Laws of Programming for References

Giovanny Lucero[1], David Naumann[2], and Augusto Sampaio[1] [*]

[1] Centro de Informática, Universidade Federal de Pernambuco
gflp@cin.ufpe.br, acas@cin.ufpe.br
[2] Dept of Computer Science, Stevens Inst of Technology,
naumann@cs.stevens.edu

Abstract. We propose a set of algebraic laws for reasoning with sequential imperative programs that use object references like in Java. The theory is based on previous work by adding laws to cope with object references. The incrementality of the algebraic method is fundamental; with a few exceptions, existing laws for copy semantics are entirely reused, as they are not affected by the proposed laws for reference semantics. As an evidence of relative completeness, we show that any program can be transformed, through the use of our laws, to a normal form which simulates it using an explicit heap with copy semantics.

1 Introduction

The inherent difficulty of reasoning with pointers has been successfully addressed using different techniques for describing spatial separation of pointers, see for example [19,7,13,3]. However, there have been few initiatives using algebraic approaches [12,20], despite its well known advantages. Transformations are used in compilers, but these rely on context conditions presented algorithmically or by means of logic (as discussed in [4]); in many cases they apply only to intermediate representations. No comprehensive set of algebraic laws has been proposed to support transformations of source programs involving references.

In [10], Hoare and Staden highlight, among other advantages, the incrementality of the algebraic method. When a new programming language concept or design pattern is added, new axioms can be introduced, keeping intact at least some axioms and theorems of the existing theory of the language. Even when the new language features have an impact on the original ones, this tends to be controllable, affecting only a few laws. On the other hand, a pure algebraic presentation is based on postulating algebraic laws, which raises the questions of consistency and completeness of the proposed set of laws.

In this paper, we explore the incrementality of algebra to extend with object references a simple non-deterministic imperative language similar to that given in the seminal "Laws of Programming" by Hoare et al [9] (*LoP* for short). Our

[*] Authors from [1] are partially supported by the National Institute of Science and Technology for Software Engineering (INES), funded by Brazil CNPq, grant 573964/2008-4. The second author is partially supported by US NSF award CNS 1228930.

language deals with references in the same way as in Java, i.e., aliasing (sharing) may only occur for fields of objects since the only operations with references are field access to read and write. There is no arithmetic on references nor operations returning addresses of variables.

Based on LoP, we generalize some laws to deal with references, in particular those related to assignments to object fields. The main difficulties are due to aliasing. We tackle them using techniques inspired in works of Morris [17] and Bornat [7] on program logic. We also propose new laws to deal with object creation and manipulation of assertions. Dynamic allocation poses challenges in semantic modeling, and thus in semantic notions of completeness [21]. As in LoP, we address the completeness of the laws by showing that any program can be transformed, by using the laws, to a normal form. Soundness of the laws is tackled in an extended report [15] by proving the validity of the laws with respect to a naive denotational semantics (which suffices because we only consider first order programs).

Our work is in a wider context of defining algebraic theories for reasoning about object-oriented programs. Previous work [6,8] defines theories for object-orientation useful to prove transformations of programs such as refactorings. However, the language used in these works has copy semantics, lacking the concept of reference and, thus, restricting the refactorings that can be characterized. By "copy semantics" we mean that it is only simple variables that are mutable; objects are immutable records (i.e., functional maps with update by copy) so aliasing cannot occur.

Our laws are intended to support program transformation and verification. They can be used, for instance, in the design of correct compilers and optimizers. Together with laws of object-orientation established in previous works, which are also valid in the context of references, they can be applied to a wider collection of refactorings and patterns which depend on the concept of reference.

In the next Section we show the abstract syntax of the language and briefly explain its constructions. Section 3 discusses how aliasing complicates the algebraic reasoning with programs and describes a substitution mechanism that deals with it. The laws are given in Section 4 and a notion of relative completeness for the proposed set of laws is shown in Section 5. Section 6 presents final considerations, including other notions of completeness, and discusses related and future works. Proofs and further details appear in the long version of this paper [15].

2 The language

The programming language we consider is sequential and imperative. It extends that in LoP by including object references as values and assignments to object fields. The language is statically typed, thus each variable and field has a statically declared type. Formalization of types is routine and omitted.

The abstract syntax is in Figure 1. In this grammar x, X, f, and K range over given sets representing names for variables, recursive commands, fields of

c	$::= x \leftarrow \mathbf{new}\, K \mid \overline{le} := \overline{e} \mid$	new instance, simultaneous assignment
	$\perp \mid [e] \mid \mathbf{skip} \mid c\,;\ c \mid$	abort, assertion, skip, sequence
	$c \lhd e \rhd c \mid c \cup c \mid e * c \mid$	conditional, non-determinism, while
	$\mu\, X \bullet c \mid X$	recursion, recursive call
le	$::= x \mid \mathbf{null}.f \mid le.f \mid le \lhd e \rhd le$	variable, field, conditional
e	$::= e\ op\ e \mid e \lhd e \rhd e \mid$	binary operator (e.g. $==$), conditional
	$x \mid e.f \mid \mathbf{null} \mid \mathbf{false} \mid 0 \mid 1 \mid \ldots$	variable, field access, constants, others
cd	$::= \mathbf{class}\ K\ \{\overline{f : T}\}$	class declaration
T	$::= K \mid \mathbf{bool} \mid \mathbf{int}$	types
$prog$	$::= \overline{cd} \bullet c$	program

Fig. 1. The Syntax of the Language

objects, and classes, respectively. The non terminal symbols cd, T, c, le and e are used for class declarations, types, commands, left expressions and expressions, respectively. As a convention, a line above syntactic elements denotes a list of them. Thus, for example, we use $\overline{e}$ to abbreviate a list $e_1, e_2, \ldots e_n$ of expressions, for some n. But the identifier $\overline{e}$ has no relation with the identifier e which represents any single expression not necessarily in $\overline{e}$. The empty list is written as "()".

A class declaration defines a named record type, i.e., $\mathbf{class}\ K\ \{\overline{f : T}\}$ declares a class with name K and fields $\overline{f : T}$. There are no methods and all the fields in the class are public. In our laws of commands, we assume there is a fixed set of class declarations which may be mutually recursive.

