFICATION*. In this case $\mathsf{e} = \mathsf{id}\Downarrow\mathsf{c}$. From rule (*recl*), we have

$$\frac{(\alpha)\ \ \mathsf{P} \vdash \mathsf{c} \diamond_{rt} \qquad (\beta)\ \ \mathcal{R}(\mathsf{P},\mathsf{c}) = \mathcal{R}(\mathsf{P},\Gamma(\mathsf{id}))}{\mathsf{P},\Gamma \vdash \mathsf{id}\Downarrow\mathsf{c} :\ \mathsf{c}\ []\ \Gamma'\ []\ \phi}$$

where $\Gamma' = (\{\mathcal{R}(\mathsf{P},\mathsf{c})\}@_{\mathsf{P}}\Gamma)[\mathsf{id} \mapsto \mathsf{c}]$ and $\phi = \{\mathcal{R}(\mathsf{P},\mathsf{c})\}$. Moreover

$$\frac{\begin{array}{l}(\gamma)\quad \sigma(\mathsf{id}) = \iota \quad \sigma(\iota) = [[\cdots]]^{\mathsf{d}} \\ \mathcal{F}s(\mathsf{P},\mathcal{R}(\mathsf{P},\mathsf{d})) = \{\mathsf{f}_1,\ldots,\mathsf{f}_r\} \\ \forall \mathsf{l} \in 1,\ldots,\mathsf{r} :\quad \mathsf{v}_\mathsf{l} = \sigma(\iota)(\mathsf{f}_\mathsf{l}) \\ \mathcal{F}s(\mathsf{P},\mathsf{c}) \setminus \{\mathsf{f}_1,\ldots,\mathsf{f}_r\} = \{\mathsf{f}_{r+1},\ldots,\mathsf{f}_{r+q}\} \\ \forall \mathsf{l} \in \mathsf{r}+1,\ldots,\mathsf{r}+\mathsf{q} :\quad \mathsf{v}_\mathsf{l} \text{ initial for } \mathcal{F}(\mathsf{P},\mathsf{c},\mathsf{f}_\mathsf{l})\end{array}}{\mathsf{id}\Downarrow\mathsf{c},\ \sigma \overset{\iota}{\rightsquigarrow}_{\mathsf{p}} \iota,\ \sigma'}$$

where $(\delta)\quad \sigma' = \sigma[\iota \mapsto [[\mathsf{f}_1 : \mathsf{v}_1,\ldots,\mathsf{f}_{r+q} : \mathsf{v}_{r+q}]]^{\mathsf{c}}]$.

We first show (3), that, by rule $(\diamond)$ of Figure 12, implies that we have to prove:

(5) $\sigma'(\iota') = [[\cdots]]^{\mathsf{c}'} \Rightarrow \mathsf{P},\ \sigma' \vdash \iota' \lhd \mathsf{c}'$ for all addresses ι', and

(6) $\Gamma'(\mathsf{id}') \neq \mathcal{U}df \Rightarrow \mathsf{P},\ \sigma' \vdash \sigma'(\mathsf{id}') \lhd \Gamma'(\mathsf{id}')$ for all identifiers id′.

From $\mathsf{P},\Gamma \vdash \sigma \diamond$ we have:

(5′) $\sigma(\iota') = [[\cdots]]^{\mathsf{d}'} \Rightarrow \mathsf{P},\ \sigma \vdash \iota' \lhd \mathsf{d}'$ for all addresses ι', and

(6′) $\Gamma(\mathsf{id}') \neq \mathcal{U}df \Rightarrow \mathsf{P},\ \sigma \vdash \sigma(\mathsf{id}') \lhd \Gamma(\mathsf{id}')$ for all identifiers id′.

