# Refactoring and Representation Independence for Class Hierarchies [Extended Abstract]

Leila Silva[*]
Universidade Federal de Sergipe, Brasil

David A. Naumann[†]
Stevens Institute of Technology, USA

Augusto Sampaio[‡]
Universidade Federal de Pernambuco, Brasil

## 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 several refactoring laws. Soundness of these laws was an open problem in an ongoing project on formal refactoring tools. The utility of the laws is shown in a case study. Shortcomings of the theory are described as a challenge to other approaches to heap encapsulation and relational reasoning for classes.

## 1. INTRODUCTION

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, 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).

This paper contributes to an ongoing project in which semantics preserving refactoring transformations are formally defined and validated. In previous work [6] 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 [7]) and relatively complete, through a normal form reduction process, and used to prove sample refactorings. Many interesting refactorings involve change of data representation, for which a theory of data refinement was developed [8]. These works impose the drastic restriction that programs do not share mutable objects (so "copy semantics" could be used).

We have also investigated the impact of reference semantics on the laws, in the rCOS refinement theory [18], 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 interdependencies 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. Banerjee and Naumann [2] 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 [3] they develop a more flexible version that caters for transferable ownership. Representation independence with references is under active investigation but recent advances [21, 12, 5, 1, 20, 11, 10] do not directly address class based languages. The technique of Koutavas and Wand [12] was adapted to a class based language [13] and used to verify the examples in [2] but those are specific programs and a single class rather than more general refactoring laws.

The works in the preceding paragraph focus on proving *contextual equivalence* of program constructs, i.e., equivalence in all contexts. (So the term *representation independence* is more suitable than data refinement.) As discussed in [6], many useful laws hold only in restricted contexts, e.g., those where method contracts are satisfied, downcasts are absent, or some confinement discipline is respected. For that reason, [6] considers whole program equivalence, via schematic laws in which contextual constraints can be expressed syntactically.

The logic of Dryer et al [10] derives equality judgements for terms and can express certain contextual assumptions; we are unable to judge the extent to which laws like ours can be derived in their logic. Their programming language features higher order functions, general references, and para-

[*]Partially supported by the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq, grant 573964/2008-4.

[†]Partially supported by US NSF awards CNS-0627338, CRI-0708330, CCF-0915611.

[‡]Partially supported by CNPq grants 309048/2006-0, 573964/2008-4.


*FTfJP* '10, June 22, 2010, Maribor, Slovenia

metric polymorphism, but not subtyping. Encoding of Java-like classes (nominal types, per-field scoping, etc) would be nontrivial. That would add complications to an already complicated reasoning system. On the other hand, the system is far more expressive and flexible than ours and we know no reason why our confinement discipline could not be expressed.

In this paper we present a representation independence theorem that can prove equivalence of two implementations of a whole hierarchy of classes. The theorem is used to prove a simulation law, which is itself used to formally justify several general refactoring laws. Those laws are in turn used to carry out a case study with several steps. One step makes direct use of the simulation law.

Remarkably, we achieve these results via a fairly straightforward generalization of results in [2]. In particular, we make confinement more flexible but it still centers on a single "owner" object and is formulated in terms of Java-like nominal types. This makes it possible to enforce confinement using a straightforward and lightweight static analysis, like that worked out in [2]. That is our motivation for basing our work on [2] rather than a more flexible confinement theory like [3]. However, in this paper we do not present a static analysis. Instead, we define confinement semantically and simply assume it where needed. We focus on the representation independence theorem and on the derived refactoring laws. The theory addresses whole program equivalence, so we prove schematic laws by structural induction on the programs involved; in this we exploit the compositionality of the denotational semantics.

We introduce the refactorings via the case study. Then we sketch the notion of confinement and results on representation independence, followed by representative refactorings. For clarity we take some liberties with notation and definitions. Full technical details and more refactorings can be found in [17].

## 2. CASE STUDY

The example begins with the program in Fig. 1. It is a banking application with two kinds of transactions. We will perform a series of program transformations involving change of data representation in class hierarchies.

The `CreditTrans` transaction records the previous balance and the value of a deposit, whereas `PayCardTrans` records the previous balance and the value drawn out to pay, for example, a credit card bill. Both transactions register the date the transaction is performed. In `CreditTrans` there is a public attribute for doing that, whereas in `PayCardTrans` this information is encapsulated as an object of class `LogDate`; and `PayCardTrans` has methods to get and set 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 that has been deposited or drawn out, respectively. The special variable `res` provides the return value of a method. All classes have a constructor, named `ctor`.