Like in Java, variables or fields of primitive types, **bool** and **int**, store values of the corresponding types. On the other hand, any variable or field having some K as its declared type does not store an object instance of K but a reference to it. This is the main difference of our language with respect to [6,8] where a copy semantics is adopted and variables/fields hold full objects.

The expression $\mathbf{null}.f$ is allowed, though it always aborts, because it may arise from the application of laws.

The language has all the programming commands given in LoP, extended with commands for creation of objects, assertions and richer left expressions for assignments to fields. We omit specification constructs like angelical choice. We use the same syntax as in LoP, e.g., $b \lhd e \rhd c$ and $e * c$ are conditional and iteration commands corresponding to the **if** and **while** of Java, respectively. In both cases e is the guard condition. For recursion we use the notation $\mu\, X \bullet c$, which binds X and defines a command where recursive calls can happen within c. The while command can be defined in terms of the μ construct: $e{*}c\ \widehat{=}\ \mu\, X \bullet c;\ X{\lhd}e{\rhd}\mathbf{skip}$.

The non-deterministic command $b \cup c$ denotes the demonic choice between b and c. The command $\perp$, also known as abort, represents the behaviour of a failing program. It is the most non-deterministic program since its execution may result in any state or may even fail to terminate.

The simultaneous assignment $\overline{le} := \overline{e}$ requires the same size for both lists, and, of course, the same type for each corresponding left and right expression. The assignment is executed by evaluating all the expressions in $\overline{e}$ and then assigning each resulting value to the corresponding left expression in $\overline{le}$. The assignment command in our language differs from that of LoP in some important aspects. First, it is allowed to have empty lists at each side of ":=". Indeed, we can define **skip** as $() := ()$, whose execution always ends successfully without performing any change. Second, left expressions are richer than those in LoP where only variables are allowed. Here, left expressions can also be fields of objects or even conditional left expressions.

Notably, it is allowed to have repeated left expressions on the left-hand side list of ":=". When this happens, the execution is non-deterministic. For example, the execution of $x, x := 1, 2$ assigns either 1 or 2 to x. This kind of non-determinism can happen even with syntactically distinct left expressions when the left-hand side list includes aliased fields, so we may as well allow it for variables as well.

The command $x \leftarrow \mathbf{new}\, K$ creates in the heap a new instance of the class K and assigns its reference to x. The fields of the new instance are initialized by default with 0 for **int**, false for **bool** and **null** for objects.

Assertions are denoted by $[e]$, where e is a boolean expression. $[e]$ is a command that behaves like **skip** when e evaluates to **true** and like $\bot$ otherwise. Our assertions use only program expressions and can be defined in terms of more basic operators: $[e] \mathrel{\widehat{=}} \mathbf{skip} \triangleleft e \triangleright \bot$. However, assertions may refer to a distinguished variable, **alloc**, which in any state holds the set of currently allocated references. This device has been used in some program logics (e.g., [3]).

As usual we model input and output by the global variables of the main program. These global variables are assumed to have implicit type declarations.

We use a, b, c to stand for commands, d, e for expressions, f, g, k for field names, m for case lists (defined in Subsection 3.2), p, q, r stand for left expressions, x, y, z stand for variables, K stands for class names and T stands for type names, i.e. primitive types and classes.

3 Aliasing and Substitution

A fundamental idea behind algebra of programs [9,16,1], as well as in Hoare Logic and many other formalisms, is that variable substitution can capture the effects that assignments produce on expressions. However, when the language has references, several obstacles emerge. The main difficulty resides in the possibility of aliasing, which happens when two or more expressions or left expressions point to a single object field.

3.1 The Problem of Aliasing

Consider the following law, as originally presented in LoP, where aliasing is not possible. It combines two sequential assignments into a single one. The notation $d^{\overline{x}}_{\overline{e}}$ denotes a simultaneous substitution on d where each variable in the list $\overline{x}$ is replaced by the corresponding expression in $\overline{e}$. This law captures the behavior of the first assignment in the sequence through syntactic substitution.

$$(\overline{x} := \overline{e}\,;\ \overline{x} := \overline{d}) \quad = \quad \overline{x} := \overline{d}^{\overline{x}}_{\overline{e}} \lhd \mathfrak{D}\overline{e} \rhd \bot \qquad (*)$$

Just like in LoP, we assume that for every expression e there is an expression $\mathfrak{D}e$ for checking if e is defined. We assume that $\mathfrak{D}e$ is always defined. For example, $\mathfrak{D}(x/y)$ is given by $y \neq 0$. Considering the left-hand side of the equation (*) above, if $\overline{e}$ is undefined it will behave as $\bot$. This justifies the checking for definedness on the right-hand side.

Under the conditions given in LoP, where the left-hand side $\overline{x}$ is a list of variables without duplicates, the law is valid. But our language allows duplicates on left-hand sides. Obviously, in this case the law does not make sense since the substitution $\overline{d}^{\overline{x}}_{\overline{e}}$ will not be well defined.

Moreover, because of the possibility of aliasing, we get in trouble if we apply the law to assignments with fields as left expressions. Note that a variable always denotes the same memory cell both in the first and in the second assignment on the left-hand side of the law. This is not true with fields. For example, in the command $x, x.f := a, b;\ x, x.f := x, x.f + 1$, the left expression $x.f$ in the second assignment may be referring to a different cell from that in the first since x may have changed to reference an object referenced by a.

It is well known that, with pointers or array, such a law is not valid if naive substitution is used for left expressions. For example, for the command $p.f := p.f + 1;\ p.f := p.f + q.f$, a naive substitution $(p.f + q.f)^{p.f}_{p.f+1}$ will give always $(p.f + 1) + q.f$, ignoring the possibility of aliasing between $p.f$ and $q.f$.

In order to generalize law (*) above for our language, we prefix a guard asserting that the left expressions of the first assignment are pairwise disjoint and, furthermore, that the left expressions of both assignments refer to the same cells. Also, we need to use a substitution mechanism that deals with aliasing and simultaneous assignments. In the following subsection we give a precise definition of substitution. We give the generalization of law (*) in Section 4 – see law (33).

