View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Elsevier - Publisher Connector

Theoretical Computer Science 433 (2012) 60–97

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Refactoring and representation independence for class hierarchies

David A. Naumann [a,*], Augusto Sampaio [b], Leila Silva [c]

[a] *Stevens Institute of Technology, 07030 Hoboken, NJ, USA*
[b] *Universidade Federal de Pernambuco, Brazil*
[c] *Universidade Federal de Sergipe, Brazil*

ARTICLE INFO

Article history:
Received 12 April 2011
Received in revised form 4 January 2012
Accepted 6 February 2012
Communicated by J. Davies

Keywords:
Refactoring
Program transformation
Class inheritance
Representation independence
Semantics
Verification

ABSTRACT

Refactoring transformations are important for productivity and quality in software evolution. Modular reasoning about semantics preserving transformations is difficult even in typed class-based languages because transformations can change the internal representations for multiple interdependent classes and because encapsulation can be violated by pointers to mutable objects. In this paper, an existing theory of representation independence for a single class, based on a simple notion of ownership confinement, is generalized to a hierarchy of classes and used to prove refactoring rules that embody transformations of complete class trees. This allows us to formalize refactorings that inherently involve class inheritance, such as Pull Up or Push Down Field; moreover, this makes it possible to generalize refactorings previously restricted to change of data representation of private attributes (like Extract Class and Encapsulate Field) to address data refinement of protected attributes, dealing with the impact that the corresponding transformations may cause in the subclasses. The utility of the proposed rules is shown in a relatively extensive case study. Shortcomings of the theory are described as a challenge to other approaches to heap encapsulation and relational reasoning for classes.
© 2012 Elsevier B.V. All rights reserved.

1. Introduction

According to Wikipedia, "code refactoring is the process of changing a computer program's source code without modifying its external functional behavior in order to improve some of the nonfunctional attributes of the software". Refactoring is an integral part of formal and informal software development processes in many organizations and tool support is provided by popular development environments such as Eclipse. However, the intent to preserve functional behavior is not always achieved [32], nor is it easily assessed. Useful refactorings may change many classes (e.g., replacing direct access to a protected field by get/set methods) and/or runtime data structures (e.g., introducing or eliminating indirection via wrapper objects). System testing can confirm whether a refactored program passes the tests of the original, and at least some unit tests will still be applicable to the refactored version. However, tests are rarely sufficient to determine whether all functional behavior has been preserved.

This paper contributes to an ongoing project in which semantics preserving refactoring transformations are formally defined and validated. In previous work [8] we present algebraic laws for a Java-like language including recursive classes and features such as inheritance. The laws are proved sound (using a predicate transformer semantics [12]) and relatively complete, through a normal form reduction process, and used to prove sample refactorings. Many interesting refactorings involve a change of data representation, for which a theory of data refinement was developed [13]. These works rely on the drastic restriction that programs do not share mutable objects, so that "copy semantics" can be used.

* Corresponding author. Tel.: +1 201 216 5608; fax: +1 201 216 8249.
E-mail address: Naumann@cs.stevens.edu (D.A. Naumann).

0304-3975/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2012.02.009

We have investigated the impact of reference semantics on the laws [37], using the rCOS [20] refinement theory, and identified a number of what we call *laws of classes* which remain sound in reference semantics, even obviously sound because they are fine grained changes of static program structure. However, rCOS does not yet provide data refinement laws with reference semantics.

The key difficulty for data refinement is the potential for references to violate encapsulation boundaries and create interdependences that are not evident in the program syntax. There have been many works on ownership and other notions of alias control intended to tame reasoning about references (e.g., [14,4,26]). In previous work [5], we develop a notion of simulation for proving equivalence between two implementations of a class, relying on a somewhat restrictive form of ownership confinement. In later work [6] we develop a more flexible version that caters for transferable ownership. In both of these works we follow Reynolds [29] in terming the main result an *abstraction theorem*. Representation independence for languages with references is under active investigation but recent advances [36,21,9,2,39,17,16] do not directly address class based languages. The technique of Koutavas and Wand [21] was adapted to a class based language [22] and used to verify the examples in [5] but those are specific programs involving a single class rather than general refactoring laws.

The works in the preceding paragraph focus on proving *contextual equivalence* of program fragments, in other words, equivalence in all contexts. As pointed out in several works (e.g., [8]), many useful laws hold only in restricted contexts,[1] such as, for instance, those where method contracts are satisfied, downcasts are absent, or some alias confinement property holds. In particular, many of the refactorings in Fowler [18] impose restrictions on the whole program, though it is not always obvious from their informal descriptions (see [28,30]). For that reason, [8] considers whole program equivalence, via schematic laws in which contextual constraints can be expressed syntactically. We do the same here.

In this paper we present an abstraction theorem for using simulation to prove equivalence of two versions of an entire hierarchy of classes. The theorem is used to prove a data refinement law, which is itself used in the proofs of some general refactoring rules. Those rules are in turn used to carry out a detailed case study with several steps. One step of the case study embodies a rather specific transformation and makes direct use of the data refinement law. Therefore, the law is useful both to prove more general transformations (contributing to sound refactoring catalogs) as well as to justify specific changes of data representation in the application being refactored.

A short summary of the ideas that have motivated the contributions of this paper can be found in our workshop paper [35]. There we illustrate the refactorings via a simplified version of the case study developed here, and briefly sketch the notion of confinement and results on representation independence.

The range of refactorings we can address are all those that involve some change of data representation. Some of these refactorings (like Pull Up or Push down Field) make explicit the impact of data refinement in the inheritance tree. Other refactorings, like Extract Class or Encapsulate Field, as originally presented [18], affect only private fields, so that the change of data representation is restricted to a single class. Nevertheless, it is useful to generalize these refactorings to encompass protected attributes as well. Although using private fields is among the good practices of object-oriented programming, there are several applications, particularly frameworks intended to be extended, whose design are heavily based on the use of protected attributes; a well-known example is Java's Swing API. Furthermore, efficiency might also dictate the use of protected attributes; the direct access of these attributes by subclasses is convenient, for instance, in the case of the implementation of some collections. Therefore, our work provides infra-structure both to propose new refactorings and to generalize well-known refactorings to allow their application in such contexts. Of course, our data refinement law can be used to justify change of data representation of an individual class, as a special case; even in this simpler case, our results extend previous work [10], which is restricted to copy semantics.

Remarkably, we achieve these results via a relatively simple generalization of results in [5]. There, a notion of ownership is defined in terms of class types declared in the program, taking advantage of nominal typing as in Java and avoiding the need for special type annotations [14,4,1,26] or specifications [6,7,34]. We adapt that approach to fit protected visibility and we improve it so that instances of library classes can be owned, without disallowing other uses of those classes. We expect that our adaptation can be enforced by a straightforward static analysis, like that worked out in [5], but that is beyond the scope of this paper. What matters here is that the approach is well suited to use with refactorings. In most cases, we found that confinement follows from the conditions already required for soundness of the various refactorings we considered. Only the Extract Class refactoring and the data refinement law have explicit conditions concerning confinement. We also follow [5] in using a denotational semantics. Compositional semantics facilitates proof of simulations and program equivalences by structural induction on program texts. Our semantics, and hence the notion of program equivalence, is idealized in the sense that it models an unbounded heap and unbounded runtime stack.

In the next section we introduce the programming language we adopt. It is an idealized fragment of Java, without threads, generics, reflection, or nested classes. Section 3 provides the denotational semantics and uses it to define program equivalence based on observable behavior. In Section 4 we formalize a notion of ownership confinement, and in Section 5 we prove an abstraction theorem for class hierarchies. In Section 6 we prove a data refinement law that directly embodies the abstraction theorem, and we prove some refactorings whose change of data representation might impact an arbitrary number of classes in an inheritance tree. Some of those refactorings are proved using the data refinement law and others are proved directly from the semantics. A case study is developed in Section 7 to illustrate the application of the refactoring

[1] This is also remarked in [22], where a specific restriction on downcasts is investigated.

cd	$::=$	**class** K **ext** L { $\overline{vis\, f : T}\quad \overline{mt}$ }	class declaration
T	$::=$	K | **bool** | **int**	data type
vis	$::=$	**pri** | **prot** | **pub**	visibility modifier for fields
$msig$	$::=$	$m(\overline{x} : \overline{T}) : U$	method signature
mt	$::=$	**meth** $msig$ { c }	public method declaration
c	$::=$	$x := e$	assign to variable
	|	$x.f := y$	field update
	|	**var** $x : T$ **in** c	local variable block
	|	$c_1;\ c_2$ | **if** x **then** c_1 **else** c_2	sequence, conditional
e	$::=$	x | **null** | **true** | 0 | 1 | 2 . . .	variable, constant
	|	$x.f$	field access
	|	$x = y$ | . . .	equality test and other prim. ops.
	|	x **is** K	type test
	|	$(K)\, x$	type cast
	|	**new** K	object construction
	|	$x.m(\overline{y})$	method call
	|	**let** x **be** e_1 **in** e_2	sequenced local binding

Fig. 1. Grammar of classes and methods. Bold keywords and punctuation marks including "{" and "}" are terminal symbols.

rules and the data refinement law. In the final section we summarize our contributions, further discuss related work, and list some challenges for future research.

The sections proceed in order of logical dependency. However, for illustrative examples, the reader is encouraged to proceed directly to Section 7. In particular, in Section 7.3 we point out the need for confinement, in one of the refactoring steps. In Section 7.4 we point out the need for confinement in a direct use of the data refinement law.

2. Programming language syntax

This section formalizes the language used in this paper. The syntax and semantics are adapted from [25] (but without interface types or first-class exceptions), which is a streamlined version of the language and denotational semantics of [5].

2.1. Grammar

The grammar is based on some given sets of names, using the following nomenclature for typical elements:

$K, L, M, N \in ClassName$	names of declared classes
S, T, U, V	data types (including class names)
b, c, d	commands
x, y, z, f	variable names (parameters, fields, and locals)
m, n	method names.

The syntax for classes and methods is in Fig. 1. Apart from the above conventions, we use $\overline{x}$ to indicate a list of variables. But the identifier $\overline{x}$ has nothing to do with the identifier x. We indicate lists of types, expressions, etc. analogously.

We use the terms "field" and "attribute" interchangeably to refer to the variables of a class. There are two distinguished variable names, **self** and **res**, which have special uses in the semantics: the receiver object and expression result, respectively. The return value of a method is given by the final value of **res**, as if every method body had at the end the statement "`return` **res**". There are three distinguished class names, **Object**, **Void**, and **None**, as well as one distinguished method name **ctor**. The latter is used as the constructor method. Whereas **Object** serves as usual as the top of the class hierarchy, **Void** plays a technical role that streamlines the formalization of program equivalence (Section 3.5). In particular, **Void** is the type of **self** in the main program. Class **Void** has no fields, no methods other than **ctor**, and no subclasses. Class **None** plays a technical role in the formalization of confinement; it never occurs in programs.

Definition 1 (*Complete Program, Class Table*)**.** A complete program is a sequence cds of class declarations together with a command c, written $cds \bullet c$.

When convenient, we abuse notations and treat a list cds of class declarations as a function, called a *class table*, that maps each declared class name K to its declaration $cds(K)$.

As usual we model input and output by the global variables of the main program c in $cds \bullet c$. To avoid clutter in the refactoring laws we refrain from formalizing syntax for the declaration of these global variables, but implicitly they do have type declarations which are used when we define well-formed programs later.

1. No parameter or local variable declaration uses the name **self** or **res**.
2. The subtype relation $\leq$ is acyclic.
3. Field names are not shadowed, that is, if $f : T$ is in $dfields(K)$ then f is not in $fields(super(K))$.
4. Method types are invariant, except for constructors. That is, if $mtype(U, m)$ is defined and $T \leq U$ and $m \neq$ **ctor** then $mtype(T, m) = mtype(U, m)$.
5. For any K, every method declaration $m(\overline{x} : \overline{T}) : U \{c\}$ in $cds(K)$ is typable in the sense that $\Gamma \vdash c$ where $\Gamma = [\mathbf{self} : K, \mathbf{res} : U, \overline{x} : \overline{T}]$.
6. In every class K, $K \neq$ **None**, there is one declaration for a method named **ctor** with return type K and body of the form c; **res** := **self**.
7. Every occurrence of an expression **new** K (in any class) has the form **let** x **be new** K **in** x.**ctor**$(\overline{y})$ and moreover there are no other invocations of **ctor**.
8. Class **Void** has superclass **Object**, no subclasses, no fields, and no methods other than the obligatory **ctor**.
9. Class **None** is declared as **class None ext Object** { }, and the class name **None** does not occur anywhere in any other class; nor in the main program.
10. Every superclass must be declared and every class is declared only once.

Fig. 2. A well-formed class table. Rules that define $\Gamma \vdash c$ appear in Fig. 3.

The syntax is in "A-normal form" [33], i.e., subexpressions in various constructs are restricted to be variables. The only command form that involves general expressions e is assignment $x := e$ to a variable.[2] We sometimes write "$le := e$" to stand for an assignment of either form $x := e$ or $x.f := y$.

For a method m with return type **Void**, we omit ": **Void**" in the declaration and treat $x.m(\overline{y})$ as a command, which desugars to **var** y : **Void in** $y := x.m(\overline{y})$ using dummy y. Typically, the body of such a method has no assignment to the distinguished variable **res** and thus the return value is the initial value of **res** (i.e., the default, **null**). We consider all methods to have public visibility.

We need to consider constructors because our main results rely on simulation relations, which are usually established by some initialization code. On the other hand, it simplifies the semantics to treat constructors like other methods as much as possible. We confine attention to programs in which each class declares one method named **ctor** and **new** K only occurs in the form (**new** K).**ctor**$(\overline{y})$ —which is sugar for **let** x **be new** K **in** x.**ctor**$(\overline{y})$. Moreover, the body of **ctor** has the form c; **res** := **self**. Thus the expression (**new** K).**ctor**$(\overline{y})$ evaluates to a reference to the freshly allocated and initialized object.

2.2. Program typing

We begin by defining some functions that extract parts of the syntax. Let

$$cds(K) \;=\; \mathbf{class}\ K\ \mathbf{ext}\ L\,\{\overline{vis\,f : T};\ \overline{mt}\}.$$

Define $super(K) = L$. Let mt be in the list $\overline{mt}$ of method declarations, so mt has the form

$$\mathbf{meth}\ m(\overline{x} : \overline{T}) : U\ \{c\}.$$

We record the type and parameter names by defining $mtype(K, m) = \overline{x} : \overline{T} \to U$. If m is inherited in K from L (i.e., is defined in L but not declared in K) then $mtype(K, m)$ is defined to be $mtype(L, m)$. Thus $mtype(K, m)$ is defined iff m is declared or inherited in K. Let $Meths(K)$ be the set of m such that $mtype(K, m)$ is defined.

For declared fields we define $dfields(K) = \overline{f : T}$. Let $visib(K, f)$ be the visibility marker of field f. To include inherited fields we define, recursively,

$$fields(K) = fields(L) \cup dfields(K).$$

In a well-formed class table this union will be disjoint.

The *subtype relation* $\leq$ is the least reflexive and transitive relation such that $super(K) = L$ implies $K \leq L$. A consequence is that, for primitive types, $T \leq U$ holds just if T is U. Moreover, in a well-formed class table, $K \leq$ **Object** for all $K \in ClassName$. We write $\overline{V} \leq \overline{T}$ to express that $\overline{V}$ and $\overline{T}$ are lists of the same length and corresponding elements are related by $\leq$.

A class table cds is *well-formed* provided it satisfies the conditions in Fig. 2, which refers to the typing rules in Fig. 3. The typing judgment has the form $\Gamma \vdash c$ where Γ is a *typing context*: a finite map from variable names to data types. The functions $mtype$ and $dfields$ depend on a class table cds, hence so does the typing relation $\vdash$, but this dependence is suppressed in the notation.

The typing rules could be streamlined if in addition we added a subsumption rule. As a technical convenience, we choose instead to use only syntax-directed rules. This lets us define the semantics by induction on typing derivations, which are essentially unique. The rules disallow variable re-declaration (shadowing): in the rule for **var**, the context $[\Gamma, x : T]$ would not be well formed if x was in the domain of Γ.

Recall that a complete program $cds \bullet c$ consists of a well-formed class table and a command c. There is an implicit context Γ such that $\Gamma \vdash c$, which we do not make explicit in the syntax (in order to streamline the laws, as in [8]). We do not model

2 One would expect the grammar for expressions to allow equality tests of the form $e = e$, method calls of the form $e.m(e)$, and so forth. These can all be desugared to our syntax by simple translations using let-expressions.

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash \textbf{true} : \textbf{bool}} \qquad \frac{}{\Gamma \vdash \textbf{null} : K}$$

$$\frac{\begin{array}{c}\Gamma \vdash x : K \quad K \le L \quad (f : T) \in \mathit{dfields}(L) \\ (\mathit{visib}(L, f) = \textbf{pri} \Rightarrow \Gamma(\textbf{self}) = L) \quad (\mathit{visib}(L, f) = \textbf{prot} \Rightarrow \Gamma(\textbf{self}) \le L)\end{array}}{\Gamma \vdash x.f : T}$$

$$\frac{\Gamma \vdash x : K \quad \mathit{mtype}(K, m) = \overline{z} : \overline{T} \to U \quad \Gamma \vdash \overline{y} : \overline{V} \quad \overline{V} \le \overline{T}}{\Gamma \vdash x.m(\overline{y}) : U}$$

$$\frac{\Gamma \vdash x : L}{\Gamma \vdash (K)\, x : K} \qquad \frac{\Gamma \vdash x : L}{\Gamma \vdash x\ \textbf{is}\ K : \textbf{bool}} \qquad \frac{}{\Gamma \vdash \textbf{new}\ K : K}$$

$$\frac{\Gamma \vdash e : T \quad [\Gamma, x : T] \vdash e1 : U}{\Gamma \vdash \textbf{let}\ x\ \textbf{be}\ e\ \textbf{in}\ e1 : U} \qquad \frac{[\Gamma, x : T] \vdash c}{\Gamma \vdash \textbf{var}\ x : T\ \textbf{in}\ c}$$

$$\frac{\Gamma \vdash e : T \quad T \le \Gamma(x) \quad x \ne \textbf{self}}{\Gamma \vdash x := e}$$

$$\frac{\begin{array}{c}\Gamma \vdash x : K \quad K \le L \quad (f : T) \in \mathit{dfields}(L) \quad \Gamma(y) \le T \\ (\mathit{visib}(L, f) = \textbf{pri} \Rightarrow \Gamma(\textbf{self}) = L) \quad (\mathit{visib}(L, f) = \textbf{prot} \Rightarrow \Gamma(\textbf{self}) \le L)\end{array}}{\Gamma \vdash x.f := y}$$

Fig. 3. Selected typing rules for expressions and commands, for a given class table.

static methods or fields; indeed, we need **self** $\in dom(\Gamma)$ to formalize confinement of the main program. Hence we assume that **self** has type **Void** in a main program.

Method m is *inherited in* K *from* L if $K \le L$, there is a declaration for m in L, and there is no declaration for m in any M such that $K \le M < L$. To make the class table explicit, we also say m is inherited from L in $cds(K)$.

Because the language has single inheritance, the subtyping relation $\le$ is a tree: if $L \le M$ and $L \le K$ then $M \le K$ or $K \le M$. If $mtype(K, m)$ is defined for some K then it is defined for all subclasses of K and there is a unique ancestor class declaring m that is least with respect to $\le$.

For any m and K such that $mtype(K, m)$ is defined, the *method depth of* K *for* m *in cds* is defined by $depth(K, m) = 1 + depth(super(K), m)$ if $mtype(super(K), m)$ is defined; otherwise, $depth(K, m) = 0$. An immediate consequence is that if $mtype(K, m)$ is defined and $depth(K, m) = 0$ then $cds(K)$ has a declaration for m.

Concerning visibility for protected fields, our access rule is a simplification of, for instance, a language like Java, which includes packages and, therefore, a more elaborate notion of scope. Our rule for type cast is slightly simpler and more general than in Java: besides up and down casts, it allows casts that are certain to fail; similarly for type test.

3. Program semantics

In this section we present a denotational semantics for our programming language. We start by defining the relevant semantic domains, followed by the semantics of expressions and commands. We then define partial orderings on the domains. Then we give the semantics of method declaration and class table. Finally, we define program equivalence.

3.1. Semantic domains

We assume a given infinite set *Ref* of *references* —abstract addresses. A *ref context* is a finite partial function r that maps references to class names. The idea is that if $o \in dom(r)$ then o is allocated and moreover o points to an object of type $r(o)$. We define the set

$$RefCtx = Ref \rightharpoonup (ClassName - \{\textbf{None}\})$$

where $\rightharpoonup$ denotes finite partial functions. The effect of disallowing class **None** in the range is that there are never instances of this class. This is feasible owing to condition 9 in Fig. 2. The need to explicitly disallow instances of **None**, even though the name is not allowed to appear in code, is a price we pay for using a denotational semantics.

Let *null* be some fixed value with $null \notin Ref$. For data types T the domain of values is defined by cases on T, for any ref context r.

$$Val(\textbf{bool}, r) = \{true, false\}$$
$$Val(\textbf{int}, r) = \mathbb{Z}$$
$$Val(K, r) = \{null\} \cup refs(K, r)$$
$$refs(K, r) = \{o \in dom(r) \mid r(o) \le K\}.$$

Note that for any ref context r we have

$$K \le L \text{ implies } Val(K, r) \subseteq Val(L, r). \quad (1)$$

Sometimes we abuse notation and write $Val(\overline{T}, r)$ for lists of values, of types $Val(T_1, r), Val(T_2, r), \ldots, Val(T_n, r)$.

The next definitions involve dependent function spaces or dependent pairs.[3] The first use of dependent types is in the definition

$$PreStore(\Gamma, r) = (x : dom(\Gamma)) \to Val(\Gamma(x), r).$$

What this means is that $PreStore(\Gamma, r)$ is a set of functions; and for any s in $PreStore(\Gamma, r)$, the domain of s is $dom(\Gamma)$, and $s(x)$ is an element of $Val(\Gamma(x), r)$ for each $x \in dom(\Gamma)$.

The domain of *stores* imposes an additional invariant on the semantics, that **self** is not null: define $Store(\Gamma, r)$ by

$$s \in Store(\Gamma, r) \text{ iff } s \in PreStore(\Gamma, r) \text{ and } (\mathbf{self} \in dom(\Gamma) \Rightarrow s(\mathbf{self}) \neq null).$$

Next we build up to program states. Define $obcontext(K)$ to be the variable context obtained by removing visibility markers from $fields(K)$. Define

$$\begin{aligned} &FieldRcrd(K, r) = Store(obcontext(K), r) \\ &Heap(r) = (o : dom(r)) \to FieldRcrd(r(o), r) \\ &State(\Gamma) = (r : RefCtx) \times Heap(r) \times Store(\Gamma, r). \end{aligned}$$

So a *heap* h is a map sending each allocated reference o to a record, $h(o)$, of the object's current field values; and $h(o)(f)$ is the value of field f. A Γ*-state* has the form (r, h, s) with h a heap for r and s a store for Γ and r.