The application class `Statement` has code, which we omit, to maintain one list of credit transactions and another list of pay card transactions, each ordered by date. The statement is built as a list of objects of type `LObject`, containing transactions of type `CreditTrans` and `PayCardTrans`, ordered by date. This list is constructed by using a standard (and private) merge procedure in the input lists.

```
class CreditTrans ext Object {
   prot cbal: int;
   prot cvalue: int;
   pub cdate: Date;
   meth ctor(): CreditTrans {res:= self} // constructor
   meth finalBal():int {res := self.cbal + self.cvalue}
}
class PayCardTrans ext Object {
    prot pbal: int;
    prot pvalue: int;
    priv pld: LogDate;
    meth ctor(): PayCardTrans {
      self.pld := (new LogDate).ctor(); res:= self}
    meth getDatePay(): Date {res:= self.pld.getD()}
    meth setDatePay(d: Date) {self.pld.setD(d)}
    meth finalBal():int {res := self.pbal - self.pvalue}
}
class LogDate ext Object {
    priv dd: Date;
    meth ctor(): LogDate {res:= self}
    meth getD(): Date {res:= self.dd)}
    meth setD(d: Date) {self.dd:= d}
}
class ListCredit ext LObject {
    priv ct: CreditTrans;
    meth ctor(): ListCredit {res:= self}
    meth getCT(): CreditTrans {res:= self.ct)}
    meth setCT(ct1: CreditTrans) {self.ct:= ct1}
}
class ListPayCardTrans ext LObject {
    priv pt: PayCardTrans;
    meth ctor(): ListPayCardTrans {res:= self}
    meth getLP(): PayCardTrans {res:= self.pt)}
    meth setLP(pt1: PayCardTrans) {self.pt:= pt1}
}
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 {
    priv lc : ListCredit;
    priv lp: ListPayCardTrans;
    meth ctor(): Statement {res:=self}
    ... // code to add transactions
    priv meth merge: LObject (l1: ListCredit,
                    l2: ListPayCardTrans) { ...
         // returns the merge of sorted lists }
    meth ShowStatement(): primString { // print stmt
      assert elements in lc and in lp ordered by date;
      LObject tmp;
      tmp:=self.merge(lc,lp);
      res:=tmp.toString()}       }
```

**Figure 1: Original version**

We will improve the design via a series of transformations leading to the final version in Fig. 2. Afterwards, we will justify that the versions are equivalent.

In the final version, 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`, which also has set and get methods to access these attributes. Lists of `CreditTrans` and `PayCardTrans` are replaced by a single list of `Transaction`, ordered by date as the previous ones. Class `Statement` is modified to reflect the unification of these lists, and the `merge` method is no longer needed.

To guarantee that both designs have the same behavior we

```
class Transaction ext Object {
    prot bal: int;
    prot value: int;
    prot dt: LogDate;
    meth ctor(): Transaction {res:= self}
    meth getDate(): Date {res:= self.dt.getD()}
    meth setDate(d: Date) {self.dt.setD(d)}
}
class CreditTrans ext Transaction {
   meth ctor(): CreditTrans {
      self.dt := (new LogDate).ctor(); res:=self}
   meth finalBal(): int {res := self.bal + self.value}
}
class PayCardTrans ext Transaction {
    meth ctor(): PayCardTrans {
       self.dt := (new LogDate).ctor(); res:=self}
    meth finalBal(): int {res := self.bal - self.value}
}
class ListTransaction ext Object {
    //list of Transactions ordered by date
    priv lt: Transaction;
    pub next: ListTransaction;
    meth ctor(): ListTransaction {res:= self}
    meth getLT(): Transaction {res:= self.lt)}
    meth setLT(lt1: Transaction) {self.lt:= lt1}
}
class Statement ext Object {
    priv llt : ListTransaction;
    meth ctor(): Statement {res:=self}
    meth ShowStatement () primString {
       assert elements in llt are ordered by date;
       ListTransaction tmp;
       tmp:=self.llt;
       res:=tmp.toString()}     }
```

**Figure 2: Final version** (`LogDate` same as in Fig. 1)

transform the code by applying laws introduced in previous work [6], as well as the laws and refactorings presented in Section 4. In this extended abstract we will not formalize all transformation steps. We introduce some of them informally and focus on the ones that involve data refinement, which use the results introduced in this paper.