Like in Java, aliasing may only occur for fields of objects. More precisely, field accesses as $p.f$ and $p.g$, where f and g are different field names, can never have the same *lvalues*, i.e, will never be aliased. On the other hand, $p.f$ and $q.f$ are aliased if and only if $p == q$, meaning that p and q hold the same value (object address). For simplicity we consider field names to be globally unique, i.e., distinct classes use distinct field names.

We define a state predicate $alias[p, q]$ that is true if and only if p and q are aliased. For convenience, we consider that any left expression is aliased with itself. We are using square brackets to stress that the arguments of *alias* are not semantic values but syntactic elements. Note however that the aliasing checking is state dependent. By recursion on structure we define

$$\begin{array}{rlrl}
alias[x, x] & \mathrel{\widehat{=}} \mathbf{true} & alias[p.f, q.g] & \mathrel{\widehat{=}} \mathbf{false} \\
alias[x, y] & \mathrel{\widehat{=}} \mathbf{false} & alias[p \lhd e \rhd q, r] & \mathrel{\widehat{=}} alias[p, r] \lhd e \rhd alias[q, r] \\
alias[x, p.f] & \mathrel{\widehat{=}} \mathbf{false} & alias[p, q] & \mathrel{\widehat{=}} alias[q, p], \quad \textit{otherwise} \\
alias[p.f, q.f] & \mathrel{\widehat{=}} p == q & &
\end{array}$$

where $x \not\equiv y$ and $f \not\equiv g$. We write $\equiv$ to mean syntactically the same.

3.2 Substitution

For substitution we borrow ideas from Bornat [7] who defines a field substitution operator for languages that treat references like in Java, as our language does. Like we do in this paper, he also assumes that distinct types of objects use distinct field names, which is useful to simplify the definition. We write $e^f_{\{f:p\mapsto d\}}$ to denote the expression obtained by syntactically replacing occurrences of the field name f by the *conditional field* $\{f : p \mapsto d\}$. A conditional field access $e_1.\{f : p \mapsto d\}$ is interpreted as being $d \triangleleft e_1 == p \triangleright e_1.f$.

As an illustrative example, consider the effect caused by the assignment $x.f := e$ on the expression $x.f + y.g + z.f$. This effect is given by the following field substitution

$$\begin{array}{lr}
(x.f+ y.g + z.f)^f_{\{f:x\mapsto e\}} = & \text{(def. of substitution)} \\
\quad x.\{f : x \mapsto e\} + y.g + z.\{f : x \mapsto e\} = & \text{(desugaring)} \\
\quad (e \triangleleft x == x \triangleright x.f) + y.g + (e \triangleleft x == z \triangleright z.f) = & (x == x \text{ is true}) \\
\quad e + y.g + (e \triangleleft x == z \triangleright z.f) &
\end{array}$$

As expected, observe that, contrasting to the initial expression, in the resulting expression $x.f$ was replaced by e and $z.f$ by a conditional indicating that if $x.f$ and $z.f$ are aliased, $z.f$ also will be replaced by e, but if there is no such aliasing $z.f$ will be kept intact.

Field substitution also works for expressions containing nested accesses to fields. For example, it is easy to see that

$$(x.f.f)^f_{\{f:y\mapsto e\}} \quad = \quad e \triangleleft (e \triangleleft x == y \triangleright x.f) == y \triangleright (e \triangleleft x == y \triangleright x.f).f$$

Because our language has simultaneous assignments, we need to extend Bornat's operator to work with simultaneous substitutions of fields and variables. We start by defining a syntax sugar. A *case expression* is defined by

$$\begin{array}{l}
\textbf{case } e \textbf{ of } e_1 \rightarrow d_1, \ldots, e_n \rightarrow d_n \textbf{ else } d_{n+1} \quad \widehat{=} \\
\qquad\qquad d_1 \triangleleft e == e_1 \triangleright (\ldots (d_n \triangleleft e == e_n \triangleright d_{n+1}) \ldots)
\end{array}$$

Note that the order of the items in the case list is important, since it is chosen the first ith branch such that $e == e_i$.

We use m to represent case list like $e_1 \rightarrow d_1, \ldots, e_n \rightarrow d_n$. We then extend the notion of conditional field by allowing case lists and define

$$e.\{f : m\} \;\widehat{=}\; \textbf{case } e \textbf{ of } m \textbf{ else } e.f$$

Consider a list $\overline{xf}$ containing variable names and field names, and a corresponding list $\overline{em}$ containing expressions and conditional fields. Suppose the list $\overline{xf}$ has no repeated variable or field names, each variable in $\overline{xf}$ corresponds positionally to an expression in $\overline{em}$, and each field f in $\overline{xf}$ corresponds to a conditional field $\{f : m\}$ in $\overline{em}$. We denote with $e^{\overline{xf}}_{\overline{em}}$ the simultaneous substitution on e of each element in $\overline{xf}$ by the corresponding element in $\overline{em}$. For example, $e^{x,f,y}_{d_1,\{f:m\},d_2}$ represents the simultaneous substitution on e of x by d_1, f by $\{f : m\}$ and y by d_2.

We relax the notation to allow duplication of fields, but not variables, in $\overline{xf}$. In this case we interpret that the corresponding conditional fields are concatenated into a single one. For example, $e^{x,f,y,f,g}_{d_1,\{f:m_1\},d_2,\{f:m_2\},\{g:m_3\}}$ (with $f \not\equiv g$) can be written as $e^{x,f,y,g}_{d_1,\{f:m_1,m_2\},d_2,\{g:m_3\}}$. Note that in the conditional field $\{f : m_1, m_2\}$ the cases on m_1 have priority over those in m_2, since the order in a case list is relevant. Our simultaneous substitution prioritizes the first listed field substitutions, which may appear arbitrary. However, in our laws all uses of simultaneous substitutions will be for *disjoint assignments*, i.e., guarded by assertions that ensures that the left expressions reference distinct locations.

Field substitution can also be applied to left expressions. However, we need a slightly different notion because left values and right values need to be treated differently. We use $le^{/f}_{\{f:m\}}$ to denote a field substitution on the left expression. The idea is that a field substitution applied on a left expression like $x.f_1.f_2 \ldots f_n$ always ignores the last field, keeping it in the result, i.e.,

$$(le.g)^{/f}_{\{f:m\}} \;\widehat{=}\; le^{f}_{\{f:m\}}.g$$

even when f and g are the same name field. The field substitution on the right side of the equation is that described above for expressions.