Let $\mathsf{d}'' = \Gamma(\mathsf{id})$: (α) and (β) imply $\mathsf{P} \vdash \mathsf{d}'' \diamond_{rt}$. From (6′), rule $(\iota \lhd)$ and (γ), we know also that $\mathsf{P} \vdash \mathsf{d} \sqsubseteq \mathsf{d}''$. Therefore, by (β), $\mathcal{R}(\mathsf{P}, \mathsf{c}) = \mathcal{R}(\mathsf{P}, \mathsf{d}'') = \mathcal{R}(\mathsf{P}, \mathsf{d})$.
From rule $(\iota \lhd)$ of Figure 12 and (5′), we get:

(7) $\sigma(\iota') = [[\cdots]]^{\mathsf{d}'} \Rightarrow \mathsf{P}, \sigma \vdash \iota' \prec \mathsf{d}'$ for all addresses ι',

(8) $\sigma(\iota') = [[\cdots]]^{\mathsf{d}'} \Rightarrow \forall \mathsf{f} \in \mathcal{F}s(\mathsf{P}, \mathsf{d}') : \mathsf{P}, \sigma \vdash \sigma(\iota')(\mathsf{f}) \prec \mathcal{F}(\mathsf{P}, \mathsf{d}', \mathsf{f})$ for all addresses ι'.

Notice that:

—$\vdash \mathsf{P} \diamond$ implies $\mathsf{P} \vdash \mathcal{F}(\mathsf{P}, \mathsf{d}', \mathsf{f}) \diamond_{ft}$, that is, for all d', f, $\mathcal{F}(\mathsf{P}, \mathsf{d}', \mathsf{f})$ cannot be a state class;

—σ' is built out of σ by reclassifying only the object at the address ι from class d to class c. That is, by keeping the fields of the object at ι that are already in $\mathcal{R}(\mathsf{P}, \mathsf{d}) = \mathcal{R}(\mathsf{P}, \mathsf{c})$ and initializing all the other fields with a value that is compatible with their type in c.

Therefore, we get

(7′) $\sigma'(\iota') = [[\cdots]]^{\mathsf{c}'} \Rightarrow \mathsf{P}, \sigma' \vdash \iota' \prec \mathsf{c}'$ for all addresses ι',

(8′) $\sigma'(\iota') = [[\cdots]]^{\mathsf{c}'} \Rightarrow \forall \mathsf{f} \in \mathcal{F}s(\mathsf{P}, \mathsf{c}') : \mathsf{P}, \sigma' \vdash \sigma'(\iota)(\mathsf{f}) \prec \mathcal{F}(\mathsf{P}, \mathsf{c}', \mathsf{f})$ for all addresses ι'.

Therefore, (5) follows from (7′) and (8′) by rule $(\iota \lhd)$ of Figure 12.

To prove (6), first we consider the case $\Gamma(\mathsf{id}') \neq \mathcal{U}df$, and $\sigma(\mathsf{id}') \neq \iota$, that is, the identifiers that are not alias for id. By construction, $\mathsf{P} \vdash \Gamma(\mathsf{id}') \leq \Gamma'(\mathsf{id}')$. This, with (6′), (δ) and Proposition 3(1), gives $\mathsf{P}, \sigma' \vdash \sigma'(\mathsf{id}') \prec \Gamma'(\mathsf{id}')$. If $\Gamma'(\mathsf{id}') =$ **bool**, we are done. Otherwise, let $\sigma'(\sigma'(\mathsf{id}')) = [[\cdots]]^{\mathsf{c}'}$. Rule $(\iota \prec)$ of Figure 12 implies $\mathsf{P} \vdash \mathsf{c}' \sqsubseteq \Gamma'(\mathsf{id})$. Condition (5) and Proposition 3(2) allow us to conclude $\mathsf{P}, \sigma' \vdash \sigma'(\mathsf{id}) \lhd \Gamma'(\mathsf{id})$.

Let instead id'' be such that $\mathsf{id}'' \neq \mathsf{id}$, $\Gamma(\mathsf{id}'') \neq \mathcal{U}df$ and $\sigma(\mathsf{id}'') = \iota$, that is, id'' is an alias for id. This implies $\mathsf{P} \vdash \mathsf{d} \sqsubseteq \Gamma(\mathsf{id}'')$. By definition, $\Gamma'(\mathsf{id}'') = \mathcal{R}(\mathsf{P}, \mathsf{c})$, if $\mathsf{P} \vdash \Gamma(\mathsf{id}'') \sqsubseteq \mathcal{R}(\mathsf{P}, \mathsf{c})$ and $\Gamma'(\mathsf{id}'') = \Gamma(\mathsf{id}'')$ otherwise. In both cases, $\mathsf{P} \vdash \mathcal{R}(\mathsf{P}, \mathsf{c}) \sqsubseteq \Gamma'(\mathsf{id}'')$, so, as before, we obtain $\mathsf{P}, \sigma' \vdash \sigma'(\mathsf{id}') \lhd \Gamma'(\mathsf{id}'')$. This concludes the proof of (6) and therefore also of (3).