Beginning with the original version, the first step is to transform the attribute `cdate` of class `CreditTrans` into a private attribute, and to provide methods get and set to access this field. Moreover, every occurrence of field `cdate` in the entire program is replaced by calls to get and set methods.

```
class CreditTrans ext Object {
    prot cbal: int;
    prot cvalue: int;
    priv cdate: Date;
    meth ctor(): CreditTrans {res:= self}
    meth getDateCr(): Date {res:= self.cdate}
    meth setDateCr(d: Date) {self.cdate := d}
    meth finalBal(): int {res:=self.cbal+self.cvalue}}...
```

After that, class `CreditTrans` is transformed to be similar to class `PayCardTrans`, by applying Refactoring 2 introduced in Section 4. This is a data refinement that replaces `cdate: Date` with `cld: LogDate` which adds a layer of indirection. Note that attribute `cld` does not occur in the original version. The object referenced by `cld` cannot be shared between instances of `CreditTrans` as otherwise the refactoring would be unsound. This restriction is imposed by means of the ownership confinement discipline briefly introduced in Section 3. Here, `CreditTrans` is considered to be the owner for class `LogDate`, so that instances of `LogDate` are not shared between transactions.

```
class CreditTrans ext Object {
   prot cbal: int;
   prot cvalue: int;
   priv cld: LogDate;
   meth ctor(): CreditTrans {
      self.cld := (new LogDate).ctor(); res:= self}
   meth getDateCr(): Date {res:= self.cld.getD()}
   meth setDateCr(d: Date) {self.cld.setD(d)}
   meth finalBal(): int {res:=self.cbal + self.cvalue}}...
```

The next step renames methods `getDateCr` and `setDateCr` to, respectively, `getDate` and `setDate`. The same procedure is applied to rename methods of class `PayCardTrans`. This renaming process must also replace all occurrences of the older get and set methods in class `Statement` by the new ones. These methods are not detailed in the description, as they belong to the merge procedure. After that, we introduce class `Transaction` and the inheritance between this class and classes `CreditTrans` and `PayCardTrans`.

```
class Transaction ext Object {
    meth ctor(): Transaction {res:= self}
}
class CreditTrans ext Transaction {
    prot cbal: int;
    prot cvalue: int;
    priv cld: LogDate;
    meth ctor(): CreditTrans {
       self.cld := (new LogDate).ctor(); res:=self}
    meth getDate(): Date {res:= self.dt.getD()}
    meth setDate(d: Date) {self.dt.setD(d)}
    meth finalBal(): int {res := self.bal + self.value}
}
class PayCardTrans ext Transaction {
    prot pbal: int;
    prot pvalue: int;
    priv pld: LogDate;
    meth ctor(): PayCardTrans {
       self.pld := (new LogDate).ctor();res:=self}
    meth getDate(): Date {res:= self.dt.getD()}
    meth setDate(d: Date) {self.dt.setD(d)}
    meth finalBal(): int {res:=self.bal-self.value}}...
```

Now, we change the visibility of `cld` and `pld` to protected, as a preparation for the next step. Then, we move the attributes of classes `CreditTrans` and `PayCardTrans` up to class `Transaction`, by repeatedly applying Refactoring 1 in Section 4. For example, `cld` and `pld` are replaced by `dt` in `Transaction` (below). This is a data refinement transformation, as the original attributes of classes `CreditTrans` and `PayCardTrans` are unified in class `Transaction`. After that, we move methods get and set of classes `CreditTrans` and `PayCardTrans` to class `Transaction`.

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

```
    meth ctor(): PayCardTrans {
       self.dt := (new LogDate).ctor(); res:=self}
    meth finalBal(): int {res:=self.bal - self.value}}...
```

The next step is to introduce a class `ListTransaction` that inherits from `LObject` and is similar to `ListCredit`.

Then we directly apply the simulation law (Law 1 in Section 4). We revise class `Statement` by replacing fields `lc` and `lp` with field `llt`, which is a list of transactions ordered by date. Thus `llt` contains what in the original version is obtained from `lc` and `lp` by method `merge`. In what follows we give a fragment of the description of the new class `Statement`.

```
class Statement ext Object{
    priv llt: ListTransaction;
    meth ctor(): Statement {res:=self}
    priv meth merge ...
    meth ShowStatement(): primString {
      assert elements in llt are ordered by date;
      ListTransaction tmp;
      tmp:=self.llt;
      res:=tmp.toString()}    }
```

To apply the simulation law, we first check the confinement conditions: the lists are owned and not shared between instances of `Statement`. Moreover, we define a local coupling that relates fields of both versions of 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:

$$merge(list(\texttt{self.lc}), list(\texttt{self.lp})) = list(\texttt{self}'\texttt{.llt})$$

Here *list* gives the abstract list represented by a pointer and *merge* is the mathematical function that merges sorted lists. 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.