4 The Laws

In this section we give an algebraic presentation for our imperative language. In general, our laws are either the same or generalizations of those in LoP. Furthermore, as one might expect, the only laws to be generalized are those related to assignment. Also, we establish a few orthogonal laws only related to references, for example one that allows to exchange a left expression by another aliased with it.

Our laws can be proved sound in a simple denotational model like that described in LoP. The main difference is that our program states include the heap as well as valuation of the variables in scope. A command denotes a relation from initial states to outcomes, where an outcome is an ordinary state or the fictitious state $\perp$ that represents both divergence and runtime error. For any initial state s, if the related outcomes include $\perp$ then s must also relate to all states; otherwise, the image of s is finite and non-empty. In this semantics, refinement is simply inclusion of relations, and equations mean equality of denotations. The semantics is parameterized on an arbitrary allocator that, for any state, returns a finite non-empty set of unused references (so **new** is boundedly non-deterministic).

A shortcoming of this simple semantics is that it does not validate laws like $x \leftarrow \mathbf{new}\, K = x \leftarrow \mathbf{new}\, K;\ x \leftarrow \mathbf{new}\, K$ for which equality of heaps is too

$$\overline{p}, \overline{q}, \overline{r} := \overline{e_1}, \overline{e_2}, \overline{e_3} = \\ \overline{q}, \overline{p}, \overline{r} := \overline{e_2}, \overline{e_1}, \overline{e_3} \quad (1)$$

$$\overline{p} := (\overline{e_1} \lhd d \rhd \overline{e_2}) = \\ (\overline{p} := \overline{e_1}) \lhd d \rhd (\overline{p} := \overline{e_2}) \quad (2)$$

$$\mu X \bullet F(X) = F(\mu X \bullet F(X)) \quad (3)$$

$$F(Y) \sqsubseteq Y \Rightarrow \mu X \bullet F(X) \sqsubseteq Y \quad (4)$$

$$b \lhd \textbf{false} \rhd c = c = c \lhd \textbf{true} \rhd b \quad (5)$$

$$b \lhd d \Rightarrow e \rhd c = \\ (b \lhd e \rhd c) \lhd d \rhd b \quad (6)$$

$$c \lhd e \rhd c = c \lhd \mathfrak{D}e \rhd \bot \quad (7)$$

$$(b \lhd e \rhd c) \lhd \mathfrak{D}e \rhd \bot = b \lhd e \rhd c \quad (8)$$

$$(a \lhd e \rhd b)\,;\; c = \\ (a\,;\; c) \lhd e \rhd (b\,;\; c) \quad (9)$$

Fig. 2. Selected laws kept intact from LoP [9].

strong. An appropriate notion of program equality considers heaps up to bijective renaming of references (as in [2]). In this paper, we do not need such laws because the normal form reduction preserves allocation order.

Figure 2 lists some laws given in LoP that do not depend on references and therefore remain intact. For lack of space, we do not explain in detail these laws here. Readers not familiar with them are referred to [9].

The refinement relation, $\sqsubseteq$, is defined by $b \sqsubseteq c \mathrel{\widehat{=}} (b \cup c) = b$. The set of programs with $\sqsubseteq$ is a semi-lattice, where $\cup$ is the meet and $\bot$ the bottom, and all the command constructions are monotonic (indeed, continuous). Law (3) on recursion says that $\mu X.F(X)$ is a fixed point and law (4) that it is the least fixed point.

4.1 Laws for Assertions

We will be especially concerned with assertions that enable reasoning about aliasing. In Figure 3 we provide a set of laws that allow to decorate commands with assertions. Some of these laws add information brought from the conditions of conditional and while commands. Others spread assertions forward in the commands deducing them from some already known assertions at the beginning. Assertions are defined as syntax sugar, and all these laws are derived except for law (22) which is proved directly in the semantics using the definition of *alias*.

Laws (10)–(21) for assertions should be familiar. Law (22) enables to replace one alias by another. The hypothesis for laws (23) and (24) may be better understood if we interpret it through partial correctness assertions (triples) from Hoare logic. Observe that when an equation $[e_1];\; c = [e_1];\; c;\; [e_2]$ holds, if and when the execution of $[e_1];\; c$ finishes, $[e_2]$ must be equivalent to **skip**, which is the same as saying that e_2 must hold. That is exactly the same meaning intended for a triple $\{e_1\}c\{e_2\}$ from Hoare logic (cf. [14]). Laws (23) and (24) state that any invariant is satisfied at the beginning of each iteration and at the end of the loop.

We give an additional law for assertions that expresses the axiom for assignment in Hoare logic. For this we need some definitions. A *path* is a left expression that does not use conditionals, i.e., a path is a variable x, or a sequence of field accesses using the dot notation like $e.f.g$, for example. Given an assignment on

$$[e];\ c = c \lhd e \rhd \bot \quad (10)$$
$$[d \wedge e] = [d];\ [e] \quad (11)$$

$$b \lhd e \rhd c = ([\mathfrak{D} e];\ b) \lhd e \rhd c \quad (12)$$
$$b \lhd e \rhd c = b \lhd e \rhd ([\mathfrak{D} e];\ c) \quad (13)$$
$$b \lhd e \rhd c = ([e];\ b) \lhd e \rhd c \quad (14)$$
$$b \lhd e \rhd c = b \lhd e \rhd ([\mathbf{not}\ e];\ c) \quad (15)$$

$$[e];\ (b \lhd d \rhd c) = ([e];\ b) \lhd d \rhd ([e];\ c) \quad (16)$$

$$[e];\ (b \cup c) = ([e];\ b) \cup ([e];\ c) \quad (17)$$

$$e * c = e * ([\mathfrak{D} e];\ c) \quad (18)$$
$$e * c = (e * c);\ [\mathfrak{D} e] \quad (19)$$
$$e * c = e * ([e];\ c) \quad (20)$$
$$e * c = (e * c);\ [\mathbf{not}\ e] \quad (21)$$

$$[alias[q, r]];\ \overline{p}, q := \overline{e}, d\ = [alias[q, r]];\ \overline{p}, r := \overline{e}, d \quad (22)$$

