True small-step reduction for imperative object oriented languages

Marco Servetto
Victoria University of Wellington
School of Engineering and Computer Science
servetto@ecs.vuw.ac.nz

Lindsay Groves
Victoria University of Wellington
School of Engineering and Computer Science
lindsay@ecs.vuw.ac.nz

ABSTRACT

Traditionally, formal semantic models of Java-like languages use an explicit model of the store which mimics pointers and ram. These low level models hamper understanding of the semantics, and development of proofs about ownerships and other encapsulation properties, since the real (graph) structure of the data is obscured by the encoding. Such models are also inadequate for didactic purposes since they rely on run-time structures that do not exist in the source program — in order to understand the meaning of an expression in the middle of the execution one is required to visualize the memory structure which is hard to relate to the abstract program state.

We present a semantic model for Java-like languages where data is encoded as part of the program rather than as a separate resource. This means that execution can be modelled more simply by just rewriting source code terms, as in semantic models for functional programs. The major challenges that need to be addressed are aliasing, circular object graphs, exceptions and multiple return methods. In this initial proposal we use local variable declarations in order to tackle aliasing and circular object graphs.

Categories and Subject Descriptors

D3.3 [**Software**]: Programming languages—*Language Constructs and Features*

Keywords

Object oriented, Small-step reduction, Imperative languages

1. INTRODUCTION

The semantic model of **functional programming languages** is conventionally defined as a small-step reduction arrow: an application of category theory over language terms [18]. Since all steps use the same language as the original program, it is easy to follow and understand small-step semantics for simple mathematical languages [6]. For example the minimal program `12-45*67-89` reduces in the following self explanatory steps: `(12-45*67-89)` → `(12-3015-89)` → `(-3003-89)` → `(-3092)`. The sequence of reductions contains more information than the end result, and provides a useful example to illustrate the semantics of the programming language itself. Indeed, in this case it shows us that the correct operator precedence and associativity treats the expression as equivalent to `(12-(45*67))-89`. A student can look at the sequence of reductions and understand the semantics of the initial expression. Moreover, a mathematician can use induction over the steps to verify that certain properties hold.

The semantic model of **imperative programming languages** (including memory, aliasing and side effects) is usually defined in terms of pointers to an external store [2, 13, 4, 5].[1] This models the hardware the programs are expected to run on: a global memory where pointers refer to records containing primitive data or other pointers. Indeed, such a formal model is a guide for an efficient implementation of such languages. Unfortunately this creates **an artificial distinction between source code and the execution model**.[2] Understanding the meaning of a reduction sequence over expressions and stores is difficult for most students, since it is as hard as understanding a memory dump: most of the relevant information is either lost or obfuscated by the encoding.

We believe that the functional model captures the semantics of computation in a more natural way, and we aim to extend this kind of model in order to handle the features of an imperative language. We aim to obtain a model where mathematical properties of object graph topologies can be formally verified significantly easier with respect to the conventional model with the store.

Our goal is not to define a new semantics, but to retain conventional Java-like semantics. We thus consider the language FJ^, defined in our companion article [19] in the conventional setting with memory and references, and we define an alternative, equivalent semantics, where a source code term is rewritten as another source code term, without using any store or other external resources, recovering the simplicity of the functional language models.

2. MAIN IDEA

The main idea is to use local variable declarations — as in the `let` construct — in order to model aliasing, state modification and (circular) references. For example under the following program

```
class D{int f; D(int f){this.f=f;}}
class C{D f; C(D f){this.f=f;}}
```

we can write the following expression:

```
C x=new C(new D(80));C y=x;x.f=new D(y.f.f+8);y.f
```

[1] An interesting exception is [16], see Section 6.

[2] That is, the execution model of most languages rely on concepts that are absent from the source code itself, often called run-time expressions. Formally, they define two languages, the source code language and the run-time language, and a transparent embedding function that maps the first into the second one. Often, going back to the source code after execution is started is not possible.

FTfJP '13, Montpellier, France

The reduction can keep the local variable x in place. On the other hand (since in FJ^, all variables are final) y can be simplified away.

```
C x=new C(new D(80)); x.f=new D(x.f.f+8); x.f
```

Then the field can be accessed and the sum can be computed

```
C x=new C(new D(80)); x.f=new D(88); x.f
```

Then the field can be updated, **directly in the declaration**.

```
C x=new C(new D(88)); x.f
```

The field can then be accessed, and the (now unused) local variable x can be simplified away, producing the final result:

```
new D(88)
```

Circular object graphs are managed in a similar way, but require the use of placeholders, introduced in FJ^ [19] and summarized in the following.

Placeholders.

A *placeholder*, as the name suggests, is a proxy or a stand-in for an uninitialised, as yet nonexistent object. Placeholders are introduced in FJ^variable declarations. A single variable declaration can declare and initialise an entire circular object graph:

```
class A{A f; A(A^ f){this.f=f;}}
...
A a1=new A(a2), //here a2 is a placeholder
A a2=new A(a1); //here a1 is a placeholder
... //here a1.f is a2 and a2.f is a1
```

In FJ^variable declarations differ from Java-style declarations in two ways. Syntactically, a series of individual variable declarations are separated with commas (,). When such newly declared variables are used in the right-hand side of those variable declarations they act as placeholders.

In the conventional semantics defined in [19], placeholders are replaced with actual values when the execution reaches the semicolon (;) sign. In this new semantics, no action is needed, since the variable name is already a value.

