φ-calculus: a purely object-oriented calculus of decorated objects

Nikolai Kudasov
Innopolis University
Innopolis, Tatarstan Republic, Russia
n.kudasov@innopolis.ru

Violetta Sim
Innopolis University
Innopolis, Tatarstan Republic, Russia
v.sim@innopolis.university

Abstract

Many calculi exist for modelling various features of object-oriented languages. Many of them are based on λ-calculus and focus either on statically typed class-based languages or dynamic prototype-based languages. We introduce untyped calculus of decorated objects, which is defined in terms of *objects* and relies on *decoration* as a primary mechanism of object extension. It is not based on λ-calculus, yet with only four basic syntactic constructions is just as complete. We prove the calculus is confluent (i.e. possesses Church-Rosser property), and introduce an abstract machine for call-by-name evaluation. Finally, we provide a sound translation to λ-calculus with records.

Keywords: models of computation, φ-calculus, object-oriented programming

1 Introduction

Formalizing object-oriented features of programming languages is an old but still vibrant topic in computer science [6]. Researchers have extended many specification languages, such as Z notation, to support object-oriented concepts to some degree [24]. Similarly, modelling languages, such as VDM, have been extended [14] to support classes, objects and a certain notion of inheritance. Later, Wolczko [22] used VDM++ notation to define semantics of object-oriented langauges.

At the same time, models based on lambda calculus have also been studied. Pierce and Turner have introduced type-theoretic foundations for object-oriented programming [23]. Fisher et al. [15] have introduced lambda calculus with object primitives able to express the capabilities of delegation-based object-oriented languages where an object is created from another one, inheriting properties of the original. Abadi and Cardelli introduced λ-calculus with primitive objects [2]. ς-calculus relies on external memory to store and modify the states of objects. The calculus has a sound type system with basic subtyping via subsumption. The work has been expanded in [1] with $\mathbf{FOb}_{<:\mu}$ calculus which is an extension of System F [17, Chapter 11]. Clarke et al. have continued this line of research [12] presenting an object calculus based on "ownership types" adding object ownership and nesting between objects. Wand has proposed a type system with row types and row polymorphism to facilitate type inference in object-oriented languages, modelled by λ-calculus with records [27]. In [11] Ciaffaglione et al. have introduced $\lambda Obj^{\oplus}$ system based on λ-calculus, with objects that can modify their interface upon receiving a message. The calculus is intended to model object evolution in prototype-based languages such as JavaScript.

The above models differ in their approach to subtyping and subclasses, but rely significantly on λ-calculus as their foundation. At the same time, many modern programming languages do not properly support λ-abstraction. For example, Java provides lambda expressions which are essentially instances of anonymous classes with a single method.

When modelling object-oriented languages some formalizations focus on statically typed class-based languages [2, 23, 27] while others prefer prototype-based approach [11].

Bugayenko [8] introduced programming language EO and its semantics in terms of an informally specified calculus which he called φ-calculus. To the best of our knowledge, his work is the only attempt to introduce an object-oriented calculus that incorporates Decorator pattern [16, Chapter 4] as the mechanism of extending objects. Bugayenko's paper expresses interesting ideas, but suffers from inaccuracies and insufficient formalization of the calculus, as the paper lacks reduction semantics and any soundness results. His paper also uses custom terminology that conflicts the established conventions in the computer science literature.

In this paper, we extract the key ideas from Bugayenko's paper and present φ-calculus, a calculus of objects with decoration as the main mechanism for object extension, in a more formal way. Moreover, our interpretation of φ-calculus is not based on λ-calculus, so object methods are also represented as objects.

In contrast with Bugayenko's work, we focus on the object-oriented core of the calculus, and do not introduce primitives such as numbers, booleans, mutable memory. Similarly to lambda calculus, these primitives can either be added via a straightforward extension, or by using an encoding, such as Church numerals. We leave details of such extensions and encodings outside the scope of this paper.

Our specific contributions are the following:

- We introduce syntax and reduction semantics of φ-calculus, a calculus of objects with decoration.

PL'18, January 01–03, 2018, New York, NY, USA
2018.

- We prove that φ-calculus possesses Church-Rosser property.
- We define normal order evaluation of φ-terms and prove its completeness, i.e. any term will be reduced to its normal form under such evaluation order, whenever the normal form exists.
- We define an abstract machine for call-by-name evaluation of φ-terms.
- We introduce translation to and from λ-calculus with records and prove this translation to be sound.
- We show how to extend φ-calculus with primitives and mutable references.

2 Calculus of decorated objects

In this section, we introduce syntax and evaluation rules, providing the intuition behind those. The central concept in φ-calculus is that of an object. In fact, every term in φ-calculus is essentially an object.

Definition 2.1. Assuming a set of labels $\mathcal{L}$ and a set of terms T, an *object* X is a mapping from $\mathcal{L}$ to $T \cup \{\varnothing, \bot\}$. The labels that map to $\bot$ are called *missing*, the labels that map to $\varnothing$ are called *void attributes* of X and labels that map to terms are called *attached attributes* of X. Void and attached attributes of X are collectively called *attributes* of X and are denoted $\text{attr}(X) \subseteq \mathcal{L}$.
An object with an non-empty set of void attributes is called *abstract*. Otherwise an object is called *concrete*.

Example 2.2. A constant mapping from $\mathcal{L}$ to $\bot$ is an *empty object* and is denoted $[\![]\!]$.

Example 2.3. Let t_1, t_2 be terms. Then the following is an object:

$$X : \mathcal{L} \to T \cup \{\varnothing, \bot\}$$
$$x \mapsto \varnothing; y \mapsto t_1; z \mapsto [\![]\!]; \ell \mapsto \bot$$

For convenience we will denote objects by listing void and attached attributes inside double brackets. For example, the object from Example 2.3 can be written succinctly as $[\![x \mapsto \varnothing, y \mapsto t_1, z \mapsto [\![]\!]]\!]$.

2.1 Syntax

The entire syntax of φ-calculus has only four syntactic constructions:

Definition 2.4. Let $\mathcal{L}$ be the set of attribute names extended with *decorator attribute* φ. Then the set of φ-terms T is defined inductively as following:

1. if $n \in \mathbb{N}$ then $\rho^n \in T$; here ρ is merely a symbol used in the syntax, it is not a variable or a meta variable;
2. if $t \in T$ and $a \in \mathcal{L}$ then $t.a \in T$;
3. if $t, u \in T$ and $a \in \mathcal{L}$ then $t(a \mapsto u) \in T$;
4. if $t_1, \ldots, t_n \in T$ and $a_1, \ldots, a_k, b_1, \ldots, b_n \in \mathcal{L}$ then $[\![a_1 \mapsto \varnothing, \ldots, a_k \mapsto \varnothing, b_1 \mapsto t_1, \ldots b_n \mapsto t_n]\!] \in T$.

The instance of rule 1 is called *locator*. The instance of rule 2 is called *attribute access*. The instance of rule 3 is called *application* or *attribute instantiation*. The instance of rule 4 is called *object term* (or just *object*).

An object term is essentially the same as object from Definition 2.1 with the restriction that mapping has finitely many attributes.