The condition (1) is (5) for $\iota' = \iota$ and $\mathsf{c}' = \mathsf{c}$.

From $\mathcal{R}(\mathsf{P}, \mathsf{c}) = \mathcal{R}(\mathsf{P}, \mathsf{d})$, we get $\phi@_{\mathsf{P}}\mathsf{c} = \phi@_{\mathsf{P}}\mathsf{d}$, which ensures statement (2).

Last, (4) follows immediately from (δ).

For the rules of Figure 4 that generate a null pointer exception, the result follows easily by inductive hypothesis.

Consider now the other rules of Figure 4. We shall show that none of these rules is applicable:

—*CONDITIONAL*. The expression e' has the form $\mathsf{e}' =$ **if** e **then** e_1 **else** e_2, and e evaluates to a value v' which is neither `true` nor `false`. Since e' is well typed, e must have type **bool**. By application of the inductive hypothesis on the evaluation of e, we obtain that e must rewrite to a value that conforms to **bool**. The only values that conform to **bool** are `true` and `false`, which gives a contradiction.

—*FIELD ACCESS and FIELD ASSIGNMENT* as above, using the fact that the expression is well typed, that the state agrees with the program and environment, and applying Proposition 2(1).

—*VARIABLE ACCESS and VARIABLE ASSIGNMENT* as above, using the fact that the expression is well typed, and that the state agrees with the program and environment.

—*METHOD CALL* as above, using the fact that the expression is well typed, that the state agrees with the program and environment, and applying Proposition 2(2) and 2(4).

We now consider exception propagation from Figure 5. Let us first consider the propagation for the case of sequences of expressions. From rule (*seq*), we have

$$\frac{(\alpha)\ \ \mathsf{P}, \Gamma \vdash \mathsf{e}_0 : \mathsf{t}_0 \,[]\, \Gamma_0 \,[]\, \phi_0 \quad (\beta)\ \ \mathsf{P}, \Gamma_0 \vdash \mathsf{e}_1 : \mathsf{t} \,[]\, \Gamma' \,[]\, \phi_1}{\mathsf{P}, \Gamma \vdash \mathsf{e}_0; \mathsf{e}_1 : \mathsf{t} \,[]\, \Gamma' \,[]\, \phi},$$

where $\phi = \phi_0 \cup \phi_1$. From the operational semantics

$$\frac{(\gamma)\ \ \mathsf{e}_0, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{dv}, \sigma'}{\mathsf{e}_0; \mathsf{e}_1, \sigma \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{dv}, \sigma'}.$$

By the inductive hypothesis on (α), $\mathsf{P}, \Gamma \vdash \sigma \diamond$, and (γ):

$(1')$ $\mathsf{dv} = \texttt{nullPntrExc}$,

$(2')$ $\mathsf{P}, \phi_0 \vdash \sigma \lhd \sigma'$,

$(3')$ $\mathsf{P}, \phi_0 @_{\mathsf{P}} \Gamma_0 \vdash \sigma' \diamond$,

$(4')$ for all id, $\Gamma_0(\mathsf{id}) = \mathcal{U}df$ implies $\sigma(\mathsf{id}) = \sigma'(\mathsf{id})$.

$(1')$ implies (1). From $(2')$, $\phi_0 \subseteq \phi$, and Proposition 3(4), we derive (2). From $(3')$, for all id, P, $\sigma' \vdash \sigma'(\mathsf{id}) \lhd \phi_0 @_{\mathsf{P}} \Gamma_0(\mathsf{id})$, and from Proposition 3(6), we get $\mathsf{P} \vdash \phi_0 @_{\mathsf{P}} \Gamma_0(\mathsf{id}) \leq \phi @_{\mathsf{P}} \Gamma_0(\mathsf{id})$. Proposition 3(2) implies: for all id, P, $\sigma' \vdash \sigma'(\mathsf{id}) \lhd \phi @_{\mathsf{P}} \Gamma_0(\mathsf{id})$. Moreover, Proposition 1(2) and (β) implies that for all id, $\phi_1 @_{\mathsf{P}} \Gamma_0(\mathsf{id}) = \phi_1 @_{\mathsf{P}} \Gamma'(\mathsf{id})$, so also $(\phi_0 \cup \phi_1) @_{\mathsf{P}} \Gamma_0(\mathsf{id}) = (\phi_0 \cup \phi_1) @_{\mathsf{P}} \Gamma'(\mathsf{id})$ by Proposition 3(6). Therefore, for all id, P, $\sigma' \vdash \sigma'(\mathsf{id}) \lhd \phi @_{\mathsf{P}} \Gamma'(\mathsf{id})$, that is (3). Finally, from $(4')$ and Proposition 1(1), we derive (4).