In Section 4 and later, we often decompose heaps into fragments. For that purpose, we say h is a *pre-heap* for r if h is like a heap except that its domain may be a subset of $dom(r)$; in particular, its object fields may contain values that are in $dom(r)$ but not in $dom(h)$. Formally:

$$PreHeap(r) = (o : dom(r)) \rightharpoonup FieldRcrd(r(o), r).$$

The most important domain is state transformers.[4]

$$STrans(\Gamma, \Gamma^*) = (\sigma : State(\Gamma)) \to \{\bot\} \cup \{\tau | \tau \in State(\Gamma^*) \wedge extState(\sigma, \tau)\}$$

Relation *extState* is defined to say that one state's ref context extends the other's[5]:

$$extState((r, h, s), (r^*, h^*, s^*)) \text{ iff } r \subseteq r^*.$$

This expresses that the type of a reference never changes. We do not model garbage collection; unreachable objects remain in the state.

The domain of state transformers subsumes meanings for methods, expressions and commands:

$$\begin{aligned} &SemExpr(\Gamma, T) = STrans(\Gamma, [\mathbf{res} : T]) \\ &SemCommand(\Gamma) = STrans(\Gamma, \Gamma) \\ &SemMeth(K, m) = STrans([\mathbf{self} : K, \overline{x} : \overline{T}], [\mathbf{res} : U]) \\ &\qquad \text{where } mtype(K, m) = \overline{x} : \overline{T} \to U. \end{aligned}$$

We sometimes use the term *state transformer type*, with notation $\Gamma \rightsquigarrow \Gamma^*$, in connection with elements of $STrans(\Gamma, \Gamma^*)$. Thus a command in context Γ denotes a state transformer of type $\Gamma \rightsquigarrow \Gamma$. An expression of type T in context Γ denotes a state transformer of type $\Gamma \rightsquigarrow [\mathbf{res} : T]$. And a method declaration with $mtype(K, m) = \overline{x} : \overline{T} \to U$ denotes a state transformer of type $[\mathbf{self} : K, \overline{x} : \overline{T}] \rightsquigarrow [\mathbf{res} : U]$.

A *method environment* is defined to be a table of meanings for all methods in all declared classes. The set, *Menv*, of method environments is defined by

$$Menv = (K : ClassName) \times (m : Meths(K)) \to SemMeth(K, m).$$

A class table *cds* denotes a method environment, $[\![cds]\!]$, defined later. The idea is that a method environment η is defined for pairs (K, m), where K is a class with method m, and moreover $\eta(K, m)$ is a state transformer suitable to be the meaning of a method of type $mtype(K, m)$. In case m is inherited in K from L, $\eta(K, m)$ will be the restriction of $\eta(L, m)$ to receive objects of type K.

---

[3] We write $(x : D) \to E$ for total functions where the type E of the range may depend on the value x of the argument, and similarly $(x : D) \times E$ for dependent pairs.

[4] We use the asterisk $(^*)$ merely to decorate identifiers, in order to save the dash $(')$ for pairs of related programs and states (starting in Section 5).

[5] Since r and r^* are partial functions which we treat as sets of pairs, $r \subseteq r^*$ says that r^* has at least the domain of r and they agree on their common domain.

3.2. Semantics of expressions and commands

A number of notations are needed. For typing contexts we write $x:T$ to express that x is mapped to T, and $[\Gamma, x:T]$ indicates the extension of Γ with a binding for x which must not occur in the domain of Γ. In the semantics we use a similar notation, e.g., $[s,\ x:v]$ extends store s to map x to v, where x is not in $dom(s)$. We write $[s \mid\ x:v]$ to update s in case x is in $dom(s)$. Note that extension of a map is distinguished from update by use of comma versus |. We abbreviate nested updates to object fields: $[h \mid\ o.f:v]$ is short for $[h \mid\ o:[h(o) \mid\ f:v]]$. Application binds most tightly, e.g., $[s,\ x:s_1(x)]$ extends s to map x to the value of $s_1(x)$. To remove an element from the domain of a function we use the minus sign, e.g., if s is a store then $s - x$ is the same store but with x removed from its domain.

In addition to working directly with states in the form (r, h, s), it is sometimes convenient to use a single identifier σ. For $\sigma = (r, h, s)$ we write $\sigma(x)$ for $s(x)$ and we write $\sigma(x.f)$ for $h(s(x))(f)$.

Let-expressions in the metalanguage are $\bot$-strict. That is, if α is $\bot$ then $\mathsf{let}\ x = \alpha\ \mathsf{in}\ \beta$ is $\bot$; otherwise the value is that of β under the binding of x as usual.

Semantics of expressions and commands. For a well-formed expression in context, i.e., a derivable judgment $\Gamma \vdash e:T$, the semantics $[\![\Gamma \vdash e:T]\!]$ is in

$$Menv \to SemExpr(\Gamma, T).$$

The semantics $[\![\Gamma \vdash e:T]\!]$ gets applied to a method environment η to yield a state transformer $[\![\Gamma \vdash e:T]\!](\eta)$ that in turn is applied to a state σ in $State(\Gamma)$. Finally, $[\![\Gamma \vdash e:T]\!](\eta)(\sigma)$ yields either $\bot$ or an element of $State([\mathbf{res}:T])$. The semantic definitions for expressions are in Fig. 4.

The semantics is given in terms of an arbitrary *allocator*, i.e., a total function $fresh : RefCtx \times ClassName \to Ref$ such that $fresh(r, K) \notin dom(r)$ for all ref contexts r. Parameter K is present so that later we can impose an additional assumption that streamlines the treatment of simulations.

Definition 2 (*Parametric Allocator*)**.** We say *fresh* is *parametric* if

$$fresh(r_1, K) = fresh(r_2, K)$$

for any r_1, r_2 such that $\{o \mid r_1(o) = K\} = \{o \mid r_2(o) = K\}$.

It is not difficult to define a set *Ref* and allocator *fresh* that is parametric, but allocators in practice are typically not parametric. See discussion near the beginning of Section 5.2. (The term ‘parametric’ is from [5] and is succinct but perhaps not informative.)

The initial state of a new object of class K is given by $defaultFieldRcrd(K)$ defined as follows. First, define $default(T)$ for each type T: $default(\mathbf{int}) = 0$, $default(\mathbf{bool}) = false$, and $default(K) = null$ for $K \in ClassName$. Then $defaultFieldRcrd(K)$ is just the mapping of $fields(K)$ to the default values for their types, so that $defaultFieldRcrd(K)$ is in $FieldRcrd(K, r)$ for all ref contexts r.

The last case in Fig. 4, for method call, deserves some explanation. Because $s(x)$ is the receiver of the call, $r(s(x))$ is the dynamic type of the receiver, so $\eta(r(s(x)), m)$ is the method meaning to be applied. It is applied to state (r, h, s_1) where the store s_1 provides the arguments. Owing to the well-formedness of the class table, the parameters $\bar{z}$ can be obtained from the static type of the receiver.

For any well-formed command in context $\Gamma \vdash c$ and method environment η, we define $[\![\Gamma \vdash c]\!](\eta)$ and observe that it is an element of $SemCommand(\Gamma)$. The semantics of commands is in Fig. 5.

3.3. Domain orderings

A class table denotes a method environment obtained as the least upper bound of a chain of approximations. For this purpose each semantic domain is given a partial ordering. The sets $Val(T, r)$, $Store(\Gamma, r)$, $Heap(r)$, and $State(\Gamma)$ are considered to be ordered by the discrete order, i.e., equality. For state transformers φ and ψ in $STrans(\Gamma_1, \Gamma_2)$ define $\varphi \le \psi$ iff for all states σ, if $\varphi(\sigma) \ne \bot$ then $\varphi(\sigma) = \psi(\sigma)$. Put differently, this is the pointwise ordering: $\varphi \le \psi$ iff $\varphi(\sigma) \le \psi(\sigma)$, for all σ. Finally, for method environments, define $\eta \le \eta'$ iff $\eta(K, m) \le \eta'(K, m)$ for all K, m.

Each of these sets is a complete partial order, that is, every ascending chain has a least upper bound. The everywhere-$\bot$ function is the least element in $STrans(\Gamma_1, \Gamma_2)$. The least method environment, η_0, is defined so that $\eta_0(K, m)$ is the everywhere-$\bot$ function for all K, m. The least upper bound of method environments has a simple characterization.

Lemma 3 (*Least Upper Bounds for Method Environments*)**.** *Suppose η is an ascending chain of method environments, i.e., $\eta_i \le \eta_{i+1}$, for all natural i. Then for any K, m, σ, there is some j such that*

$$lub(\eta)(K, m)(\sigma) = \eta_k(K, m)(\sigma) \quad \text{for all } k \ge j.$$

This follows from a similar property for ascending chains of state transformers of a fixed type.

$[\![\Gamma \vdash x : T]\!](\eta)(r, h, s) = (r, h, [\mathbf{res} : s(x)])$

$[\![\Gamma \vdash \mathbf{true} : \mathbf{bool}]\!](\eta)(r, h, s) = (r, h, [\mathbf{res} : \mathit{true}])$ and similarly for other literals

$[\![\Gamma \vdash x = y : \mathbf{bool}]\!](\eta)(r, h, s) =$
 let $v =$ (if $s(x) = s(y)$ then true else false) in $(r, h, [\mathbf{res} : v])$

$[\![\Gamma \vdash \mathbf{new}\, K : K]\!](\eta)(r, h, s) =$
 let $o = \mathit{fresh}(r, K)$ in let $r_0 = [r,\ o : K]$ in
  let $h_0 = [h,\ o : \mathit{defaultFieldRcrd}(K)]$ in $(r_0, h_0, [\mathbf{res} : o])$

$[\![\Gamma \vdash x.f : T]\!](\eta)(r, h, s) =$ if $s(x) \neq \mathit{null}$ then $(r, h, [\mathbf{res} : h(s(x))(f)])$ else $\perp$

$[\![\Gamma \vdash (K)\, x : K]\!](\eta)(r, h, s) =$
 if $s(x) = \mathit{null} \vee r(s(x)) \leq K$ then $(r, h, [\mathbf{res} : s(x)])$ else $\perp$

$[\![\Gamma \vdash x\ \mathbf{is}\ K : \mathbf{bool}]\!](\eta)(r, h, s) =$
 let $v =$ (if $s(x) \neq \mathit{null} \wedge r(s(x)) \leq K$ then true else false) in $(r, h, [\mathbf{res} : v])$

$[\![\Gamma \vdash \mathbf{let}\ x\ \mathbf{be}\ e\ \mathbf{in}\ e1 : U]\!](\eta)(r, h, s) =$
 let $(r_0, h_0, s_0) = [\![\Gamma \vdash e : T]\!](\eta)(r, h, s)$ in
 let $s_1 = [s,\ x : s_0(\mathbf{res})]$ in $[\![\Gamma \vdash e1 : U]\!](\eta)(r_0, h_0, s_1)$

$[\![\Gamma \vdash x.m(\overline{y}) : U]\!](\eta)(r, h, s) =$
 if $s(x) = \mathit{null}$ then $\perp$ else let $s_1 = [\mathbf{self} : s(x), \overline{z} : s(\overline{y})]$ in $\eta(r(s(x)), m)(r, h, s_1)$
 where $\overline{z} : \overline{T} \to U = \mathit{mtype}(K, m)$

Fig. 4. Semantics of expressions.

$[\![\Gamma \vdash x := e]\!](\eta)(r, h, s) =$
 let $(r_1, h_1, s_1) = [\![\Gamma \vdash e : T]\!](\eta)(r, h, s)$ in $(r_1, h_1, [s \mid\ x : s_1(\mathbf{res})])$

$[\![\Gamma \vdash x.f := y]\!](\eta)(r, h, s) =$
 if $s(x) = \mathit{null}$ then $\perp$ else let $o = s(x)$ in $(r, [h \mid\ o.f : s(y)], s)$

$[\![\Gamma \vdash \mathbf{if}\ x\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!](\eta)(r, h, s) =$
 if $s(x) = \mathit{true}$ then $[\![\Gamma \vdash c_1]\!](\eta)(r, h, s)$ else $[\![\Gamma \vdash c_2]\!](\eta)(r, h, s)$

$[\![\Gamma \vdash \mathbf{var}\ x : T\ \mathbf{in}\ c]\!](\eta)(r, h, s) =$
 let $(r_1, h_1, s_1) = [\![\Gamma, x : T \vdash c]\!](\eta)(r, h, [s,\ x : \mathit{default}\, T])$ in $(r_1, h_1, s_1 - x)$

$[\![\Gamma \vdash c_1;\ c_2]\!](\eta)(r, h, s) =$
 let $(r_1, h_1, s_1) = [\![\Gamma \vdash c_1]\!](\eta)(r, h, s)$ in $[\![\Gamma \vdash c_2]\!](\eta)(r_1, h_1, s_1)$

Fig. 5. Semantics of commands.

3.4. Semantics of method declarations and class tables

To define the semantics of a class table we need to obtain the semantics of each method declaration from the semantics of its body. Suppose class K has declaration

$$\mathbf{meth}\ m(\overline{x} : \overline{T}) : U\ \{c\} \tag{2}$$

Its meaning is a state transformer of type $[\mathbf{self} : K, \overline{x} : \overline{T}] \rightsquigarrow [\mathbf{res} : U]$, and the meaning of c is a state transformer of type $\Gamma \rightsquigarrow \Gamma$ with $\Gamma = [\mathbf{self} : K, \overline{x} : \overline{T}, \mathbf{res} : U]$. Basically, the method meaning is obtained from the denotation of c by initializing **res** and then projecting out of the final state.

Formally, suppose η is a method environment and K declares m by Eq. (2). Then $[\![K, m]\!](\eta)$ is the element of $\mathit{SemMeth}(K, m)$ defined by

$$[\![K, m]\!](\eta)(r, h, s) = \begin{array}{l} \text{let } s_0 = [s,\ \mathbf{res} : \mathit{default}(U)] \text{ in} \\ \text{let } (r_1, h_1, s_1) = [\![\overline{x} : \overline{T}, \mathbf{self} : K, \mathbf{res} : U \vdash c]\!](\eta)(r, h, s_0) \text{ in} \\ (r_1, h_1, [\mathbf{res} : (s_1(\mathbf{res}))]). \end{array}$$

Definition 4 (*Semantics of Class Table*)**.** The semantics of a class table *cds*, written $[\![\mathit{cds}]\!]$, is the least upper bound of the ascending chain η defined as follows, for each K and every m declared or inherited[6] in K:

$\eta_0(K, m) = \lambda\sigma \bullet \perp$
$\eta_{j+1}(K, m) = [\![K, m]\!](\eta_j)$ if m is declared in K
$\eta_{j+1}(K, m) = \eta_{j+1}(L, m)$ if m is inherited from L in K.

[6] To be very precise for an inherited method, if $\mathit{mtype}(K, m) = \overline{x} : \overline{T} \to U$ then $\eta_{j+1}(K, m)$ should apply to stores for $[\overline{x} : \overline{T}, \mathbf{self} : K]$ whereas $\eta_{j+1}(L, m)$ applies to stores for $[\overline{x} : \overline{T}, \mathbf{self} : L]$. But the latter contains the former, owing to Eq. (1).

To show that the chain η defined above is ascending, we need $[\![K, m]\!]$ to be a monotonic function on method environments. This in turn relies on the same property for command semantics, $[\![\Gamma \vdash c]\!]$.[7]

Definition 5 (*Semantics of Complete Program*)**.** Let $cds \bullet c$ be a program where the implicit context for c is Γ. Its semantics, written $[\![cds \bullet c]\!]$, is the transformer of Γ-states defined by $[\![cds \bullet c]\!] = [\![\Gamma \vdash c]\!]([\![cds]\!])$.

3.5. Program equivalence

We consider that the inputs and outputs of a main program are given in its global variables, i.e., the heap is not directly observable. The primitive type **int** of unbounded integers is suitable for input/output (e.g., it can encode strings), whereas reference values are not interesting without the heap. However, for simplicity we compare the entire store of global variables, including values of all types including references. The key point is to be able to compare states for two different class tables, where private fields may differ in the heap but stores remain comparable.

The initial heap ought to be empty, but we are not modeling static methods so a main program has **self** : **Void**, and the value of **self** is always non-null. For the sake of the following definition, we say a *Void-only state* is a state (r, h, s) such that for every $o \in dom(r)$ we have $r(o) = \textbf{Void}$.

Definition 6 (*Visible Equivalence of State Transformers*)**.** Consider class tables *cds* and *cds*′ and contexts Γ and Γ^* that only involve types present both *cds* and in *cds*′. Let φ be a state transformer of type $\Gamma \rightsquigarrow \Gamma^*$ for *cds* and φ' a state transformer of type $\Gamma \rightsquigarrow \Gamma^*$ for *cds*′. Define $\varphi \doteq \varphi'$ iff for all Γ-states σ that are Void-only, either $\varphi(\sigma) = \bot = \varphi'(\sigma)$ or both are non-$\bot$ and $\varphi(\sigma)(x) = \varphi'(\sigma)(x)$ for all $x \in dom(\Gamma)$.

This definition is designed to cater for the situation where φ and φ' are acting on states associated with different class tables *cds* and *cds*′. In that situation, the Γ-states for *cds* are different from those for *cds*′ since different type objects may exist and have different fields, and thus the sets of Γ-states are slightly different. However, if the types in $rng(\Gamma)$ are present in both *cds* and *cds*′ then the stores can be compared (they contain references and values of primitive type, e.g., integers) and the Void-only states of *cds* are the same as the Void-only states of *cds*′.

For the equation $\varphi(\sigma)(x) = \varphi'(\sigma)(x)$ to hold for all variables amounts to saying the two stores are equal as functions. Equality for references is sensible provided that the allocator is parametric (Definition 2).

Our main concern is with comparing two different class tables. For that purpose, if we have comparable class tables *cds* and *cds*′ we write $\vdash$, $\vdash'$ for the typing relations determined by *cds*, *cds*′ respectively, and similarly for the auxiliary functions, such as *mtype*, *mtype*′. We also write $[\![-]\!]$, $[\![-]\!]'$ for the respective semantics, and omit the dash on $\vdash$ inside $[\![-]\!]'$. We assume that the same allocator, *fresh*, is used for both $[\![-]\!]$ and $[\![-]\!]'$.

Definition 7 (*Program and Class Equivalence*)**.** Let $cds \bullet c$ and $cds' \bullet c'$ be well-formed and suppose c and c' are typable in context Γ. Define

$$cds \bullet c = cds' \bullet c'$$

iff $[\![\Gamma \vdash c]\!](\eta) \doteq [\![\Gamma \vdash c']\!]'(\eta')$ where $\eta = [\![cds]\!]$ and $\eta' = [\![cds']\!]'$.

4. Ownership confinement

The purpose of confinement is to restrict possible dependences via pointers, to facilitate local reasoning. In fact we restrict dependency by disallowing the existence of certain pointers. We consider an instance-oriented notion of confinement: a single object can "own" some others – called its *reps* – that may not be pointed to except by each other and by the owner.

Two class names determine the confinement policy. The name Own is supposed to be some class for which certain other objects are to be encapsulated. More precisely, each instance of any type K, $K \le Own$, has some associated objects that it "owns". A second class name, say *Rep*, is designated, and owned objects are instances of *Rep* or its subclasses.[8] A confined program maintains an all-states invariant that amounts to "owners as dominators": all paths to an owned object go through its owner. This is the usual property enforced by ownership type systems [11]. Because we are not concerned with ownership hierarchy in general, but only ownership by instances of the designated class Own, we formulate the all-states invariant differently, using partitions instead of paths.[9]

The all-states invariant is that objects in the heap can be partitioned into disjoint regions, depicted in Fig. 6. Each owner is in a region by itself, say Oh_i, with an associated region Rh_i that consists of its owned reps. Aside from owners and their reps, all other objects are termed "clients". Their part of the heap is partitioned into a region *CRh* consisting of instances of type $\le$ *Rep*, and a region *Ch* of all remaining objects.[10]

7 For a very similar semantics, the straightforward but tedious proofs have been machine checked in PVS [25,27].

8 This could easily be generalized to several class names, or to an interface type if that feature were included in the language.

9 Our notion imposes a slight additional restriction, that un-owned instances of class $\le$ *Rep* are not directly referenced by owned reps.

10 For readers familiar with [5], we note that by distinguishing *CRh* from *Ch* we gain important flexibility in the use of library classes.

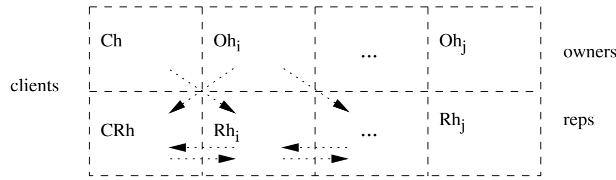


Fig. 6. Confinement scheme for island i. Dashed boxes are partition blocks. Dotted lines indicate prohibited references.

It is convenient but slightly misleading to use the term "free reps" for objects in $\mathcal{C}Rh$: whereas objects of Rh_i are the encapsulated representation of the owner in Oh_i, objects in $\mathcal{C}Rh$ are no different from other clients except that they happen to have type $\leq Rep$.

The formalization designates that all instances in $\mathcal{C}Rh$ and Rh_i regions have type $\leq Rep$, each Oh_i is a single instance of type $\leq Own$, and Ch has no instance of type $\leq Rep$ or $\leq Own$. Given a partition of this form, confinement disallows certain references between regions. Specifically: (a) clients (i.e., Ch and $\mathcal{C}Rh$) do not point to owned reps (in the Rh_i); (b) the owner and reps in one island, say Oh_i and Rh_i, do not point to reps that are free or in another island (say Rh_j with $j \neq i$); and (c) the references from an owner to its reps are private or protected fields declared in class Own or a subclass of Own.[11] References from reps to owners and to clients are allowed.

A command is *confined* provided that it maintains the confinement invariant and a class table is confined provided that its methods all maintain confinement. The details are slightly delicate; the rest of this section is devoted to their formalization.

4.1. Confinement of states

We write $K \not\lesseqgtr L$ to abbreviate $K \not\leq L \wedge L \not\leq K$. We assume that class names Own and Rep are given, such that $Own \not\lesseqgtr Rep$ and thus $Val(Own, r) \cap Val(Rep, r) = \{null\}$, for all r. Moreover, we assume that $Own \neq$ **Void** and $Rep \neq$ **Void** (so these classes are mutually incomparable).