Locators allow to reference enclosing objects, and consequently, their attributes. For example, consider object $[\![x \mapsto \rho^0.y, y \mapsto [\![]\!]]\!]$. Here ρ^0 references the closest enclosing object, which is the entire term, and so attribute x is attached to a term that essentially references attribute y of the same object. The index in the locator tells us how many levels of enclosing objects one needs to go to arrive at the referenced object. In fact, locators are effectively de Bruijn indices [13] of nested objects. For example, locator ρ^2 in the term $[\![x \mapsto [\![y \mapsto [\![z \mapsto \rho^2]\!]]\!]]\!]$ references the entire term. This correspondence is made more precise in Section 5 where we give a translation to lambda calculus.

The idea behind attribute access terms is fairly straightforward: we simply intend to extract the associated value. For example, evaluating $[\![x \mapsto [\![]\!]]\!].x$ should produce an empty object. However, because of locators, we cannot do a simple extraction in general and have to perform locator substitution: evaluating $[\![x \mapsto \rho^0]\!].x$ requires understanding what object ρ^0 references. This limitation makes it very different from syntactically similar construction in λ-calculus with records.

Attribute instantiation attaches terms to void attributes. Importantly, attached attributes may reference void attributes as in $[\![x \mapsto \varnothing, y \mapsto \rho^0.x]\!]$. Obviously, accessing attribute y of such an object would require accessing void attribute x which does not have an attached value. However, after instantiating the attribute $[\![x \mapsto \varnothing, y \mapsto \rho^0.x]\!](x \mapsto [\![]\!])$, we can now access attribute y and get the empty object. This example shows that locators are not just references to syntactic objects, but can also reference the result of attribute instantiation.

Decorator attribute φ plays an important role in evaluation. This attribute contains the component of the decorator, and an object with φ will redirect attribute access to its component whenever the object does not possess the attribute itself. For example, accessing attribute $.x$ of the object $[\![\varphi \mapsto [\![x \mapsto t_1]\!], y \mapsto t_2]\!]$ will be redirected to accessing $.\varphi.x$ of the object since the original object does not have x as its attribute.

2.2 Locators

Before we can properly talk about evaluation, we need to define how locators work. As locators are de Bruijn indices of nested objects, we have to be able to adjust locator indices

when moving terms in and out of objects, and replace locators with an actual object they reference when that object disappears.

Attribute instantiation requires putting a term in an object. This requirement may demand updating certain locators. Consider term $[\![x \mapsto \varnothing]\!](x \mapsto \rho^0)$ where ρ^0 references some outer object. Simply replacing $\varnothing$ with ρ^0 will result in $[\![x \mapsto \rho^0]\!]$ which would change the meaning of ρ^0. Instead, since we are placing ρ^0 inside of an object, we have to increment its index. In general though, given $t(a \mapsto u)$ we do not increment all locators in u, but only those referencing outside of u. For example, if $u \equiv [\![y \mapsto \rho^0, z \mapsto \rho^1]\!]$ then we only increment ρ^1, since otherwise its reference object will change when we put u in an object. So we define locator increment as follows:

Definition 2.5. *Locator increment* $t \uparrow^n$ is defined inductively on φ-terms:

$$\rho^m \uparrow^n := \rho^m \quad \text{if } m < n \tag{1}$$

$$\rho^m \uparrow^n := \rho^{m+1} \quad \text{if } n \le m \tag{2}$$

$$t.a \uparrow^n := t \uparrow^n .a \tag{3}$$

$$t_1(a \mapsto t_2) \uparrow^n := t_1 \uparrow^n (a \mapsto t_2 \uparrow^n) \tag{4}$$

$$[\![a_1 \mapsto \varnothing, \ldots, b_1 \mapsto t_1, \ldots, b_n \mapsto t_n]\!] \uparrow^n$$
$$:= [\![a_1 \mapsto \varnothing, \ldots, b_1 \mapsto t_1 \uparrow^{n+1}, \ldots, b_n \mapsto t_n \uparrow^{n+1}]\!] \tag{5}$$

We write $t \uparrow$ short for $t \uparrow^0$.

When accessing an attribute a of an object $u \equiv [\![\ldots, a \mapsto t, \ldots]\!]$, we cannot simply return t as it may contain locators that need adjustment. We can split locators in t into three groups:

1. Locators that reference some object *inside* of t require no adjustment as corresponding objects are still present after extraction.
2. Locators that reference object u, which we are extracting from, have to be substituted with u itself, to avoid losing information when we extract the subterm t.
3. Locators that reference some object *outside* of u have to be decremented since one nested object (u) disappeared.

We define locator substitution with the usual notation $t[\rho^n \mapsto u]$ meaning that we intend to substitute all locators referencing the same object as ρ^n in t with term u (with all locators updated properly).

Definition 2.6. *Locator substitution* $t[\rho^n \mapsto u]$ is defined inductively on φ-terms (see Figure 1).

Definition 2.7. A φ-term t is called *closed* if all its locators reference some object in t. Otherwise it is *open*.

2.3 Evaluation

In this section we define small step reduction semantics for φ-calculus. We introduce four congruence rules, to enable reduction in subterms, as well as three main reduction rules. Attribute access and attribute instantiation provide two reduction rules given an object term. Then, considering the presence of the decorator attribute φ, we get one extra rule for attribute access. Figure 1 shows the complete set of reduction rules.

Rule DOT_c formalizes the idea that extracting attribute c from an object t is straightforward as long as we substitute all locators that reference to t in the resulting term. Rule DOT_c^φ specifies further that whenever c is not an attribute of t but φ is, we should extract c from $t.\varphi$. Importantly, we do extract the term attached to φ immediately. Such extraction would require to check recursively whether we should go to $t.\varphi.\varphi$ and further. Instead we want our rules to perform just a single step of reduction.

Rule APP_c shows that attaching a term u to the attribute c requires incrementing locators in u.

2.3.1 Decorated instantiation.

As we are following the idea of Decorator pattern [16, Chapter 4], in φ-calculus only concrete objects are supposed to be decorated. That said, one could consider a variation of our calculus where decoration and later instantiation of abstract objects is possible as well. For example, we can introduce the following evaluation rule:

$$\frac{t \equiv [\![\ldots, \varphi \mapsto t_\varphi, \ldots]\!] \qquad c \notin \text{attr}(t)}{t(c \mapsto u) \rightsquigarrow [\![\ldots, \varphi \mapsto t_\varphi(c \mapsto u \uparrow), \ldots]\!]}\ \text{APP}_c^\varphi$$

For the rest of this paper we will assume φ-calculus without APP_c^φ rule. However, all results still hold for a variation of the calculus with it.

2.4 Modelling object-oriented languages

Bugayenko [8] introduced his calculus as a semi-formal mathematical model for EO programming language. EO language is an object-oriented programming language that relies on decoration instead of inheritance to work with object hierarchies. In this subsection, we give examples that relate concepts from class-based and prototype-based object-oriented code to terms in φ-calculus.

2.4.1 Class-based.

Classes can be modelled as the so called "object factories" — objects with a special method new, that produces an instance of the class. To model inheritance, this method can take an object as input and extend it with all the necessary methods using decoration.