The purpose of placeholders is similar to recursive bindings in OCaml [17], where for example, the following code initializes a recursive data-structure:

OCaml

```
type t = A of t
let rec a1 = A( a2 )
  and a2 = A( a1 )
```

While placeholders allow arbitrary expressions to be placed on the right hand side, OCaml imposes heavy restrictions [9]: *"the right-hand side of recursive value definitions to be constructors or tuples, and all occurrences of the defined names must appear only as constructor or tuple arguments."*

The FJ^ type system uses placeholder types (such as A^ in the example) to ensure placeholders are used in a safe way, however, they can be considered equivalent to the normal object type (A in this case) for the purposes of the current article. Indeed, here we do not consider typing issues at all, since both the conventional and newly proposed semantics of FJ^ are independent of the type system, and are designed to get stuck in error conditions, such as message-not-understood. The only reason we keep the symbol (^) in the syntax is for consistency with [19].[3]

Placeholder declarations can appear in the original program, or can be the result of a field assignment; as the following code shows:

```
interface I{}
class A implements I{I f;A(I^ f){this.f=f;}}
class B implements I{B(){}}
```

The expression

(step-0)
```
A a1=new A(new B());A a2=new A(a1);
    A a3=(a1.f=a2);a2.f.f
```

is reduced in one step into

(step-1)
```
A a1=new A(a2),A a2=new A(a1);
    A a3=a2;a2.f.f
```

where the semicolon (;) is replaced with a comma (,), since the field update operation has merged two variable declarations into a single placeholder declaration. This step reduces the subexpression inside the declaration of a3, which updates the field f of a1 with the value a2. Thus, the term `new B()` is now replaced by a2.

The unused variable a3 can now be simplified away and then the expression reduces to

(step-2)
```
A a1=new A(a2), A a2=new A(a1); a1.f
```

and finally to

(step-3)
```
A a1=new A(a2), A a2=new A(a1); a2
```

which is a value, and is equivalent to `A a2=new A(new A(a2));a2`.

The emergence of different, but equivalent, terms is a pervasive phenomenon in our semantics.

For example `C x1=new C();C x2=x1;C x3=x2;x2.m(x3)` is clearly identical to `C x1=new C();x1.m(x1)`.[4]

In order to capture this informal idea of equivalence, we we define a concept of normalization: [5] if a variable is used only once then the variable declaration can be removed and the single variable occurrence can be replaced by the corresponding initialization expression. In a normalized term, if a value is a variable, then we are dealing with an aliased object.

3. SYNTAX AND SEMANTICS OF FJ^

We now define the syntax of FJ^, and show how its semantics can be defined in terms of simple small-step reductions.

3.1 Syntax

The syntax of FJ^ is shown in Figure 1. The language is essentially a simple subset of Java, similar to FJ [15], with the addition of field assignments and local definition blocks.

A program (p) is a list of class and interface declarations. A class declaration (cd) has a class name, a list of implemented interfaces, a list of field declarations, a constructor, and a list of method declarations. An interface declaration (id) has an interface name, a set of extended interfaces, and a set of method headers. Each field declaration (fds) has a type and a field name. A constructor (k) takes a parameter for each field of the class, and simply initialises those fields. Note that the argument types are placeholders. A method header (mh) has a result type, a method name, and a list of argument names along with their types. Each method declaration (mds) is a method header along with a method body, which is an expression. A type (T) is a class or interface name, optionally marked with (^) to indicate that it is a placeholder. An expression (e) is either a variable name, a **new** expression (constructor call), a field access, field assignment, method call, or a local declaration block, consisting of a list of local variable declarations and an expression. Each local variable declaration (es) has a type, a variable name and an expression providing the value for the variable; the local variable declarations are separated by commas. A value (v) is a variable name, a **new** expression in which the arguments are all values, or a

[3]Well, just note that in [19] we use the symbol (') instead of (^). The symbol (') has proved itself typographically confounding.

[4]Non-well guarded variable declarations, like `C x=x` are not allowed [19]. In FJ^ the reduction gets stuck on this kind of declaration; thus the FJ^ type system statically prevents such cases.

[5]That is, equivalent terms normalize to the same term.

p	::= $cd_1 \ldots cd_n \;\; id_1 \ldots id_k$	program	e	::= $x \mid$ **new** $\mathcal{C}$**(** e_1**,** $\ldots$**,** e_n**)** $\mid e_0.f$		
cd	::= **class** $\mathcal{C}$ **implements** $\mathcal{C}_1$**,** $\ldots$**,** $\mathcal{C}_n$**{** $fds\ k\ mds$**}**	class declaration		$\mid e_0.f{=}e_1 \mid e_0.m$**(** e_1**,** $\ldots$**,** e_n**)** $\mid$ **(** es**;** e**)**		expression
id	::= **interface** $\mathcal{C}$ **extends** $\mathcal{C}_1$**,** $\ldots$**,** $\mathcal{C}_n$**{** mh_1**;** $\ldots$ mh_k**;}**	interface declaration	es	::= $T_0\ x_0{=}e_0$**,** $\ldots$**,** $T_n\, x_n{=}e_n$		local var dec
fds	::= $\mathcal{C}_1\ f_1$**;** $\ldots$ $\mathcal{C}_n\ f_n$**;**	field declaration	v	::= $x \mid$ **new** $\mathcal{C}$**(** v_1**,** $\ldots$**,** v_n**)** $\mid$ **(** vs**;** v**)**		values
k	::= $\mathcal{C}$**(** $\mathcal{C}_1\hat{}\ f_1$**,** $\ldots$**,** $\mathcal{C}_n\hat{}\ f_n$**){ this.**$f_1{=}f_1$**;** $\ldots$ **this.**$f_n{=}f_n$**;}**	constructor	vs	::= $T_0\ x_0{=}v_0$**,** $\ldots$**,** $T_n\ x_n{=}v_n$		local val dec
mh	::= $T\ m$**(** $T_1\ x_1$**,** $\ldots$**,** $T_n\ x_n$**)**	method header	x, y, z	::= $\ldots$		variable names
mds	::= $mh_1\ e_1$**;** $\ldots$ $mh_n\ e_n$**;**	method declaration	$\mathcal{C}$	::= $\ldots$		class/interf. names
T	::= $\mathcal{C} \mid \mathcal{C}\hat{}$	type				