We say pre-heaps h_1 and h_2 are *disjoint* if $dom(h_1) \cap dom(h_2) = \varnothing$. Let $h_1 * h_2$ be the union of h_1 and h_2 if they are disjoint, and undefined otherwise. The following notations are used to express the absence of certain references between objects in different pre-heaps.

Definition 8 ($\not\rightsquigarrow, \not\rightsquigarrow^{\overline{g}}$)**.** Given two pre-heaps h_1 and h_2, to say that no object in h_1 contains a reference to an object in h_2, we define $\not\rightsquigarrow$ by

$$h_1 \not\rightsquigarrow h_2 \text{ iff } \forall o \in dom(h_1) \bullet \forall f \in dom(h_1(o)) \bullet h_1(o)(f) \notin dom(h_2).$$

To say that no object in h_1 contains a reference to an object in h_2 except via a field in $\overline{g}$, we define $\not\rightsquigarrow^{\overline{g}}$ by

$$h_1 \not\rightsquigarrow^{\overline{g}} h_2 \text{ iff } \forall o \in dom(h_1) \bullet \forall f \in dom(h_1(o)) \bullet h_1(o)(f) \in dom(h_2) \Rightarrow f \in \overline{g}. \quad \Box$$

We shall partition the heap h as $Ch * \mathcal{C}Rh * \cdots$ where $Ch * \mathcal{C}Rh$ contains client objects and the rest is partitioned into islands of the form $Oh * Rh$ consisting of a singleton pre-heap Oh with an owner object and a pre-heap Rh of its representation objects. The "mainland" $Ch * \mathcal{C}Rh$ is partitioned to separate those instances of Rep (and its subclasses) that are used in clients from those owned. In such a partition, the pre-heaps Ch, $\mathcal{C}Rh$, Oh, and Rh need not be closed (i.e., they are not self-contained heaps). Moreover, some of the sub-heaps may be empty, so we are slightly abusing the term "partition".

Definition 9 (*Admissible Partition*)**.** An *admissible partition* (with k islands) for ref context r is a set P of pairwise disjoint pre-heaps for r

$$P = \{Ch, \mathcal{C}Rh, Oh_1, Rh_1, \ldots, Oh_k, Rh_k\}$$

with $k \geq 0$, that partition r in the sense that[12]

$$dom(r) = dom(Ch) \cup dom(\mathcal{C}Rh) \cup (\cup_i\, dom(Oh_i * Rh_i)).$$

Furthermore, the following must hold for all i with $1 \leq i \leq k$:

- $dom(Oh_i) \subseteq refs(Own, r)$ and $size(dom(Oh_i)) = 1$ (owners)
- $dom(Rh_i) \subseteq refs(Rep, r)$ (owned reps)
- $dom(\mathcal{C}Rh) \subseteq refs(Rep, r)$ (free reps)
- $dom(Ch) \cap refs(Own, r) = \varnothing$ and $dom(Ch) \cap refs(Rep, r) = \varnothing$ (clients) $\Box$

---

[11] Item (c) is different from [5] and caters for changing the implementation of the entire hierarchy of classes $\leq Own$ rather than just Own.

[12] Note that we allow Rh_i to be empty and moreover k may be 0; there may be only Ch and $\mathcal{C}Rh$, and these too may be empty.

Having established enough notation, we can finally formulate the key notion. In terms of partitions, we can express the sense in which reps belong to a particular owner: the absence of certain references crossing the boundaries of islands, as indicated in Fig. 6.

Definition 10 (*Confined by Partition*)**.** Heap h and ref context r are *confined by partition* P, written $conf(r, h, P)$, iff there is k such that

- h is in $Heap(r)$
- P is an admissible partition for r with k islands
- $h = Ch * CRh * Oh_1 * Rh_1 * \cdots * Oh_k * Rh_k$
- The following conditions hold for all j, i with $j \neq i$.

1. $Ch \not\rightsquigarrow Rh_j$ (clients do not point to owned reps)
2. $Oh_j \not\rightsquigarrow Rh_i$ and $Oh_j \not\rightsquigarrow CRh$ (owners do not share reps)
3. $Oh_j \not\rightsquigarrow^{\overline{g}} Rh_j$ (owned reps are private and protected to Own)
   where $\overline{g} = \{f \mid \exists K \leq Own \bullet f \in dfields(K) \wedge visib(f) \in \{\mathbf{pri}, \mathbf{prot}\}\}$
4. $Rh_j \not\rightsquigarrow Rh_i$ and $CRh \not\rightsquigarrow Rh_i$ and $Rh_j \not\rightsquigarrow CRh$ (reps are confined to their islands) □

Aside from the condition $Rh_j \not\rightsquigarrow CRh$, the conditions enforce "owner as dominator" with each Oh_i owning Rh_i. The condition $Rh_j \not\rightsquigarrow CRh$ does not seem onerous and streamlines our formalization.

A heap may be partitioned in several ways, because there is no inherent order on islands and because unreachable reps can be put in any island. The definitions and results do not depend on choice of partition.

Definition 11 (*Extension of a Partition,* $\trianglelefteq$)**.** Let P be an admissible partition for r, with k islands. Define $P \trianglelefteq P^*$ iff there is a ref context r^* such that $r \subseteq r^*$ and P^* is admissible for r^*; moreover, it is an extension in the sense of the following conditions, where $P^* = \{Ch^*, CRh^*, Oh_1^*, Rh_1^*, \ldots, Oh_n^*, Rh_n^*\}$.

- $n \geq k$
- $dom(Ch) \subseteq dom(Ch^*)$ and $dom(CRh) \subseteq dom(CRh^*)$
- $dom(Oh_j) \subseteq dom(Oh_j^*)$ and $dom(Rh_j) \subseteq dom(Rh_j^*)$, for all $j \leq k$. □

For a state to be confined means that its heap is confined and its store is suitably restricted in terms of the type of some object. That object is typically the receiver of the current method invocation. The restriction on the store treats locals like the fields of the receiver.

Definition 12 (*Confined State*)**.** A state $\sigma = (r, h, s)$ is *confined for object* $o \in dom(r)$ *and partition* P, written $conf(\sigma, o, P)$, iff

- $conf(r, h, P)$
- $o \in dom(Ch * CRh) \Rightarrow rng(s) \cap refs(Rep, r) \subseteq dom(CRh)$
- $\forall i \bullet o \in dom(Oh_i * Rh_i) \Rightarrow rng(s) \cap refs(Rep, r) \subseteq Rh_i$.

In case $\mathbf{self} \in dom(\Gamma)$ and $\sigma \in State(\Gamma)$, we write $conf(\sigma)$ iff there is some P such that $conf(\sigma, \sigma(\mathbf{self}), P)$.

Note that $conf(r, h, P)$ implies that P is admissible for r.

Lemma 13 (*Void-Only Confined*)**.** *If* σ *is Void-only then it is confined for any o and any admissible partition P.*

This is a straightforward consequence of definitions; note that there is a unique admissible partition for a Void-only state.

Lemma 14 (*None State Confined*)**.** *If Rep is* ***None*** *then every state is confined.*

This is because, by semantics, no instances of **None** exist. There is a unique partition in this case, for any state and regardless of what Own is.

4.2. Confinement of commands, expressions and methods

Our main results apply to programs for which confinement is an invariant. In keeping with the compositional structure of the semantics, we formulate this invariance for each of the semantic notions.

A state transformer ψ of type $\Gamma \rightsquigarrow \Gamma^*$ is *confined* iff for all σ, σ^*, P, o, if $o = \sigma(\mathbf{self})$, $conf(\sigma, o, P)$, and $\sigma^* = \psi(\sigma)$, then $\exists P^* \bullet P \trianglelefteq P^* \wedge conf(\sigma^*, o, P^*)$. Observe that there is no constraint in case $\psi(\sigma) = \bot$. Also, Γ^* need not include **self**. But the definition only makes sense in case $\mathbf{self} \in dom(\Gamma)$ (in which case $\sigma(\mathbf{self})$ is non-null). This is indeed the case for all Γ used in the semantics.

For a method environment to be confined, each method meaning should be confined, and moreover public methods of Own subclasses should not return reps. The latter condition we express in terms of types, as follows.

Method environment η is *confined*, written $conf(\eta)$, if and only if $\eta(K, m)$ is a confined state transformer for all K and all $m \in Meths(K)$. Moreover, if $K \leq Own$ and $mtype(K, m) = \overline{x} : \overline{T} \to U$ then $U \not\leq Rep$. The latter condition ensures that owned reps are not returned to clients; it would be slightly awkward to phrase that semantically.

Confinement of arguments means that the store passed in the semantics of method call is confined for the callee.[13] That is, owned reps are not passed to clients. Formally, a call $\Gamma \vdash x.m(\overline{y})$ has *confined arguments* provided the following holds. For all σ, P, if $conf(\sigma, \sigma(\mathbf{self}), P)$ and $\sigma(x) \neq \mathbf{null}$ and σ_1 is the argument state with store $[\mathbf{self} : \sigma(x), \overline{z} : \sigma(\overline{y})]$ then $conf(\sigma_1, \sigma(x), P)$.

Expression $\Gamma \vdash E : T$ is *confined* iff $[\![\Gamma \vdash E : T]\!](\eta)$ is a confined state transformer, for all confined η, and moreover its subexpressions are confined, and if it is a method call then it has confined arguments.

Command $\Gamma \vdash c$ is *confined* iff $[\![\Gamma \vdash c]\!](\eta)$ is a confined state transformer, for all confined η, and moreover all constituent commands and expressions of c are confined.

A class table *cds* is *confined* if $[\![cds]\!]$ is a confined method environment and so are the approximants η_i used to define $[\![cds]\!]$ (see Definition 4).

A program $cds \bullet c$ is *confined* if *cds* is confined and $\Gamma \vdash c$ is confined (where Γ declares the globals of c).

It can be proved that a program is confined provided that its method bodies are confined, plus some restrictions on the parameter and return types of methods (see [5] for a similar proof). It should be straightforward to adapt the static analysis in [5] to enforce confinement as we have defined it here; but we have not checked the details.

Lemma 15 (*None Confined*)**.** *For any Own, if Rep is* ***None*** *then every expression, command, class table, and program is confined.*

This is an easy consequence of Lemma 14.

5. Representation independence

This section formulates and proves the abstraction theorem. First, we make precise the idea of comparing two class tables that differ only in their implementation of the hierarchy of classes K such that $K \leq Own$ for some designated class Own. Then we define local coupling: a relation between single instances of classes $\leq Own$ for the two implementations. This lifts to relations on states, state transformers, and method environments. The abstraction theorem says that if methods of subclasses of Own preserve the coupling then so do all methods of all classes.

5.1. Comparing class tables

We compare two implementations of a hierarchy of classes, namely all the subclasses of the designated Own. They can have completely different declarations, so long as methods of the same signatures are present in both. They can use different reps, distinguished by class name *Rep* for one implementation and *Rep*′ for the other. We allow $Rep = Rep'$. For simplicity, we assume that both *Rep* and *Rep*′ are declared in each of the two compared class tables.

Definition 16 (*Comparable Class Tables, Non-Rep Class*)**.** Consider the following class names Own, Rep, Rep', such that $Own \not\leq Rep$ and $Own \not\leq Rep'$. We say K is a *non-rep* class iff $K \not\leq Rep$ and $K \not\leq Rep'$. Well formed class tables *cds* and *cds*′ are *comparable* provided the following conditions hold.

1. *cds* and *cds*′ are identical except for their values on all $K, K \leq Own$. (In particular, $cds(Rep) = cds'(Rep)$ and $cds(Rep') = cds'(Rep')$.)
2. $super = super'$ (In particular $super(K) = super'(K)$, for all $K \leq Own$.)
3. $mtype = mtype'$ (In particular, for any m and any $K \leq Own$, either $mtype(K, m)$ and $mtype'(K, m)$ are both undefined or both are defined and equal.)
4. The public fields of $cds(K)$ and $cds'(K)$ are the same, for all $K, K \leq Own$. □

The notation for confinement has Own and *Rep* implicit, so we write $conf'$ for confinement with respect to Own and *Rep*′.

Note that the typing relations $\Gamma \vdash -$ and $\Gamma \vdash' -$ are the same unless $\Gamma(\mathbf{self}) \leq Own$. Similarly, $dfields(K) = dfields'(K)$ unless $K \leq Own$.

The following straightforward consequence of condition (3) is needed.

Lemma 17 (*Depth Agree*)**.** *If $mtype(K, m)$ is defined then $depth(K, m) = depth'(K, m)$, for all K and all m in $Meths(K)$.*

This may seem a rather strong requirement, since all the methods, with the same signature, have to be present in both versions of the class K. (Though a method may be declared in one version of the class and inherited in the other.) Nevertheless, this is required only for refactoring transformations that involve data refinement, and hence confinement. It does not preclude the definition of refactorings that capture, for instance, method elimination, which can be applied in contexts where confinement is not required.

[13] We formalize a condition that is simple and satisfactory for our purposes. But it is unnecessarily strong: it would suffice for it to be true for reachable states, e.g., if we had an assert before the call.

5.2. Coupling relations and simulations

In this section we assume given comparable class tables *cds* and *cds'*. The definitions are organized as follows. A *local coupling* $\mathcal{R}$ is a suitable relation on islands. This induces a family of lifted *coupling relations* $\hat{\mathcal{R}}$, for heaps, for states, for state transformers, etc. Then comes the definition of *simulation*, a coupling that is preserved by all methods of classes $K, K \le Own$, and established by their constructors.

For precision, the notation distinguishes between the different kinds of relations, e.g., $\hat{\mathcal{R}}\ \Gamma\ \sigma\ \sigma'$ says states σ, σ' are coupled, and $\hat{\mathcal{R}}\ (\Gamma \rightsquigarrow \Gamma^*)\ \varphi\ \varphi'$ is used for state transformers of the indicated type. For clarity, however, we write $\hat{\mathcal{R}}\sigma\sigma'$ or $\hat{\mathcal{R}}\varphi\varphi'$ when there seems to be no risk of confusion.

The definitions to follow are suitable under the assumption that the semantics uses a parametric allocator (Definition 2). The point is that two comparable programs can have different allocation behavior on owned reps, without influence on allocation for other types. So coupling relations can be defined making heavy use of equality of references. For practical purposes, the parametricity assumption is unrealistic. Instead, the technical development can be changed: in place of equality of references one can use partial equivalence relations based on bijective renamings of references. In [5], the simplifying assumption is used for expository purposes, and then the version with partial equivalence relations is worked out fully. Here, we do only the simplified version, because the generalization to partial equivalence relations does not pose additional difficulties.

Definition 18 (*Local Coupling,* $\mathcal{R}$)**.** A *local coupling* is a binary relation $\mathcal{R}$ on pairs (r, h) with $h \in PreHeap(r)$, that is local to an island, in the following sense: For any r, h, r', h', if $\mathcal{R}(r, h)(r', h')$ then there is a reference o with $r(o) \le Own$ and $r(o) = r'(o)$ and there are partitions $h = Oh * Rh$ and $h' = Oh' * Rh'$ such that

1. $dom(Oh) = \{o\} = dom(Oh')$
2. $dom(Rh) \subseteq refs(Rep, r)$ and $dom(Rh') \subseteq refs(Rep', r')$
3. $h(o)(f) = h'(o)(f)$ for all $f \in dom(fields(r(o)))$ such that either f is public or there is K such that $Own < K$ and f is a private or protected field declared in K. □

The last condition says that only the private and protected fields declared within classes $\le Own$ are exempt from being equated. These are exactly the fields that are allowed to differ between *cds* and *cds'* according to Definition 16.

Two implementations of Own and its subclasses may preserve the coupling while making quite different use of owned reps. For this reason, we need to make an exception for owned reps in the following definition. To this end, we define "*non-rep and confined*" as follows:

$$nrconf(\sigma, o, P) \text{ iff } conf(\sigma, o, P) \text{ and } \neg\exists i \bullet o \in dom(Rh_i)$$
$$nrconf(\sigma) \text{ iff } \exists P \bullet nrconf(\sigma, \sigma(\mathbf{self}), P).$$

Definition 19 (*Coupling Relations,* $\hat{\mathcal{R}}$)**.** Suppose $\mathcal{R}$ is a local coupling.

For heaps $h \in Heap(r)$ and $h' \in Heap(r')$, we define $\hat{\mathcal{R}}(r, h)(r', h')$ iff there are partitions P, P' with the same number, k, of islands, such that

- $conf(r, h, P)$ and $conf'(r', h', P')$
- $\mathcal{R}\ (r, Oh_i * Rh_i)\ (r', Oh'_i * Rh'_i)$ for all i in $1..k$
- $Ch * CRh = Ch' * CRh'$

Given typing context Γ and states $\sigma = (r, h, s)$ and $\sigma' = (r', h', s')$ with $\sigma \in State(\Gamma)$ and $\sigma' \in State'(\Gamma)$, we define
$\hat{\mathcal{R}}\ \Gamma\ \sigma\ \sigma'$ iff $\hat{\mathcal{R}}(r, h)(r', h')$ and $s = s'$, i.e., $s(x) = s'(x)$ for all x in $dom(\Gamma)$.

For Γ-outcomes α, i.e., α is either $\perp$ or a Γ-state, we define
$\hat{\mathcal{R}}\ \Gamma\ \alpha\ \alpha'$ iff $\alpha = \perp = \alpha'$ or both are non-$\perp$ and relate as states.

For state transformers φ, ψ of type $\Gamma \rightsquigarrow \Gamma^*$ (with $\mathbf{self} \in dom(\Gamma)$) we define
$\hat{\mathcal{R}}\ (\Gamma \rightsquigarrow \Gamma^*)\ \varphi\ \psi$ iff
$\forall \sigma, \sigma' \bullet \hat{\mathcal{R}}\ \Gamma\ \sigma\ \sigma' \wedge nrconf(\sigma) \wedge nrconf'(\sigma') \Rightarrow \hat{\mathcal{R}}\ \Gamma^*\ (\varphi(\sigma))\ (\psi(\sigma'))$.

For methods environments η, η', we define
$\hat{\mathcal{R}}\ \eta\ \eta'$ iff $\hat{\mathcal{R}}\ ([\mathbf{self}: K, \overline{x}: \overline{T}] \rightsquigarrow [res: U])\ (\eta(K, m))\ (\eta'(K, m))$
for all K, m such that $mtype(K, m) = \overline{x}: \overline{T} \to U$. □

Note that the condition $\hat{\mathcal{R}}\ \Gamma\ \sigma\ \sigma'$ is defined even in case $\Gamma(\mathbf{self}) \le Own$. This is appropriate for the initial state for a call to a method of Own or a subclass thereof. But it is not always appropriate for intermediate states of two different implementations of such a method. In general, two commands that denote related state transformers may go through intermediate states that are quite different, and a proof that the two commands denote related state transformers may involve quite separate reasoning about those two commands.

Apropos the definition of $\hat{\mathcal{R}}$ for state transformers, note that the antecedent $\hat{\mathcal{R}}\ \Gamma\ \sigma\ \sigma'$ implies confinement of the heaps; the antecedents $nrconf(\sigma)$ and $nrconf'(\sigma')$ add that the store is confined for **self** and moreover **self** is not an owned rep.[14]

If $\hat{\mathcal{R}}\ \Gamma\ (r, h, s)\ (r', h', s')$ then the ref contexts agree on all except owned reps. Let us make this precise.

Lemma 20 (*Ref Context Agree*)**.** *Suppose* $\hat{\mathcal{R}}\ \Gamma\ (r, h, s)\ (r', h', s')$. *Then we have* (a) *if* o *is in* $dom(r) - dom(r')$ *then* $r(o) \leq Rep$ *and if* o *is in* $dom(r') - dom(r)$ *then* $r'(o) \leq Rep'$. *And* (b) *if* o *is in* $dom(r) \cap dom(r')$ *then* $r(o) = r'(o)$.

The following two properties hold for any Γ, even if $\Gamma(\textbf{self}) \leq Own$. Like Lemma 20, they are direct consequences of the definitions.

Lemma 21 (*Identity Extension*)**.** *For any* σ, *if the heap of* σ *is Void-only then* $\hat{\mathcal{R}}\ \Gamma\ \sigma\ \sigma$.

Lemma 22 (*Identity Relation*)**.** *For any* σ, σ', *if* $\hat{\mathcal{R}}\ \Gamma\ \sigma\ \sigma'$ *then* $\sigma(x) = \sigma'(x)$ *for all* $x \in dom(\Gamma)$.

Finally we are ready to define the notion of simulation. To do so, we need to express that the corresponding implementations of the constructors in classes $\leq Own$ establish the local coupling for their initial islands. However, the semantics is defined in terms of heaps, not pre-heaps, and construction of a new owner object can involve interaction with other Own instances and client objects. In effect, item 1 below spells out part of the semantics of **new** in order to precisely say what it means for constructors to establish $\mathcal{R}$.

Definition 23 (*Simulation*)**.** Let $\eta \in \mathbb{N} \to [\![Menv]\!]$ (resp. $\eta' \in \mathbb{N} \to [\![Menv]\!]'$) be the approximation chain in the definition of $[\![cds]\!]$ (resp. $[\![cds']\!]'$). Local coupling $\mathcal{R}$ is a *simulation* iff the following conditions hold for every i.

1. (constructors establish $\mathcal{R}$) For any r, h, r', h', o, K, suppose $K \leq Own$ and $mtype(K, \textbf{ctor}) = \overline{z}:\overline{T} \to K$. Suppose that $\hat{\mathcal{R}}(r, h)(r', h')$ and $o \notin dom(r \cup r')$ and $\overline{v} \in Val(\overline{T}, r \cap r')$ and let
   $r_0 = [r,\ o:K]$ and $h_0 = [h,\ o:defaultFieldRcrd(K)]$
   $r'_0 = [r',\ o:K]$ and $h'_0 = [h',\ o:defaultFieldRcrd'(K)]$
   $\sigma = (r_0, h_0, [\textbf{self}:o, \overline{z}:\overline{v}])$
   $\sigma' = (r'_0, h'_0, [\textbf{self}:o, \overline{z}:\overline{v}])$
   $\alpha = \eta_i(K, \textbf{ctor})(\sigma)$
   $\alpha' = \eta'_i(K, \textbf{ctor})(\sigma')$
   If $conf(\sigma)$ and $conf(\sigma')$ then $\hat{\mathcal{R}}\ [\textbf{res}:K]\ \alpha\ \alpha'$.
2. (methods preserve $\hat{\mathcal{R}}$) For every $K \leq Own$ and every non-**ctor** method m in $Meths(K)$, let $mtype(K, m) = \overline{x}:\overline{T} \to U$ in the conditions
   (a) $\hat{\mathcal{R}}\ \ \eta_i\ \eta'_i \Rightarrow \hat{\mathcal{R}}\ ([\textbf{self}:K, \overline{x}:\overline{T}] \rightsquigarrow [\textbf{res}:U])\ ([\![K, m]\!](\eta_i))\ ([\![K, m]\!]'(\eta'_i))$ if m is declared in $cds(K)$ and in $cds'(K)$
   (b) $\hat{\mathcal{R}}\ \ \eta_i\ \eta'_i \Rightarrow \hat{\mathcal{R}}\ ([\textbf{self}:K, \overline{x}:\overline{T}] \rightsquigarrow [\textbf{res}:U])\ ([\![K, m]\!](\eta_i))\ ([\![L, m]\!]'(\eta'_i))$ if m is declared $cds(K)$ and is inherited from L in $cds'(K)$
   (c) the condition symmetric to (2b), if m is inherited in $cds(K)$ but declared in $cds'(K)$. □

The gist of the theorem is that if the methods of subclasses of Own are related by $\hat{\mathcal{R}}$ then all methods are. We can now express this conclusion as $\hat{\mathcal{R}}\ \ [\![cds]\!]\ [\![cds']\!]'$.