If $[d \wedge e];\ c = [d \wedge e];\ c;\ [d]$ then

$$[d];\ e * c = [d];\ e * ([d];\ c) \quad (23)$$
$$[d];\ e * c = [d];\ (e * c);\ [d] \quad (24)$$

Fig. 3. Laws for assertions.

paths $\overline{p} := \overline{e}$, we define functions *vf* and *em* to build a corresponding substitution. Define $vf[x] = x$ if x is a variable, $vf[p.f] = f$ and $vf[\overline{p}]$ is obtained applying *vf* to every element of $\overline{p}$. Also, for each e_i in $\overline{e}$, we define $em[e_i] = e_i$, if the corresponding $vf[p_i]$ is a variable, and $em[e_i] = \{f : p_i \mapsto e_i\}$ if $vf[p_i] = f$. Finally, $em[\overline{e}]$ is obtained applying *em* to each element of $\overline{e}$.

Let $\overline{p}$ be a list of paths $(p_1, p_2, \ldots, p_n)$. Define the disjointness state predicate

$$disj[\overline{p}] \mathrel{\widehat{=}} \forall\, i, j \bullet i \neq j \Rightarrow \mathbf{not}\ alias[p_i, p_j]$$

using "∀" as shorthand for conjunction. We have the law

$$[disj[\overline{p}]];\ [d^{vf[\overline{p}]}_{em[\overline{e}]}];\ \overline{p} := \overline{e} \quad = \quad [disj[\overline{p}]];\ \overline{p} := \overline{e};\ [d] \quad (25)$$

The assertion at the beginning of both sides of the equation ensures that the substitution $d^{vf[\overline{p}]}_{em[\overline{e}]}$ is well defined. The law states that if d is a post-condition of the assignment $\overline{p} := \overline{e}$ then $d^{vf[\overline{p}]}_{em[\overline{e}]}$ must be a precondition.

4.2 Laws for Assignment

Many of the laws for assignment (Figure 4) are guarded by assertions that say the assigned locations are disjoint.

Law (26) stipulates that attempting to evaluate the right-hand side outside its domain has a behaviour wholly arbitrary. We express that by prefixing the assertion $[\mathfrak{D}\overline{e}]$ on the assignment. Because left expressions also may be undefined, we also have law (27). The definition of $\mathfrak{D} p$ is straightforward if, for this purpose, we consider left expressions as being expressions. In Section 4.3 we determine when a field access $e.f$ is defined.

$$\begin{aligned}
\overline{p} := \overline{e} &= [\mathfrak{D}\overline{e}];\ \overline{p} := \overline{e} && (26)\\
\overline{p} := \overline{e} &= [\mathfrak{D}\overline{p}];\ \overline{p} := \overline{e} && (27)\\
\overline{p}, q, q := \overline{e}, d_1, d_2 &= \overline{p}, q := \overline{e}, d_1 \ \cup\ \overline{p}, q := \overline{e}, d_2 && (28)\\
[\forall\, i \bullet \textbf{not}\ \mathit{alias}[p_i, q]];\ \overline{p}, q := \overline{e}, q &= [\mathfrak{D} q \wedge \forall\, i \bullet \textbf{not}\ \mathit{alias}[p_i, q]];\ \overline{p} := \overline{e} && (29)\\
\textbf{skip} &= () := () && (30)\\
\overline{p}, (q \lhd d \rhd r) := \overline{e}, e &= (\overline{p}, q := \overline{e}, e) \lhd d \rhd (\overline{p}, r := \overline{e}, e) && (31)
\end{aligned}$$

Let $\overline{p}$ be a list of paths $(p_1, p_2, \ldots, p_n)$, then we have

$$[\mathit{disj}[\overline{p}]];\ \overline{p} := \overline{e};\ (b \lhd d \rhd c) = [\mathit{disj}[\overline{p}]];\ (\ \overline{p} := \overline{e};\ b \lhd d^{\mathit{vf}[\overline{p}]}_{\mathit{em}[\overline{e}]} \rhd \overline{p} := \overline{e};\ c\ ) \quad (32)$$

$$\begin{pmatrix} [\mathit{disj}[\overline{p}] \wedge \mathit{alias}[\overline{p}, \overline{q}^{/\mathit{vf}[\overline{p}]}_{\mathit{em}[\overline{e}]}]]; \\ \overline{p} := \overline{e};\quad \overline{q} := \overline{d} \end{pmatrix} = \begin{pmatrix} [\mathit{disj}[\overline{p}] \wedge \mathit{alias}[\overline{p}, \overline{q}^{/\mathit{vf}[\overline{p}]}_{\mathit{em}[\overline{e}]}] \wedge \mathfrak{D}\overline{e}\ ]; \\ \overline{p} := \overline{d}^{\mathit{vf}[\overline{p}]}_{\mathit{em}[\overline{e}]} \end{pmatrix} \quad (33)$$

Fig. 4. Laws for Assignment

Law (28) characterizes the behaviour of simultaneous assignments to repeated left expressions as being non-deterministic. In a simultaneous assignment, if the same left expression q receives simultaneously the values of expressions d_1 and d_2, there will occur a non-deterministic choice between both values and one of them will be assigned to q.

Law (29) is a generalization of a similar one given in LoP, but it is conceived to deal with non-determinism. It establishes that a simple assignment of the value of a left expression back to itself has no effect, when the expression is defined. We can add (or eliminate) such a dummy assignment to a simultaneous assignment whenever no non-determinism is introduced (eliminated). Note that if $\overline{p}, q$ is composed only by non-repeated variables then the assertion will be trivially true, and thus law (29) becomes the same established in LoP.

Law (31), when used from left to right, eliminates conditionals in left expressions. If we have a conditional left expression, we can pass the responsibility for checking the condition to a conditional command. This law resembles assignment law (2) in Figure 2 in that it behaves the same but for expressions on the right-hand side of assignments.