Example 2.8. Consider the following code snippet in Java:

```
class Base { Integer g() { return 3; } }
class Derived extends Base {
  Integer f() { return 2 + this.g(); }
}
Derived d = new Derived();
```

Syntax

$$
\begin{array}{rll}
t := & & \text{(terms)} \\
& t.a & \text{(attribute)} \\
& \mid t_1(a \mapsto t_2) & \text{(application)} \\
& \mid [\![a_1 \mapsto \varnothing, \ldots, a_k \mapsto \varnothing, b_1 \mapsto t_1, \ldots, b_n \mapsto t_n]\!] & \text{(object)} \\
& \mid \rho^n & (n\text{-th parent object locator})
\end{array}
$$

Evaluation

$$\frac{t_i \rightsquigarrow t_i'}{[\![\ldots, b_i \mapsto t_i, \ldots]\!] \rightsquigarrow [\![\ldots, b_i \mapsto t_i', \ldots]\!]}\ \text{cong}_{\text{OBJ}} \qquad \frac{t \rightsquigarrow t'}{t.a \rightsquigarrow t'.a}\ \text{cong}_{\text{DOT}}$$

$$\frac{t \rightsquigarrow t'}{t(a \mapsto u) \rightsquigarrow t'(a \mapsto u)}\ \text{cong}_{\text{APP}^{\text{L}}} \qquad \frac{u \rightsquigarrow u'}{t(a \mapsto u) \rightsquigarrow t'(a \mapsto u')}\ \text{cong}_{\text{APP}^{\text{R}}}$$

$$\frac{t \equiv [\![\ldots, c \mapsto t_c, \ldots]\!]}{t.c \rightsquigarrow t_c\,[\rho^0 \mapsto t]}\ \text{DOT}_c \qquad \frac{t \equiv [\![\ldots]\!] \qquad c \notin \text{attr}(t) \qquad \varphi \in \text{attr}(t)}{t.c \rightsquigarrow t.\varphi.c}\ \text{DOT}_c^{\varphi}$$

$$\frac{t \equiv [\![a_1 \mapsto \varnothing, \ldots, a_k \mapsto \varnothing, c \mapsto \varnothing, b_1 \mapsto t_1, \ldots, b_n \mapsto t_n]\!]}{t(c \mapsto u) \rightsquigarrow [\![a_1 \mapsto \varnothing, \ldots, a_k \mapsto \varnothing, c \mapsto u\uparrow, b_1 \mapsto t_1, \ldots, b_n \mapsto t_n]\!]}\ \text{APP}_c$$

Locator substitution

$$
\begin{aligned}
\rho^n[\rho^m \mapsto u] &:= \rho^n \quad \text{if } n < m \\
\rho^n[\rho^n \mapsto u] &:= u \\
\rho^n[\rho^m \mapsto u] &:= \rho^{n-1} \quad \text{if } n > m \\
t.a[\rho^n \mapsto u] &:= t[\rho^n \mapsto u].a \\
t_1(a \mapsto t_2)[\rho^n \mapsto u] &:= t_1[\rho^n \mapsto u](a \mapsto t_2[\rho^n \mapsto u]) \\
[\![a_1 \mapsto \varnothing, \ldots, b_1 \mapsto t_1, \ldots, b_n \mapsto t_n]\!][\rho^n \mapsto u] &:= [\![a_1 \mapsto \varnothing, \ldots, b_1 \mapsto t_1[\rho^{n+1} \mapsto u\uparrow], \ldots, b_n \mapsto t_n[\rho^{n+1} \mapsto u\uparrow]]\!]
\end{aligned}
$$

Figure 1. Syntax and evaluation rules for φ-calculus.

These declarations roughly correspond to the following φ-terms:

$$
\begin{aligned}
\text{Base} &:= [\![\text{new} \mapsto [\![\text{this} \mapsto \varnothing,\ \varphi \mapsto \rho^0.\text{this},\ g \mapsto 3]\!]]\!] \\
\text{Derived} &:= [\![\text{new} \mapsto [\![\text{this} \mapsto \varnothing, \\
&\qquad \varphi \mapsto \text{Base.new}(\text{this} \mapsto \rho^0.\text{this}), \\
&\qquad f \mapsto 2.\text{add}(n \mapsto \rho^0.g)]\!]]\!] \\
d &:= \text{Derived.new}(\text{this} \mapsto [\![ ]\!])
\end{aligned}
$$

The above example aims to provide intuition for φ-calculus. A proper mapping from Java to φ-calculus would require more technical details, such as dealing with mutable attributes, generics, interfaces, and other features.

2.4.2 Prototype-based. Prototypes in object-oriented languages, such as JavaScript, work similarly to decorators in φ-calculus: when looking for a method in a JavaScript object, the interpreter checks object's *own properties* first and then, if such properties are absent, the interpreter proceeds to look for the method in the object's *prototype*, unless the prototype is *null*. Importantly, JavaScript's objects are mutable allowing for dynamic prototypes and properties, whereas in φ-calculus objects are immutable.

Example 2.9. Consider the following code snippet in JavaScript:

```
let A = function() { this.x = 3; }
A.prototype.f = function() { return this.x; }
let b = new A();
```

This snippet would translate to φ-calculus as follows:

$$
\begin{aligned}
\text{A} :=& [\![\text{new} \mapsto [\![\text{x} \mapsto 3, \varphi \mapsto \rho^1.\text{prototype}]\!], \\
&\quad \text{prototype} \mapsto [\![\text{f} \mapsto [\![\text{this} \mapsto \varnothing, \varphi \mapsto \rho^0.\text{this.x}]\!]]\!]]\!] \\
\text{b} :=& \text{A.new}
\end{aligned}
$$

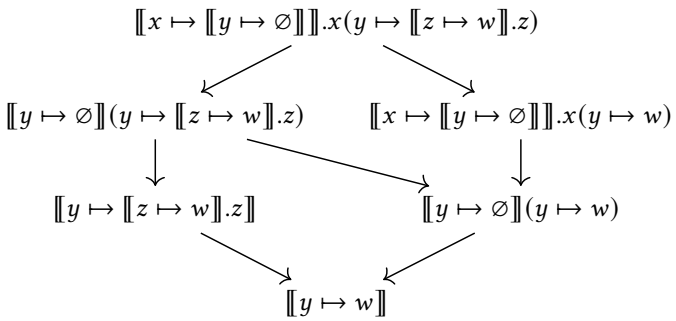
3 Confluence

Intuitively, we think of φ-terms as programs in terms of objects. Moreover, we assume a single meaning to each such program. In other words, every program either diverges (e.g. falls into an infinite loop) or produces the final object (normal form). To justify the use of "the" in "the normal form" we require the uniqueness of normal form.

In this section we prove a more general result. We show that φ-calculus possesses the following property: if some term t can be reduced in different ways to terms u and v then there exists some term w such that both u and v reduce to w. This is known as Church-Rosser property:

Definition 3.1. A relation $\xrightarrow{X}$ on terms is said to satisfy *Church-Rosser property* if for any terms t, u and v, if $t \xrightarrow{X} u$ and $t \xrightarrow{X} v$, then there exists some term w such that $u \xrightarrow{X} w$ and $v \xrightarrow{X} w$.

In general, a φ-term can be rewritten in different ways, since it may contain several redexes. For example, here is a graph with all possible reductions for a term:



Other terms may have infinite rewrite sequences, e.g.:

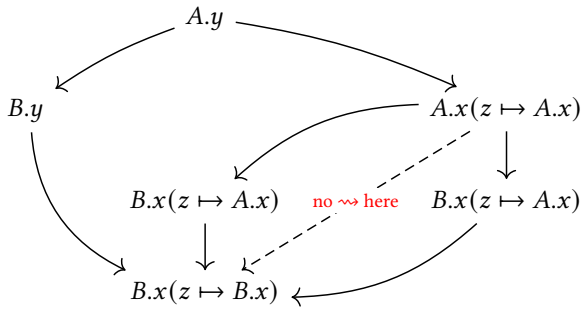
$$\llbracket x \mapsto \xi.y, y \mapsto \xi.x \rrbracket.x$$
$$\llbracket x \mapsto \xi.y, y \mapsto \xi.x \rrbracket.y$$

In these two examples, we can see diamond property satisfied for $\rightsquigarrow$, but unfortunately, the property does not hold in general.