Figure 1: Expression Syntax

local declaration block in which the local variables are all assigned values and the expression is a value.

Local value declarations vs can be seen as functions: $\mathsf{dom}(vs)$ is the set of declared variables, $vs(x)$ is the corresponding value and $vs_1 \cup vs_2 = vs$ holds if $\mathsf{dom}(vs_1) \cap \mathsf{dom}(vs_2) = \emptyset$ and vs is composed by the declarations in vs_1 and vs_2 in any order. Similarly, programs can be seen as functions from class names to their declarations, so $p(C)$ returns the declaration of class C.

A program is well formed if all the names used are declared, and all the names after **extends** and **implements** clauses correspond to interface names. Moreover, all the variable/parameter names declared within a given method body must be unique.

3.2 Reduction

The semantics of our language is defined in terms of small-step reductions, which successively transform subexpressions into simpler ones. Because an expression being reduced may occur within nested local declaration blocks, and because local declarations may be added or deleted during the reduction process, we abstract from the program structure to collect the surrounding local definitions into an *environment* (σ), which is a list of lists of local value declarations. Reduction is then defined as an arrow over pairs consisting of an environment and an expression. At the top level, an expression is reduced starting and ending with an empty environment, so although we use environments in describing intermediate steps, the result we obtain is described entirely within our programming language.

The pair $\sigma \mid e$, where $\sigma = vs_1 \parallel \ldots \parallel vs_n$, describes an expression e occurring within the scope of n nested local declaration blocks, whose declarations are $vs_1, \ldots, vs_n$, with vs_1 being the outermost block. A reduction $\sigma|e \underset{p}{\rightarrow} \sigma'|e'$ means that in the context of program p and environment σ, expression e reduces to e' with new environment σ'. This is formalised in terms of *evaluation contexts* (*eval-ctx*), which are expressions with a hole in any of the places where subexpression substitution is allowed, namely, the receiver of a field access, field assignment or method call, the right hand side of a field assignment where the receiver is a value, an argument of a **new** or method call, provided that all previous arguments (and the receiver in the case of a method call) are values, or the right hand side of a declaration in a local variable declaration block where the right hand sides of all preceding declarations are values.

Reduction rules are shown in Figure 2, and are explained as follows:

- (TOP-LEVEL): e reduces to e' if e can be reduced to e' starting and ending with an empty environment.
- (R-CTX): If e, within environment σ, reduces to e' with new environment σ', then an occurrence of e within any evaluation context with environment σ, can be replaced by e' also with new environment σ'. Note we do not allow nested declarations of the same variable, so application of this rule may require α-conversion to avoid name clashes.
- (R-VD-IN): The above rule does not apply to the expression part of a local declaration block, since in this case the reduction may affect the local declarations as well as the environment. To handle this case, we add the local declaration list to the environment, and reduce the expression in the context of that environment. The resulting block is then obtained by combining the new expression with the new version of the declaration list. Again, α-conversion may be required. Note that this is effectively using the environment as a stack.
- (R-VD-GET), (R-VD-SET): A field access or assignment on a receiver specified by a local declaration block is reduced by moving the access or assignment into the body of the local declaration block.
- (R-NEW-GET), (R-NEW-SET): A field access or assignment on a receiver specified by a **new** expression is reduced to the field value or assigned value respectively.
- (R-VAR-GET): A field access on a variable, $x.f$, is reduced to a new variable, say z, and a new definition setting z to the value of $x.f$ is added to the environment. The operator $\sigma[\text{+=}\, z := x.f]$ finds the declaration of x. If x is assigned the result of a constructor call in which the argument value corresponding to field f is v and has type T, the constructor call is modified so that this occurrence of v is replaced by z, and the new assignment $T\ z{=}v$ is added. If x is assigned the value of a local declaration block, $T\ x{=}$**(**vs_0**;** v_0**)**, the operator is recursively propagated to the value v_0. Since the resulting value v_1 has to be moved to the same level as x, the declarations in vs_0 and vs_1 declaring variables whose v_1 depends on are also "flattened" to the same level as x. If x is assigned the value of a variable, it can be simplified using the rule (S-VAR-EMBED), described below, before being reduced.
- (R-VAR-SET): A field set on a variable, $z.f := v$, is reduced to the assigned value (v), and the environment is updated to record the new value of $z.f$. The operator $\sigma[z \nwarrow v]$ finds the level in the environment σ where z is declared, and moves into that level all declarations in later levels which declare variables that v depends on. The operator $[z.f := v]$ then finds the declaration of z. If z is assigned the result of a constructor call, the constructor call is modified so that the argument value corresponding to field f is replaced by v. If z is assigned the value of a local declaration block, $T\ z{=}$**(**$vs\, v_0$**)**, the update is applied directly to v_0. If x is assigned the value of a variable, it can be simplified using the rule (S-VAR-EMBED), described below, before being reduced.
- (R-METH): A method call where the receiver and arguments are all values is reduced by introducing a local declaration block with variables for the receiver and the arguments, and with the method body as the expression. Note that using this rule may involve α-conversion (as explained above for rule (R-CTX)), for example, where a recursive method call is evaluated.