The next steps eliminate method `merge` as well as classes `ListCredit` and `ListPayCardTrans` which are no longer used. After that, `LObject` does not occur anywhere except as the superclass of `ListTransaction`. We can move fields and methods of `LObject` to `ListTransaction` and afterwards eliminate `LObject`. The result is the final version in Fig. 2.

# 3. CONFINEMENT AND SIMULATION

*Language and semantics.* Our results are formalized for a sequential class based language as seen in the case study. A *class table* is a collection of class declarations, subject to Java-like typing rules. We write $\Gamma \vdash c$ to indicate that command $c$ is well formed in context $\Gamma$, where $\Gamma$ maps a finite set of variables to their types (and the class table is implicit). For example, suppose $m$ is a method in class $K$ with parameter list $\overline{x} : \overline{T}$ and return type $U$. (We let $x, y$ range over variable names and $T, U$ over types, including class names and primitive types like **int**. Overlines indicate lists.) The body of $m$ should be typable in the context $\Gamma = [\mathbf{self} : K, \mathbf{res} : U, \overline{x} : \overline{T}]$. We use a conventional model of states: a state $\sigma$ of type $\Gamma$ assigns type-consistent values to the variables $dom(\Gamma)$, and has a typed heap that maps references to records whose fields are the declared and inherited attributes. Commands denote state transformers, as do expressions (to model side effects of method calls). To cater for mutual recursion between classes and method implementations, the semantics $[\![\Gamma \vdash c]\!](\eta)$ of a command is parameterized by the *method environment* $\eta$ that provides semantics for the methods of the class table. The meaning of a class table *cds* is a method environment, $[\![cds]\!]$, obtained as a limit of approximations which correspond to the semantics with a bound on recursion depth.[1]

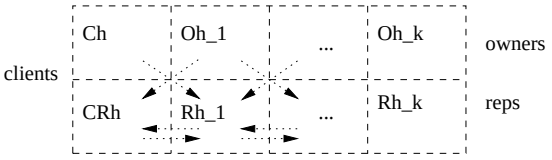


**Figure 3: Confinement scheme.** Dashed boxes are partition blocks and there are $k$ owner islands. Dotted lines indicate disallowed references.

A complete program is comprised of a well formed class table and a command in context $\Gamma \vdash c$. To compare different programs, we do not consider the heap directly observable, only the values of variables. For two state transformers in the same context $\Gamma$, we define $\varphi \doteq \varphi'$ iff when applied to an "empty" initial state $\sigma$, either $\varphi(\sigma)$ and $\varphi'(\sigma)$ are both $\perp$ (denoting divergence) or the resulting states $\tau = \varphi(\sigma)$ and $\tau' = \varphi'(\sigma)$ have the same values for every variable $x \in dom(\Gamma)$ of primitive type.[2] The initial state needs to have an empty heap so that it makes sense for both *cds* and *cds'*, which may declare different private and protected fields for some classes.

Define $cds \bullet \Gamma \vdash c = cds' \bullet \Gamma \vdash c'$ iff $[\![\Gamma \vdash c]\!](\eta) \doteq [\![\Gamma \vdash c']\!](\eta')$ where $\eta = [\![cds]\!]$ and $\eta' = [\![cds']\!]$. We often elide $\Gamma$. Finally, for sets *cs* and *cs'* of classes (with the same public interface), define $cs =_{cds,c} cs'$ iff both $cds\, cs \bullet c$ and $cds\, cs' \bullet c$ are well formed and $cds\, cs \bullet c = cds\, cs' \bullet c$.

*Confinement.* The purpose of confinement is to restrict possible dependencies via pointers, to facilitate local reasoning. In particular, we consider an instance-oriented notion of confinement: a single object can "own" some others — called its *reps*— that must not be pointed to except by each other and by the owner. Notions of ownership have been formalized in class based languages using annotated types [9], ghost state [15], and separation logic [4, 19]. Here we formalize a somewhat restrictive notion of confinement, adapted from [2] and defined in terms of constraints expressed using ordinary types. This takes advantage of nominal typing and has the benefit of relative simplicity.

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 $K$, $K \leq Own$, 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. A confined program maintains an all-states in-

[1]The semantics, from [2, 14], is not fully abstract, but it is sound and suffices to validate the laws of interest.