All the other cases are proven in a similar way. We outline the proof for the last propagation rule that deals with the fact that the exception may be generated during the evaluation of the body of the method. For this case, we can proceed with the same proof as for the case of correct method call, till statement (9) (included), just replacing

$$(\zeta')\ \ \mathsf{e}'', \sigma'' \overset{\mathsf{p}}{\rightsquigarrow} \mathsf{dv}, \sigma'''$$

for (ζ). We then add:

Applying the inductive hypothesis to (6), (9), and (ζ'), we derive that

$(1''')$ $\mathsf{dv} = \texttt{nullPntrExc}$,

$(2''')$ $\mathsf{P}, \phi''' \vdash \sigma'' \lhd \sigma'''$,

$(3''')$ $\mathsf{P}, \phi''' @_{\mathsf{P}} \Gamma''' \vdash \sigma''' \diamond$,

$(4''')$ for all id, $\Gamma'''(\mathsf{id}) = \mathcal{U}df$ implies $\sigma''(\mathsf{id}) = \sigma'''(\mathsf{id})$.

The proof of (1) derives directly from (1‴), whereas the proofs of (2) and (4) are as for the case of correct method call (noting that we have almost the same inductive hypotheses).

To complete the proof of this case we have to show (3), that is:

$$\mathsf{P}, \phi@_{\mathsf{P}}\Gamma' \vdash \sigma' \diamond.$$

As for the proof of correct method call we can show that for all id such that $\Gamma'(\mathsf{id}) \neq \mathcal{U}df$, $\mathsf{P}, \sigma' \vdash \sigma'(\mathsf{id}) \lhd \Gamma'(\mathsf{id})$. From Proposition 3(6), $\mathsf{P} \vdash \Gamma'(\mathsf{id}) \leq \phi@_{\mathsf{P}}\Gamma'(\mathsf{id})$. So from Proposition 3(2), $\mathsf{P}, \sigma' \vdash \sigma'(\mathsf{id}) \lhd \phi@_{\mathsf{P}}\Gamma'(\mathsf{id})$. This concludes the proof. □

Remark 3. The weaker guarantee of well formedness for the resulting store σ' in the second case of Lemma 1 is due to the fact that the interruption of execution of e might prevent setting the type of `this` and parameters to the types specified in Γ'. For instance, for program P_0, state classes d', d'', which are not subclasses of each other, and d their root superclass, σ_0, Γ_0 and e_0 with $\sigma_0(\texttt{this}) = [[\cdots]]^{\mathsf{d}'}$ and $\Gamma_0(\texttt{this}) = \mathsf{d}'$, and $\mathsf{e}_0 = \mathsf{null.f}; \texttt{this} \Downarrow \mathsf{d}''$, typing produces:

$$\mathsf{P}_0, \Gamma_0 \vdash \mathsf{e}_0 : \mathsf{d}'' \,[]\, \Gamma_0[\texttt{this} \mapsto \mathsf{d}''] \,[]\, \{\mathsf{d}\},$$

whereas execution produces:

$$\mathsf{e}_0, \sigma_0 \overset{\mathsf{p}_0}{\rightsquigarrow} \texttt{nullPntrExc}, \sigma_0.$$

In σ_0, the receiver `this` is bound to an object of class d'. So, $\mathsf{P}_0, \Gamma_0[\texttt{this} \to \mathsf{d}''] \not\vdash \sigma_0 \diamond$. However, $\{\mathsf{d}\}@_{\mathsf{P}_0}(\Gamma_0(\texttt{this})) = \mathsf{d}$, and $\mathsf{P}_0 \vdash \mathsf{d}' \sqsubseteq \mathsf{d}$. Thus, $\mathsf{P}_0, \{\mathsf{d}\}@_{\mathsf{P}}\Gamma_0 \vdash \sigma_0 \diamond$ holds.