Figure 3 defines the operators used in the reduction relation.

σ ::= $vs_1 \parallel \ldots \parallel vs_n$ scope

$\mathcal{E}^{\mathrm{r}}$ ::= **new** $\mathcal{C}$ **(** v_1, …, v_k, $\square$, e_1, …, e_n **)** | $\square$.f | $\square$.f=e | v.f=$\square$
| $\square$.m **(** e_1, …, e_n **)** | v_0.m **(** v_1, …, v_k, $\square$, e_1, …, e_n **)** | **(** vs, T x=$\square$, es; e **)** eval-ctx

$e \xrightarrow{p} e'$, $\sigma|e \xrightarrow{p} \sigma'|e'$

(TOP-LEVEL) $\dfrac{\emptyset|e \xrightarrow{p} \emptyset|e'}{e \xrightarrow{p} e'}$ (R-CTX) $\dfrac{\sigma|e \xrightarrow{p} \sigma'|e'}{\sigma|\mathcal{E}^{\mathrm{r}}[e] \xrightarrow{p} \sigma'|\mathcal{E}^{\mathrm{r}}[e']}$ (R-VD-IN) $\dfrac{\sigma\|vs|e \xrightarrow{p} \sigma'\|vs'|e'}{\sigma|\mathbf{(}vs; e\mathbf{)} \xrightarrow{p} \sigma'|\mathbf{(}vs'; e'\mathbf{)}}$

(R-VD-GET) $\dfrac{}{\sigma|\mathbf{(}vs; v_0\mathbf{)}.f \xrightarrow{p} \sigma|\mathbf{(}vs; v_0.f\mathbf{)}}$ (R-NEW-GET) $\dfrac{}{\sigma|\mathbf{new}\,\mathcal{C}\mathbf{(}v_1, \ldots, v_n\mathbf{)}.f_i \xrightarrow{p} \sigma|v_i}$ with $p(\mathcal{C}) = \mathbf{class}\ \mathcal{C}\ \mathbf{implements}\ \_\{\mathcal{C}_1\ f_1; \ldots \mathcal{C}_n\ f_n; \_\}$

(R-VAR-GET) $\dfrac{}{\sigma|x.f \xrightarrow{p} \sigma[\mathbf{+=}\ z := x.f]|z}$ with z fresh

(R-VD-SET) $\dfrac{}{\sigma|\mathbf{(}vs; v_0\mathbf{)}.f=v_1 \xrightarrow{p} \sigma|\mathbf{(}vs; v_0.f=v_1\mathbf{)}}$ (R-NEW-SET) $\dfrac{}{\sigma|\mathbf{new}\,\mathcal{C}\mathbf{(}v_1, \ldots, v_n\mathbf{)}.f=v \xrightarrow{p} \sigma|v}$ (R-VAR-SET) $\dfrac{}{\sigma|z.f=v \xrightarrow{p} \sigma[z \nwarrow v][z.f := v]|v}$

(R-METH) $\dfrac{}{\sigma|v_0.m\mathbf{(}v_1, \ldots, v_n\mathbf{)} \xrightarrow{p} \sigma|\mathbf{(}\mathcal{C}\ \mathbf{this}=v_0, T_1\ x_1=v_1, \ldots, T_n\ x_n=v_n; e\mathbf{)}}$
with
$\mathsf{typeOf}_\sigma(v_0) = \mathcal{C}$
$p(\mathcal{C}) = \mathbf{class}\ \mathcal{C}\ \mathbf{implements}\ \_\{\_\ T\ m\mathbf{(}T_1\ x_1, \ldots, T_n\ x_n\mathbf{)}\ e\_\}$

Figure 2: Reduction

3.3 Simplification

In addition to the above reduction rules, we have a small set of simplification/normalization rules which can be used to simplify an expression in ways that do not depend on the environment (see Figure 4). As for α-conversion, simplification can be used at any point during the reduction.

The rules are formalised in terms of the standard full expression context, which does not enforce any evaluation order.

- (S-CTX): If expression e_0 can be simplified to e_1, then any occurrence of e_0 in the locations identified by the definition of contexts (i.e. an argument of a constructor or method call, the receiver of a field access, field assignment or method call, the value in a field assignment or local variable declaration, or the expression part of the local variable declaration block) can be replaced by e_1.
- (S-VAR-EMBED): A local variable declaration which either: (i) assigns a value to a variable which is used at most once in the enclosing local variable declaration block, or (ii) assigns a variable to another, can be removed by replacing any use of the variable by the assigned value or variable. The notation $e[x := v]$ denotes ordinary variable substitution.
- (S-VS-OUT): A local declaration block with no declarations can be replaced by its expression part.