[2]In fact our formalization compares reference types as well, and relies on a simplifying assumption that the allocator is "parametric" in a certain sense. This lets us avoid working with bijective renamings of references which are needed to handle unobservable differences in allocation. The simplification and the generalization both follow [2] closely.

variant that objects in the heap can be partitioned into certain disjoint regions; see Fig. 3. 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 all *Rep* instances that have no owner, and a region $Ch$ of all remaining objects. Whereas objects of $Rh_i$ are the encapsulated representation of the owner in $Oh_i$, objects in *CRh* are no different from other clients except that they happen to have type $\leq Rep$.

The formalization designates that all instances in *CRh* 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 *CRh*) do not point to owned reps (in the $Rh_i$); (b) the owner and reps in one *island*, say $Oh_i * Rh_i$ (using $*$ for union of disjoint sub-heaps) do not point to reps in another island (say $Rh_j$ with $j \neq i$) or not owned; and (c) the references from an owner to its reps are private or protected fields declared in class *Own* or a subclass of *Own*.

For a state to be *confined*, the heap should be confined and moreover the values of local variables are restricted in accord with **self**. For example, if **self** is the object in $Oh_i$ then the locals cannot point to $Rh_j$ with $j \neq i$.

A command is *confined* provided that it maintains the confinement invariant and moreover the arguments it passes in method calls are confined for the callee. Furthermore, a confined command must preserve partitions in the sense that new reps can be added to an island but a rep cannot be transferred from one owner to another. Transfer is disallowed in many works, including [10], but allowed in, e.g., [16, 15]. The latter provide a notion of ownership that is hierarchical in the sense that owners can be owned; that is suited to program specification and documentation. Here, ownership does not serve to describe program structure but rather is used temporarily, to justify a particular transformation.

A class table is *confined* provided that its methods preserve confinement.

As noted earlier, we do not prove confinement, but rather assume it where needed. We express the assumption in a global way: the complete program must be confined. However, confinement is formulated in a way that caters for modular static analysis that would provide checks on the relevant class hierarchy that entail confinement for the whole program.

***Couplings, simulation, and the abstraction theorem.*** The abstraction theorem provides a technique for proving program equivalence by means of simulations. It pertains to two class tables, *cds* and $cds'$, that are the same except for the implementation of *Own* and its subclasses; both versions have the same public interface. A class *Rep* (resp. $Rep'$) is designated, for which *cds* (resp. $cds'$) is confined.

A *local coupling* is a relation $\mathcal{R}$ on islands; specifically, a set of pairs $(Oh * Rh, Oh' * Rh')$ where $Oh * Rh$ is an island for *cds* and $Oh' * Rh'$ is an island for $cds'$. We lift $\mathcal{R}$ to a relation $\hat{\mathcal{R}}$ on complete states $\sigma, \sigma'$ by stipulating that (a) $\sigma$ and $\sigma'$ are partitioned into the same number of owner islands, (b) $\mathcal{R}$ connects each corresponding pair of owner islands, and (c) objects in $Ch$ and *CRh* have equal field values (and so do public fields of owner objects).

We lift $\hat{\mathcal{R}}$ to a relation on state transformers (also written $\hat{\mathcal{R}}$) in the usual way: Two state transformers are related if, when applied to related initial states, they return related final states. Two method environments are related if the corresponding state transformers denoted by method declarations are related.

A *simulation* is a local coupling $\mathcal{R}$ such that (a) the corresponding implementations of constructors of classes $\leq Own$ establish $\mathcal{R}$, and (b) methods of those classes are related by $\hat{\mathcal{R}}$. The *Abstraction Theorem* says that if $\mathcal{R}$ is a simulation, then the method environments $[\![cds]\!]$ and $[\![cds']\!]$ are related. It rests on this *Preservation Lemma*: Suppose *cds* and $cds'$ are connected by a simulation $\mathcal{R}$. If $\Gamma \vdash c$ and $\Gamma(\mathbf{self}) \not\leq Own$ then $\hat{\mathcal{R}}$ relates $[\![\Gamma \vdash c]\!]([\![cds]\!])$ to $[\![\Gamma \vdash c]\!]([\![cds']\!])$.

Note that $\hat{\mathcal{R}}$ is designed so that if $\Gamma(\mathbf{self}) \not\leq Own$ then related states for $\Gamma$ are observably equal. To prove equality of two complete programs, we find a simulation and then use the preservation lemma for the main command. That is formalized as Law 1 in the next section.