Theorem 24 (*Abstraction*)**.** *Suppose cds and cds′ are confined (for* Own, Rep *and* Own, Rep' *respectively),* $\mathcal{R}$ *is a simulation, and the allocator is parametric. Then* $\hat{\mathcal{R}}\ \ [\![cds]\!]\ [\![cds']\!]'$.

The proof is based on the following Lemmas. In the following, we confine attention to method environments in the approximation chains for class tables cds and cds'. This is only needed for the case: **new** L with $L \leq Own$, for which we use the assumption that $\mathcal{R}$ is a simulation — specifically, the condition in Definition 23 that the constructor establishes $\hat{\mathcal{R}}$. The other condition in Definition 23, that methods preserve $\hat{\mathcal{R}}$, is used in the proof of Theorem 24.

Lemma 25 (*Preservation by Expressions*)**.** *Suppose that the following hold:*

- $\mathcal{R}$ *is a simulation*
- *cds and cds′ are confined, for* Own, Rep *and* Own, Rep' *respectively*
- η_i *and* η'_i *are method environments at the same level, i, in the approximation chain defining the semantics of cds and cds′ respectively*
- $\hat{\mathcal{R}}\ \ \eta_i\ \eta'_i$
- $\Gamma(\textbf{\textit{self}}) \nleq Own$
- $\Gamma \vdash e:T$ *is confined*

Then we have $\hat{\mathcal{R}}\ (\Gamma \rightsquigarrow [\textbf{res}:T])\ ([\![\Gamma \vdash e:T]\!](\eta_i))\ ([\![\Gamma \vdash' e:T]\!]'(\eta'_i))$.

[14] By contrast, the definition for state transformers in [5] does not impose an ownership condition on **self**. Instead, the definition for method environments excludes classes $K \leq Rep$. That works because all instances of Rep are treated as owned.

Proof. Note that the hypotheses imply that η_i and η'_i are confined. We drop the subscripts and write η and η' in the proof.

We use induction on the derivation of $\Gamma \vdash e : T$. In each case, we assume $\sigma = (r, h, s)$ and $\sigma' = (r', h', s')$, with $\sigma \in State(\Gamma)$ and $\sigma' \in State'(\Gamma')$. We assume that $\hat{\mathcal{R}}\ \Gamma\ \sigma\ \sigma'$ and $nrconf(\sigma)$ and $nrconf'(\sigma')$. We must show that $\hat{\mathcal{R}}\ [\mathbf{res} : T]\ \alpha\ \alpha'$ where α and α' are the corresponding outcomes, i.e., $\alpha = [\![\Gamma \vdash e : T]\!](\eta)(\sigma)$ and $\alpha' = [\![\Gamma \vdash e : T]\!]'(\eta')(\sigma')$.

Case $\Gamma \vdash (L)\, x : L$. From $\hat{\mathcal{R}}\ \Gamma\ \sigma\ \sigma'$ we have $s(x) = s'(x)$. Let $o = s(x)$. By hypothesis of the Lemma, $\Gamma(\mathbf{self})$ is not $\leq Own$. Also, $\sigma(\mathbf{self})$ is not owned, by hypotheses $nrconf(\sigma)$. So, by $conf(\sigma)$, o is not an owned rep in σ. For the same reasons, o is not an owned rep in σ'. Thus by $\hat{\mathcal{R}}\ \ \sigma\ \sigma'$ we have $r(o) = r'(o)$. By semantics, if $r(o) \not\leq L$ then the outcomes are both $\bot$. Otherwise, α is $(r, h, [\mathbf{res} : o])$ and α' is $(r', h', [\mathbf{res} : o])$. So $\hat{\mathcal{R}}\ [\mathbf{res} : L]\ \alpha\ \alpha'$ follows from $\hat{\mathcal{R}}\ \Gamma\ \sigma\ \sigma'$ by definition of $\hat{\mathcal{R}}$.

Cases $\Gamma \vdash x\ \mathbf{is}\ L : \mathbf{bool}$ and $\Gamma \vdash x = y : \mathbf{bool}$. Similar to the preceding case.

Case $\Gamma \vdash x.f : T$. As above, we have $s(x) = s'(x)$. If $s(x) = null$ then both outcomes are $\bot$. Otherwise, note first that for the same reasons as in the case of cast above, by confinement $s(x)$ is not an owned rep. Thus by definition of $\hat{\mathcal{R}}$ we have $\sigma(x.f) = \sigma'(x.f)$. So by semantics α is $(r, h, [\mathbf{res} : \sigma(x.f)])$ and α' is $(r', h', [\mathbf{res} : \sigma(x.f)])$ and we have $\hat{\mathcal{R}}\ [\mathbf{res} : L]\ \alpha\ \alpha'$.

Case $x.m(\overline{y})$. From $\hat{\mathcal{R}}\ \Gamma\ \sigma\ \sigma'$ we have $s(x) = s'(x)$ and $s(\overline{y}) = s'(\overline{y})$. If $s(x) = null$ then both outcomes are $\bot$. Otherwise, let $o = s(x)$. By hypothesis of the Lemma, $\Gamma(\mathbf{self}) \not\leq Own$ and $\sigma(\mathbf{self})$ is not an owned rep, so by confinement o is not an owned rep. Thus $\hat{\mathcal{R}}\ \Gamma\ \sigma\ \sigma'$ implies $r(o) = r'(o)$. Let $L = r(o)$ and suppose $mtype(L, m)$ is $\overline{z} : \overline{T} \rightarrow U$. Let $s_1 = [\mathbf{self} : o, \overline{z} : s(\overline{y})]$. We have $\hat{\mathcal{R}}\ [\mathbf{self} : L, \overline{z} : \overline{T}]\ (r, h, s_1)\ (r', h', s_1)$. Moreover, the states (r, h, s_1) and (r', h', s_1) are confined, because by hypothesis of the Lemma $\Gamma \vdash e : T$ is confined which includes the confinement of arguments. Now α is $\eta(L, m)(r, h, s_1)$ and α' is $\eta'(L, m)(r', h', s_1)$. So by $\hat{\mathcal{R}}\ \ \eta\ \eta'$ we get $\hat{\mathcal{R}}\ \ \alpha\ \alpha'$.

Case **let** x **be** e **in** $e1$. By induction on e and $e1$; details are omitted.

Case **new** L. For this case, we will distinguish sub-cases on whether $L \leq Own$. But first, recall that according to definition of well formed class table in Section 2, this only occurs in the form **let** x **be new** L **in** $x.\mathbf{ctor}(\overline{y})$.

For sub-case $L \not\leq Own$, we argue as follows. Let $o = fresh(r, L)$ and $o' = fresh(r', L)$. Using Lemma 20, we have $\{o \mid r(o) = L\} = \{o \mid r'(o) = L\}$, so by parametricity of the allocator we have $o = o'$. Let $r_0 = [r,\ o : L]$ and $h_0 = [h,\ o : defaultFieldRcrd(L)]$; *mutatis mutandis* for r'_0 and h'_0. Note that $\hat{\mathcal{R}}\ [\mathbf{self} : L]\ (r_0, h_0, [\mathbf{self} : o])\ (r'_0, h'_0, [\mathbf{self} : o])$, and thus the interpretations of **new** L in these two states are related. That is,

$$\hat{\mathcal{R}}\ (\Gamma \rightsquigarrow [\mathbf{res} : L])\ ([\![\mathbf{new}\ L]\!](\eta)(r_0, h_0, [\mathbf{self} : o]))\ ([\![\mathbf{new}\ L]\!]'(\eta')(r'_0, h'_0, [\mathbf{self} : o])).$$

We obtain a confining partition by extending the given ones (suppose they are $Ch * CRh * Oh_1 * Rh_1 * \cdots$ and $Ch^* * CRh^* * Oh_1^* * Rh_1^* * \cdots$ respectively). For r_0, h_0, we add the new object to Ch or to CRh depending on whether $L \leq Rep$. For r'_0, h'_0, we add the new object to Ch^* or to CRh^* depending on whether $L \leq Rep'$. It follows that $conf(\sigma_0)$ and $conf'(\sigma'_0)$. Using preservation by method call (an earlier case in this proof) and unfolding the semantics of **let** expressions, we get that the outcomes from **let** x **be new** L **in** $x.\mathbf{ctor}(\overline{y})$ are related.

For sub-case $L \leq Own$, we exploit that **new** L occurs in the **let** expression with its constructor invocation. Let $o, o', r_0, r'_0, h_0, h'_0$ be as in the preceding paragraph. Let $\overline{v} = s(\overline{y})$ and note that $\overline{v} = s'(\overline{y})$ from $\hat{\mathcal{R}}\ \ \Gamma\ \ \sigma\ \ \sigma'$. Let $\sigma_0 = (r_0, h_0, [\mathbf{self} : o, \overline{z} : \overline{v}])$ (where $\overline{z}$ are the parameters of **ctor** for class L) and let $\sigma'_0 = (r'_0, h'_0, [\mathbf{self} : o, \overline{z} : \overline{v}])$. We extend the given confining partitions by adding a new owner island. By hypothesis, the arguments are confined, i.e., $conf(\sigma_0)$ and $conf'(\sigma'_0)$. Now σ_0, σ'_0 satisfy the conditions on σ, σ' in Definition 23(1) of simulation. So by Definition 23(1) and using $\hat{\mathcal{R}}\ \ \eta\ \eta'$ we get $\hat{\mathcal{R}}\ [\mathbf{res} : L]\ (\eta(L, \mathbf{ctor})(\sigma_0))\ (\eta'(L, \mathbf{ctor})(\sigma'_0))$ and then by semantics of **let** we get that the outcomes from **let** x **be new** L **in** $x.\mathbf{ctor}(\overline{y})$ are related. □

Lemma 26 (*Preservation by Commands*)**.** *Suppose that the following hold:*

- $\mathcal{R}$ *is a simulation*
- *cds and cds′ are confined, for* Own, Rep *and* Own, Rep' *respectively*
- η_i *and* η'_i *are method environments at the same level, i, in the approximation chain defining the semantics of cds and cds′ respectively*
- $\hat{\mathcal{R}}\ \ \eta_i\ \eta'_i$
- $\Gamma(\mathbf{self}) \not\leq Own$
- $\Gamma \vdash c$ *is confined*

Then we have $\hat{\mathcal{R}}\ (\Gamma \rightsquigarrow \Gamma)\ ([\![\Gamma \vdash c]\!](\eta_i))\ ([\![\Gamma \vdash' c]\!]'(\eta'_i))$.

Proof. By structural induction on the derivation of $\Gamma \vdash c$. As in the proof of Lemma 25, we assume σ, σ' are related, with $nrconf(\sigma)$ and $nrconf'(\sigma')$, to prove that the corresponding outcomes α and α' are related. And we drop subscripts on η_i, η'_i.

Case $x := e$. By hypothesis e is confined. By Lemma 25 we have

$$\hat{\mathcal{R}}\ (\Gamma \rightsquigarrow [\mathbf{res} : T])\ ([\![\Gamma \vdash e : T]\!](\eta))\ ([\![\Gamma \vdash' e : T]\!]'(\eta')).$$

Hence $\hat{\mathcal{R}}$ $[\mathbf{res}:T]$ β β' where $\beta = [\![\Gamma \vdash e:T]\!](\eta)(\sigma)$ and $\beta' = [\![\Gamma \vdash e:T]\!]'(\eta')(\sigma')$. Either both are $\bot$ are both are states with the same values for **res**. Either way, the semantics of $x := e$ gives related outcomes.

Case $x.f := y$. From $\hat{\mathcal{R}}$ Γ σ σ' we have $s(x) = s'(x)$ and $s(y) = s'(y)$. If $s(x) = null$ then both outcomes are $\bot$. Otherwise, note first that by hypothesis of the Lemma, $\Gamma(\mathbf{self}) \not\leq Own$. By antecedent in the definition of $\hat{\mathcal{R}}$, $\sigma(\mathbf{self})$ is not an owned rep. So by confinement $s(x)$ is not an owned rep. Similarly, $s'(x)$ is not an owned rep. So by semantics $[h \mid s(x).f : s(y)]$ is related to $[h' \mid s(x).f : s(y)]$ as required.

Case **if** x **then** c_1 **else** c_2. From $\hat{\mathcal{R}}$ Γ σ σ' we have $s(x) = s'(x)$. If $s(x)$ is true, we use the induction hypothesis for c_1; otherwise we use induction on c_2.

Case $c_1;\ c_2$. By induction on c_1 we get related outcomes; if not $\bot$, then these are related states and we can use induction on c_2.

The remaining cases are straightforward. $\square$

Now we return to the abstraction Theorem.

Proof (*of Theorem 24*)**.** We show that the relation holds for each step in the approximation chain in the semantics of class tables. That is, we show by induction on j that

$$\hat{\mathcal{R}}\ \ \eta_j\ \eta'_j \quad \text{for every } j \in \mathbb{N}.$$

The result $\hat{\mathcal{R}}$ $[\![cds]\!]$ $[\![cds']\!]'$ then follows by fixpoint induction, as $[\![cds]\!]$ and $[\![cds']\!]'$ are defined to be the fixpoints of these ascending chains.

For the base case, we have $\hat{\mathcal{R}}$ $([\mathbf{self}:K, \overline{x}:\overline{T}] \rightsquigarrow [\mathbf{res}:U])$ $(\eta_0(K,m))$ $(\eta'_0(K,m))$ for every (K,m) because $\lambda\sigma \bullet \bot$ relates to itself. Hence $\hat{\mathcal{R}}$ η_0 η'_0.

For the induction step, suppose

$$\hat{\mathcal{R}}\ \ \eta_j\ \eta'_j. \tag{$*$}$$

We must show $\hat{\mathcal{R}}$ η_{j+1} η'_{j+1}, that is, for every m with $mtype(K,m)$ defined:

$$\hat{\mathcal{R}}\ ([\mathbf{self}:K, \overline{x}:\overline{T}] \rightsquigarrow [\mathbf{res}:U])\ (\eta_{j+1}(K,m))\ (\eta'_{j+1}(K,m)) \tag{$\dagger$}$$

where $mtype(K,m) = \overline{x}:\overline{T}\rightarrow U$. For arbitrary m we show ($\dagger$) for all K with $mtype(K,m)$ defined, using a nested induction on $depth(K,m)$. Note that we have $depth'(K,m) = depth(K,m)$ from Lemma 17.

The base case of the nested induction is the unique K with $depth(K,m) = 0$; here m is declared in both $cds(K)$ and $cds'(K)$. We go by cases on K. If $K \leq Own$, we get ($\dagger$) from the assumption that $\hat{\mathcal{R}}$ is a simulation. In detail: Using ($*$) and Definition 23(2a) we get

$$\hat{\mathcal{R}}\ ([\mathbf{self}:K, \overline{x}:\overline{T}] \rightsquigarrow [\mathbf{res}:U])\ ([\![K,m]\!](\eta_j))\ ([\![K,m]\!]'(\eta'_j))$$

whence we have ($\dagger$) by definition of η_{j+1} and η'_{j+1}. The other case is that $K \not\leq Own$. Then by Definition 16(1) of comparable class tables we have $cds(K) = cds'(K)$ and in particular both class tables have the same declaration

$$\mathbf{meth}\ m(\overline{x}:\overline{T}):U\ \{c\}.$$

To show ($\dagger$), let $\Gamma = [\mathbf{self}:K, \mathbf{res}:U, \overline{x}:\overline{T}]$. Then by Lemma 26, considering that $\hat{\mathcal{R}}$ η_j η'_j, we get that

$$\hat{\mathcal{R}}\ (\Gamma \rightsquigarrow \Gamma)\ ([\![\Gamma \vdash c]\!](\eta_j))\ ([\![\Gamma \vdash' c]\!]'(\eta'_j)).$$

It follows from the semantics and definition of $\hat{\mathcal{R}}$ that

$$\hat{\mathcal{R}}\ ([\mathbf{self}:K, \overline{x}:\overline{T}] \rightsquigarrow [\mathbf{res}:U])\ ([\![K,m]\!](\eta_j))\ ([\![K,m]\!]'(\eta'_j))$$

and so ($\dagger$) holds by definition of η_{j+1} and η'_{j+1}. This concludes the base case of the nested induction. The appeal to Lemma 26 depends on $conf(\eta_i)$ and $conf'(\eta'_i)$ which holds by the hypothesis of the Theorem that cds and cds' are confined.

For the induction step of the nested induction, suppose $depth(K,m) > 0$. Using the definition of $depth$, the induction hypothesis is

$$\hat{\mathcal{R}}\ ([\mathbf{self}:L, \overline{x}:\overline{T}] \rightsquigarrow [\mathbf{res}:U])\ (\eta_{j+1}(L,m))\ (\eta'_{j+1}(L,m)) \tag{$\ddagger$}$$

where $L = super(K)$. If m is declared in both $cds(K)$ and $cds'(K)$ then the argument is the same as in the base case of the nested induction. If m is inherited in both $cds(K)$ and $cds'(K)$ then ($\dagger$) follows from ($\ddagger$) because the semantics defines $\eta_{j+1}(K,m)$ to be $\eta_{j+1}(L,m)$. (And by definition $\hat{\mathcal{R}}$ $[\mathbf{self}:L, \overline{x}:\overline{T}]$ is the same as $\hat{\mathcal{R}}$ $[\mathbf{self}:K, \overline{x}:\overline{T}]$ on state-pairs where both are defined.) The remaining possibility is that m is declared in $cds(K)$ and inherited in $cds'(K)$ from some L (or the other way around, in which case the argument is symmetric). Then $K \leq Own$, by comparability of cds and cds' (Definition 16). Using the simulation property Definition 23(2b) and ($*$) we get

$$\hat{\mathcal{R}}\ ([\mathbf{self}:T, \overline{x}:\overline{T}] \rightsquigarrow [\mathbf{res}:U])\ ([\![K,m]\!](\eta_j))\ ([\![L,m]\!]'(\eta'_j))$$

and thus ($\dagger$) by definition of η_{j+1} and η'_{j+1}. That concludes the proof. $\square$

One would like to combine Theorem 24 and Lemma 26 to conclude that the meaning of a command c outside class Own relates to itself, in the environments $[\![cds]\!]$ and $[\![cds']\!]'$, but this is not immediate because Lemma 26 only applies to the approximating environments. So we need a separate result.

Theorem 27 (*Preservation for Complete Program*)**.** *Suppose $\mathcal{R}$ is a simulation and cds and cds′ are confined. Consider any class $K \not\leq Own$ and any confined $\Gamma \vdash c$ with $\Gamma(\textbf{self}) = K$. Then*

$$\hat{\mathcal{R}}\ (\Gamma \rightsquigarrow \Gamma)\ ([\![\Gamma \vdash c]\!]([\![cds]\!]))\ ([\![\Gamma \vdash' c]\!]'([\![cds']\!]')).$$

Proof. From Lemma 26 by fixpoint induction, similar to the proof of Theorem 24. □

These results allow us to prove a data refinement law for an inheritance hierarchy, as well as some representative refactorings that also impact on entire class trees.

6. Laws and refactorings

Equivalence of programs can be proved using a transformational approach based on algebraic laws. In this section we present some algebraic laws and refactorings that may be used to prove equivalence of object-oriented programs. Broadly, in this context, refactorings can be regarded as algebraic rules, which have a larger granularity than laws, expressing a more robust transformation. Refactorings can be proved by the application of several laws, possibly including change of data representation. Nevertheless, there is no precise technical difference between laws and refactorings. Here we keep consistency with previous work [8,10], where laws form a comprehensive set that is complete in the sense of being expressive enough to reduce an arbitrary program to an imperative normal form; refactorings are restructuring transformations as proposed by Fowler [18] and others. We are not concerned with completeness here; we present some general laws and refactorings that illustrate both applications of the confinement theory presented in previous sections and (as shown in the next section) the development of a representative case study.

The laws and refactorings in our previous work [8,10] are defined in the context of copy semantics. Furthermore, apart from the inherent limitation of adopting copy semantics, transformations involving change of data representation of class hierarchies were not formalized; a data refinement law in this context was only postulated. Our simulation theory [13] justifies a data refinement law only for a single class and its private fields. As an initial effort to promote the algebraic laws to a reference semantics context, we have analyzed the validity of these laws [37] based on the rCOS framework [20]. As aliasing occurs at the level of commands, through assignments and parameter passing mechanisms, laws that involve only transformation in the structure of the classes, like the ones in Section 6.1, are valid on both copy and reference semantics, and the proofs of these laws on both semantics follow similar reasoning. In this paper we do not prove the laws of classes in our semantics.

The main contributions of this section are the formalization of the data refinement law (in Section 6.1) and the new refactorings that ensure the behavior preserving transformation of data representation in entire class trees (in Section 6.2). The refactorings are proved by using the data refinement law, some laws of classes and the semantics of our language.

Preliminaries. Laws and refactorings are stated as equivalences of the form:

$$cds \bullet c = cds' \bullet c'.$$

In general, classes can be introduced, removed or modified, and the main program can also be affected, as a consequence of the transformation. In most cases, however, a single or only a few class declarations, say cds_1, are affected, in which case we write the equation

$$cds_1 =_{cds,c} cds'_1 \tag{3}$$

as an abbreviation of $cds_1\ cds \bullet c = cds'_1\ cds \bullet c$ (for which see Definition 7).

All laws and refactorings in this section have side conditions that must be satisfied to allow the law (refactoring) application. We use the arrow $\rightarrow$ to express the side conditions necessary to apply the law (refactoring) from left to right. The arrow $\leftarrow$ has the same purpose, but concerning a right to left application. When the condition applies in both directions we use $\leftrightarrow$. Class, attribute and method declarations of class K are denoted by cd_k, ads_k and mts_k, respectively. The conditions are formulated in terms of well formed code as defined in Figs. 2 and 3.