Law (34) below can be used together with (31) for transforming any assignment in an equivalent one where left expressions have no conditional. This transformation can be useful to enable the use of law (25).

$$(p \lhd e \rhd q).f \quad = \quad p.k \lhd e \rhd q.f \qquad (34)$$

Law (32) states that when we have a disjoint assignment to paths followed by a conditional command, the assignment can be distributed rightward through the conditional, but changing occurrences of the assigned paths in the condition to reflect the effects of the assignment. Note that the assertion ensures that the assignment is disjoint, and therefore the substitution applied on the condition d is well defined.

$$x \leftarrow \mathbf{new}\, K \;=\; x \leftarrow \mathbf{new}\, K;\; x.f := \mathit{default}(f) \tag{35}$$

$$[y == \mathbf{alloc}];\; x \leftarrow \mathbf{new}\, K \;=\; [y == \mathbf{alloc}];\; x \leftarrow \mathbf{new}\, K;\; [x \neq \mathbf{null} \wedge x \notin y \wedge \mathbf{alloc} == y \cup \{x\}] \tag{36}$$

if x does not occur in $\overline{p}, \overline{e}$, then

$$x \leftarrow \mathbf{new}\, K;\; \overline{p} := \overline{e} \;=\; \overline{p} := \overline{e};\; x \leftarrow \mathbf{new}\, K \tag{37}$$

if x does not occur in d, then

$$x \leftarrow \mathbf{new}\, K;\; (b \lhd d \rhd c) \;=\; (x \leftarrow \mathbf{new}\, K;\; b) \lhd d \rhd (x \leftarrow \mathbf{new}\, K;\; c) \tag{38}$$

Fig. 5. Laws for **new**.

Law (33) is our version for the law (*) already discussed in Subsection 3.1. This law permits merging two successive assignments to the same locations (variables or fields). The first conjunct in the assertion guarantees that the first assignment is disjoint, thus the substitutions $\overline{q}^{/vf[\overline{p}]}_{em[\overline{e}]}$ and $\overline{d}^{vf[\overline{p}]}_{em[\overline{e}]}$ will be well defined. In particular, note that $\overline{q}^{/vf[\overline{p}]}_{em[\overline{e}]}$ and $\overline{d}^{vf[\overline{p}]}_{em[\overline{e}]}$ denote the left value of $\overline{q}$ and the value of $\overline{d}$, respectively, after the execution of the assignment $\overline{p} := \overline{e}$. The second conjunct in the assertion states that the left expressions $\overline{p}$ and $\overline{q}$ are referring to same locations, and thus the assignments can be combined into a single one.

4.3 Laws for the new Command

The **new** construction in our language is a command. It cannot be handled as an expression because it alters the state of the program and our approach assumes side-effect-free expressions. The laws for **new** are given in Figure 5. Recall that there is a distinguished variable **alloc** which does not occur in commands except in assertions; its value is always the set of allocated references. Any attempt to access a field of a non allocated reference will be undefined. Thus, we have as definition that $\mathfrak{D} e.f \;\widehat{=}\; \mathfrak{D} e \wedge e \in \mathbf{alloc} \wedge f \in \mathit{fields}(e)$, where $\mathit{fields}(e) = \mathit{fields}(\mathit{type}(e))$ and $\mathit{type}(e)$ returns the class name of e. Recall that our laws are in an implicit context $\overline{cd}$ of class declarations and context Γ for types of variables; so the type of e is statically determined.

Law (35) determines that any field of a new object will be initialized with the default value. The value of $\mathit{default}(f)$ is 0, **false** or **null** depending on the type of f declared in K. Law (36) establishes that $x \leftarrow \mathbf{new}\, K$ assigns to x a fresh reference of type K and adds it to **alloc**. Law (37) allows to exchange the order of a **new** followed by an assignment, if the assignment does not depends on the created new object. Finally, law (38) distributes a **new** command to the right, inside a conditional, if the condition does not depend on the created new object.

5 Completeness

Our main result says that any command can be reduced to an equivalent one that simulates the original command by using a temporary variable representing explicitly the heap through copy semantics. The simulating program never accesses fields in the heap, neither for writing nor for reading. Instead, it just uses the explicit heap where any update is done by means of copy semantics. Reduction of a program to this form is used as a measure of the comprehensiveness of the proposed set of laws.

The explicit representation of the heap is given by a mapping from references to explicit objects and, in turn, every explicit object is represented by a mapping from field names to values. We assume there is a type, *Object*, of all object references, and a type, *Value*, of all primitive values (including **null**) and elements of *Object*. We also assume the existence of a type *Heap* whose values are mappings with the signature $Object \to (FieldName \to Value)$. *FieldName* is another primitive type whose values are names of fields. We assume the expression language includes functional updates of finite maps; we use Z notation so $h \oplus \{x \mapsto e\}$ denotes the map like h but with x mapped to the value of e.

The reduction is made in two stages. In the first stage, an arbitrary command is transformed (using the laws) into an equivalent one whose assignments are all disjoint and with paths as left expressions.

Theorem 1. *For any command c there is a provably equivalent one such that each assignment in it is prefixed by an assertion and follows the form $[dis]$; $\overline{p} := \overline{e}$ where $\overline{p}$ is a list of paths and dis ensures that there is no pair of aliased paths in $\overline{p}$.*

The proof uses the following ideas. In order to have just paths on left expressions, we can use systematically assignment laws (34) and (31) for eliminating conditionals. For example, it is easy to prove

$$x.f, (y \lhd e_1 \rhd y.g).f := e_2, e_3 \quad = \quad x.f, y.f := e_2, e_3 \ \lhd e_1 \rhd \ x.f, y.g.f := e_2, e_3 \qquad (\dagger)$$

After the elimination of conditionals, we can use systematically the derived law stated below to transform all the assignments to disjoint ones.

$$\begin{array}{rl} \overline{p}, q, r := \overline{d}, e_1, e_2 = & \quad \overline{p}, q := \overline{d}, e_1 \cup \ \overline{p}, q := \overline{d}, e_2 \\ & \lhd alias[q, r] \rhd \\ & \quad [\textbf{not}\ alias[q, r]];\ \overline{p}, q, r := \overline{d}, e_1, e_2 \end{array} \qquad (\ddagger)$$

To illustrate the use of our laws, we give the proof of this derived law.