# 4. SIMULATION AND REFACTORINGS

In this section we present a simulation law and two refactorings that may be used to perform sound data refinement transformations of object-oriented programs. Refactorings are regarded as algebraic rules, which have a larger granularity than laws, expressing more robust transformations.

The refactorings are proved by using the simulation law and some laws of classes presented in previous works. A direct application of the law occurs in the case study, when fields `lc` and `lp` are replaced by field `llt`.

Algebraic laws and refactoring rules for an object-oriented language tend to have several side conditions to ensure both well-formedness of the transformed program fragments as well as semantic validity. Here we informally discuss the well-formedness conditions and include as side conditions only those concerned with semantic preservation.

We represent attribute and method declarations of class $K$ by $ads_k$ and $mts_k$, respectively. We also use substitution notation $W[\alpha/\beta]$, meaning that a phrase $\beta$ is replaced by phrase $\alpha$ in the context $W$.

## 4.1 Simulation Law

The following simulation law 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, i.e., it is preserved by corresponding versions of methods.

LAW 1. $\langle simulation \rangle$

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

**provided**

1. *cs* and $cs'$ are hierarchies with root *Own*, with the same classes, public attributes and methods; and *cds* has no subclasses of *Own*;
2. $cds, cs$ is confined for $Own, Rep$ and $cds, cs'$ is confined for $Own, Rep'$;
3. $\Gamma \vdash c$, $\Gamma \vdash' c$, and $\Gamma(\mathbf{self})$ is non-rep and $\not\leq Own$;
4. $c$ is confined in $cds, cs$ and also in $cds, cs'$;
5. there exists $\mathcal{R}$ that is a simulation

Note that confinement is required not only for the relevant class hierarchy, $cs$, but also in $cds$ and $c$. (Indeed, confinement is defined for complete class tables, whereas $cs$ need not be complete on its own.) But as noted earlier, a static analysis would be designed so that if $cs$ is confined then confinement of $cds$ and $c$ holds automatically (cf. [2]).

## 4.2 Refactorings

In this section we present two refactorings involving data refinement, which are used to justify transformations in the case study. In the first one, only fields of the owner class are changed, no other objects. To fit the confinement theory, we choose for *Rep* a class *Junk* that is never instantiated; thus the confinement conditions are vacuously true. If such class does not exist we can introduce it, and after performing the transformation we can eliminate it using the laws presented in [6].

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/Push Down Field*⟩.

On the left-hand side of the refactoring, 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 then $K$ 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 $z$.

REFACTORING 1. ⟨*Pull Up/Push Down Field*⟩

| | | |
|---|---|---|
| **class** $M$ **ext** $N\{$<br>$ads_m$;<br>$mts_m$<br>$\}$<br>**class** $L$ **ext** $M\{$<br>**prot** $x : T$; $ads_l$;<br>$mts_l$<br>$\}$<br>**class** $K$ **ext** $M\{$<br>**prot** $y : T$; $ads_k$;<br>$mts_k$<br>$\}$<br>$cds_1$ | $=_{cds,c}$ | **class** $M$ **ext** $N\{$<br>**prot** $z : T$; $ads_m$;<br>$mts_m$<br>$\}$<br>**class** $L$ **ext** $M\{$<br>$ads_l$;<br>$mts'_l$<br>$\}$<br>**class** $K$ **ext** $M\{$<br>$ads_k$;<br>$mts'_k$<br>$\}$<br>$cds'_1$ |

**where**

$mts'_k = mts_k[z/x] \qquad mts'_l = mts_l[z/y]$

$cds_1$ contains the subclasses of $M$ other then $K$ and $L$ and $cds$ contains no subclasses of $M$;

$cds'_1 = cds_1[u.z, v.z/u.x, v.y]$, for all variables $u$ of type $L_1$, $L_1 \leq L$ and all variables $v$ of type $K_1$, $K_1 \leq K$.

**provided**

$c$ and all classes are confined for $[M, Junk/Own, Rep]$.

To guarantee well-formedness, when the refactoring is applied from left to right, $z$ does not occur in $cd_m$, $cd_l$ , $cd_k$, $cds$, $cds_1$ and $c$. Furthermore, when the refactoring is applied from right to left, $x$ (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 $cds'_1$.

One might think that type $T$ in class $M$ should be our choice for *Rep*, in case $T$ is not primitive. However, validity of this refactoring does not require a restriction on sharing, and such situations are encompassed by our simulation law.