In the refactorings, though not the laws, we use the substitution notation $W[\alpha/\beta]$, meaning that a phrase β is replaced by phrase α in the context W. Owing to the A-normal form of our syntax (Fig. 1), the transformations we need do not involve bound variables and capture avoidance. We replace

- class names, as in $[K/L]$
- attribute and method names, as in $[u.x/u.y]$ and $[u.m/u.n]$ where u is a variable, x, y are attributes, m, n are methods
- method calls for attribute reads and writes, in particular: changing a field update $u.x := v$ to $u.set\,X(v)$ and field access expression $u.x$ to $u.get\,X()$.

The last case yields code that is sugared, so it abbreviates transformations using the desugarings. Further clarification about our use of substitution appears just before Refactoring 1 ⟨*pull up field*⟩.

6.1. Laws of classes

The purpose of Law 1 ⟨*class elimination*⟩ is to introduce (from right to left) or to eliminate (from left to right) a class declaration, provided the side conditions shown below are satisfied. In this law, cd_1 is the class declaration being eliminated/introduced, *cds* is the remaining set of class declarations of the program, and *c* is the main command.

Law 1 (*Class Elimination*)**.**

$$\boxed{cds\ cd_1\ \bullet\ c} \quad = \quad \boxed{cds\ \bullet\ c}$$

(→) 1. *The class declared in* cd_1 *is not referred to in cds or c;*

(←) 1. *The name of the class declared in* cd_1 *is distinct from those of all classes declared in cds;*
2. *the superclass appearing in* cd_1 *is either* **Object** *or declared in cds;*
3. *the attribute and method names declared by* cd_1 *are not declared by its superclasses in cds, except in the case of* ***ctor*** *and method redefinitions.*

The following law is also a simple one: it is used to introduce new private attributes or to eliminate unused ones.

Law 2 (*Attribute Elimination*)**.**

Left		Right

class K ***ext*** L {
  pri $x : T;\ ads_k$
  mts_k
}
	$=_{cds,c}$	***class*** K ***ext*** L {
		  ads_k
		  mts_k
		}

(→) 1. *attribute x does not appear in* mts_k*;*

(←) 1. *x is not declared in* ads_k *nor as an attribute by a superclass or subclass of K in cds.*

The following law allows us to move a protected attribute *x* from a class *K* to a superclass *L*, and vice-versa. To move the attribute up to *L*, it is required that this does not generate a name conflict: no subclass of *L*, other than *K*, can declare an attribute with the same name.

Observe that the second proviso precludes an expression such as **self**.*x* from appearing in mts_l, but does not preclude **self**.*z*.*x* (which is sugar for **let** *y* **be self**.*z* **in** *y*.*x*), for an attribute $z : K$ declared in *L*. The latter expression is valid in mts_l no matter whether *x* is declared in *K* or in *L*.

Law 3 (*Move Attribute to Superclass*)**.**

Left		Right

class L ***ext*** N {
  ads_l
  mts_l
}
class K ***ext*** L {
  prot $x : T;\ ads_k$
  mts_k
}
	$=_{cds,c}$	***class*** L ***ext*** N {
		  prot $x : T;\ ads_l$
		  mts_l
		}
		class K ***ext*** L {
		  ads_k
		  mts_k
		}

(→) 1. *The attribute name x is not declared by the subclasses of L in cds;*

(←) 1. *attribute x does not appear in* mts_l *nor in declarations of subclasses of L, other than K, in cds, as part of any expression of the form y.x, where y is a variable of type M, for any* $M \leq L$ *and* $M \not\leq K$*.*

A method that is not called can be eliminated. Conversely, we can always introduce a new method in a class, provided we avoid naming conflicts.

Law 4 (*Method Elimination*)**.**

Left		Right

class K ***ext*** L {
  ads_k
  meth $m(\overline{y} : \overline{v}) : U\{c_1\};$
  mts_k
}
	$=_{cds,c}$	***class*** K ***ext*** L {
		  ads_k
		  mts_k
		}

(→) 1. *a method call u.m, with u of type M, such that* $M \leq K$*, cannot occur in cds, c nor in* mts_k*;*

(←) 1. *m is not declared in* mts_k *nor in any superclass or subclass of K in cds, in case that m is not a* ***ctor****.*

The next law changes the superclass of a class, from **Object** to an existing class, or vice-versa. The side conditions are numerous, but easy to follow.

Law 5 (*Change Superclass: From* **Object** *to Any Class*).

class K **ext** **Object** { ads_k mts_k }	$=_{cds,c}$	**class** K **ext** L { ads_k mts_k }

($\rightarrow$)
1. *Class L is declared in cds and all attributes in ads_k and in subclasses of K are distinct from those declared in L and in superclasses of L;*
2. *$K \neq$ **Void** and L is not be subclass of K;*
3. *Methods in mts_k and in subclasses of K that have the same name must have the same parameter declaration of methods declared or inherited by L;*

($\leftarrow$)
1. *K or any of its subclasses in cds is not used in type casts or tests involving any expression of type L or of any supertype of L;*
2. *There are no assignments of the form $x := e$, for any x whose declared type is L or any superclass of L and any e whose type is K or any subclass of K;*
3. *Expressions of type K or of any subclass of K are not used as value arguments in calls with a corresponding formal parameter whose type is L or any superclass of L;*
4. *Expressions whose declared type is L or any of its superclasses are not assigned to the result variable **res** of methods whose declared result is of type K or any subclass of K;*
5. *$w.x$, for any $w : K$, does not appear in cd_k, cds or c for any public or protected attribute x of L or of any of its superclasses;*
6. *There is no call $u.m$, for any u of type M, $M \leq K$, and any m, such that m is declared in L or in any of its superclasses, but not redefined in M, $M \leq K$.*

The following law is an important contribution of this section. It considers changing data representation in a hierarchy of classes. It allows us to relate two versions of the private and protected attributes in a hierarchy of classes, by means of a local coupling $\mathcal{R}$. The law requires that the coupling relation is a simulation: it is preserved by corresponding versions of methods (Definition 23).

Law 6 (*Data Refinement*).

cs	$=_{cds,c}$	cs'

($\leftrightarrow$)
1. *cs and cs′ are hierarchies with root Own, and cds has no subclasses of Own;*
2. *class tables cds, cs and cds, cs′ are comparable for class Own;*
3. *cds, cs is confined for Own, Rep;*
4. *cds, cs′ is confined for Own, Rep′;*
5. *$\Gamma \vdash c$ and $\Gamma \vdash' c$;*
6. *c is confined in cds, cs and also in cds, cs′;*
7. *$\mathcal{R}$ is a simulation.*

Proof. As class tables cds, cs and cds, cs' are confined and $\mathcal{R}$ is a simulation, then by Theorem 24 we have $\hat{\hat{\mathcal{R}}}\,\eta\,\eta'$ where $\eta = [\![cds\, cs]\!]$ and $\eta' = [\![cds\, cs']\!]$. According to Definition 7 (program and class equivalence) and abbreviation (3), we must show $[\![\Gamma \vdash c]\!](\eta) \doteq [\![\Gamma \vdash' c]\!]'(\eta')$, so consider any σ that is void-only. By Lemma 21 we have $\hat{\mathcal{R}}\ \Gamma\ \sigma\ \sigma$. In a main program, **self** has type **Void** (Section 2.2), so we have $\Gamma(\textbf{self}) \not\leq Own$ and $\Gamma(\textbf{self})$ is non-rep. Thus we can appeal to Theorem 27, whence by Definition 19 (coupling relation), we get $\hat{\mathcal{R}}\ \Gamma\ \alpha\ \alpha'$, where $\alpha = [\![\Gamma \vdash c]\!](\eta)(\sigma)$, $\alpha' = [\![\Gamma \vdash' c]\!]'(\eta')(\sigma)$. By Lemma 22 we have $\alpha(x) = \alpha'(x)$ for every variable in scope. Thus, by Definition 6 (visible equivalence of state transformers), we have $[\![\Gamma \vdash c]\!](\eta) \doteq [\![\Gamma \vdash' c]\!]'(\eta')$. □

6.2. Refactoring rules

In this section we present representative refactorings that illustrate systematic transformations in the context of complete class hierarchies. In some refactorings, like the first one in the sequel, only fields of the owner class are changed; no other object is affected. In such cases, to fit the confinement theory, we choose for *Rep* the class **None** that is never instantiated, so that the confinement conditions are vacuously true (Lemma 15).

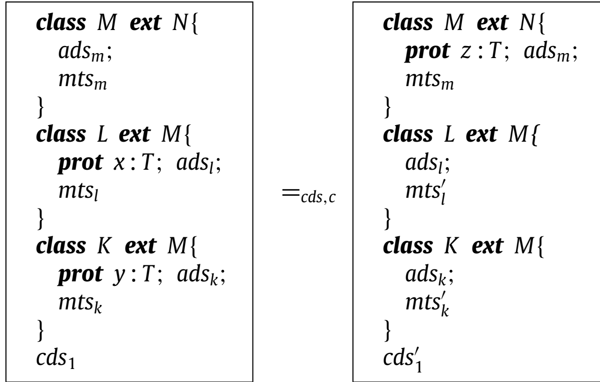
Subclasses developed independently can have attributes with the same purpose. These attributes may have different names, but if they have the same type and are used in a similar way, they can be unified, eliminating duplication. This

is the purpose of Refactoring 1 ⟨*pull up field*⟩. As the laws and refactorings are presented as equations, each one actually corresponds to two transformations. For example, Refactoring ⟨*pull up field*⟩, when applied from right to left, corresponds to ⟨*push down field*⟩ [18]. The names we have chosen capture the left to right transformation.

On the left-hand side of the rule, the classes L and K are subclasses of M. The class L declares an attribute x, whereas the class K declares an attribute named y. Both attributes have the same type. The sequence of class declaration cds_1 contains the subclasses of M other than K and L. On the right-hand side, attributes x and y are unified as attribute z. Occurrences of x and y in mts'_k, mts'_l and cds'_1 are renamed to z.

The transformation of cds_1 is schematic: the attribute x can occur in field reference expressions $u.x$ and assignments $u.x := \ldots$ for many different variables u. Rather than using special notation for schematic variables, we simply use the phrase "for every variable u", meaning that the schema is applied simultaneously for all u.

Refactoring 1 (*Pull up Field*).



where

$$mts'_k = mts_k[u.z,\ v.z/u.x,\ v.y], \textit{for every variable } u \textit{ of type } L_1, L_1 \le L \textit{ and every variable } v \textit{ of type } K_1, K_1 \le K;$$
$$mts'_l = mts_l[u.z,\ v.z/u.x,\ v.y], \textit{for every variable } u \textit{ of type } L_1, L_1 \le L \textit{ and every variable } v \textit{ of type } K_1, K_1 \le K;$$
$$cds'_1 = cds_1[u.z,\ v.z/u.x,\ v.y], \textit{for every variable } u \textit{ of type } L_1, L_1 \le L \textit{ and every variable } v \textit{ of type } K_1, K_1 \le K.$$

(↔) 1. *cds contains no subclasses of M;*

(→) 1. *z is not declared in* cd_m, cd_l, cd_k, *nor in any subclass or superclass of M in cds and* cds_1;

(←) 1. *x (resp., y) is not declared in* ads_m, ads_l *(resp.,* ads_k*), nor in any subclass or superclass of L (resp., K) in cds and* cds'_1;
2. *z does not occur in* mts_m.

Proof. We begin by transforming the left-hand side of the rule, by applying Law 3 ⟨*move attribute to superclass*⟩ twice (from left to right) to move attributes x and y to class M, reaching the following intermediate program. Observe that all methods are the same as in the program on the left-hand side; the only change is the promotion of attributes x and y to class M.

```
class M ext N{
  prot x : T;
  prot y : T;
  ads_m;
  mts_m
}
class L ext M{
  ads_l;
  mts_l
}
class K ext M{
  ads_k;
  mts_k
}
cds_1
```

We proceed by relating this intermediate program to the right-hand side of the rule, by instantiating Law 6 ⟨*data refinement*⟩ by $[(cd_m\ cd_l\ cd_k\ cds_1)/cs]$, $[(cd'_m\ cd'_l\ cd'_k\ cds'_1)/cs']$, and $[M, \textbf{None}, \textbf{None}/Own, Rep, Rep']$. Clearly, cs and cs'

are hierarchies with root Own and, by hypothesis, *cds* has no subclasses of Own. By Definition 16 (comparable class tables), *cds*, *cs* and *cds*, *cs*′ are comparable. We define a local coupling $\mathcal{R}$ as follows. For clarity we write it as a formula over **self** : M and **self**′ : M (i.e., instances of the old and new versions):

$$\begin{aligned} & type(\mathbf{self}) = type(\mathbf{self}') \wedge (\mathbf{self\ is}\ L \Rightarrow \mathbf{self}'.z = \mathbf{self}.x) \\ & \quad \wedge (\mathbf{self\ is}\ K \Rightarrow \mathbf{self}'.z = \mathbf{self}.y) \\ & \wedge \forall f \in fields(type(\mathbf{self})) \bullet f \neq x \wedge f \neq y \Rightarrow \mathbf{self}'.f = \mathbf{self}.f\,. \end{aligned}$$

This depends only on fields of M, so there are no rep objects that need to be confined. In order to use Law 6, however, we need to designate *Rep* and *Rep*′ for which the classes and c are confined. We choose **None** for both of them; then the programs are both confined according to Lemma 15. To prove that $\mathcal{R}$ is a simulation, we prove a stronger claim:

> For every sub-expression e of the original code (i.e., any method in the hierarchy rooted in M: mts_m, mts_l, mts_k, or cds_1), if e' is the transformed version (mts_m, mts'_l, mts'_k, cds'_1) then $[\![e]\!](\eta)$ relates to $[\![e']\!](\eta')$ via $\hat{\mathcal{R}}$, for all η, η' with $\hat{\mathcal{R}}\eta\eta'$.

Furthermore, we prove a corresponding claim for commands. The proofs go by structural induction and rely on particular conditions in $\mathcal{R}$. To be precise, we need to consider expressions in context, noting that if $\Gamma \vdash e : U$ for some Γ and U then $\Gamma \vdash e' : U$.

Note that the claims are similar to Lemmas 25 and 26. There are two differences: here we consider in addition the case that $\Gamma(\mathbf{self}) \leq Own$, and in general e' here is not identical to e. We consider the interesting cases, sketching the main points and omitting routine use of the definitions as these should be clear to readers familiar with the proofs of Lemmas 25 and 26. In each case we suppose $\hat{\mathcal{R}}\ \Gamma\ \sigma\ \sigma'$ and $\hat{\mathcal{R}}\eta\eta'$.

Consider the case of field access to x, i.e., e is $w.x$ for some variable w. Thus e' is $w.z$. By semantics and $\hat{\mathcal{R}}\ \Gamma\ \sigma\ \sigma'$, either $\sigma(w)$ and $\sigma'(w)$ are both null or neither is. In case both are null, the outcomes $[\![w.x]\!]$ and $[\![w.z]\!]$ are both $\perp$ and hence related by $\hat{\mathcal{R}}$. If neither is null then using $\hat{\mathcal{R}}\ \Gamma\ \sigma\ \sigma'$ we have $\sigma(w) = o = \sigma'(w)$ for some o of type $\leq M$, and there is some i such that o references the objects in Oh_i and Oh'_i, using the partitions obtained in accord with $\hat{\mathcal{R}}\ \Gamma\ \sigma\ \sigma'$. Moreover Oh_i and Oh'_i are connected by $\mathcal{R}$. By definition of $\mathcal{R}$ we get $\sigma(o.x) = \sigma'(o.z)$, concluding the proof that $\hat{\mathcal{R}}\ ([\![w.x]\!](\eta)(\sigma))\ ([\![w.z]\!](\eta')(\sigma'))$.

The case of field access to y is similar to that of x. In case e is $w.f$ for f distinct from x and y, e' is $w.f$. Using $\hat{\mathcal{R}}\ \Gamma\ \sigma\ \sigma'$ we get in the non-null case that $\sigma(w) = o = \sigma'(w)$ for some o. Now we distinguish two sub-cases, in both of which the values $\sigma(o.f)$ and $\sigma'(o.f)$ are the same: If o has type $\leq M$ then we rely on the specific definition of $\mathcal{R}$ as in the previous paragraph; otherwise, o is not in an island and we rely on the general definition of $\hat{\mathcal{R}}$ (Definition 19).

For other expression forms the argument is straightforward, using induction on e for **let** expressions.

For commands, the interesting cases are $w.x := v$ and $w.y := v$. These are transformed to $w.z := v$. As in the case of field access $w.x$, $\sigma(w)$ and $\sigma'(w)$ may be null, in which case both commands yield $\perp$. Otherwise, there is some o with $\sigma(w) = o = \sigma'(w)$ and i with o referencing the objects in Oh_i and Oh'_i. We have $\sigma(v) = \sigma'(v)$ so the update of $\sigma(o.x)$ to $\sigma(v)$ and $\sigma'(o.z)$ to $\sigma'(v)$ is just what is required to maintain $\mathcal{R}$ on Oh_i and Oh'_i, whence the updated states are related by $\hat{\mathcal{R}}$. □

Method names play an important role concerning the legibility of object-oriented systems. Refactoring 2 ⟨*rename method*⟩ allows us to change the name of a method. On the left-hand side of this refactoring, there is a method named m. This method can be called at any point in the whole program. Applying this refactoring the name of m is changed to n. Renaming a method inside a class affects not only the clients of such class, but also the classes in the same hierarchy in which the method is present.

Refactoring 2 (*Rename Method*).

class K ***ext*** L {
 ads_k;
 meth $m(\overline{z} : \overline{T}) : U\{c_1\}$
 mts_k
 }
 $cds_1\ cds_2 \bullet c$
	=	***class*** K ***ext*** L {
		 ads_k;
		 meth $n(\overline{z} : \overline{T}) : U\{c'_1\}$
		 mts'_k
		 }
		 $cds'_1\ cds'_2 \bullet c'$

where

cds_1 contains all the declarations of subclasses of K;
cds_2 contains all other class declarations than those in cds_1, and might include calls to m;
for all $x : M$, $M \leq K$ the following substitutions hold:
$c'_1 = c_1[x.n/x.m]$;
$cds'_1 = cds_1\ [\mathbf{meth}\ n(\overline{z} : \overline{T}) : U\ \{c_1[x.n/x.m]\},\quad x.n/\ ;$
$\qquad\qquad \mathbf{meth}\ m(\overline{z} : \overline{T}) : U\ \{c_1\},\qquad x.m]$
$cds'_2 = cds_2[x.n/x.m]$;
$mts'_k = mts_k[x.n/x.m]$;
$c' = c[x.n/x.m]$.

(→) 1. *L does not declare or inherit a method named m;*
2. *n is not declared in mts_k nor in any subclass or superclass of K in cds_1 and cds_2;*

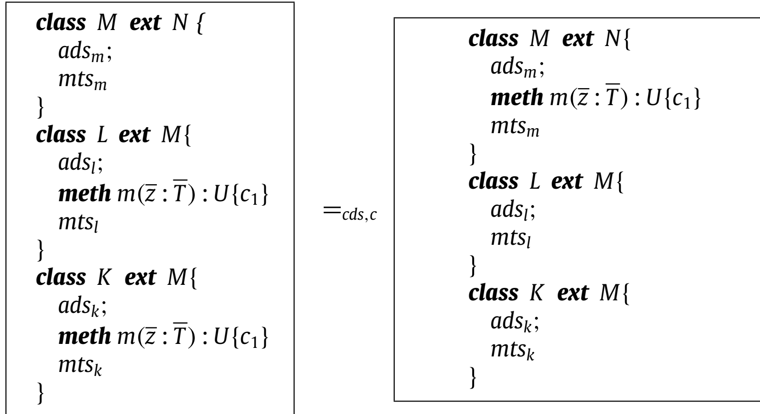
(←) 1. *L does not declare or inherit a method named n;*
2. *m is not declared in mts'_k nor in any subclass or superclass of K in cds'_1 and cds'_2.*

Proof. For pairs M, p of a class name and method name, let $\sim$ be the bijection with $M, m \sim M, n$ in case $M \leq K$, and otherwise $M, p \sim M, q$ iff $p = q$. Let $\eta = [\![cds_1\, cds_2]\!]$ and $\eta' = [\![cds'_1\, cds'_2]\!]$. We claim that for every M, p, q with $M, p \sim M, q$ we have $\eta(M, p) = \eta(M, q)$, i.e., they are identical state transformers. Using the claim, we can show $[\![c]\!](\eta) = [\![c']\!](\eta')$ by structural induction on c. The point is that c' is just c except for using from η' the method named n that has the same denotation as the one named m in η. Note that we show c and c' denote identical state transformers, which implies the visible equivalence $[\![c]\!](\eta) \doteq [\![c']\!](\eta')$ required by Definition 7 for this refactoring.

The claim is proved by fixpoint induction, based on a second claim: that $\eta_i(M, p) = \eta_i(M, q)$ for each i, where these are the environments in the approximation chains defining $[\![cds_1\, cds_2]\!]$ and $[\![cds'_1\, cds'_2]\!]$ (Definition 4). To prove the second claim, consider any M, p, q with $M, p \sim M, q$. If M declares p with body d (in $cds_1\, cds_2$) then the body of M, q is d' with m renamed to n, and we get $[\![d]\!](\eta_i) = [\![d']\!](\eta'_i)$ by structural induction on d as for c above. □

In a class hierarchy, methods having the same body, in different branches of the hierarchy, may be condensed in a single method of the common superclass. This is the purpose of Refactoring 3 ⟨*pull up method*⟩. When applying this rule from left to right, we move methods with the same definition to the superclass of the classes. In practice, the most common use of this rule involves its application for pulling up methods. On the other hand, if a method is called just on objects of a particular subclass, we can push the method down and then remove it from classes whose objects are not target of calls to the method.

Refactoring 3 (*Pull up Method*).



(↔) *private attributes do not appear in c_1;*

(→) 1. *m is not declared in any superclass of M in cds;*
2. *m is not declared in mts_m, and can only be declared in a class M_1, for any $M_1 \leq M$, if it has parameters $\overline{z} : \overline{T}$;*
3. *protected attributes declared in ads_l and in ads_k do not appear in c_1;*

(←) 1. *m is not declared in mts_k or mts_l;*
2. *$u.m$, for any expression u of type $M_1, M_1 \leq M, M_1 \not\leq K, M_1 \not\leq L$, does not appear in cds, c, mts_m, mts_k or mts_l.*

Proof. Let $\eta = [\![cds\, cd_k\, cd_l\, cd_m]\!]$ and $\eta' = [\![cds\, cd'_k\, cd'_l\, cd'_m]\!]$ where cd'_k is the version of K on the right (and similarly for L and M). Note that η' is defined on M, m whereas η is not. We claim that for every class J and every $p \in Meths(J)$, except for $J = M$ and $p = m$, we have $\eta(J, p) = \eta'(J, p)$. From the claim it follows that $[\![c]\!](\eta)$ is identical to $[\![c]\!](\eta')$, because the provisos ensure that the semantics of the main program c does not depend on the meaning of m in M.