$(\overline{p}, q := \overline{d}, e_1 \cup \overline{p}, q := \overline{d}, e_2) \lhd alias[q,r] \rhd$ [**not** $alias[q,r]]; \overline{p}, q, r := \overline{d}, e_1, e_2$

$=$ by assert. (15)

$\quad (\overline{p}, q := \overline{d}, e_1 \cup \overline{p}, q := \overline{d}, e_2) \lhd alias[q,r] \rhd \overline{p}, q, r := \overline{d}, e_1, e_2$

$=$ by assign. (28)

$\quad \overline{p}, q, q := \overline{d}, e_1, e_2 \lhd alias[q,r] \rhd \overline{p}, q, r := \overline{d}, e_1, e_2$

$=$ by assert. (14)

$\quad [alias[q,r]]; \overline{p}, q, q := \overline{d}, e_1, e_2 \lhd alias[q,r] \rhd \overline{p}, q, r := \overline{d}, e_1, e_2$

$=$ by assert. (22)

$\quad [alias[q,r]]; \overline{p}, q, r := \overline{d}, e_1, e_2 \lhd alias[q,r] \rhd \overline{p}, q, r := \overline{d}, e_1, e_2$

$=$ by assert (14) & LoP (7)

$\quad \overline{p}, q, r := \overline{d}, e_1, e_2 \lhd \mathfrak{D} alias[q,r] \rhd \bot$

$=$ by $\mathfrak{D}(\overline{p}, q, r) \Rightarrow \mathfrak{D} alias[q,r]$ & LoP (6,8)

$\overline{p}, q, r := \overline{d}, e_1, e_2.$

Continuing the example (†), using repeatedly law (‡), we obtain the following program where all assignments are disjoint.

$$= \left(\begin{array}{rc} & x.f := e_2 \cup x.f := e_3 \\ \lhd x == y \rhd & [x \neq y];\ x.f, y.f := e_2, e_3 \\ \lhd e_1 \rhd \qquad & x.f := e_2 \cup x.f := e_3 \\ \lhd x == y.g \rhd & [x \neq y.g];\ x.f, y.g.f := e_2, e_3 \end{array} \right)$$

Roughly speaking, the second stage of the reduction is to transform the command in the intermediate form (obtained from the first stage, with disjoint assignments) to an equivalent one that first loads the implicit heap into an explicit heap $h : Heap$, then simulates the original command always using h instead of object fields and finally, when it finishes, restores back the contents of h to the implicit heap, i.e, updates all the object fields accordingly as they are represented in h. The domain of h will be the entire set of allocated references, i.e. **alloc**. To keep this domain we use variables mirroring **alloc** before and after the transformed command.

Following our example, suppose that e_2 is $z + x.f$. The assignment $x.f := e_2$ will be simulated using the explicit heap h by

$$h := h \oplus \{x \mapsto \{h(x) \oplus \{f \mapsto z + h(x)(f)\}\}$$

where h is updated by copying a new mapping equal to the original except for the value of $h(x)$, which is updated accordingly. Note that $h(x)$ represents the object referenced by x, $h(x)(f)$ represents the value of the field f of this object.

For any c, we will define a command $S[c][h, al]$ that simulates c using the explicit heap h and the **alloc** mirror al. We will also define command $load[h, al]$

that loads the contents of the objects into the explicit heap h, and $store[h, al]$ that restores back h into the objects. In these terms we can state our main result.

Theorem 2. *For any command c in the intermediate form and where h, al_0 and al_1 do not occur free we have that*

$$\begin{aligned}&[al_0 == \mathbf{alloc}];\ c;\ [al_1 == \mathbf{alloc}];\ load[h, al_1];\ al_0 := al_1 = \\ &\qquad [al_0 == \mathbf{alloc}];\ load[h, al_0];\ S[c][h, al_0];\ [al_1 == \mathbf{alloc}];\ store[h, al_1]\end{aligned}$$

Because the variable h is free on the right-hand side, we need the $load[h, al_1]$ after c on the left-hand side. That is according to the standard interpretation that free variables are the input and output of commands. al_0 and al_1 can be interpreted as variables that bring the value of **alloc** at the points of the assertions. An alternative would be to introduce h, al_0 and al_1 on the right-hand side as local variables; then the assertions could be removed from both sides. But, for brevity in this paper, we omit local variable blocks.

The formal definitions of *load* and *store* are given by

$$\begin{aligned}load[h, al] &\mathrel{\widehat{=}} h := \{r \mapsto \{f \mapsto r.f \mid f \in fields(r)\} \mid r \in al\} \\ store[h, al] &\mathrel{\widehat{=}} r.f \underset{(r,f)\in al\times fields(r)}{:=} h(r)(f)\end{aligned}$$

Here, $fields(r) = fields(type(r))$ where $type(r)$ returns the class name of r.

We use an indexed multiple assignment $r.f \underset{(r,f)\in al\times fields(r)}{:=} h(r)(f)$. This is an abuse of notation since the index set $al \times fields(r)$ depends on the value of al which is determined at runtime (but is finite). This, and similar constructions in the definition of *load* and in (43) and (44), could be avoided using loops; but that would complicate the normal form proof with little benefit.

The definition for the simulation $S[c][h, al]$ for commands is sketched in Figure 6. The operator $\boxplus$ used in (43) is similar to map overriding ($\oplus$) with the difference that the right operand is an uncurried map. We also define the simulation $R[e][h]$ for expressions. The proof of theorem 2 is by induction on c.