Example 3.2. Consider the following φ-terms A and B:

$$A \equiv \llbracket x \mapsto \llbracket a \mapsto \llbracket z \mapsto \varnothing \rrbracket \rrbracket.a, y \mapsto \rho^0.x(z \mapsto \rho^0.x) \rrbracket$$
$$B \equiv \llbracket x \mapsto \quad \llbracket z \mapsto \varnothing \rrbracket \quad , y \mapsto \rho^0.x(z \mapsto \rho^0.x) \rrbracket$$

Note that $A \rightsquigarrow B$ by cong_{OBJ} and DOT_a. Reduction of $A.y$ illustrates that substitution can introduce multiple redexes, that cannot be reduced in a single step of $\rightsquigarrow$:



To prove confluence for φ-calculus we follow these steps:

1. We introduce parallel reduction on φ-terms; this kind of reduction possesses the diamond property. Using parallel reduction in a proof of confluence is due to Tait and Löf [4, Section 3.2]
2. We show that parallel reduction is equivalent to regular reduction.
3. We show that parallel reduction possesses the diamond property via complete development, following Takahashi's technique [25].
4. We prove confluence for regular reduction via equivalence with parallel reduction, also using the fact that confluence for parallel reduction follows from its diamond property via [20, Lemma 1.17].

While regular reduction performs exactly one reduction step somewhere in a term, the idea of parallel reduction is to perform arbitrary number of reductions *in parallel*. Performing reductions in parallel intuitively means that we do not reduce a term after performing substitution or adding $.\varphi$. Figure 2 gives the rules for parallel reduction. Observe that φ-terms in Example 3.2 satisfy the diamond property with single step parallel reduction: $A.x(z \mapsto A.x) \Rrightarrow B.x(z \mapsto B.x)$ by $\text{cong}^{\Rrightarrow}_{\text{APP}}$.

One important property of parallel reduction is that "doing nothing" is also a parallel reduction. We justify this with the following proposition:

Proposition 3.3 (Reflexivity of parallel reduction)**.** *Let t be a φ-term. Then $t \Rrightarrow t$.*

Proof. Straightforward by structural induction on t. □

Since our goal is to prove confluence for regular reduction via parallel reduction, we need to establish that the two kinds of reduction are equivalent, meaning that if one term reduces regularly to another term, then those terms are also related via parallel reduction and vice versa.

Proposition 3.4 (Equivalence of $\Rrightarrow$ and $\rightsquigarrow$)**.** *Parallel reduction ($\Rrightarrow$) is equivalent to regular reduction ($\rightsquigarrow$):*

1. *$t \rightsquigarrow t'$ implies $t \Rrightarrow t'$*
2. *$t \overset{*}{\rightsquigarrow} t'$ implies $t \overset{*}{\Rrightarrow} t'$*
3. *$t \Rrightarrow t'$ implies $t \overset{*}{\rightsquigarrow} t'$*
4. *$t \overset{*}{\Rrightarrow} t'$ implies $t \overset{*}{\rightsquigarrow} t'$*

Proof. Straightforward by structural induction. □

In the following proofs we need to know that if $t \Rrightarrow t'$ and $u \Rrightarrow u'$ then $t[\rho^0 \mapsto u] \Rrightarrow t'[\rho^0 \mapsto u']$. We prove a slightly more general lemma:

Lemma 3.5 (Substitution lemma)**.** *Let t, t', u, u' be φ-terms and $t \Rrightarrow t'$ and $u \Rrightarrow u'$. Then $t[\rho^i \mapsto u] \Rrightarrow t'[\rho^i \mapsto u']$.*

To show that parallel reduction satisfies the diamond property, we adapt Takahashi's technique [25] and define complete development of a term, which is intuitively the maximum possible one-step parallel reduction of a term. The

$$\frac{t_1 \Rrightarrow t_1' \quad \ldots \quad t_n \Rrightarrow t_n'}{[\![a_1 \mapsto \varnothing, \ldots, a_k \mapsto \varnothing, b_1 \mapsto t_1, \ldots, b_n \mapsto t_n]\!] \Rrightarrow [\![a_1 \mapsto \varnothing, \ldots, a_k \mapsto \varnothing, b_1 \mapsto t_1', \ldots, b_n \mapsto t_n']\!]}\ \mathrm{cong}^{\Rrightarrow}_{\mathrm{OBJ}}$$

$$\frac{}{\rho^n \Rrightarrow \rho^n}\ \mathrm{cong}^{\Rrightarrow}_{\rho} \qquad \frac{t \Rrightarrow t'}{t.a \Rrightarrow t'.a}\ \mathrm{cong}^{\Rrightarrow}_{\mathrm{DOT}} \qquad \frac{t \Rrightarrow t' \quad u \Rrightarrow u'}{t(a \mapsto u) \Rrightarrow t'(a \mapsto u')}\ \mathrm{cong}^{\Rrightarrow}_{\mathrm{APP}}$$

$$\frac{t \Rrightarrow t' \quad t' \equiv [\![\ldots, c \mapsto t_c, \ldots]\!]}{t.c \Rrightarrow t_c\,[\rho^0 \mapsto t']}\ \mathrm{DOT}^{\Rrightarrow}_{c} \qquad \frac{t \Rrightarrow t' \quad t' \equiv [\![\ldots]\!] \quad c \notin \mathrm{attr}(t') \quad \varphi \in \mathrm{attr}(t')}{t.c \Rrightarrow t'.\varphi.c}\ \mathrm{DOT}^{\varphi\Rrightarrow}_{c}$$

$$\frac{t \Rrightarrow t' \quad t' \equiv [\![a_1 \mapsto \varnothing, \ldots, a_k \mapsto \varnothing, c \mapsto \varnothing, b_1 \mapsto t_1, \ldots, b_n \mapsto t_n]\!] \quad u \Rrightarrow u'}{t(c \mapsto u) \Rrightarrow [\![a_1 \mapsto \varnothing, \ldots, a_k \mapsto \varnothing, c \mapsto u' \uparrow, b_1 \mapsto t_1, \ldots, b_n \mapsto t_n]\!]}\ \mathrm{APP}^{\Rrightarrow}_{c}$$

Figure 2. Parallel reduction rules for φ-calculus.

idea is that if $t \Rrightarrow t'$ then simply by performing all parallel reductions that were "skipped" when producing t', we get from t' to the complete development of t. More formally:

Definition 3.6. Let t be a φ-term. Then a term t^+ denotes the *complete development* of t, defined recursively as follows:

$$(\rho^n)^+ := \rho^n$$

$$(t.a)^+ := \begin{cases} t_a[\rho^0 \mapsto t^+], & \text{if } t^+ \equiv [\![\ldots, a \mapsto t_a, \ldots]\!] \\ t^+.\varphi.a, & \text{if } a \notin \mathrm{attr}(t^+) \text{ and } \varphi \in \mathrm{attr}(t^+) \\ t^+.a & \text{otherwise} \end{cases}$$

$$(t(a \mapsto u))^+ := \begin{cases} [\![a \mapsto u^+ \uparrow, \ldots]\!], & \text{if } t^+ \equiv [\![a \mapsto \varnothing, \ldots]\!] \\ t^+(a \mapsto u^+) & \text{otherwise} \end{cases}$$

$$([\![a_1 \mapsto \varnothing, \ldots, a_k \mapsto \varnothing, b_1 \mapsto t_1, \ldots, b_n \mapsto t_n]\!])^+ := [\![a_1 \mapsto \varnothing, \ldots, a_k \mapsto \varnothing, b_1 \mapsto t_1^+, \ldots, b_n \mapsto t_n^+]\!]$$

The definition clearly shows that we basically follow rules of parallel reduction, but always choose the rule that does the most work. In particular, if both $\mathrm{APP}^{\Rrightarrow}_{c}$ and $\mathrm{cong}^{\Rrightarrow}_{\mathrm{APP}}$ are applicable, we prioritize rule $\mathrm{APP}^{\Rrightarrow}_{c}$ since it performs strictly more reductions. Consequently, a term can always be parallel reduced to its complete development:

Proposition 3.7. *Let t be a φ-term. Then $t \Rrightarrow t^+$.*

Proof. Straightforward by induction on t. □

Before we prove the diamond property, we need a simpler result for just one half of the diamond:

Proposition 3.8. *Let t, t' be φ-terms and $t \Rrightarrow t'$. Then $t' \Rrightarrow t^+$.*

Corollary 3.9 (Diamond property of parallel reduction)**.** *Let t, u, v be φ-terms and $t \Rrightarrow u$ and $t \Rrightarrow v$. Then there exists a φ-term w such that $u \Rrightarrow w$ and $v \Rrightarrow w$.*

Proof. Let $w \equiv t^+$. With 3.8, $u \Rrightarrow w$ and $v \Rrightarrow w$. □

Corollary 3.10 (Confluence of parallel reduction)**.** *Let t, u, v be φ-terms and $t \overset{*}{\Rrightarrow} u$ and $t \overset{*}{\Rrightarrow} v$. Then there exists a φ-term w such that $u \overset{*}{\Rrightarrow} w$ and $v \overset{*}{\Rrightarrow} w$.*

Proof. Follows from the diamond property (see [20, Lemma 1.17]). □

Theorem 3.11 (Confluence)**.** *Let t, u, v be φ-terms and $t \overset{*}{\rightsquigarrow} u$ and $t \overset{*}{\rightsquigarrow} v$. Then there exists a φ-term w such that $u \overset{*}{\rightsquigarrow} w$ and $v \overset{*}{\rightsquigarrow} w$.*

Proof. Follows from Proposition 3.4 and Corollary 3.10. □

As usual, a term t is in *normal form* if it has no redexes. With this definition we have a trivial corollary of Theorem 3.11:

Corollary 3.12. *Every φ-term has at most one normal form.*

Example 3.13. Term $[\![x \mapsto \rho^0.y, y \mapsto \rho^0.x]\!].x$ does not have a normal form.

A more strict definition of normal forms is presented in Figure 3.

3.1 Completeness of normal order evaluation

In this subsection, we define normal order reduction strategy and prove that this strategy reduces a term to its normal form, if such a form exists.

Similarly to λ-calculus, normal order reduction in φ-calculus is defined in terms of head reduction. Head reduction does not apply to redexes inside objects and to arguments of object application. If possible, normal order reduction performs head reduction, otherwise it performs reduction in the leftmost subterm that can be reduced.

Our proof of completeness of normal order evaluation relies on the standardization theorem: if t reduces to u, then there exists a reduction path, where head reductions are performed first, and internal reductions follow.

Contrary to the head reduction, internal reduction applies to redexes inside objects and in arguments of application. During the proof, we exploit internal parallel reduction which is the intersection of parallel reduction and reflexive-transitive closure of the internal regular reduction.

Normal form

$$t \text{ has normal form if } t \equiv \begin{cases} \rho^n & \\ [\![a_1 \mapsto \varnothing, \ldots, a_k \mapsto \varnothing, b_1 \mapsto t_1, \ldots, b_n \mapsto t_n]\!], & \text{if } t_j \text{ is in NF} \\ s.a, & \text{if } s \text{ is in NF and } s.a \text{ is not a redex} \\ s(a \mapsto u), & \text{if } s \text{ and } u \text{ are in NF and } s(a \mapsto u) \text{ is not a redex} \end{cases}$$

Head reduction

$$\frac{t \overset{h}{\rightsquigarrow} t'}{t.a \overset{h}{\rightsquigarrow} t'.a}\ \text{cong}^h_{\text{DOT}} \qquad \frac{t \overset{h}{\rightsquigarrow} t'}{t(a \mapsto u) \overset{h}{\rightsquigarrow} t'(a \mapsto u)}\ \text{cong}^h_{\text{APP}}$$

$$\frac{t \equiv [\![\ldots, c \mapsto t_c, \ldots]\!]}{t.c \overset{h}{\rightsquigarrow} t_c\ [\rho^0 \mapsto t]}\ \text{DOT}_c \qquad \frac{t \equiv [\![\ldots]\!] \qquad c \notin \text{attr}(t) \qquad \varphi \in \text{attr}(t)}{t.c \overset{h}{\rightsquigarrow} t.\varphi.c}\ \text{DOT}^{\varphi}_c$$

$$\frac{t \equiv [\![a_1 \mapsto \varnothing, \ldots, a_k \mapsto \varnothing, c \mapsto \varnothing, b_1 \mapsto t_1, \ldots, b_n \mapsto t_n]\!]}{t(c \mapsto u) \overset{h}{\rightsquigarrow} [\![a_1 \mapsto \varnothing, \ldots, a_k \mapsto \varnothing, c \mapsto u \uparrow, b_1 \mapsto t_1, \ldots, b_n \mapsto t_n]\!]}\ \text{APP}_c$$

Normal order

$$t.a \overset{n.o.}{\rightsquigarrow} \begin{cases} s, & \text{if } t.a \overset{h}{\rightsquigarrow} s \\ t'.a, & \text{else if } t \overset{n.o.}{\rightsquigarrow} t' \end{cases}$$

$$t(a \mapsto u) \overset{n.o.}{\rightsquigarrow} \begin{cases} s, & \text{if } t(a \mapsto u) \overset{h}{\rightsquigarrow} s \\ t'(a \mapsto u), & \text{else if } t \overset{n.o.}{\rightsquigarrow} t' \\ t(a \mapsto u'), & \text{else if } u \overset{n.o.}{\rightsquigarrow} u' \end{cases}$$

$$[\![a_1 \mapsto \varnothing, \ldots, b_1 \mapsto t_1, \ldots, b_i \mapsto t_i, \ldots b_n \mapsto t_n]\!] \overset{n.o.}{\rightsquigarrow} [\![a_1 \mapsto \varnothing, \ldots, b_1 \mapsto t_1, \ldots, b_i \mapsto t'_i, \ldots b_n \mapsto t_n]\!], \text{if } t_i \overset{n.o.}{\rightsquigarrow} t'_i$$