The claim is proved by fixpoint induction, based on a similar claim for the approximate method environments. To prove the second claim, for approximants η_i and η'_i, the key point is that $\eta_{i+1}(L, m)$ is defined to be $[\![c_1]\!](\eta_i)$, but in turn $\eta'_{i+1}(L, m)$ is defined to be $\eta'_{i+1}(M, m)$ where $\eta'_{i+1}(M, m)$ is defined to be $[\![c_1]\!](\eta'_i)$. Similarly for K, m. For other classes/methods, the declarations are identical. □

Refactoring 4 ⟨*encapsulate field*⟩ hides a public attribute and provides get and set methods for it. In the original version of this refactoring, as presented in [18], the attribute visibility is changed to private. As with other refactorings, we generalize this transformation to consider protected attributes instead.

The class K on the left-hand side of the refactoring includes a public attribute x. The context for this class is the sequence of classes cds_1, cds_2 and the command c. In class K, on the right-hand side, the attribute x is protected and get and set

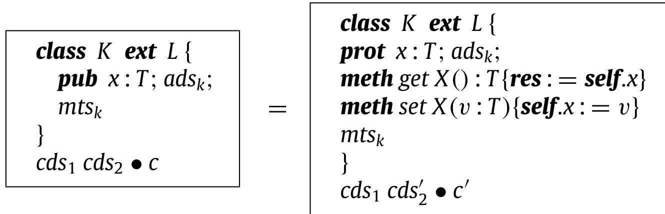
methods are declared. The context for class K is the sequence of classes cds'_1, cds'_2 and the command c'. Direct accesses to x by client classes are replaced with calls to get and set methods on the right-hand side. Nevertheless, for subclasses of K, direct accesses to x are still allowed.

To apply this refactoring from left to right, the methods *get X* and *set X* must not be already declared in K nor in any of its superclasses or subclasses.

This refactoring leads to changes in the context of the class to which the refactoring is applied since all direct accesses by clients to a previously public, and now protected, attribute must now be indirect, by using get and set methods. Assignments of the form $w.x := u$, with $w : K_1$ for $K_1 \leq K$, are replaced by $w.set\,X(u)$. Assignments of the form $u := w.x$ are replaced with calls $u := w.get\,X()$. More generally field accesses $w.x$ on the right side of a variable assignment $u := \ldots$ are replaced by $w.get\,X()$, which we express schematically: the substitution of $u := e[w.get\,X()]$ for $u := e[w.x]$ is intended to mean that every subexpression $w.x$ of e becomes $w.get\,X()$.

For clarity, we give the substitutions using syntax sugar (the call $w.set\,X(u)$ as a command), so the complete transformation includes a desugaring not shown. As always, the substitution is to be applied to un-sugared code.

Refactoring 4 (*Encapsulate Field*).



where

cds_1 contains the declarations of the subclasses of K, and cds_2 the remaining classes.
For all $w : K_1$, such that $K_1 \not\leq K$, and all e and u we have:
$cds'_2 = cds_2[w.set\,X(u),\ u := e[w.get\,X()]\ /\ w.x := u,\ u := e[w.x]];$
$c' = c[w.set\,X(u),\ u := e[w.get\,X()]\ /\ w.x := u,\ u := e[w.x]].$

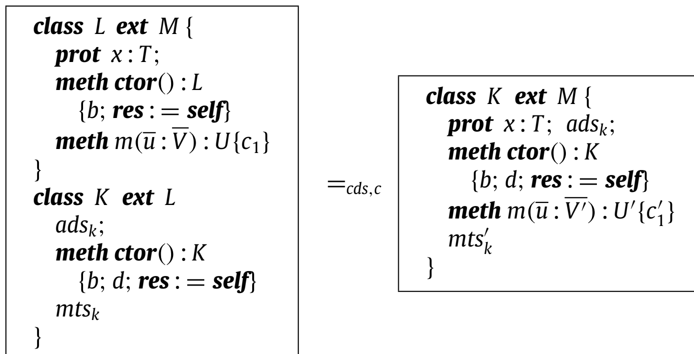
$(\rightarrow)$ *get X and set X are not declared in K nor in any superclass or subclass of K.*

Proof. In this situation, the semantic domains are the same for the two versions: attribute x is present in objects of type K regardless of its visibility. Let $\eta = [\![cd_k\, cds_1\, cds_2]\!]$ and $\eta' = [\![cd'_k\, cds_1\, cds'_2]\!]$. We claim that $\eta(M, m) = \eta'(M, m)$ for all classes M and all methods $m \in Meths(M)$ except *get X* and *set X* in $M \leq K$. Observe also that the denotations of *get X* and *set X* in classes $M \leq K$ do nothing more or less than getting and setting the value of x, as the code is stipulated explicitly in cd'_k and by proviso there are no overrides in subclasses. The proof of the claim uses a similar claim for the approximate method environments. That in turn uses the observation, from which it follows that the transformed versions have identical denotations to the originals.

Finally, from the claim we get $[\![c]\!](\eta) = [\![c']\!](\eta')$, as c does not use *get X* or *set X* in c' and we can again use the observation to connect calls to *get X* and *set X* in c' to accesses/updates in c. □

After moving attributes and methods up or down in a hierarchy, a class may end up not adding any valuable feature. Such a class can be merged with another class, resulting in an empty class that can then be removed. This is the purpose of Refactoring 5 ⟨*collapse hierarchy – subclass*⟩. In this refactoring the attributes and methods of a class L are moved to a subclass K in the hierarchy, according to some conditions. In a richer language, the command b in class K would be a super call.

Refactoring 5 (*Collapse Hierarchy – Subclass*).



where

$$mts'_k = mts_k[K/L];\ \overline{V}' = \overline{V}[K/L];\ U' = U[K/L];\ c'_1 = c_1[K/L]$$

($\rightarrow$) 1. *m is not declared in mts_k;*
2. *Class L is not referenced anywhere in cds, c or inside cd_k;*

($\leftarrow$) 1. *Class L is not declared in cds.*

Proof. Let cd'_k be the declaration of K on the right side. Note that the semantic domains for $cds\ cd_k\ cd_l$ are slightly different from the domains for $cds\ cd'_k$, because the states in the latter do not include references of type L. The provisos ensure that the programs do not create objects of type L. But we want to reason compositionally. A method, say p, in mts_k (and its subclasses) denotes a state transformer where the values for **self** (and parameters of type $\leq K$) include references of (exactly) type L, and the heap contains objects of (exactly) type L. So it cannot be directly compared with the denotation of p in the class table of the right side.

What we can do is define a suitable inclusion of states. For example, let $\Gamma_0 = [\textbf{self} : K]$ and define inc to be the inclusion of $State'(\Gamma_0)$ into $State(\Gamma_0)$.

Let $\eta = [\![cds\ cd_k\ cd_l]\!]$ and $\eta' = [\![cds\ cd'_k]\!]'$. We claim that $\eta(K, p) \circ inc = \eta'(K, p)$ for every method p in K (including $p = m$). Indeed, we define suitable inclusion functions for all contexts Γ and claim $\eta(J, p) \circ inc = \eta'(J, p)$ for every class J and method p in $Meths(J)$. From the claim it follows that $[\![c]\!](\eta) = [\![c]\!](\eta')$ by the proviso on c.

Of course, to prove the claim we need a similar claim for the approximate method environments. The argument then goes by structural induction on method bodies, using that in the version $cds\ cd_k\ cd_l$ objects of type L are never created owing to provisos of the law. □

A common and crucial practice in object-oriented software development is the identification of *hidden* abstractions inside a model (typically a class), and the proper representation of such abstractions as independent units (other classes). This is the purpose of Refactoring 6 ⟨*extract class*⟩. When applied from left to right, this refactoring creates a new class L with some attributes and methods from an original class K, which is then transformed to include an attribute of type L. In K, original direct accesses to attributes that are now in L are replaced with calls to the get and set methods of L. Original methods of K that act on the attributes that were moved to L are moved to L as well; in K we keep delegating methods: they just call the corresponding methods of L. Particularly, the method m_1 acts only on the attribute x, indicating that it is an operation on the attribute x. Therefore, it is part of the interface of L.

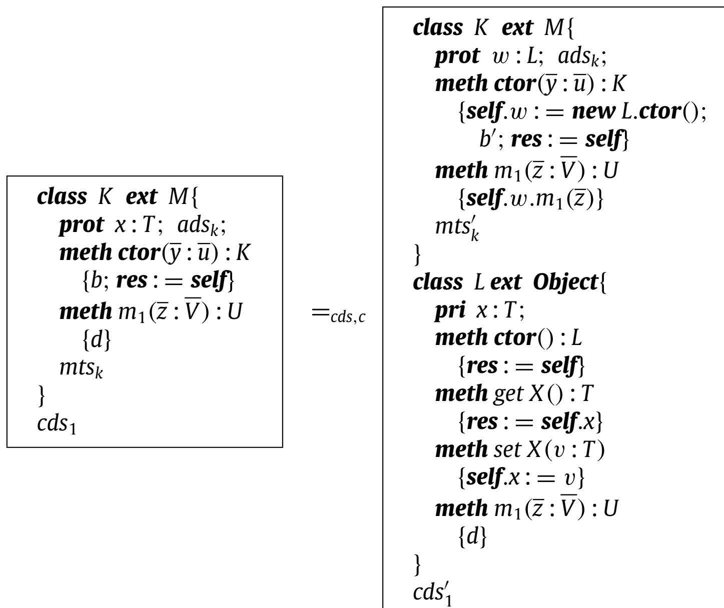
As in the previous refactoring, both on the left- and on the right-hand side of this refactoring we single out the subclasses of K that might also be affected by these changes, since x is a protected attribute, unlike in the original version proposed by Fowler where it is a private field. The modifications in the subclasses are the same as those in the methods of class K.

As in Refactoring ⟨*encapsulate field*⟩, note that we use a schematic substitution: to say every subexpression $v.x$ in expression e gets replaced by $v.w.get\,X()$ we write $[u := e[v.w.get\,X()]/u := e[v.x]$. Furthermore, we use the sugared expression $v.w.get\,X()$ for clarity, but the transformation also includes desugaring.

Without the restriction on parameter types, d could mention say $z.x$, which would not make sense in class L. The restriction can be dropped by insisting that K declare $set\,X$ and $get\,X$ methods, and d would be rewritten using them.

This rule is unusual in that, in one direction, we need to impose confinement as an explicit proviso.

Refactoring 6 (*Extract Class*)**.**



where *for every* u, v *we replace*
$mts'_k = mts_k[u := e[v.w.get\,X()],\ v.w.set\,X(u)\ /\ u := e[v.x],\ v.x := u]$
$cds'_1 = cds_1[u := e[v.w.get\,X()],\ v.w.set\,X(u)\ /\ u := e[v.x],\ v.x := u]$
$b' = b[u := e[v.w.get\,X()],\ v.w.set\,X(u)\ /\ u := e[v.x],\ v.x := u]$
and in addition, in cds'_1 *every* ***ctor*** *has the following prefixed to the beginning of its body:* $\textbf{self}.w := \textbf{new}\ L.\textbf{ctor}()$;

($\leftrightarrow$)
1. *None of the parameter types* $\overline{V}$ *is a subtype of* K*;*

($\rightarrow$)
1. *The class* L *is not declared in cds, nor in* cds_1*;*
2. *The attribute* w *is not declared in* ads_k *nor in any subclass or superclass of* K*;*
3. *Except for* x*, no attribute of* K *occurs in* d*;*

($\leftarrow$)
1. *The name* L *does not occur in cds or* c*;*
2. $cds\ cds'_1\ cd_k\ cd_l \bullet c$ *is confined for* $[K, L/Own, Rep]$.

Proof. Begin with the left-hand side and apply Law 1 ⟨*class elimination*⟩ to add the declaration cd_l of class L. Now we have comparable class tables and we can use Law 6 ⟨*data refinement*⟩, which we instantiate by $[(cd_k, cd_l, cds_1)/cs]$, $[(cd'_k, cd_l, cds'_1)/cs']$, $[K, \textbf{None}, L/Own, Rep, Rep']$. Confinement of the left-hand side for K, **None** holds by Lemma 15. Confinement of the right-hand side for K, L holds for the following reasons. If the law is applied right-to-left, it is an explicit proviso. If the law is applied left-to-right, it should be clear that the transformed version maintains the invariant that the object referenced from w is never shared.[15] We define a local coupling $\mathcal{R}$ as follows:

$$\begin{aligned} &\textbf{self} = \textbf{self}' \wedge type(\textbf{self}) = type'(\textbf{self}') \\ &\wedge \textbf{self}.x = \textbf{self}'.w.x \wedge type'(\textbf{self}'.w) = L \\ &\wedge \forall f \in fields(K) \bullet f \neq x \Rightarrow \textbf{self}.f = \textbf{self}'.f\,. \end{aligned}$$

Notice that this depends on[16] the value of $\textbf{self}.w.x$, so confinement is necessary. It remains to prove that this is a simulation. To this end we consider any η_i, η'_i in the semantic approximation chains[17] and assume $\hat{\mathcal{R}}\ \eta_i, \eta'_i$. We consider four main cases, as the transformation acts differently on different methods.

The first main case is for the methods of class L. These are identical on both sides (see the definition of cs and cs'), and moreover the type of **self** is $\not\leq Own$ (i.e., $\not\leq K$). So we obtain the simulation condition directly by Lemma 26.

The second main case is for method m_1 in class K. Consider related states σ, σ'. Owing to the provisos, in the left-hand version the body d only accesses the x field of **self** (but not other instances of K), and no other fields of **self**. The right-hand version of m_1 in K consists of the invocation $\textbf{self}.w.m_1(\overline{z})$ of method m_1 in L, and that method has the same body d but interpreted so that $\textbf{self}.x$ refers to attribute x of L. So the effect on $\textbf{self}.x$ and $\overline{z}$ in the left version is the same as the effect on $\textbf{self}.w.x$ and $\overline{z}$ in the right version, as needed to preserve the relation.

The third main case is for constructors. We transform

$$\textbf{meth ctor}(\overline{y} : \overline{u}) : K\{b;\ \textbf{res} := \textbf{self}\}$$

in class K to

$$\textbf{meth ctor}(\overline{y} : \overline{u}) : K\{\textbf{self}.w := \textbf{new}\ L.\textbf{ctor}();\ b';\ \textbf{res} := \textbf{self}\}$$

where b' is b with x replaced by $w.get\,X()$ or $w.set\,X(\ldots)$ as appropriate. Constructors of subclasses of K are transformed following the same pattern. For the simulation property we must show that the constructors establish $\hat{\mathcal{R}}$. In states σ, σ' with $\hat{\mathcal{R}}\,\sigma\sigma'$, consider the states τ and τ' extending σ, σ' with a fresh reference o mapped to new K objects. Extend the given confining partitions with a new pair of islands for o. Let τ'' be obtained from τ' by applying $\textbf{self}.w := \textbf{new}\ L.\textbf{ctor}()$. By definition of $\mathcal{R}$ we get $\hat{\mathcal{R}}\,\tau\,\tau''$. From these states we execute b, respectively b'. To complete the argument that the outcomes are related, we proceed just as in the fourth main case.

The fourth main case is for methods of classes other than L, aside from constructors and aside from m_1. We claim that for every sub-expression e of one of these methods, we have $\hat{\mathcal{R}}\,([\![e]\!](\eta_i))([\![e']\!](\eta'_i)')$ and similarly for commands. The claim is proved by induction on the typing derivation of the expression or command, as is the similar claim used to prove ⟨*pull up field*⟩. The interesting cases are those where the transformation changes the code. We consider these in turn, in initial states σ, σ' with $\hat{\mathcal{R}}\,\sigma\sigma'$.

- For an assignment $u.x := e$, the transformed version is $u.w.set\,X(e)$. Owing to the condition $type'(\textbf{self}'.w) = L$ in the coupling, the call $set\,X(e)$ dispatches in σ' to the method declared in class L, which sets the x field to the value of e, whereas in σ the assignment $u.x := e$ sets x to the value of e. So $\hat{\mathcal{R}}$ holds on the final states.

---

15 One way to prove this formally would be via a static analysis adapted from the one in [5] and then show by structural induction on the left-hand program that the transformed version satisfies the constraints of that analysis.

16 As well as the type of $\textbf{self}'.w$, though that is immutable and so not a confinement issue.

17 Unlike in the proof of ⟨*pull up field*⟩, where we can consider any environments, here we rely on the specific semantics of some method calls.

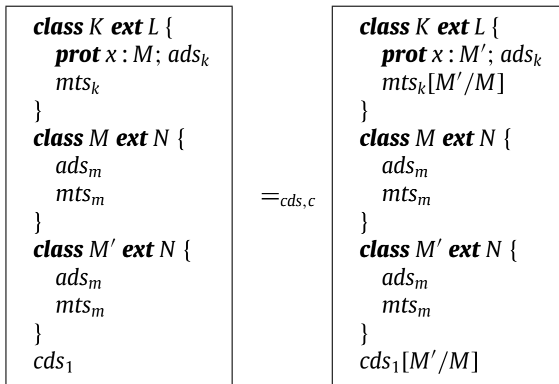
- For an expression $u.x$, the transformed version is $u.w.get\,X()$. The call $get\,X()$ dispatches in σ' to the method declared in class L, which retrieves the value of field x, which is according to $\hat{\mathcal{R}}$ the value of $u.x$ in σ. □

In order to support type reuse, it should be possible to change the type of an attribute with another structurally equivalent one, but with a different name. This is the purpose of Refactoring 7 ⟨*change name of attribute type*⟩. When applied from left to right, its effect is to change the type of attribute x, in class K, from M to M'; occurrences of M in the methods of class K are also replaced with M'. As with other refactorings, we state this transformation for protected attributes, and so the replacement of M with M' is performed on subclasses cds_1 of class K as well. Although this refactoring is of more general utility, it is a particular useful complement to ⟨*extract class*⟩ when the class to be extracted already exists, as illustrated in two circumstances in the case study developed in the next section.

The side conditions ensure that the transformation is valid by carrying out a simple replacement of M with M' (or the other way round, for the reverse application). The idea is that attribute x can be initialized but not otherwise written. Its fields can be read and written, and methods invoked on it, but its value does not flow to other variables, fields, or parameters. For example, assuming that y has type M, an assignment like $\mathbf{self}.y := x$ in class K, on the left-hand side of the refactoring, might not be well-typed on the right-hand side, where x has type M'.

In reading the provisos, please keep in mind that in our syntax (Fig. 1) the attribute x only occurs after a dot as in $u.x$. It cannot occur as a bare identifier as in equality test $x = y$, nor type test or cast on x, nor in a bare assignment $x := y$.

Refactoring 7 (*Change Name of Attribute Type*).



(→)

1. *cds has no subclass of K and N is not a subclass of K;*
2. *The only occurrences of M in mts_k and cds_1 are in initializations $w.x := (\mathbf{new}\ M).\mathbf{ctor}(\ldots)$ of x and in the variable declarations for assignments to fields, item 5 below. In our syntax, the initializations take the form $\mathbf{var}\ y : M\ \mathbf{in}\ y := (\mathbf{new}\ M).\mathbf{ctor}(\ldots); w.x := y$ for some w, y;*
3. *There is no occurrence of M in ads_k;*
4. *Initialization of x is restricted to objects of the* exact *type M; objects of subclasses of M are not allowed;*
5. *The only occurrences of x in mts_k and cds_1 are*
   - *assignments to fields a, which in our desugared syntax take the form $\mathbf{var}\ y : M\ \mathbf{in}\ y := w.x; y.a := u$ for some y, w, u;*
   - *reading of fields a; in our syntax, reads occur in expressions of the form $\mathbf{let}\ y\ \mathbf{be}\ w.x\ \mathbf{in}\ y.a$, for some y, w;*
   - *invocation of methods; in our syntax, calls occur in expressions of the form $\mathbf{let}\ y\ \mathbf{be}\ w.x\ \mathbf{in}\ y.m(\overline{z})$, for some $y, w, \overline{z}$.*

(←)

1. *Same conditions as above, replacing M with M'.*

Proof. We instantiate Law 6 ⟨*data refinement*⟩ by $[K, M, M'/Own, Rep, Rep']$. The provisos above impose strong restrictions on the use of field x; as a result, it is a unique reference in the sense that there is no sharing in the heap and moreover the value of x (i.e., of some $o.x$) is never stored in a variable except in local blocks of the form $\mathbf{var}\ y : M\ \mathbf{in}\ y := w.x; y.a := u$ where it is used only to access attribute a. A consequence is that the left-hand side program is confined for K, M and the right for K, M'.

We define a local coupling $\mathcal{R}$ by

$$\mathbf{self} = \mathbf{self}' \wedge type(\mathbf{self}) = type'(\mathbf{self}') \wedge \forall f \in ads_m \bullet \mathbf{self}.x.f = \mathbf{self}'.x.f \\ \wedge type(\mathbf{self}.x) = M \wedge type'(\mathbf{self}'.x) = M'.$$

Note that we do not require $\mathbf{self}.x = \mathbf{self}'.x$. It remains to prove that this is a simulation. To this end we consider any η, η' such that $\hat{\mathcal{R}}\,\eta\,\eta'$. We must show, for any method declaration mt in mts_k or cds_1 that the semantics of mt is related to the

semantics of $mt[M'/M]$. We claim that for every sub-expression e of one of these methods, we have $\hat{\mathcal{R}}\,([\![e]\!](\eta_i))([\![e']\!](\eta_i')')$, where e' is the transformed version, and similarly for commands. The claim is proved by induction on the typing derivation of the expression or command, as is the similar claim used to prove ⟨*pull up field*⟩. The interesting cases are those where x or M is involved. We consider these, in initial states σ, σ' with $\hat{\mathcal{R}}\,\sigma\sigma'$, to show $\hat{\mathcal{R}}\,([\![e]\!](\eta_i)(\sigma))([\![e']\!](\eta_i')'(\sigma'))$ for expressions and $\hat{\mathcal{R}}\,([\![c]\!](\eta_i)(\sigma))([\![c']\!](\eta_i')'(\sigma'))$ for commands.

An initialization **var** $y : M$ **in** $y := ($**new** $M)$.**ctor**$(\ldots)$; $w.x := y$ in mts_k or cds_1 is transformed by renaming M to M'. This establishes the coupling for $w.x$, as the new object has type M or M' accordingly. The constructors of M and M' are identical, so the field values are equated as required by $\mathcal{R}$.