4. DIDACTIC APPLICATIONS

A student needs the support of an expert mathematician to verify his steps while resolving a mathematical expression, while a programming language simply provides the solution and intermediate steps are hidden. For example, Bob wonders what happens to $12 - 45 * 67 - 89$. He might reduce it with the following steps: $12 - 45 * 67 - 89 = 12 - 3015 - 89 = 12 - 2926 = -2914$. The result is incorrect; but he is unable to spot the error. If he is also taking a programming course, he might ask the solution of the language, as in `System.out.println(12-45*67-89);`, which prints the correct result: `-3092`.

Neither of these approaches is satisfactory for Bob. In the former he sees intermediate steps, but has no confidence in the result. In the latter he gets a correct result, but does not understand how it was obtained.

A didactic tool can show all the steps and help Bob finally understand the right precedence of operators.

This issue is not limited to simple mathematical expressions. Indeed, a student trying to understand the semantics of a programming language will often write and run several small programs. When the result is unexpected, he/she has to guess where the computation process differs from his/her understanding. Running the program gets a result, but the procedure to obtain such a result is out of reach for the student: compilation and many other intermediate optimization steps hide the original structure and make it hard to follow the whole process.

Traditionally, the semantic model of Java-like languages mimics pointers and RAM. This low level model is inappropriate as a didactic instrument to explain the semantics of the language to a student, since the real (graph) structure of the data is lost in the encoding.[6] In our proposed model a source code term is rewritten as another source code term without using external resources, but encoding the data as part of the program itself.

With such a formal definition, is easy to write a graphical educational tool showing the reduction steps one at a time, as in the Racket Stepper [8], a similar didactic tool working only in a limited, purely functional setting.

Such a tool would allow students to visualize reduction steps in order to understand the semantics of Java-like languages; this is

[6]Of course, the heap still contains all the information, but is not a suitable representation for didactic purposes.

$\mathcal{E} ::= \texttt{new}\,\mathcal{C}\texttt{(}e_1\texttt{, ...,}\,\square\texttt{, ...,}\,e_n\texttt{)} \mid \square.f \mid \square.f\texttt{=}e \mid e.f\texttt{=}\square \mid \square.m\texttt{(}e_1\texttt{, ...,}\,e_n\texttt{)}$
$\mid e_0.m\texttt{(}e_1\texttt{, ...,}\,\square\texttt{, ...,}\,e_n\texttt{)} \mid \texttt{(}es_1\texttt{,}\ T\ x\texttt{=}\square\texttt{,}\ es_2\texttt{;}\ e\texttt{)} \mid \texttt{(}es\texttt{;}\ \square\texttt{)}$ simpl-ctx

$e_0 \rightarrow e_1$

(S-CTX) $\dfrac{e_0 \rightarrow e_1}{\mathcal{E}[e_0] \rightarrow \mathcal{E}[e_1]}$

(S-VAR-EMBED) $\dfrac{}{\texttt{(}vs\texttt{,}\ T\ x\texttt{=}v\texttt{,}\ es\texttt{;}\ e_0\texttt{)} \rightarrow \texttt{(}vs\texttt{,}\ es\texttt{;}\ e_0\texttt{)}[x := v]}$
with
x used 0 or 1 times in $\texttt{(}vs\texttt{,}\ T\ x\texttt{=}v\texttt{,}\ es\texttt{;}\ e_0\texttt{)}$
or v is a variable

(S-VD-OUT) $\dfrac{}{\texttt{(}e\texttt{)} \rightarrow e}$

Figure 4: Simplification

Definition for $\sigma[\texttt{+=}\, z := x.f]$
$(vs_1 \| \ldots \| vs_n)[\texttt{+=}\, z := x.f] = vs_1[\texttt{+=}\, z := x.f] \| \ldots \| vs_n[\texttt{+=}\, z := x.f]$
$(vs_1,\ T x\texttt{=}v,\ vs_2)[\texttt{+=}\, z := x.f] = vs_1,\ (T x\texttt{=}v)[\texttt{+=}\, z := x.f],\ vs_2$
$vs[\texttt{+=}\, z := x.f] = vs$ with $x \notin \mathsf{dom}(vs)$
$T\ x\texttt{=(}vs_0\texttt{;}\ v_0\texttt{)}\,[\texttt{+=}\, z := x.f] = vs_2,\ T\ x\texttt{=(}vs_3\texttt{;}\ v_1\texttt{)},\ T\ z\texttt{=}v_2$
iff $T\ x\texttt{=}v_0[\texttt{+=}\, z := x.f] = vs_1,\ T\ x\texttt{=}v_1,\ T\ z\texttt{=}v_2$
and $vs_0, vs_1 = vs_2 \cup vs_3$ and $\mathsf{dom}(vs_2) = \mathsf{dep}(vs_0, vs_1, v_1)$
$T\ x\texttt{=}\,\texttt{new}\,\mathcal{C}\texttt{(}v_1\texttt{, ...,}\ v_i\texttt{, ...,}\ v_n\texttt{)}\,[\texttt{+=}\, z := x.f_i] =$
$\quad T\ x\texttt{=}\,\texttt{new}\,\mathcal{C}\texttt{(}v_1\texttt{, ...,}\ z\texttt{, ...,}\ v_n\texttt{)}, \mathcal{C}_i\ z\texttt{=}v_i$
$\quad$ if $p(\mathcal{C}) = \texttt{class}\ \mathcal{C}\ \texttt{implements\_}\{\mathcal{C}_1\ f_1\texttt{;}\ \ldots \mathcal{C}_n\ f_n\texttt{;\_}\}$