The proof first applies a law (from [6]) to move attribute to superclass. After that, instantiate Law 1⟨*simulation*⟩ with $[(cds_m, cds_l, cds_k, cds_1)/cs]$, $[(cds'_m, cds'_l, cds'_k, cds'_1)/cs']$, $[M, Junk, Junk/Own, Rep, Rep']$. Moreover, define the local coupling $\mathcal{R}_1$ as follows.

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

Formally, $\mathcal{R}_1$ is a set of pairs of partial heaps; each of those heaps has the form of an island. For clarity, we use a formula where **self** refers to the owner of one island and **self′** to the owner of the other. Note that the objects of type $\leq M$ are owners, but the rep part of an island is empty: the coupling imposes a condition on the $x/y/z$ fields but not on the objects they reference.

To prove $\mathcal{R}_1$ is a simulation, we prove something stronger: For every sub-expression $e$ (resp. every sub-command $c$) of the original code (of the hierarchy rooted in $M = Own$), if $e'$ is the transformed version then $[\![e]\!]$ relates to $[\![e']\!]$ via $\hat{\mathcal{R}}_1$ (i.e., they preserve the simulation). The proof goes by structural induction and relies on particular conditions in $\mathcal{R}_1$.

Refactoring 2 ⟨*Change Attribute Types*⟩ expresses a data refinement of the attribute $x$ of class $K$. On the left-hand side of the rule $x$ is of type $T$. On the right-side, this attribute is replaced by an attribute $z$ of type $L$. Class $L$ has an attribute $xx$ of the same type, $T$, as $x$. This refactoring is useful to improve reusability of a class.

On the right-hand side, in $K$, original direct accesses to attributes that are now in $L$ are replaced with calls to the *get* and *set* methods of $L$. We express the data transformation of only one attribute and this can be straightforwardly generalized to several attributes, with their respective *get* and *set* methods.

REFACTORING 2. ⟨*Change Attribute Types*⟩

| | | |
|---|---|---|
| **class** $K$ **ext** $M\{$<br>**priv** $x : T$; $ads_k$;<br>**meth ctor**$(\overline{y} : \overline{u}) : K$<br>$\{C;$ **res** := **self**$\}$<br>**meth** $getX() : T$<br>$\{$**res** := **self**$.x\}$<br>**meth** $setX(v : T)$<br>$\{$**self**$.x := v)\}$<br>$mts_k$<br>$\}$<br>**class** $L$ **ext Object**<br>**priv** $xx : T$;<br>**meth ctor**$() : L$<br>$\{$**res** := **self**$\}$<br>**meth** $getXX() : T$<br>$\{$**res** := **self**$.xx\}$<br>**meth** $setXX(v : T)$<br>$\{$**self**$.xx := v)\}$<br>$\}$ | $=_{cds,c}$ | **class** $K$ **ext** $M\{$<br>**priv** $z : L$; $ads_k$;<br>**meth ctor**$(\overline{y} : \overline{u}) : K$<br>$\{C;$ **self**$.z :=$<br>**new** $L$**.ctor**$();$<br>**res** := **self**$\}$<br>**meth** $getX() : T$<br>$\{$**res** :=<br>**self**$.z.getXX()\}$<br>**meth** $setX(v : T)$<br>$\{$**self**$.z.setXX(v)\}$<br>$mts'_k$<br>$\}$<br>**class** $L$ **ext Object**<br>**priv** $xx : T$;<br>**meth ctor**$() : L$<br>$\{$**res** := **self**$\}$<br>**meth** $getXX() : T$<br>$\{$**res** := **self**$.xx\}$<br>**meth** $setXX(v : T)$<br>$\{$**self**$.xx := v)\}$<br>$\}$ |

**where**

$$mts'_k = mts_k[le := e[\mathbf{self}.z.getXX()],\ \mathbf{self}.z.setXX(e)\ /\ le := e[\mathbf{self}.x],\ \mathbf{self}.x := e]$$

**provided**

cds, c and $cd_k$, $cd_l$ and $cd_n$, for each $N < K$ inside $cds$, are confined for $[K, L/Own, Rep]$.

To guarantee well-formedness, when the refactoring is applied from left to right, the class $L$ is not declared in $cds$ and the attribute $z$ is not declared in $ads_k$ nor in any subclass or superclass of $K$ . Furthermore, when the refactoring is applied from right to left, the class $L$ is not used in $cds$ or $c$.

In this refactoring, confinement requires that the objects referenced by $z$ fields are not shared between instances of $K$. With sharing, the refactoring would not be sound without significant additional restrictions.