Initialization of x to a subtype of M would be problematic but is disallowed by proviso 4. Moreover, item 2 disallows M used in casts or type tests in mts_k and cds_1.

A field assignment **var** $y : M$ **in** $y := w.x$; $y.a := u$ in mts_k or cds_1 gets M changed to M' but this does not affect the semantics, and the value of $y.a$ ends up the same because σ and σ' agree on the value of u.

Invocation of some method m on x, written **let** y **be** $w.x$ **in** $y.m(\overline{z})$, is not transformed. The two semantics are related because, by $\hat{\mathcal{R}}\,\sigma\,\sigma'$ the types of x are M, M' respectively, and these two classes provide identical code for m.

For code not in subclasses of K, we appeal to Theorem 27. □

We presented some representative laws and refactorings, as well as addressed soundness of the data refinement law and of the refactorings. These can then be used for program transformation in an algebraic style.

7. Case study

To illustrate a sequence of program transformations involving change of data representation in class hierarchies, we refactor a simple bank account application, particularly focused on transactions log and statement generation. The first subsection presents the initial version of the program. The second one shows the target program we aim to systematically obtain. The final sections give the development steps in detail.

7.1. Original version

Consider the program below, called here *original version*. The description is sugared for readability. In this example, a bank application has two separated parts dealing with transactions. The `CreditTrans` transaction increments the balance of an account when a value is deposited, whereas `PayCardTrans` transaction decrements the balance when a value is drawn out. Both transactions register the date and time the transaction is performed. In `CreditTrans` there are two public attributes for doing that, whereas in `PayCardTrans` this information is encapsulated as an object of class `LogDate`; and `PayCardTrans` has methods get and set to access this information. Attributes `cbal` and `pbal` store the balance before the transaction in `CreditTrans` and `PayCardTrans`, respectively. Attributes `cvalue` and `pvalue` store the amount of money to be deposited or drawn out, respectively. All classes have a constructor, named `ctor`.

Moreover, consider class `Statement` that uses lists of transactions `CreditTrans` and `PayCardTrans` to construct the statement of an account. We assume that these lists are ordered by date and time. These lists are constructed by invoking the method `addTrans`, which itself calls `addCreditTrans` or `addPayCardTrans`, depending on the transaction type.

The statement is recorded as a list of objects `LObject`, comprising transactions of type `CreditTrans` and `PayCardTrans`. Transactions in the statement are ordered by date and time and the method `merge` is responsible to merge the ordered lists. It uses the method `leqDateTime` to determine the order of occurrence of two transactions. Moreover, we assume that there exists a built-in function `toString` that prints the elements of list `LObject`. For simplicity, we omit the code of methods `merge`, `addCreditTrans` and `addPayCardTrans`.

The main program creates an instance of class `Statement` and iteratively stores the relevant transactions, by calling the method `addTrans`. As a final action it invokes the method `showStatement` of class `Statement`.

```
class CreditTrans ext Object {
   prot cbal: int;
   prot cvalue: int;
   pub cdate: Date;
   pub ctime: Time;
   meth ctor(): CreditTrans {res:= self}
   meth finalBal():int  {res:= self.cbal + self.cvalue}
}
class PayCardTrans ext Object {
    prot pbal: int;
    prot pvalue: int;
    prot pld: LogDate;
    meth ctor(): PayCardTrans {
       self.pld:= (new LogDate).ctor(); res:= self }
    meth getDatePay(): Date {res:= self.pld.getD()}
    meth getTimePay(): Time {res:= self.pld.getT()}
    meth setDatePay(d: Date) {self.pld.setD(d)}
```

```
    meth setTimePay(t: Time) {self.pld.setT(t)}
    meth finalBal():int {res:= self.pbal - self.pvalue}
}
class LogDate ext Object {
    pri dd: Date;
    pri tt: Time;
    meth ctor(): LogDate {res:= self}
    meth getD(): Date {res:= self.dd}
    meth getT(): int {res:= self.tt}
    meth setD(d: Date) {self.dd:= d}
    meth setT(t: Time) {self.tt:= t}
}
class ListCredit ext LObject {
    pri ct: CreditTrans;
    meth ctor(): ListCredit {res:= self}
    meth getCT(): CreditTrans {res:= self.ct}
    meth setCT(ct1: CreditTrans) {self.ct:= ct1}
}
class ListPayCard ext LObject {
    pri pc: PayCardTrans;
    meth ctor(): ListPayCard {res:= self};
    meth getPC(): PayCardTrans {res:= self.pc)}
    meth setPC(pc1: PayCardTrans) {self.pc:= pc1}
}
class LObject ext Object {
    prot next: LObject;
    meth ctor():Lobject {res:= self}
    meth getNext():LObject {res:= self.next}
    meth setNext(n1: LObject) {self.next:= n1}
}
class Statement ext Object{
    pri lc : ListCredit;
    pri lp: ListPayCard;
    meth ctor (): Statement {res:=self}
    meth addCreditTrans(ct: CreditTrans){
      /* add an element in lc ordered by date and time */
    }
    meth addPayCardTrans(pc: PayCardTrans){
      /* add an element in lp  ordered by date and time */
    }
    meth addTrans(t: Object){
       if (t is CreditTrans)
       then self.addCreditTrans((CreditTrans) t)
       else if (t is PayCardTrans)
            then self.addPayCardTrans((PayCardTrans) t)
            else // some error message
    }
    meth leqDateTime: bool (l1: ListCredit, l2:ListPayCard) {
      /* returns true if l1 occurs earlier than l2 */
       res:= false;
       if (l1.getCT().cdate <  l2.getPC().getDatePay()) or
          (l1.getCT().cdate = l2.getPC().getDatePay() and
          l1.getCT().ctime ≤ l2.getPC().getTimePay()) then
          res:= true
    }
    meth merge: LObject (l1: ListCredit, l2: ListPayCard) {
      /* returns the merge of l1 and l2 in res */
    }
    meth showStatement(): primString {
       res:= self.merge(lc,lp).toString()
    }
}
// Main program skeleton
var st: Statement in
var t: Object in
  st:= (new Statement).ctor();
  while /* there are transactions */ do
    t:= /* get a new transaction */
    st.addTransactions(t);
  od
  st.showStatement()
```

Here we use a loop for clarity. For brevity, our formal theory allows recursive methods but does not include loops.

7.2. Final version

Our aim is to improve the design to reach the final program shown next, called *final version*. By applying sound laws and refactorings from the previous section, we progressively justify that, despite the several structural changes, the main program in the final version has the same behavior as the original one.

In this program, classes `CreditTrans` and `PayCardTrans` are subclasses of class `Transaction`. The original attributes of classes `CreditTrans` and `PayCardTrans` are now unified as attributes of class `Transaction`. Transaction also has methods set and get to provide access to the date and time elements of `dt`.

Note that the balance and value attributes now have type `FValue`, which encapsulates a financial value with addition and subtraction operations. FValue keeps the internal representation of a value as an integer, for simplicity. The homonymous methods named `finalBal` of `CreditTrans` and of `PayCardTrans` are then implemented in terms of operations provided by `FValue`. In practice, this class could be implemented using, for instance, Java's `BigDecimal`.

Lists of `CreditTrans` and `PayCardTrans` are replaced by a single list of `Transaction`, ordered by date and time as the previous ones. Class `Statement` is modified to reflect the unification of these lists, as the `merge` method is no longer necessary. The methods `addCreditTrans` and `addPayCardTrans` in class `Statement` are also eliminated, as transactions are now treated uniformly in the new design. The main program remained intact, but the methods it invokes from class `Statement` have been modified. So the preservation of behavior must be ensured.

Our formalization does not include the `super` construct. But in the case study, we use `super()` in constructors, as an abbreviation for copying the code from the superclass constructor, excluding its trailing assignment `res:= self`.

```
class Transaction ext Object{
    prot bal: FValue;
    prot value: FValue;
    prot dt: LogDate;
    meth ctor(): Transaction {
         self.bal:= (new FValue).ctor();
         self.value:= (new FValue).ctor();
         self.dt:= (new LogDate).ctor();
         res:= self}
    meth getDate(): Date {res:= self.dt.getD()}
    meth getTime(): Time {res:= self.dt.getT()}
    meth setDate(d: Date) {self.dt.setD(d)}
    meth setTime(t: Time) {self.dt.setT(t)}
}
class FValue ext Object{
    pri value: int;
    meth ctor(): FValue {res:= self}
    meth getValue(): int {res:= self.value}
    meth setValue(v: int) {self.value:= v}
    meth add(fv: FValue) {self.value:= self.value + fv.getValue()}
    meth sub(fv: FValue) {self.value:= self.value - fv.getValue()}
}
class CreditTrans ext Transaction {
   meth ctor(): CreditTrans {super(); res:= self}
   meth finalBal():int {var fv: FValue in fv = new FValue().ctor();
                        fv.setValue(self.bal.getValue())
                        fv.add(self.value); res:= fv.getValue()}
}
class PayCardTrans ext Transaction {
    meth ctor(): PayCardTrans {super(); res:= self}
    meth finalBal():int {var fv: FValue in fv = new FValue().ctor();
                        fv.setValue(self.bal.getValue())
                        fv.sub(self.value); res:= fv.getValue()}
}
class LogDate ext Object {
    pri dd: Date;
    pri tt: Time;
    meth ctor(): LogDate {res:= self}
    meth getD(): Date {res:= self.dd}
    meth getT(): int {res:= self.tt}
    meth setD(d: Date) {self.dd:= d}
    meth setT(t: Time) {self.tt:= t}
}
class ListTransaction ext Object {
    // list of Transactions ordered by date and time
    pri tr: Transaction;
    pri next: LTransaction
    meth ctor(): ListTransaction {res:= self}
    meth getTR(): Transaction {res:= self.tr)}
    meth setTR(tr1: Transaction) {self.tr:= tr1}
```

```
    meth getNext():LTransaction {res:=self.next}
    meth setNext(n1: LTransaction) {self.next:= n1}
}
class Statement ext Object{
    pri lt : ListTransaction;
    meth ctor (): Statement {res:=self}
    meth addTrans(t: Object){
      /* add an element in lt ordered by date and time */
    }
    meth showStatement () primString {
       res:=self.lt.toString()
    }
}
```

We omit the main program because it is exactly as in the original version.

7.3. Transforming the original version to the final one: first steps

To guarantee that the main program in both designs have the same behavior, we perform a series of code transformations, based on the laws and refactorings presented in Section 6.

Beginning with the original version, first we transform class `CreditTrans` to encapsulate date and time fields, achieving a configuration similar to class `PayCardTrans`; see below. To perform this transformation we apply twice Refactoring 4 ⟨*encapsulate field*⟩ to transform `cdate` and `ctime` into protected attributes and to provide get and set methods to access these fields from client classes. Observe that the code of method `leqDateTime` in the client class `Statement` also changes to reflect this transformation. The rest of the program remains the same.

When describing the transformations we show only the pieces of the program that have been modified.

```
class CreditTrans ext Object {
    prot cbal: int;
    prot cvalue: int;
    prot cdate: Date;
    prot ctime: Time;
    meth ctor(): CreditTrans {res:= self}
    meth getDateCr(): Date {res:= self.cdate}
    meth getTimeCr(): Time {res:= self.ctime}
    meth setDateCr(d: Date) {self.cdate:= d}
    meth setTimeCr(t: Time) {self.ctime:= t}
    meth finalBal():int {res:= self.cbal + self.cvalue}
}
...
class Statement ext Object{
    pri lc : ListCredit;
    pri lp: ListPayCard;
    meth ctor (): Statement {res:=self}
    ... // same code
    meth leqDateTime(l1: ListCredit, l2:ListPayCard): bool {
      /* returns true if l1 occurs earlier than l2 */
       res:=false;
       if (l1.getCT().getDateCr() <  l2.getPC().getDatePay()) or
          (l1.getCT().getDateCr() = l2.getPC().getDatePay() and
          l1.getCT().getTimeCr() ≤ l2.getPC().getTimePay()) then
          res:= true;}
    ... // the rest of the class is unchanged
```

Next we apply Refactoring 6 ⟨*extract class*⟩ to perform a data refinement in class `CreditTrans`. In this transformation, fields `cdate` and `ctime` are mapped to a new field `cld`, which is of a new type, say `LogDate'`, that includes attributes of types `Date` and `Time`. Strictly, we are using a slight generalization of ⟨*extract class*⟩, since, as formulated, for simplicity, it handles a single field *x*. Note also that we are applying the refactoring from left to right.

Were we to be applying the refactoring from right to left, we would need to establish the confinement proviso (for owner class `CreditTrans` with rep class `LogDate`). Confinement in this case would mean that the `LogDate` objects referenced by field `cld` are not shared, as indeed they are not in the transformed program.

The new type `LogDate'` has the same structure as `LogDate`, but a new name since Refactoring ⟨*extract class*⟩ generates a new class. To reuse class `LogDate` we next apply Refactoring 7 ⟨*change name of attribute type*⟩ to change the type of `cld` from `LogDate'` to `LogDate`. The class `LogDate'` is not referenced any longer and thus can be eliminated using Law 1 ⟨*class elimination*⟩. Observe that, by using this refactoring we are separating concerns in class `CreditTrans`, as fields related to date fits better in a separate class.

```
class CreditTrans ext Object {
    prot cbal: int;
    prot cvalue: int;
```

```
    prot cld: LogDate;
    meth ctor(): CreditTrans {
         self.cld:= (new LogDate).ctor(); res:= self}
    meth getDateCr(): Date {res:= self.cld.getD()}
    meth getTimeCr(): Time {res:= self.cld.getT()}
    meth setDateCr(d: Date) {self.cld.setD(d)}
    meth setTimeCr(t: Time) {self.cld.setT(t)}
    meth finalBal():int {res:= self.cbal + self.cvalue}
}
class LogDate ext Object {
    pri dd: Date;
    pri tt: Time;
    meth ctor(): LogDate {res:= self}
    meth getD(): Date {res:= self.dd}
    meth getT(): int {res:= self.tt}
    meth setD(d: Date) {self.dd:= d}
    meth setT(t: Time) {self.tt:= t}
} ...
```

The next step is to apply Refactoring 2 ⟨*rename method*⟩ to rename methods `getDateCr`, `getTimeCr`, `setDateCr` and `setTimeCr` to, respectively, `getDate`, `getTime`, `setDate` and `setTime`. The same procedure is applied to rename methods in class `PayCardTrans`. Observe that the application of this refactoring replaces all occurrences of the older get and set methods in class `Statement`.

```
class CreditTrans ext Object {
    prot cbal: int;
    prot cvalue: int;
    prot cld: LogDate;
    meth ctor(): CreditTrans {
         self.cld:= (new LogDate).ctor(); res:= self}
    meth getDate(): Date {res:= self.dt.getD()}
    meth getTime(): Time {res:= self.dt.getT()}
    meth setDate(d: Date) {self.dt.setD(d)}
    meth setTime(t: Time) {self.dt.setT(t)}
    meth finalBal():int {res:= self.bal + self.value}
}
class PayCardTrans ext Object {
    prot pbal: int;
    prot pvalue: int;
    prot pld: LogDate;
    meth ctor(): PayCardTrans {
       self.pld:= (new LogDate).ctor(); res:= self }
    meth getDate(): Date {res:= self.dt.getD()}
    meth getTime(): Time {res:= self.dt.getT()}
    meth setDate(d: Date) {self.dt.setD(d)}
    meth setTime(t: Time) {self.dt.setT(t)}
    meth finalBal() {res:= self.bal - self.value}
}...
class Statement ext Object{
    pri lc : ListCredit;
    pri lp: ListPayCard;
    meth ctor (): Statement {res:=self};
    ... // same code
    meth leqDateTime(l1: ListCredit, l2:ListPayCard): bool {
      /* returns true if l1 occurs earlier than l2 */
       res:=false;
       if (l1.getCT().getDate() <  l2.getPC.getDate()) or
          (l1.getCT().getDate() =  l2.getPC.getDate() and
          l1.getCT().getTime() ≤ l2.getPC().getTime()) then
          res:= true;}
    ... // the rest of the class is unchanged
}
```

Then we introduce class `Transaction` and make `CreditTrans` and `PayCardTrans` inherit from `Transaction`. So, we apply Law 1 ⟨*class elimination*⟩, from right to left, to introduce class `Transaction`. After that, in order to introduce the inheritance relationship, we apply twice Law 5 ⟨*change superclass*⟩.

```
class Transaction ext Object {}

class CreditTrans ext Transaction {
    prot cbal: int;
    prot cvalue: int;
    prot cld: LogDate;
```

```
    meth ctor(): CreditTrans {
       self.cld:= (new LogDate).ctor(); res:= self}
    meth getDate(): Date {res:= self.cld.getD()}
    meth getTime(): Time {res:= self.cld.getT()}
    meth setDate(d: Date) {self.cld.setD(d)}
    meth setTime(t: Time) {self.cld.setT(t)}
    meth finalBal():int {res:= self.cbal + self.cvalue}
}
class PayCardTrans ext Transaction {
    prot pbal: int;
    prot pvalue: int;
    prot pld: LogDate;
    meth ctor(): PayCardTrans {
       self.pld:= (new LogDate).ctor(); res:= self}
    meth getDate(): Date {res:= self.pld.getD()}
    meth getTime(): Time {res:= self.pld.getT()}
    meth setDate(d: Date) {self.pld.setD(d)}
    meth setTime(t: Time) {self.pld.setT(t)}
    meth finalBal():int {res:= self.pbal - self.pvalue}
} ...
```

To unify the attributes of classes `CreditTrans` and `PayCardTrans` by moving them to class `Transaction`, we apply three times Refactoring 1 ⟨*pull up field*⟩, from left to right. Observe that at this point data refinements are performed and all occurrences of the old fields are now replaced by the new fields.

The next step is to apply Law 4 ⟨*method elimination*⟩ from right to left to introduce method `ctor` in class `Transaction`.

```
class Transaction ext Object {
    prot bal: int;
    prot value: int;
    prot dt: LogDate;
    meth ctor(): Transaction {
       self.dt:= (new LogDate).ctor(); res:= self}
}
class CreditTrans ext Transaction {
    meth ctor(): CreditTrans {super(); res:= self}
    meth getDate(): Date {res:= self.dt.getD()}
    meth getTime(): Time {res:= self.dt.getT()}
    meth setDate(d: Date) {self.dt.setD(d)}
    meth setTime(t: Time) {self.dt.setT(t)}
    meth finalBal():int {res:= self.bal + self.value}
}
class PayCardTrans ext Transaction {
    meth ctor(): PayCardTrans {super(); res:= self}
    meth getDate(): Date {res:= self.dt.getD()}
    meth getTime(): Time {res:= self.dt.getT()}
    meth setDate(d: Date) {self.dt.setD(d)}
    meth setTime(t: Time) {self.dt.setT(t)}
    meth finalBal():int {res:= self.bal - self.value}
}...
```

Then we move methods get and set of both classes to class `Transaction`. This is performed by applying Refactoring 3 ⟨*pull up method*⟩ four times, reaching the following program.

```
class Transaction ext Object {
    prot bal: int;
    prot value: int;
    prot dt: LogDate;
    meth ctor(): Transaction {
       self.dt:= (new LogDate).ctor(); res:= self}
    meth getDate(): Date {res:= self.dt.getD()}
    meth getTime(): Time {res:= self.dt.getT()}
    meth setDate(d: Date) {self.dt.setD(d)}
    meth setTime(t: Time) {self.dt.setT(t)}
}
class CreditTrans ext Transaction {
    meth ctor(): CreditTrans {super(); res:= self}
    meth finalBal(): int {res:= self.bal + self.value}
}
class PayCardTrans ext Transaction {
    meth ctor(): PayCardTrans {super(); res:= self}
    meth finalBal():int {res:= self.bal - self.value}
}
...
```

7.4. Transforming the original version to the final one: data refinement and remaining steps

The next step is to apply Law 1 ⟨*class elimination*⟩ in order to introduce class `ListTransaction`. This yields the following.

```
class ListTransaction ext LObject {
    // list of Transactions ordered by date and time
    pri tr: Transaction;
    meth ctor(): ListTransaction {res:= self}
    meth getTR(): Transaction {res:= self.tr)}
    meth setTR(tr1: Transaction) {self.tr:= tr1}
}
```

Next, we directly apply Law 6 ⟨*data refinement*⟩. We revise class `Statement` by replacing fields `lc` and `lp` with field `lt`, which is a list of transactions ordered by date. Thus `lt` contains what in the original version is obtained from `lc` and `lp` by method `merge`. Moreover we eliminate the methods `addCreditTrans` and `addPayCardTrans`, by applying twice Law 4 ⟨*method elimination*⟩, since the method `addTrans` now directly include transactions in `lt`, and does not call `addCreditTrans` or `addPayCardTrans` anymore.

In what follows we give a fragment of the new class `Statement`.

```
class Statement ext Object{
    pri lt: ListTransaction;
    meth ctor(): Statement {res:=self}
    meth addTrans ...
    meth leqDateTime ...
    meth merge ...
    meth showStatement(): primString {
      res:=self.lt.toString()}
}
```

To apply the data refinement law, we first check the confinement conditions: the lists are owned and not shared between instances of `Statement`, which is *Own*. Let $Rep = LObject$ and $Rep' = ListTransaction$. Moreover, we define a local coupling that relates fields of both classes `Statement`, and the contents of their respective lists. Let us write it as a formula where variables `self` and `self'` refer to instances of the old and new versions of `Statement`, respectively:

$$\begin{aligned}&merge(list(\texttt{self.lc}), list(\texttt{self.lp})) = list(\texttt{self'.lt})\\ &\wedge asc(list(\texttt{self.lc})) \wedge asc(list(\texttt{self.lp})).\end{aligned} \tag{4}$$

Here *list* gives the abstract list represented by a pointer, *asc* says the list is sorted, and *merge* is the mathematical function that merges sorted lists. Equal lists implies that corresponding elements of the lists have same data. That is, if l and l' are lists of the same length then

$$\forall i \bullet l[i] = l'[i] \Rightarrow l'.\texttt{toString}() = l'.\texttt{toString}(). \tag{5}$$

Note that the first conjunct implies that $list(\texttt{self'.lt})$ is ascending.

We have to show that this relation – extended to complete states – is preserved by the corresponding implementations of methods of `Statement`. For example, in related states, the strings returned by the two versions of `showStatement` are equal.

Without confinement separating owners from each other, it would not be enough to reason in terms of a coupling like (4) defined in terms of a single instance of `Statement`. Without confinement keeping clients separate from the internal representation, it would not suffice to check preservation only for the methods of `Statement`.