Definition for $\sigma[x.f := v]$
$(vs_1 \| \ldots \| vs_n)[x.f := v] = vs_1[x.f := v] \| \ldots \| vs_n[x.f := v]$
$(vs_1,\ T\ x\texttt{=}v,\ vs_2)[x.f := v] = vs_1, (T\ x\texttt{=}v)[x.f := v], vs_2$
$vs[x.f := v] = vs$ with $x \notin \mathsf{dom}(vs)$
$T\ x\texttt{=(}vs\texttt{;}\ v_0\texttt{)}\,[x.f := v] = T\ x\texttt{=(}vs\texttt{;}\ v_1\texttt{)}$
with $T\ x\texttt{=}v_0[x.f := v] = T\ x\texttt{=}v_1$
$T\ x\texttt{=}\,\texttt{new}\,\mathcal{C}\texttt{(}v_1\texttt{, ...,}\ v_i\texttt{, ...,}\ v_n\texttt{)}\,[x.f_i := v] =$
$\quad T\ x\texttt{=}\,\texttt{new}\,\mathcal{C}\texttt{(}v_1\texttt{, ...,}\ v\texttt{, ...,}\ v_n\texttt{)}$
$\quad$ if $p(\mathcal{C}) = \texttt{class}\ \mathcal{C}\ \texttt{implements\_}\{\mathcal{C}_1\ f_1\texttt{;}\ \ldots \mathcal{C}_n\ f_n\texttt{;\_}\}$

Definition for $\sigma[x \nwarrow v]$
$(\sigma \| vs)[x \nwarrow v] = \sigma \| vs$ if $x \in \mathsf{dom}(vs)$
$(\sigma \| vs_0 \| vs)[x \nwarrow v] = (\sigma \| vs_0, vs_1)[x \nwarrow v] \| vs_2$
if $x \notin \mathsf{dom}(vs),\ vs = vs_1 \cup vs_2$ and $\mathsf{dom}(vs_1) = \mathsf{dep}(vs, v)$

Definition for $\mathsf{dep}(vs, v)$
$\mathsf{dep}(vs, v)$ is the smallest set S such that
(i) $\mathsf{freeVar}(v) \cap \mathsf{dom}(vs) \subseteq S$
(ii) $\forall x \in S, \mathsf{dep}(vs, vs(x)) \subseteq S$
where $\mathsf{freeVar}(e)$ denotes the free variables in e

Definition for $\mathsf{typeOf}_\sigma(v)$
$\mathsf{typeOf}_\sigma(\,\texttt{new}\,\mathcal{C}\texttt{(\_)}) = \mathcal{C}$
$\mathsf{typeOf}_\sigma(\texttt{(}vs\texttt{;}\ v\texttt{)}) = \mathsf{typeOf}_{\sigma \| vs}(v)$
$\mathsf{typeOf}_\sigma(x) = \mathsf{typeOf}_\sigma(v)$ iff $T\ x\texttt{=}v \in \sigma$

Figure 3: Definitions

profoundly different from what other tools (e.g. [14]) do. While showing intermediate steps in the computation they rely on some sort of graphic representation of the memory, which takes the focus of the programmer away from the source code.

In particular, our approach can illustrate dynamic dispatch in a very direct way. For example, given the following program:

```
interface I{int m(I i);}
class A implements I{ int m(I i){return 8;}
class B implements I{ int m(I i){return 2;}
class Factory{I make(){return new A();}}
```

we can show the following reductions:

```
(step-0) I x=new Factory().make();x.m(x)
(step-1) I x=new A();x.m(x)
(step-2) I x=new A();8
(step-3) 8
```

where a factory method and a local variable declaration hide the concrete type of our object.

Another important context where our approach can support learning is recursion. Consider the following definition of `pow`, written in a extension of FJ$^\wedge$ with **`boolean`**, **`int`**, **`if`** and **`static`** methods. We model **`if`** as an expression in order to more easily integrate it with our core calculus.

```
class M{...
  static int pow(int n,int e){
    //if(e<0)error();let us ignore errors here
    if(e==0) then 1 else  M.pow(n,e-1)*n
    }}
```

This straightforward code contains recursion, and our reduction can show how recursion works in a very clear way. One of the main points of this example is to show how method invocation is managed.[7]

```
(step-0) M.pow(8,2)
(step-1) int n=8,int e=2;
                 if(e==0)then 1;else M.pow(n,e-1)*n
(step-2) int n=8, int e=2;
                if(false)then 1;else M.pow(n,e-1)*n
(step-3) int n=8, int e=2;M.pow(n,e-1)*n
(step-4) int n=8, int e=2;M.pow(n,1)*n
(step-5) int n=8;M.pow(n,1)*n
(step-6) int n=8;int n1=n,int e=1;
           (if(e==0)then 1;else M.pow(n1,e-1)*n1)*n
(step-7) int n=8;int e=1;
             (if(e==0)then 1;else M.pow(n,e-1)*n)*n
(step-8) int n=8;int e=1;
            (if(false)then 1;else M.pow(n,e-1)*n)*n
(step-9) int n=8;int e=1;M.pow(n,e-1)*n*n
(step-10) int n=8;int e=1;M.pow(n,0)*n*n
(step-11) int n=8;M.pow(n,0)*n*n
(step-12) int n=8;int n1=n,int e=0;
         (if(e==0)then 1;else M.pow(n1,e-1)*n1)*n*n
(step-13) (int n=8;int e=0;
           (if(e==0)then 1;else M.pow(n,e-1)*n)*n*n
(step-14) int n=8;int e=0;
          (if(true)then 1;else n*M.pow(n,e-1)*n)*n*n
(step-15) int n=8;int e=0;1*n*n
(step-16) (int n=8;1*n*n)
(step-17) (int n=8;8*n)
(step-18) (int n=8;64)
(step-19) 64
```