ACKNOWLEDGMENTS

$\mathcal{F}ickle_{II}$ has benefited from constructive criticism on $\mathcal{F}ickle$–99 from Walt Hill, Viviana Bono, Luca Cardelli, Andrew Kennedy, Giorgio Ghelli, and anonymous POPL'00 reviewers. Christopher Anderson, Lorenzo Bettini, Ross Jarman, and the anonymous FOOL'01, ECOOP'01 referees gave useful feedback on $\mathcal{F}ickle$. We are particularly grateful to Davide Ancona and the TOPLAS referees for careful reading and detailed suggestions, which improved greatly the submitted version.

REFERENCES

ABADI, M. AND CARDELLI, L. 1996. *A Theory of Objects*. Springer, Berlin.

AGESEN, O., BAK, L., CHAMBERS, C., CHANG, B., HÖLZLE, U., MALONEY, J., SMITH, R., AND UNGAR, D. 1992. The SELF Programmers's Reference Manual, version 2.0. Tech. rep., SUN Microsystems.

AGESEN, O., PALSBERG, J., AND SCHWARTZBACH, M. I. 1995. Type inference of self: Analysis of objects with dynamic and multiple inheritance. *Soft.-Pract. Exper. 25*, 9, 975–995.

ANCONA, D., ANDERSON, C., DAMIANI, F., DROSSOPOULOU, S., GIANNINI, P., AND ZUCCA, E. 2001. An effective translation of Fickle into Java. In *Proceedings of the ICTCS'01*. Lecture Notes in Computer Science, vol. 2002. Springer, Berlin, 215–234.

ANCONA, D., ANDERSON, C., DAMIANI, F., DROSSOPOULOU, S., GIANNINI, P., AND ZUCCA, E. 2002. Translating $\mathcal{F}ickle_{\mathrm{II}}$ into Java. In preparation.

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

BERTINO, E. AND GUERRINI, G. 1995. Objects with multiple most specific classes. In *Proceedings of ECOOP'95*. Lecture Notes in Computer Science, vol. 952. Springer, Berlin, 102–126.

BONO, V., BUGLIESI, M., DEZANI-CIANCAGLINI, M., AND LIQUORI, L. 1999. A subtyping for extensible, incomplete objects. *Funda. Inf. 38*, 4, 325–364.

CANNING, P., COOK, W., HILL, W., AND OLTHOFF, W. 1989. Interfaces for strongly typed object oriented languages. In *Proceedings of OOPSLA'89*. ACM, New York, 457–467.

CARDELLI, L., DONAHUE, J., GLASSMAN, L., JORDAN, M., KALSOW, B., AND NELSON, G. 1989. Modula-3 Report (revised). Tech. rep., DEC Systems Research Center.

CHAMBERS, C. 1993. Predicate classes. In *Proceedings of ECOOP'93*. Lecture Notes in Computer Science, vol. 707. Springer, Berlin, 268–296.

CHAMBERS, C. AND LEAVENS, G. 1995. Type checking modules for multimethods. *ACM Trans. Prog. Lang. Syst. 17*, 6, 805–843.

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

DELINE, R. AND FÄHNDRICH, M. 2001. Enforcing high-level protocols in low-level software. In *Proceedings of PLDI'01*. ACM, New York, 59–69.

DI GIANANTONIO, P., HONSELL, F., AND LIQUORI, L. 1998. A Lambda calculus of objects with self-inflicted extension. In *Proceedings of OOPSLA'98*. ACM Press, New York, 166–178.

DROSSOPOULOU, S. 2002. Three Case Studies in $\mathcal{F}ickle_{\mathrm{II}}$. Tech. rep., Imperial College. Available from http://www.di.unito.it/~damiani/papers/dor.html.

DROSSOPOULOU, S., DAMIANI, F., DEZANI-CIANCAGLINI, M., AND GIANNINI, P. 2001. Fickle: Dynamic object reclassification. In *Proceedings of ECOOP'01*. Lecture Notes in Computer Science, vol. 2072. Springer, Berlin, 130–149. A shorter version is available in: Electronic proceedings of FOOL8 (http://www.cs.williams.edu/~kim/FOOL/).

DROSSOPOULOU, S., DEZANI-CIANCAGLINI, M., DAMIANI, F., AND GIANNINI, P. 1999a. Objects dynamically changing class. Tech. rep., Imperial College. Available from http://www.di.unito.it/~dezani/odcc.html.