To prove the coupling is preserved, we have that, by definition of *merge*, $l[0]$ is the first element of *lc* or *lp*, considering date and time. Thus, by assert of *lt*, $l[0] = l'[0]$. So, the initial states σ and σ' are related ($\hat{\mathcal{R}}\ \ \sigma\ \ \sigma'$). Let τ and τ' be the final states after applying method `showStatement`. Thus, $list(\tau(tmp)) = merge(list(\sigma(\texttt{self.lc})), list(\sigma(\texttt{self.lp})))$ and $list(\tau'(tmp)) = list(\sigma'(tmp)) = list(\sigma'(\texttt{self.lt}))$. Note that the following equation holds: $list(\sigma'(\texttt{self.lt})) = merge(list(\sigma(\texttt{self.lc})), list(\sigma(\texttt{self.lp})))$. Then, as $\hat{\mathcal{R}}\ \ \sigma\ \ \sigma'$, $list(\tau(tmp)) = list(\tau(tmp'))$. By Eq. (5), the results of both versions of class `Statement` are the same. After applying the law, we have:

```
class Transaction ext Object {
    prot bal: int;
    prot value: int;
    prot dt: LogDate;
    meth ctor(): Transaction {
       self.dt:= (new LogDate).ctor(); res:= self}
    meth getDate(): Date {res:= self.dt.getD()}
    meth getTime(): Time {res:= self.dt.getT()}
    meth setDate(d: Date) {self.dt.setD(d)}
    meth setTime(t: Time) {self.dt.setT(t)}
}
class CreditTrans ext Transaction {
    meth ctor(): CreditTrans{super(); res:= self}
    meth finalBal():int {res:= self.bal + self.value}
```

```
}
class PayCardTrans ext Transaction {
    meth ctor(): PayCardTrans {super(); res:= self}
    meth finalBal():int {res:= self.bal - self.value}
}
class LogDate ext Object {
    pri dd: Date;
    pri tt: Time;
    meth ctor(): LogDate {res:= self}
    meth getD(): Date {res:= self.dd}
    meth getT(): int {res:= self.tt}
    meth setD(d: Date) {self.dd:= d}
    meth setT(t: Time) {self.tt:= t}
}
class ListCredit ext LObject {
    pri ct: CreditTrans;
    meth ctor(): ListCredit {res:= self}
    meth getCT(): CreditTrans {res:= self.ct}
    meth setCT(ct1: CreditTrans) {self.ct:= ct1}
}
class ListPayCard ext LObject {
    pri pc: PayCardTrans;
    meth ctor(): ListPayCard {res:= self}
    meth getPC(): PayCardTrans {res:= self.pc)}
    meth setPC(pc1: PayCardTrans) {self.pc:= pc1}
}
class ListTransaction ext LObject {
    //list of transactions ordered by date and time
    pri tr: Transaction;
    meth ctor(): ListTransaction {res:= self}
    meth getTR(): Transaction {res:= self.tr)}
    meth setTR(tr1: Transaction) {self.lt:= tr1};
}
class LObject ext Object {
    meth ctor():Lobject {res:= self}
    pri next: LObject
    meth getNext():LObject {res:=self.next}
    meth setNext(n1: LObject) {self.next:= n1}
}
class Statement ext Object{
    pri lt: ListTransaction;
    meth ctor(): Statement {res:=self}
    meth addTrans ...
    meth leqDateTime ...
    meth merge ...
    meth showStatement(): primString {
      res:=self.lt.toString()}
}
```

We can eliminate method `merge` because it is no longer used, by applying Law 4 ⟨*method elimination*⟩. Analogously, we can eliminate `leqDateTime`. Moreover, classes `ListCredit` and `ListPayCard` are no longer referenced, so we can eliminate these classes by applying twice Law 1 ⟨*class elimination*⟩.

At this stage, `LObject` does not occur anywhere except as the superclass of `ListTransaction`. Using Refactoring 5 ⟨*collapse hierarchy – subclass*⟩, we can collapse the hierarchy. The result is now close to the final intended design of our case study, as presented early in this section. The only missing step is to represent financial values (with respect to the attributes `bal` and `value` of class `Transaction`) as a separate abstraction, captured by the class `FValue`. Note that the application of the ⟨*extract class*⟩ refactoring affects the subclasses `CreditTrans` and `PayCardTrans` of `Transaction`. Particularly, the homonymous methods `finalBal()` of these classes are modified to reflect the new representation of financial values. This complete transformation is achieved by the following sequence of steps:

- First we consider the `value` attribute. Using Refactoring 6 ⟨*extract class*⟩, we extract an initial version of class `FValue`, and modify the subclasses of `Transaction` as a consequence of changing the type of `value` from `int` to `FValue`. Class `Transaction` is modified only concerning the type of the attribute `value` and the constructor method `ctor`, which now needs to create an instance of `FValue`.

```
class Transaction ext Object {
   prot bal: int;
   prot value: FValue;
   prot dt: LogDate;
   meth ctor(): Transaction {
      self.dt:= (new LogDate).ctor();
      self.value:= (new FValue).ctor(); res:= self}
```

```
   \\ all other methods as before
}
class FValue ext Object{
   pri value: int;
   meth ctor(int v): FValue {value:= v; res:= self}
   meth getValue(): int {res:= self.value}
   meth setValue(v: int) {self.value:= v}
}
class CreditTrans ext Transaction {
   meth ctor(): CreditTrans {super(); res:= self}
   meth finalBal():int {res:= self.bal + self.value.getValue()}
}
class PayCardTrans ext Transaction {
   meth ctor(): PayCardTrans {super(); res:= self}
   meth finalBal():int {res:= self.bal - self.value.getValue()}
}
```

- We then carry out a similar transformation considering the `bal` attribute, also applying Refactoring 6 ⟨*extract class*⟩. This generates a class with the same structure as `FValue`, but with a different name, say `BValue`. So we apply Refactoring 7 ⟨*change name of attribute type*⟩ to change the type of `bal` to be `FValue`. The class `BValue` is not referenced any longer and thus can be eliminated using Law 1 ⟨*class elimination*⟩. The subclasses of `Transaction` are modified accordingly.

```
class Transaction ext Object {
   prot bal: FValue;
   prot value: FValue;
   prot dt: LogDate;
   meth ctor(): Transaction {
      self.dt:= (new LogDate).ctor();
      self.bal:= (new FValue).ctor();
      self.value:= (new FValue).ctor(); res:= self}
   \\ all other methods are as before
}
class FValue ext Object { /* same as before */ }

class CreditTrans ext Transaction {
   meth ctor(): CreditTrans {super(); res:= self}
   meth finalBal():int
      {res:= self.bal.getValue() + self.value.getValue()}
}
class PayCardTrans ext Transaction {
   meth ctor(): PayCardTrans {super(); res:= self}
   meth finalBal():int
      {res:= self.bal.Value() - self.value.getValue()}
}
```

- The final step involves introducing methods `add` and `sub` to class `FValue`, using Law 4 ⟨*method elimination*⟩. With this we achieve the final form of class `FValue`, as presented early in this section:

```
class FValue ext Object{
  pri value: int;
  meth ctor(): FValue {res:= self}
  meth getValue(): int {res:= self.value}
  meth setValue(v: int){self.value:= v}
  meth add(fv: FValue){self.value:= self.value + fv.getValue()}
  meth sub(fv: FValue){self.value:= self.value - fv.getValue()}
}
```

Now it is necessary some algorithm refinement to show that the bodies of the homonymous methods `finalBal` in classes `CreditTrans` and `PayCardTrans` can be transformed to use methods `add` and `sub`. For instance, taking the class `CreditTrans` as illustration, we need to prove it can be transformed into:

```
 class CreditTrans ext Transaction {
   meth ctor(): CreditTrans {super(); res:= self}
   meth finalBal():int
     {var fv: FValue in fv:= (new FValue).ctor();
      fv.setValue(self.bal.getValue())
      fv.add(self.value); res:= fv.getValue()}
 }
```

The program fragment that defines the body of the method `finalBal` introduces a local variable `fv` that stores the output of the method `finalBal` in class `CreditTrans`. The initial value stored in the freshly created `FValue` object assigned to `fv` is the value of the field `bal`. Then the method `add` is used to increment the value stored in `fv` with the integer stored in attribute `value`. Finally, the resulting value is yielded by the method, which is precisely that previously returned by the

statement `res:= self.bal.getValue() + self.value.getValue()`. The fact that the behavior of the body of the method `finalBal` is preserved can be easily proved using some laws of commands and method calls, but this detailed proof is not relevant in the context of this work.

This concludes the development of our case study. At this stage it has the form presented in Section 7.2, which is the intended target of our successive series of refactoring applications embodying transformations that impact class hierarchies.

8. Conclusion

Object oriented languages feature a notion of *protected* scope whereby a declaration in one class K is visible in the entire hierarchy of subclasses of K. The feature is extensively used due to both performance considerations and software engineering considerations, as in the context of extensible frameworks. As with private visibility, protected visibility does not itself provide a strong encapsulation boundary due to the possibility of sharing. Some form of alias control is also needed. To provide sufficient encapsulation for correctness-preserving transformations, we defined a notion of ownership confinement suited to class hierarchies. On this basis we defined local coupling relations and proved an abstraction theorem which says the simulation property for methods within a class hierarchy extends to simulation for a complete program. We used the simulation theorem to prove a data refinement law by which the implementation of a class hierarchy can be transformed to a different implementation, using different protected (and private) attributes and different method implementations. We proved a number of refactoring transformations and illustrated their application in an extended case study. Particularly, we propose a generalization of Refactoring ⟨*extract class*⟩, which is among the complex refactorings that have been dubbed a Rubicon for refactoring tools to cross [3].

Although our primary goal is to ensure correctness of refactorings, our insistence on formality in dealing with case studies has led us to improve on the informal refactorings in the literature. Another benefit is that we reduce the number of rules by explicitly treating them as conditional equivalences. For instance, Fowler distinguishes between ⟨*pull up field*⟩ and ⟨*push down field*⟩, whereas in our case this is captured by a single equation.

Most of our refactoring rules do not impose confinement as an explicit proviso, although the general Law 6 of ⟨*data refinement*⟩ and Refactoring 6 ⟨*extract class*⟩ do. In using data refinement to prove Refactoring 1 ⟨*pull up field*⟩ we must satisfy the confinement conditions but the refactoring is only concerned with the fields of a single object so confinement is merely a technicality in the proof. On the other hand, Refactoring 6 ⟨*extract class*⟩ involves multiple objects and is not sound without confinement. Confinement in this case is not a consequence of the other provisos, when the refactoring is used in the direction of un-extracting; so we made it an explicit proviso. The proof of Refactoring 7 ⟨*change name of attribute type*⟩ also relies on non-trivial confinement, but in this case confinement is a consequence of the other provisos. In our case study, we needed to explicitly use the data refinement law only once, for a very specific data refinement. Otherwise we applied laws and refactorings, which are of more general use.

In this paper we used program semantics and in some cases laws, including data refinement, to prove a number of refactorings. Ideally, program semantics and data refinement are used to prove a core set of laws and refactorings, from which more elaborate ones are derived in an algebraic style, as has been done in earlier work [10]. The laws and refactorings can then be used to prove more elaborate transformations involving design and architectural patterns.

The basic laws and many refactorings have been formally justified in previous work that was based on "copy semantics" and so applicable to programs that use cloning but never shared references. The fact that our language includes object references, as opposed to copy semantics, has allowed us to avoid some restrictions imposed as side conditions on the refactorings presented in [10]. For example, the version of the ⟨*extract class*⟩ refactoring in [10] forbids objects as parameters to the method $m_1()$, since, for instance, passing self as parameter would cause inconsistencies in the refactorings presented there; here we have no such restrictions.

In summary, the distinguishing features of our approach to refactorings are:

- focus on refactoring programs with potential effects on arbitrary inheritance hierarchies;
- algebraic presentation of rules as complete programs, with explicit side conditions on other classes or on the main program, ensuring soundness of the transformations; and
- reference semantics that allows sharing of objects, but controlled by a confinement discipline.

Related work addresses one aspect or another but, to our knowledge, our work is the first attempt to formalize and systematize algebraic transformations involving arbitrary class trees.

For instance, rCOS [20] presents a mathematical characterization of object-oriented concepts together with a calculus that supports both structural and behavioral refinement of object-oriented designs, in the context of reference semantics. However, the proposed laws are stated in terms of equivalences in all (main program) contexts. Also, despite the definitions and theorems on data refinement, rCOS does not address algebraic laws or refactorings for data refinement in the context of inheritance.

We follow Opdyke [28] in formulating laws with syntactic applicability conditions (often called "preconditions" in the refactoring literature). In contrast, Shäfer and de Moor [30] advocate applicability conditions of a different form: a refactoring consists of a pattern of transformation, that the code must match, and specified structural properties like binding relationships, that must be preserved. (In [40] these are termed "invariants" but this is dropped in the later paper.) They

argue that syntactic provisos are doomed to be overly restrictive or wrong in dealing with the full complications of Java. Our language is much simpler but even so it is difficult to find satisfactory provisos. Shäfer et al [40,31,30] also observe that the properties they identify, e.g., binding, control, and dataflow structure, are usually necessary if not sufficient for behavioral equivalence. Their work provides refactorings that are more often correct than those implemented by other tools, but for critical software behavioral equivalence remains the goal. We leave it as important future work to find techniques for proving correctness of refactorings specified in the manner of Shäfer et al.

A few works prove refactoring laws in the setting of very simple languages without object-oriented features. Sultana and Thompson [38] prove refactorings for simple lambda calculi; indeed, they use the Isabelle/HOL proof assistant to mechanically check the proofs. In [25,27], we have mechanized the semantics of a language similar to the one used here, in the PVS proof assistant. A natural path to explore would be to build on that work to formalize our confinement theory and the refactorings, and to check all the proofs. In previous work [15], we explored the use of the term rewriting system CafeOBJ for the refactorings proposed in [10], restricted to copy semantics. In that work, the syntax of the language was encoded, and some basic laws postulated as axioms. Proofs of some refactorings were derived from the basic laws, in an algebraic style.

Ownership confinement has been formalized in class based languages using annotated types [11], ghost state [24], and separation logic [7,34]. Here we used a somewhat restrictive notion of confinement, adapted from [5], enforced by constraints expressed in terms of ordinary types (in a nominal type system like that of Java). This notion suffices in cases like Refactoring 7 ⟨*change name of attribute type*⟩ where natural provisos suffice to ensure confinement. On the other hand, confinement is an explicit proviso in Refactoring 6 ⟨*extract class*⟩, which is probably sound under less restrictive confinement disciplines than the one formalized here.

Simulation theories for much more sophisticated languages are a topic of current interest (e.g., [36,21,9,2,39,17,16]). Refactoring transformations for object-oriented and other programming languages provide interesting and important challenges for semantic models and simulation techniques.

In the future we plan to consider protected methods as well, since these are also extensively used in framework design; however, they are much simpler than protected fields, since the impact on transformations is restricted to algorithmic (control) refinement, in opposition to data refinement. We also plan to develop further case studies and to explore the derivation of architectural patterns from the laws and refactorings. As a complementary work, we are investigating a comprehensive set of command laws in the context of reference semantics. An ultimate goal for the refactoring tools in software development environments is to provide transformations that are sound for the full programming language and with respect to the program semantics implemented by the compiler, runtime, and hardware. We envision a verified or verifying refactorer, to complement a verifying or verified compiler [19,23].

References

[1] Jonathan Aldrich, Craig Chambers, Ownership domains: separating aliasing policy from mechanism, in: European Conference on Object-Oriented Programming, in: LNCS, vol. 3086, 2004, pp. 1–25.
[2] Amal Ahmed, Derek Dreyer, Andreas Rossberg, State-dependent representation independence, in: ACM Symposium on Principles of Programming Languages, 2009, pp. 340–353.
[3] Aharon Abadi, Ran Ettinger, Yishai A. Feldman, Re-approaching the refactoring Rubicon, in: Proceedings of the 2nd ACM Workshop on Refactoring Tools, WRT'08, 2008, pp. 10:1–10:4.
[4] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Ownership types for object encapsulation, in: ACM Symposium on Principles of Programming Languages, 2003, pp. 213–223.
[5] Anindya Banerjee, David A. Naumann, Ownership confinement ensures representation independence for object-oriented programs, Journal of the ACM 52 (6) (2005) 894–960.
[6] Anindya Banerjee, David A. Naumann, State based ownership, reentrance, and encapsulation, in: European Conference on Object-Oriented Programming, in: LNCS, vol. 3586, 2005, pp. 387–411.
[7] Gavin M. Bierman, Matthew J. Parkinson, Separation logic and abstraction, in: ACM Symposium on Principles of Programming Languages, 2005, pp. 247–258.
[8] Paulo Borba, Augusto Sampaio, Ana Cavalcanti, Márcio Cornélio, Algebraic reasoning for object-oriented programming, Science of Computer Programming 52 (1–3) (2004) 53–100.
[9] Lars Birkedal, Hongseok Yang, Relational parametricity and separation logic, Logical Methods in Computer Science 4 (2) (2008) 1–27.
[10] Márcio Cornélio, Ana Cavalcanti, Augusto Sampaio, Sound refactorings, Science of Computer Programming 75 (3) (2010) 106–133.
[11] David Clarke, Sophia Drossopoulou, Ownership, encapsulation and the disjointness of type and effect, in: ACM Conf. on Object-Oriented Programming Languages, Systems, and Applications, November 2002, pp. 292–310.
[12] Ana Cavalcanti, David A. Naumann, A weakest precondition semantics for refinement of object-oriented programs, IEEE Transactions on Software Engineering 26 (8) (2000) 713–728.
[13] Ana Cavalcanti, David A. Naumann, Forward simulation for data refinement of classes, in: Formal Methods Europe, in: LNCS, vol. 2391, 2002, pp. 471–490.
[14] David G. Clarke, James Noble, John M. Potter, Simple ownership types for object containment, in: Jørgen Lindskov Knudsen (Ed.), European Conference on Object-Oriented Programming, in: LNCS, vol. 2072, 2001, pp. 53–76.
[15] Antônio Carvalho, Leila Silva, Márcio Cornélio, Using CafeOBJ to mechanise refactoring proofs and application, Electronic Notes in Theoretical Computer Science 184 (2007) 39–61.
[16] Derek Dreyer, Georg Neis, Andreas Rossberg, Lars Birkedal, A relational modal logic for higher-order stateful ADTs, in: ACM Symposium on Principles of Programming Languages, 2010, pp. 185–198.
[17] Ivana Filipović, Peter O'Hearn, Noah Torp-Smith, Hongseok Yang, Blaming the client: On data refinement in the presence of pointers, Formal Aspects of Computing 22 (5) (2010) 547–583.
[18] Martin Fowler, Refactoring: Improving the Design of existing Code, Addison Wesley, 2000.
[19] Tony Hoare, The verifying compiler: a grand challenge for computing research, Journal of the ACM 50 (January) (2003) 63–69.
[20] He Jifeng, Xian Li, Zhiming Liu, rCOS: a refinement calculus of object systems, Theoretical Computer Science 1–2 (365) (2006) 109–142.
[21] Vasileios Koutavas, Mitchell Wand, Bisimulations for untyped imperative objects, in: European Symposium on Programming, in: LNCS, vol. 3924, 2006, pp. 146–161.

[22] Vasileios Koutavas, Mitchell Wand, Reasoning about class behavior, in: Informal proc. of FOOL/WOOD, 2007.
[23] Xavier Leroy, Formal verification of a realistic compiler, Communications of the ACM 52 (July) (2009) 107–115.
[24] K. Rustan, M. Leino, Peter Müller, Object invariants in dynamic contexts, in: European Conference on Object-Oriented Programming, in: LNCS, vol. 3086, 2004, pp. 491–516.
[25] Gary T. Leavens, David A. Naumann, Stan Rosenberg, Preliminary definition of core JML. Technical Report CS Report 2006-07, Stevens Inst. Tech., 2006.
[26] Peter Müller, Arsenii Rudich, Ownership transfer in Universe Types, in: ACM Conf. on Object-Oriented Programming Languages, Systems, and Applications, 2007, pp. 461–478.
[27] David A. Naumann, Verifying a secure information flow analyzer, in: 18th International Conference on Theorem Proving in Higher Order Logics, in: LNCS, vol. 3603, 2005, pp. 211–226.
[28] William F. Opdyke, Refactoring object-oriented frameworks, Ph.D. thesis, University of Illinois at Urbana-Champaign. 1992.
[29] John C. Reynolds, Types, abstraction, and parametric polymorphism, in: R.E.A. Mason (Ed.), Information Processing'83, North-Holland, 1984, pp. 513–523.
[30] Max Schäfer, Oege de Moor, Specifying and implementing refactorings, in: ACM Conf. on Object-Oriented Programming Languages, Systems, and Applications, 2010, pp. 286–301.
[31] Max Schäfer, Torbjörn Ekman, Oege de Moor, Challenge proposal: verification of refactorings, in: Proceedings of the 3rd ACM workshop on Programming Languages Meets Program Verification, PLPV, 2009, pp. 67–72.
[32] Max Shäfer, Torbjörn Ekman, Ran Ettinger, Mathieu Verbaere, Refactoring bugs, 2010. http://progtools.comlab.ox.ac.uk/projects/refactoring/bugreports.
[33] Amr Sabry, Matthias Felleisen, Reasoning about programs in continuation passing style, Lisp and Symbolic Computation 6 (3/4) (1993) 289–360.
[34] Jan Smans, Bart Jacobs, Frank Piessens, Implicit dynamic frames: combining dynamic frames and separation logic, in: European Conference on Object-Oriented Programming, in: LNCS, vol. 5653, 2009, pp. 148–172.
[35] Leila Silva, David A. Naumann, Augusto Sampaio, Refactoring and representation independence for class hierarchies, in: FTfJP — Formal Techniques for Java-like Programs, ACM Digital Library, 2010, pp. 1–6.
[36] Eijiro Sumii, Benjamin C. Pierce, A bisimulation for type abstraction and recursion, Journal of the ACM 54 (5) (2007).
[37] Leila Silva, Augusto Sampaio, Zhiming Liu, Laws of object-orientation with reference semantics, in: Antonio Cerone, Stefan Gruner (Eds.), Sixth IEEE International Conference on Software Engineering and Formal Methods, 2008, pp. 217–226.
[38] Nik Sultana, Simon Thompson, Mechanical verification of refactorings, in: ACM Symposium on Partial Evaluation and Semantics-based Program Manipulation, 2008, pp. 51–60.
[39] Eijiro Sumii, A complete characterization of observational equivalence in polymorphic lambda-calculus with general references, in: Computer Science Logic, in: LNCS, vol. 5771, 2009, pp. 455–469.
[40] Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, Oege Moor, Stepping stones over the refactoring rubicon, in: European Conference on Object-Oriented Programming, in: LNCS, vol. 5653, 2009, pp. 369–393.