After the student has understood this reduction sequence, it is possible to notice how step 5 (where the recursive call is ready to be called) is the same as step 17 (where the recursive call is completely executed) but the call of the recursive function is replaced with the

[7] To be consistent with our formalism integers are treated in the same way as other objects.

result. In this way, the inductive understanding of the `pow` definition can be consolidated. This particular example could be encoded also in FJ, however, a suitably modified example could show how side effects do not fit naturally in the inductive reasoning, since they can be externally visible, that is, the environment around the recursive call can be influenced by the execution itself.

5. RELATIONSHIP WITH OWNERSHIP

Object oriented programming allow complex object graphs to be hidden behind behavioural interfaces. For each object that "the user can perceive and observe" there can be a whole set of intercommunicating objects, cooperating to provide a complex behaviour. Ownership [7] keeps such "private representation objects" **private**.

By using nested variable declarations to represent values, we can induce a hierarchical structure. This is one of the main advantages with respect to the conventional model of memory and pointers. Assuming appropriate classes `C` and `D`, in the following code

```
C x=(
  D y= new D(y,z),
  C z= new C(y,12);
  z)
```

the local variable `x` denotes a value composed of two objects defined in a mutually recursive fashion. Thanks to the usual well-known scoping rules, it is clear that the value `y` is just part of the internal representation of `x`, and is not visible at other points of the program.

Even though our reduction does not enforce any kind of ownership, hierarchies of variable declarations naturally show the ownership tree present at a given point in the execution. Moreover, our reduction carefully represents the only two cases where there is a violation in the ownership tree: field access can force a local variable to be exported (operator $\sigma[\text{+=}\, z := x.f]$), pushing it nearer to the current execution point; and field update can require a local variable to be pushed back in the stack (operator $\sigma[x \nwarrow v]$), in order to allow the updated object to refer to it. In the following, two examples clarify the difference between these two cases.

Field access ownership violation.

```
class D{}  class C{ D f; C(D f){this.f=f;}}
...
c1=new C(new D());c1.f
```

becomes

```
D d1=new D(),C c1=new C(d1);d1
```

That is, the value `new D()` that was originally owned by `c1`, is now an independent value `d1`. A version of FJ$^\wedge$ with ownership annotation could get stuck at this reduction step, allowing the correctness proof of the type system to fall more naturally into the normal progress + subject reduction mechanism.

Field update ownership violation.

```
class D{}  class C{ D f; C(D f){this.f=f;}}
...
C c1=new C(new D());
new C(C c2=new C(d1),D d1=new D();c1.f=d1)
```

becomes

```
C c1=new C(d1),D d1=new D();
new C(C c2=new C(d1);d1)
```

where the local variable `d1`, which was originally a part of the data used in the expression of the `C` constructor parameter, is now visible at top level. Again, it would be possible to add some annotations to FJ$^\wedge$ to ensure that a value does not escape a specific scope.

Ownership and inductive definitions.

To the best of our knowledge, in ownership systems either the owner object must be created before the owned ones or ownership transfer (when allowed) is required to create the owned objects **before** their owners. Both solutions hamper the developing of intuitive functional patterns where a composite structure is inductively generated.

A well defined inductive generation of a data-structure produces a well encapsulated hierarchical value; that is, for any terminating method call, in the absence of field update operations over the method parameters, the method invocation reduces into a value of the form `new` $\mathcal{C}$`(_)` or `(_;_)` and all the objects created during the invocation are dominated by this value. For example:

```
class C{ D m(){...}}
...
... new C().m() ...
```

No matter what the body of `C.m` is, the method call will not modify/add any external local variable (potentially present in the dots), and will produce a self contained object graph. Thus the resulting `D` object could safely own all of its fields, no matter how/when they are created. Note that, since our system does not modify in any way the well known Java semantics, this property holds already in any Java program (without `static` fields). Our approach simply makes this property self evident.

6. RELATED WORK

To the best of our knowledge only Java Jr. [16] defines a semantics for a Java-like language with state that, at the same time, does not introduce run-time expressions and does not rely on a separate store. All other small-step semantics for Java-like languages with field update use some sort of memory/store and some concept of run-time expression (e.g. object references) that can not be expressed in the original program. Still, many works in the literature have arguably similar objectives:

- FJ [15] is a Java-like language whose semantics is defined without memory/run-time expressions; but it has no field update.
- Classic Java [13] models a more complete subset of Java, including state as a store.
- Many works have extended FJ with field update, but always reintroducing stores and run-time expressions as in Classic Java.
- Andersen [3] defined a symbolic execution for C programs, but his proposal keeps memory allocation functions in place, since he wisely does not assume a fixed behaviour for any library code or operative system primitive.
- Felleisen's syntactic theory of state [11, 12] mangles the store in the expressions, but it relies on labelled values; a kind of run-time expression modelling the value and the location address at the same time.
- A different strategy is used for "abstract object creation" [1] by the Key theorem prover. A run-time expression, called parallel update, is added to the language and is used to represent the store state inside the code: all the object creations and field updates are preserved and consulted by field accesses.