DROSSOPOULOU, S., EISENBACH, S., AND KHURSHID, S. 1999b. Is the Java type system sound? *Theory Pract. Obj. Syst. 5*, 1, 3–24.

ERNST, M. D., KAPLAN, C., AND CHAMBERS, C. 1998. Predicate dispatching: A unified theory of dispatch. In *Proceedings of ECOOP'98*. Lecture Notes in Computer Science, vol. 1445. Springer, Berlin, 186–211.

FIDGETT, D. 2002. Extending $\mathcal{F}ickle_{\mathrm{II}}$, Imperial College, final year thesis—to appear.

FISHER, K., HONSELL, F., AND MITCHELL, J. C. 1994. A Lambda calculus of objects and method specialization. *Nord. J. Comput. 1*, 1, 3–37.

FISHER, K. AND MITCHELL, J. C. 1995. A Delegation-based Object Calculus with Subtyping. In *Proceedings of FCT'95*. Lecture Notes in Computer Science, vol. 965. Springer, Berlin, 42–61.

FREEMAN, T. AND PFENNING, F. 1991. Refinement types for ML. In *Proceedings of SIGPLAN '91*. ACM, New York, 268–277.

GHELLI, G. AND PALMERINI, D. 1999. Foundations of extended objets with roles (*extended abstract*). In Electronic Proceedings of *FOOL6*. Available from http://www.cs.williams.edu/~kim/FOOL/FOOL6.html.

GOLDBERG, A. AND ROBSON, D. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, Mass.

HÜRSCH, W. 1994. Should superclasses be abstract? In *Proceedings of ECOOP'94*. Lecture Notes in Computer Science, vol. 821. Springer, Berlin, 12–31.

JARMAN, R. 2000. Fickle: A study in objects, Imperial College, final year thesis.

JARMAN, R. AND DROSSOPOULOU, S. 2000. Examples in Fickle. Available from http://www.di.unito.it/~damiani/papers/dor.html.

KRISTENSEN, B., MADSEN, O., MOLLER-PEDERSON, B., AND NYGAARD, K. 1987. The BETA programming language. In *Research Directions in Object-Oriented Programming*. MIT Press, Boston, Mass., 7–48.

LUCASSEN, M. AND GIFFORD, D. K. 1988. Polymorphic effect systems. In *Proceedings of POPL'88*. ACM, New York, 47–57.

RAVARA, A. AND VASCONCELOS, V. T. 2000. Typing non-uniform concurrent objects. In *Proceedings of CONCUR'00*. Lecture Notes in Computer Science, vol. 1877. Springer, Berlin, 474–488.

RÉMY, D. 1995. From classes to objects via subtyping. In *Proceedings of ESOP'98*. Lecture Notes in Computer Science, vol. 1381. Springer, Berlin, 200–220.

RIECKE, J. C. AND STONE, C. A. 1998. Privacy via subsumption. In Electronic Proceedings of *FOOL5*. Available from http://www.cs.williams.edu/~kim/FOOL/FOOL5.html.

SCHEER, T. AND PRINGLE, S. 1998. Ten practical limitations of object orientation. OOPSLA Poster Session, Available from http://www.acm.org/sigplan/oopsla/oopsla98/fp/posters/10.htm.

SERRANO, M. 1999. Wide classes. In *Proceedings of ECOOP'99*. Lecture Notes in Computer Science, vol. 1628. Springer, Berlin, 391–415.

SHUTTLEWOOD, A. 2002. Implementing $\mathcal{F}ickle_{II}$ on the JVM, Imperial College, final year thesis—to appear.

STROM, R. E. AND YELLIN, D. M. 1993. Extending typestate checking using conditional liveness analysis. *IEEE Trans. Soft. Eng. 19*, 5, 478–485.

TAILVASAARI, A. 1993. Object oriented programming with modes. *J. Obj. Orient. Prog. 6*, 3, 27–32.

TALPIN, J.-P. AND JOUVELOT, P. 1992. Polymorphic type, region and effect inference. *J. Funct. Prog. 2*, 3, 245–271.

WALKER, D., CRARY, K., AND MORRISETT, G. 2001. Typed memory management via static capabilities. *ACM Trans. Prog. Lang. Syst. 22*, 4, 701–771.

Received May 2001; revised December 2001; accepted March 2002