To prove this refactoring we instantiate Law 1 $\langle simulation \rangle$: let $cds_1 = cds_n\, cd_k$ and instantiate $[cds_1/cs]$, $[cds'_1/cs']$, $[(cds\, cd_l - cds_n)/cds]$, $[K, L, L/Own, Rep, Rep']$. The local coupling $\mathcal{R}$ defined as follows:

$$\mathtt{self} = \mathtt{self}' \wedge type(\mathtt{self}) = type(\mathtt{self}') \wedge \mathtt{self}.x = \mathtt{self}.z.xx \wedge (\forall f \in cds_K \bullet \mathtt{self}.f = \mathtt{self}'.f).$$

We follow similar reasoning as in the proof of Refactoring 1; here the induction is on $mts_k$.

# 5. DISCUSSION

As far as we are aware, no previous work in the literature has proposed a simulation law for arbitrary inheritance hierarchies. This law has been formally derived from the abstraction theorem, which is itself a generalization of the one originally presented in [2] for a single class. Furthermore, the simulation law has been used both to prove general refactorings, as well as in more specific contexts, as illustrated by the case study.

The case study demonstrates the generality of the refactoring rules, but our confinement conditions rule out some interesting programs such as those in which ownership is transferred. We intend to investigate more flexible notions, catering for refactorings of interest. We hope other researchers will take up the challenge to prove the refactoring rules using alternative techniques.

# 6. ACKNOWLEDGMENTS

We would like to thank the anonymous referees, whose comments helped us to improve this paper.

# 7. REFERENCES

[1] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *POPL*, pages 340–353, 2009.

[2] A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *J. ACM*, 52(6), 2005.

[3] A. Banerjee and D. A. Naumann. State based ownership, reentrance, and encapsulation. In *ECOOP*, volume 3586 of *LNCS*, pages 387–411, 2005.

[4] G. Bierman and M. Parkinson. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.

[5] L. Birkedal and H. Yang. Relational parametricity and separation logic. *Logical Methods in Comp. Sci.*, 4(2), 2008.

[6] P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornélio. Algebraic reasoning for object-oriented programming. *Sci. Comput. Programming*, 52(1-3):53–100, 2004.

[7] A. L. C. Cavalcanti and D. Naumann. A weakest precondition semantics for refinement of object-oriented programs. *IEEE Trans. Softw. Eng.*, 26(8):713–728, 2000.

[8] A. L. C. Cavalcanti and D. A. Naumann. Forward simulation for data refinement of classes. In *Formal Methods Europe*, volume 2391 of *LNCS*, pages 471–490, 2002.

[9] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, pages 292–310, Nov. 2002.

[10] D. Dreyer, G. Neis, A. Rossberg, and L. Birkedal. A relational modal logic for higher-order stateful ADTs. In *POPL*, pages 185–198, 2010.

[11] I. Filipović, P. O'Hearn, N. Torp-smith, and H. Yang. Blaming the client: On data refinement in the presence of pointers. *Formal Aspects Comp.*, 2010.

[12] V. Koutavas and M. Wand. Bisimulations for untyped imperative objects. In *ESOP*, volume 3924 of *LNCS*, pages 146–161, 2006.

[13] V. Koutavas and M. Wand. Reasoning about class behavior. Informal proc. of FOOL/WOOD, 2007.

[14] G. T. Leavens, D. A. Naumann, and S. Rosenberg. Preliminary definition of core JML. Technical Report CS Report 2006-07, Stevens Inst. Tech., 2006.

[15] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, pages 491–516, 2004.

[16] P. Müller and A. Rudich. Ownership transfer in Universe Types. In *OOPSLA*, pages 461–478, 2007.

[17] L. Silva, D. A. Naumann, and A. Sampaio. Refactoring and representation independence for class hierarchies. Draft journal paper, `www.cs.stevens.edu/~naumann/pub/clHieDraft.pdf`.

[18] L. Silva, A. Sampaio, and Z. Liu. Laws of object-orientation with reference semantics. In *SEFM*, pages 217–226, 2008.

[19] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, pages 148–172, 2009.

[20] E. Sumii. A complete characterization of observational equivalence in polymorphic lambda-calculus with general references. In *Computer Science Logic*, volume 5771 of *LNCS*, pages 455–469, 2009.

[21] E. Sumii and B. C. Pierce. A bisimulation for type abstraction and recursion. *J. ACM*, 54(5), 2007.