6 Conclusions and related work

We established a comprehensive algebraic presentation for programs that deal with references in the way Java does. Taking as a basis the classical work in LoP [9], we explored the incrementality of algebra by generalizing laws of assignment and formulating others. Our normal form theorem gives a sense in which the laws are complete. Perhaps a more standard notion of completeness is whether every semantically true equation is provable. What we know is that our denotational model is not fully abstract with respect to contextual equivalence, owing to the missing laws about **new** mentioned in Section 4. For our first order language those laws can be validated by quotienting modulo renaming, e.g., by using FM-sets, though for higher order programs that would be unsatisfactory [21].

$$R[x][h] \mathrel{\widehat{=}} x \tag{39}$$

$$R[p.f][h] \mathrel{\widehat{=}} h(R[p][h])(f) \tag{40}$$

$$R[e_1 \lhd d \rhd e_2][h] \mathrel{\widehat{=}} R[e_1][h] \lhd R[d][h] \rhd R[e_2][h] \tag{41}$$

$$S[\bot][h, a] \mathrel{\widehat{=}} \bot \tag{42}$$

$$\begin{aligned} &S[\overline{x}, \overline{p} := \overline{e}, \overline{d}][h, al] \mathrel{\widehat{=}} \\ &\quad \overline{x},\ h := R[\overline{e}][h],\ h \boxplus \{(R[p_i][h], f_i) \mapsto R[d_i][h]\} \mid p_i.f_i \in \overline{p} \land d_i \in \overline{d}\} \end{aligned} \tag{43}$$

$$\begin{aligned} S[x \leftarrow \mathbf{new}\, K][h, al] \mathrel{\widehat{=}}\ & x \leftarrow \mathbf{new}\, K;\ al := al \cup \{x\}; \\ & h := h \oplus \{x \mapsto \{f \mapsto \mathit{default}(f) \mid f \in \mathit{fields}(K)\} \end{aligned} \tag{44}$$

$$S[c_1 \lhd d \rhd c_2][h, al] \mathrel{\widehat{=}} S[c_1][h, al] \lhd R[d][h] \rhd S[c_2][h, al] \tag{45}$$

Fig. 6. The simulations R and S using explicit heap (selected cases)

In [20] some laws are presented as an initial attempt to characterize references. However, they are not comprehensive; the law for combining assignments does not consider the possibility of aliasing on left expressions. Furthermore, the laws depend on provisos that are hard to verify statically. Other initiatives like abstract separation algebras [11] do not deal with a concrete notation for manipulation of fields. Transformations used in compilers are conditioned on alias analyses that are not expressed algebraically. Staton proves Hilbert-Post completeness for a equational theory of ML-style references. The language lacks field update and null (and includes higher order functions and an unusual operator ref^n). The laws capture commutativity of allocations and aspects of locality, but the step from this work to handling refactorings in object-oriented programs is big.

A fundamental notion for our laws is field substitution. Inspired by the works of Bornat [7] and Morris [17] on program correctness, we adapt Bornat's definition for our language by extending it to deal with simultaneous assignments.

The problem approached in this paper is related with the difficulties for defining algebraic theories like those in [9,16,1] for a language with references. Like in [7], we only tackle the problem of pointer aliasing. Other kinds of aliasing, like parameter aliasing or aliasing through higher order references, are out of the scope of our work. For higher order programs, more complicated machinery is needed, e.g., content quantification as proposed in [5].

Another difficulty not addressed in this paper is caused by aliasing in combination with recursive data structures. This usually requires dealing with assertions with inductive formulas, and reasoning with substitution and aliasing can explode into numerous conditionals [7]. Note, however, that assertions in our laws use no inductive predicates, only boolean expressions involving pointer equalities. We intend to provide mechanisms for local reasoning as those proposed in [19,7,13,3] in an extension of our theory when dealing with a more expressive object-oriented language. In particular, we would like to connect the present work with that of Silva et al [18] where denotational semantics is used to prove refactoring laws from which others are derived. The goal is algebraic proof of refactoring laws based ultimately on basic laws like our extension of LoP, just as the works [6,8] do for object-oriented progams with copy semantics. In future work we will explore other laws for **new**, seeking to avoid the use of the implicit variable **alloc** in assertions.

References

1. R.J.R. Back and J. von Wright. *Refinement calculus: a systematic introduction.* Springer Verlag, 1998.
2. A. Banerjee and D. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *J. ACM*, 52:894–960, 2005.
3. A. Banerjee, D. A. Naumann, and S. Rosenberg. Local reasoning for global invariants, part I: Region logic. *Journal of the ACM*, 60:18:1–18:56, 2013.
4. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, pages 14–25, 2004.
5. M. Berger, K. Honda, and N. Yoshida. A logical analysis of aliasing in imperative higher-order functions. *SIGPLAN Not.*, 40(9):280–293, September 2005.
6. P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornélio. Algebraic reasoning for object-oriented programming. *Science of Computer Programming*, 2004.
7. Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, volume 1837 of *LNCS*, pages 102–126. Springer-Verlag, 2000.
8. M. Cornélio, A. Cavalcanti, and A. Sampaio. Sound refactorings. *Science of Computer Programming*, 2010.
9. C. A. R. Hoare and *et. al.* Laws of programming. *Communications of the ACM*, 30(8), 1987.
10. C. A. R. Hoare and S. Staden. In praise of algebra. *Formal Aspects of Computing*, pages 423–431, 2012.
11. T. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent Kleene algebra and its foundations. *J. Log. Algebr. Program.*, 80(6):266–296, 2011.
12. He Jifeng, Xiaoshan Li, and Zhiming Liu. rCOS: A refinement calculus of object systems. *Theoretical Computer Science*, 365(1–2):109 – 142, 2006.
13. I.T. Kassios. The dynamic frames theory. *Formal Aspects of Computing*, 23:267–288, 2011.
14. Dexter Kozen. On Hoare logic and Kleene algebra with tests. *ACM Trans. Comput. Logic*, July 2000.
15. G. Lucero, D. Naumann, and A. Sampaio. Laws of programming for references (long version). `www.cs.stevens.edu/~naumann/pub/LuceroNSfull.pdf`, 2013.
16. C. Morgan. *Programming from specifications.* Prentice-Hall, Inc., 1990.
17. J. Morris. A general axiom of assignment. In Manfred Broy and Gunther Schmidt, editors, *Theoretical Foundations of Programming Methodology.* 1982.
18. D. A. Naumann, A. Sampaio, and L. Silva. Refactoring and representation independence for class hierarchies. *Theoretical Computer Science*, 433:60–97, 2012.
19. P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic*, 2001.
20. L. Silva, A. Sampaio, and Zhiming Liu. Laws of object-orientation with reference semantics. In *IEEE Software Engineering and Formal Methods*, 2008.
21. S. Staton. Completeness for algebraic theories of local state. In *FoSSaCS*, volume 6014 of *LNCS*, pages 48–63, 2010.