PLT Redex [10] is a domain specific language (and a suite of software tools) to define language semantics using reduction. One of the features offered by PLT Redex is a visualization tool that could be used to implement a visualizer for our small-step semantics.

Racket Stepper [8] was developed before PLT Redex, and, relying on some run-time expressions, extracts a small-step semantics from Racket. The stepper accurately models the functional setting of Racket. In the Stepper, bindings are lifted to top-level; while it does not currently step through mutable bindings the author argues it can be easy extended in such a way.

7. DISCUSSION

While our simple formalism does not model object identity, the == operation can still be supported: in a normalized term, every aliased object is referenced with a specific, unique variable name, while non-variable values represent non-aliased objects. Thus, $x{==}x$ reduces to **`true`** and every other case of $v_1{==}v_2$ reduces to **`false`**. The idea that object identity can be represented by a variable name is also present in Java Jr [16].

The Racket community seems to support small-step reduction as an effective way to learn the semantics of a language [10, 8]. Throughout this paper we argue that a language designed to be understandable should support a readable small-step reduction in the language itself, without any additional machinery; that is, a *true* small-step reduction.

We choose to model a true small-step reduction for FJ^, that is, an extension to Java with placeholders. It is indeed possible to provide a true small-step reduction without using placeholders and extending the notion of values to include explicit field setting when their execution would introduce circular references. That is, in our first example with

```
interface I{}
class A implements I{I f;A(I^ f){this.f=f;}}
class B implements I{B(){}}
```

the term

```
A a1=new A(new B());
A a2=new A(a1);
a1.f=a2;
```

would be a value. Another approach could be to rely on **`null`**s and use as values normalized initialization where all the fields are initialized with **`null`** and manually set to the right value afterwards.[8] In this setting the code in our example would be normalized to the following:

```
A a1=new A(null);
A a2=new A(null);
a1.f=a2;
a2.f=a1;
```

Thus, true small-step reduction of Java is possible without placeholders. However, we believe the placeholder version to be much more readable.

8. FUTURE WORK AND CONCLUSION

In this work we provide an alternative semantics for Java-like languages, showing how aliasing and circular object graphs can be supported without resorting to stores or run-time expressions. In the future we would like to prove the equivalence between our alternative semantics and the conventional one.

Other important future work is to extend our approach to exceptions and multiple return methods. Defining the semantics of exceptions should be reasonably easy, but since the exception object itself could refer to local variables, we need to deal carefully with stack unwinding.

We plan to explore a language design where **`return`** and other statements could all be expressed as syntactic sugar over an extension of FJ^ with exceptions. In the future we aim to produce an invertible sugaring and de-sugaring process, so that a term in the rich language (with **`return`**, **`if`** and so on) could be translated (de-sugared) in the minimal FJ^ language, evaluated for some step and then sugared back in the more high level language.

[8]Sightly different variations of this solution were independently synthesized by the authors of this paper, one reviewer and the designer of the Key theorem prover [1].

9. REFERENCES

[1] Wolfgang Ahrendt, Frank S. de Boer, and Immo Grabe. Abstract object creation in dynamic logic. In Ana Cavalcanti and Dennis Dams, editors, *FM*, volume 5850 of *Lecture Notes in Computer Science*, pages 612–627. Springer, 2009.

[2] Jim Alves-Foss, editor. *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*. Springer, 1999.

[3] Lars Ole Andersen. Program analysis and specialization for the C Programming Language. Technical report, 1994.

[4] Gavin M. Bierman and Matthew J. Parkinson. Effects and effect inference for a core Java calculus. *Electr. Notes Theor. Comput. Sci.*, 82(7):82–107, 2003.

[5] G.M. Bierman, M.J. Parkinson, and A. M. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical report, 2003.

[6] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):pp. 346–366, 1932.

[7] David G. Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In Bjørn N. Freeman-Benson and Craig Chambers, editors, *OOPSLA*, pages 48–64. ACM, 1998.

[8] John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In David Sands, editor, *ESOP*, volume 2028 of *Lecture Notes in Computer Science*, pages 320–334. Springer, 2001.

[9] Manuel Fähndrich and Songtao Xia. Establishing object invariants with delayed types. In *OOPSLA*, pages 337–350, 2007.

[10] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[11] Matthias Felleisen and Daniel P. Friedman. A syntactic theory of sequential state. *Theor. Comput. Sci.*, 69(3):243–287, 1989.

[12] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992.

[13] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 171–183, New York, NY, USA, 1998. ACM.

[14] Philip Guo. Online Python Tutor: Embeddable web-based program visualization for cs education. In *Proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE)*, 2013.

[15] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA1999*, pages 132–146. ACM Press, 1999.

[16] Alan Jeffrey and Julian Rathke. Java Jr: Fully abstract trace semantics for a core Java language. In Shmuel Sagiv, editor, *ESOP*, volume 3444 of *Lecture Notes in Computer Science*, pages 423–438. Springer, 2005.

[17] X. Leroy. The Objective Caml system (release 2.00). Available at `http://paulliac.inria.fr/caml`, August 1998.

[18] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[19] Marco Servetto, Julian Mackay, Alex Potanin, and James Noble. The billion dollar fix: Safe modular circular initialisation with placeholders and placeholder types. In *ECOOP*, 2013. to appear.