Regular internal reduction

$$\frac{t_i \rightsquigarrow t'_i}{[\![\ldots, b_i \mapsto t_i, \ldots]\!] \overset{i}{\rightsquigarrow} [\![\ldots, b_i \mapsto t'_i, \ldots]\!]}\ \text{cong}^i_{\text{OBJ}} \qquad \frac{t \overset{i}{\rightsquigarrow} t'}{t.a \overset{i}{\rightsquigarrow} t'.a}\ \text{cong}^i_{\text{DOT}}$$

$$\frac{t \overset{i}{\rightsquigarrow} t'}{t(a \mapsto u) \overset{i}{\rightsquigarrow} t'(a \mapsto u)}\ \text{cong}^i_{\text{APP}^{\text{L}}} \qquad \frac{u \rightsquigarrow u'}{t(a \mapsto u) \overset{i}{\rightsquigarrow} t'(a \mapsto u')}\ \text{cong}^i_{\text{APP}^{\text{R}}}$$

Parallel internal reduction

$$\frac{t_1 \Rrightarrow t'_1 \quad \ldots \quad t_n \Rrightarrow t'_n}{[\![a_1 \mapsto \varnothing, \ldots, a_k \mapsto \varnothing, b_1 \mapsto t_1, \ldots, b_n \mapsto t_n]\!] \overset{i}{\Rrightarrow} [\![a_1 \mapsto \varnothing, \ldots, a_k \mapsto \varnothing, b_1 \mapsto t'_1, \ldots, b_n \mapsto t'_n]\!]}\ \text{cong}^{\overset{i}{\Rrightarrow}}_{\text{OBJ}}$$

$$\frac{}{\rho^n \overset{i}{\Rrightarrow} \rho^n}\ \text{cong}^{\overset{i}{\Rrightarrow}}_{\rho} \qquad \frac{t \overset{i}{\Rrightarrow} t'}{t.a \overset{i}{\Rrightarrow} t'.a}\ \text{cong}^{\overset{i}{\Rrightarrow}}_{\text{DOT}} \qquad \frac{t \overset{i}{\Rrightarrow} t' \qquad u \Rrightarrow u'}{t(a \mapsto u) \overset{i}{\Rrightarrow} t'(a \mapsto u')}\ \text{cong}^{\overset{i}{\Rrightarrow}}_{\text{APP}}$$

Figure 3. Head, internal and normal order reductions

The proof is developed in a close relation to the one of Takahashi [25]. First, we show that one step of parallel reduction can be decomposed to multiple head reductions and one internal parallel reduction step.

Lemma 3.14 (Main Lemma). *$t \Rrightarrow s$ implies $t \overset{h^*}{\rightsquigarrow} r \overset{i}{\Rrightarrow} s$ for some r.*

Lemma 3.15 (Substitution Lemma for $\overset{h}{\rightsquigarrow}$). *If $t \overset{h}{\rightsquigarrow} s$, then $t[\rho^n \mapsto q] \overset{h}{\rightsquigarrow} s[\rho^n \mapsto q]$.*

Lemma 3.16 (Substitution Lemma for $\overset{i}{\Rightarrow}$). *If $t \overset{i}{\Rightarrow} s$ and $q \overset{i}{\Rightarrow} r$, then $t[\rho^n \mapsto q] \overset{i}{\Rightarrow} s[\rho^n \mapsto r]$.*

Then, we show that if head reduction follows internal parallel reduction, reductions can be reordered so that head reductions occur first.

Lemma 3.17 (Standardizing Reductions). *For any φ-terms t, r, s such that $t \overset{i}{\Rightarrow} r \overset{h}{\rightsquigarrow} s$, there exists φ-term q, such that $t \overset{h^*}{\rightsquigarrow} q \overset{i}{\Rightarrow} s$.*

Proof. By induction on the structure of $r \overset{h}{\rightsquigarrow} s$. □

Standardization theorem is then a corollary:

Corollary 3.18. *$t \overset{*}{\rightsquigarrow} s$ implies $t \overset{h^*}{\rightsquigarrow} r \overset{i^*}{\rightsquigarrow} s$ for some φ-term r.*

Proof. Recall that equivalence of $\overset{*}{\rightsquigarrow}$ and $\overset{*}{\Rightarrow}$ (4) implies that $t \overset{*}{\Rightarrow} s$.

By induction on $\overset{*}{\Rightarrow}$,

1. if $t \equiv s$, then $t \overset{h^*}{\rightsquigarrow} r \overset{i^*}{\rightsquigarrow} s$ for $r \equiv s$
2. else, $t \Rightarrow q$ and $q \overset{*}{\Rightarrow} s$. By Main Lemma, there exists p, such that $t \overset{h^*}{\rightsquigarrow} p \overset{i}{\Rightarrow} q$. By induction hypothesis, there exists r', such that $q \overset{h^*}{\rightsquigarrow} r' \overset{i^*}{\rightsquigarrow} s$. Repeated application of the Standardizing Reductions Lemma propagates $\overset{i}{\Rightarrow}$ in $p \overset{i}{\Rightarrow} q \overset{h^*}{\rightsquigarrow} r'$ to the end, and the equivalence of $\overset{i}{\Rightarrow}$ and $\overset{i^*}{\rightsquigarrow}$ completes the proof.

□

Theorem 3.19 (Completeness of normal order evaluation). *If t has normal form s, then $t \overset{n.o.^*}{\rightsquigarrow} s$.*

Proof. With the corollary, $t \overset{h^*}{\rightsquigarrow} r \overset{i^*}{\rightsquigarrow} s$ for some φ-term r.

By induction on the structure of s,

1. if $s \equiv \rho^n$, then $r \equiv \rho^n$. As $t \overset{h^*}{\rightsquigarrow} r$, $t \overset{n.o.^*}{\rightsquigarrow} r$, which follows from the definition of $\overset{n.o.}{\rightsquigarrow}$.
2. if $s \equiv [\![a_1 \mapsto \varnothing, \ldots, a_k \mapsto \varnothing, b_1 \mapsto t_1, \ldots, b_n \mapsto t_n]\!]$, the $r \equiv [\![a_1 \mapsto \varnothing, \ldots, a_k \mapsto \varnothing, b_1 \mapsto t'_1, \ldots, b_n \mapsto t'_n]\!]$ and t_j is NF of t'_j. By induction hypothesis, $t'_j \overset{n.o.^*}{\rightsquigarrow} t_j$, so $r \overset{n.o.^*}{\rightsquigarrow} s$, hence $t \overset{n.o.^*}{\rightsquigarrow} s$.
3. if $s \equiv q.a$, then q is in NF and $q.a$ is not a redex, hence $r \equiv q'.a$, and q is NF of q'. By induction hypothesis, $q' \overset{n.o.^*}{\rightsquigarrow} q$, and since $q.a$ is not a redex, $q'.a \overset{n.o.^*}{\rightsquigarrow} q.a$. So, $t \overset{h^*}{\rightsquigarrow} r \overset{n.o.^*}{\rightsquigarrow} s$, and by definition of $\overset{n.o.}{\rightsquigarrow}$, $t \overset{n.o.^*}{\rightsquigarrow} s$.
4. if $s \equiv q(a \mapsto u)$, then q and u are in NF and $q(a \mapsto u)$ is not a redex, hence $r \equiv q'(a \mapsto u')$, and q is NF of q' and u is NF of u'. By induction hypothesis, $q' \overset{n.o.^*}{\rightsquigarrow} q$ and $u' \overset{n.o.^*}{\rightsquigarrow} u$, and since $q(a \mapsto u)$ is not a redex, $q'(a \mapsto u') \overset{n.o.^*}{\rightsquigarrow} q(a \mapsto u') \overset{n.o.^*}{\rightsquigarrow} q(a \mapsto u)$. So, $t \overset{n.o.^*}{\rightsquigarrow} s$.

□

4 Abstract machine

Bugayenko [8] gives graph-based operational semantics. Although the general idea is clear, his description lacks precision, in particular regarding the handling of parent locators. Still, the existing implementations of EO programming language and descriptions of graph-based semantics hint strongly that intended semantics are those of a non-strict evaluation.

In this section, we introduce an abstract machine à la Krivine that performs call-by-name reduction of φ-terms, therefore computing their weak head normal form.

4.1 Call-by-name abstract machine

Here we present *term-actions-parents* abstract machine (TAP machine) for call-by-name evaluation of φ-terms.

We begin by introducing configurations of TAP machine:

Definition 4.1. An *object closure* is a tuple (t, e) of a φ-term t and a parent stack, described below. A *parent* is a tuple (t, o) of an object φ-term t and a partial mapping o (called *application mapping*) from $\mathcal{L}$ to a set of object closures. A *parent stack* is a finite sequence of parents. An empty parent stack is denoted ϵ. An *action* is either an attribute access denoted $.a$ for some attribute $a \in \mathcal{L}$, or an application denoted $(a \mapsto c)$ for some attribute $a \in \mathcal{L}$ and an object closure c. An *action stack* is a finite sequence of actions. An empty action stack is denoted ϵ. A *configuration* of TAP machine is a triple $\langle T, A, P \rangle$, where T is either a φ-term in focus or an empty symbol ϵ, A is a stack of actions, and P — a parent stack.

The machine operates by following transition rules between the configurations. Figure 4 gives the transition rules. The first two rules instruct how to dereference parent locator ρ^n. Attribute access and application terms are broken down into a smaller term and an action. For application $t(a \mapsto u)$ we save the current parent stack e and put an action $(a \mapsto (u, e))$ on the stack of actions. This effectively captures the necessary context required to compute term u later. Rule 10 puts an object term on the stack with an empty application mapping (denoted $\varnothing$). The remaining four rules describe effects of actions on the parent on the top of the stack. If the parent object on the stack has an attribute that we want to access, we extract the corresponding subterm and set it as our new current term. On the other hand, if the attribute is mapped by the parent application mapping to some object closure, then we take the term from that closure as our new current term and replace current parent stack with the one from the closure. If a parent has no required attribute, but has φ, we simply add $.\varphi$ action to the action stack. Finally, an application action merely updates the parent at the top of the parent stack, by replacing its application

mapping correspondingly: $o \cup \{a \mapsto (u, e')\}$ denotes a mapping that maps attribute a to object closure (u, e') and maps any other attribute x to $o(x)$.

An object closure can be converted back to a φ-term by converting every parent into a φ-term and then performing locator substitution, instantiating corresponding parents. A parent (t, o) is converted into a φ-term by joining its object term t with its application mapping and converting every object closure in that mapping to a φ-term. Any configuration $\langle t, A, P\rangle$ (or $\langle \epsilon, A, p : P\rangle$) can be converted back to a φ-term by appending the stack of actions, where object closures are converted to φ-terms, to the term produced from the closure (t, P) (resp. (t, P) where t is produced from p).

Proposition 4.2 (Soundness of TAP machine)**.** *Let t be a closed φ-term. Then starting from configuration $C_0 = \langle t, \epsilon, \epsilon\rangle$ TAP machine operates for finitely many steps if and only if t has a weak head normal form. Moreover, if it stops with configuration C_n then this configuration corresponds to the weak head normal form of t.*

Proof. Each reduction in call-by-name evaluation sequence corresponds unambiguously to zero or more transitions of the TAP machine. Transitions from equivalent configurations, corresponding to the same φ-term, destructure current term, so there can only be a finite sequence of them. □

5 Translation to λ-calculus

In this section we compare φ-calculus with λ-calculus, present translation rules from one to the other, and prove soundness of the translation.

5.1 λ-calculus with records

We will use Mitchell Wand's λ-calculus with records [27], including both record extension (via **with**-expression) and record concatenation. As we will be translating locators approximately to de Bruijn indices in λ-terms, we will focus on a nameless variation of the syntax.

One important detail for us will be the computation rule for record extension. We will consider a term of the form e **with** $\{\ldots\}$ in weak head normal form, and extend the λ-calculus with the following evaluation rules:

$$(e \textbf{ with } \{a = e_a, \ldots\}).a \longrightarrow e_a$$
$$(e \textbf{ with } \{\ldots\}).a \longrightarrow e.a \quad \text{if } a \text{ is not in } \{\ldots\}$$

This slight modification of Wand's calculus allows strictly more terms to avoid diverging computation, and is crucial for translation of decorators from φ-calculus.

5.2 Translation from φ-calculus to λ-calculus

To translate φ-terms to λ-terms, one must understand how to represent objects. Since records have nothing like void attributes, we cannot map void attributes directly to record attributes. So, instead, we will represent objects as functions taking records with instantiated void attributes. For example, we would like to represent an empty object $[\![\,]\!]$ as a constant function $\lambda\{\}$, and an object $[\![x \mapsto \varnothing, y \mapsto [\![\,]\!]]\!]$ as a function $\lambda\{x = 0.x, y = \lambda\{\}\}$.

Since locators enable referencing outer terms, for translation we will also make use of the fixpoint combinator. In particular, a term $[\![x \mapsto \rho^0]\!]$ should be translated into $\mathrm{fix}(\lambda\lambda\{x = \lambda(2\ (1\ \textbf{with}\ 0)\})$. Note that the outermost λ introduces the translated object (represented as a function) that is then referenced as 2 in 2 (1 **with** 0). 1 references the instantiated void attributes passed to the outer term, and 0 references the instantiated void attributes passed to the locator ρ^0.

In general, translation objects terms involves two λ-abstractions. One abstraction is used together with the fixed point combinator, to allow locators. And another one is used to properly represent void attributes. So, when translating locator ρ^n we need to represent it as a that should reference the proper outer term and corresponding void attributes. That is why we translate ρ^n to $\lambda\underline{2n+2}(\underline{2n+1} \parallel \underline{0})$. All the other translation rules follow naturally and are presented in Figure 5.

5.3 Soundness of translation

The translation is sound if it commutes with computation. That is, given φ-terms t and u such that $t \rightsquigarrow_\varphi u$, we have $\mathrm{trans}_{\varphi\to\lambda}(t) \approx \mathrm{trans}_{\varphi\to\lambda}(u)$. Intuitively, by $e_1 \approx e_2$ we mean that e_1 and e_2 are observationally equivalent. More precisely, $e_1 \approx e_2$ if and only if e_1 is $\beta\eta\zeta$-equivalent to e_2. Here by ζ-equivalence we mean the obvious congruence rules, like associativity of $\parallel$: $x \parallel (y \parallel z) \approx_\zeta (x \parallel y) \parallel z$.

Proposition 5.1. *Let t, u be φ-terms and $t \rightsquigarrow_\varphi u$. Then*

$$\underset{\varphi\to\lambda}{\mathrm{trans}}(t) \approx \underset{\varphi\to\lambda}{\mathrm{trans}}(u)$$

Proof. Straightforward by induction on $t \rightsquigarrow_\varphi u$. □

Theorem 5.2 (Soundness of $\mathrm{trans}_{\varphi\to\lambda}$)**.** *Let t, t' be φ-terms and e be a λ-term such that $t \overset{*}{\rightsquigarrow}_\varphi t'$. Then*

$$\underset{\varphi\to\lambda}{\mathrm{trans}}(t) \approx \underset{\varphi\to\lambda}{\mathrm{trans}}(u)$$

Proof. Follows from Proposition 5.1 and confluence of λ-calculus. □

5.4 Translation from λ-calculus to φ-calculus

The translation from φ-calculus aimed to map attributes of objects in φ-terms to attributes of records in λ-calculus. Unfortunately, such mapping is impossible in the backwards direction, unless we drop the record concatenation. Indeed, record concatenation cannot be translated to φ-calculus directly, as the latter does not support any mechanism for merging objects.

There exist two remaining options for translation from λ-calculus. First, we could translate only the segment without record concatenation. Such translation is possible under assumption that attributes map to attributes. However, as

Initial configuration

$$t \longrightarrow \langle t, \epsilon, \epsilon \rangle$$

Transition rules for configurations

$$\langle \rho^0, p, e \rangle \longrightarrow \langle \epsilon, p, e \rangle \tag{6}$$

$$\langle \rho^{n+1}, p, (c, o) : e \rangle \longrightarrow \langle \rho^n, p, e \rangle \tag{7}$$

$$\langle t.a, p, e \rangle \longrightarrow \langle t, .a : p, e \rangle \tag{8}$$

$$\langle t(a \mapsto u), p, e \rangle \longrightarrow \langle t, (a \mapsto (u, e)) : p, e \rangle \tag{9}$$

$$\langle [\![a_1 \mapsto \varnothing, \ldots, b_1 \mapsto t_1, \ldots, b_n \mapsto t_n]\!], p, e \rangle \longrightarrow \langle \epsilon, p, ([\![a_1 \mapsto \varnothing, \ldots, b_1 \mapsto t_1, \ldots, b_n \mapsto t_n]\!], \varnothing) : e \rangle \tag{10}$$

$$\langle \epsilon, .a : p, ([\![a \mapsto u, \ldots]\!], o) : e \rangle \longrightarrow \langle u, p, ([\![a \mapsto u, \ldots]\!], o) : e \rangle \tag{11}$$

$$\langle \epsilon, .a : p, ([\![a \mapsto \varnothing, \ldots]\!], \{(a \mapsto (u, e')), \ldots\}) : e \rangle \longrightarrow \langle u, p, e' \rangle \tag{12}$$

$$\langle \epsilon, .a : p, (t, o) : e \rangle \longrightarrow \langle \epsilon, .\varphi.a : p, (t, o) : e \rangle, \tag{13}$$

$$\text{if } t \equiv [\![\ldots]\!],\ \varphi \notin \text{attr}(t),\ c \notin \text{attr}(t)$$

$$\langle \epsilon, (a \mapsto (u, e')) : p, ([\![a \mapsto \varnothing, \ldots]\!], o) : e \rangle \longrightarrow \langle \epsilon, p, ([\![a \mapsto \varnothing, \ldots]\!], o \cup \{a \mapsto (u, e')\}) : e \rangle, \tag{14}$$

$$\text{if } o \text{ is not defined for attribute } a$$

Figure 4. TAP machine for call-by-name evaluation of φ-terms.

Syntax of nameless λ-calculus with records

$$\begin{aligned}
e &:= \underline{n} && \text{(de Bruijn index)} \\
&\mid \lambda e && \text{(abstraction)} \\
&\mid (e_1) e_2 && \text{(application)} \\
&\mid \{a_1 = e_1, \ldots, a_n = e_n\} && \text{(record)} \\
&\mid e.a && \text{(attribute)} \\
&\mid e \ \textbf{with} \ \{a_1 = e_1, \ldots, a_n = e_n\} && \text{(record extension)} \\
&\mid e_1 \parallel e_2 && \text{(record concatenation)} \\
&\mid \text{fix}\, e && \text{(fixed point)}
\end{aligned}$$

Translation from φ-calculus to λ-calculus

$$\begin{aligned}
\underset{\varphi \to \lambda}{\text{trans}}(\rho^n) &:= \lambda \underline{(2n+2)} (\underline{(2n+1)} \parallel \underline{0}) \\
\underset{\varphi \to \lambda}{\text{trans}}(t.a) &:= (\underset{\varphi \to \lambda}{\text{trans}}(t)\ \{\}).a \\
\underset{\varphi \to \lambda}{\text{trans}}(t(a \mapsto u)) &:= \lambda(\text{inc}_\lambda(\underset{\varphi \to \lambda}{\text{trans}}(t)))(\underline{0} \ \textbf{with} \ \{a = \text{inc}_\lambda(\underset{\varphi \to \lambda}{\text{trans}}(u))\}) \\
\underset{\varphi \to \lambda}{\text{trans}}([\![\ldots, a_i \mapsto \varnothing, \ldots, b_j \mapsto t_j, \ldots, \varphi \mapsto \ldots]\!]) &:= \text{fix}(\lambda\lambda((\underline{1}\ \underline{0}).\varphi\ \{\}) \ \textbf{with} \ \{\ldots, a_i = 0.a_i, \ldots, b_j = \underset{\varphi \to \lambda}{\text{trans}}(t_j), \ldots\}) \\
\underset{\varphi \to \lambda}{\text{trans}}([\![\ldots, a_i \mapsto \varnothing, \ldots, b_j \mapsto t_j, \ldots]\!]) &:= \text{fix}(\lambda\lambda\{\ldots, a_i = 0.a_i, \ldots, b_j = \underset{\varphi \to \lambda}{\text{trans}}(t_j), \ldots\})
\end{aligned}$$

Figure 5. Translation from φ-calculus to λ-calculus with records.

record concatenation is important for translation of locators from φ-calculus, this option is not ideal, as we only have full translation in one direction. Second, we can encode attributes and records, for example, using Church encoding. This would effectively translate λ-terms with records to mere λ-terms, which can then be translated to φ-calculus.

As we do not see a satisfactory translation from λ-calculus to φ-calculus, we propose, as a potential future work, an extension of φ-calculus with object concatenation.

5.5 φ-calculus versus λ-calculus

φ-calculus shares some common features with various λ-calculi as both are confluent term-rewriting systems capturing the notion of computability. Yet, φ-calculus focused on objects differs from λ-calculus in the following important ways:

1. φ-calculus does not rely on λ-terms to represent functions. In φ-calculus everything is an object (in the sense of Definition 2.1).
2. The attribute access in φ-calculus is much more powerful than that of λ-calculus with records. In fact attribute access shares certain similarities with with β-reduction in λ-calculus because of the substitution involved.
3. Because of the decorators, φ-calculus requires no explicit analogue of fixpoint combinator. In a sense, objects have a recursive let-construction built into them.
4. The locators in φ-calculus are arguably more natural than de Bruijn indices used in λ-calculi.

As both calculi are Turing complete, λ-calculus can be encoded in φ-calculus and vice versa. However, φ-calculus shares enough similarities with λ-calculus with records to enable intuitive yet sound translations in one direction, and a partial translation in the other. These translations can be used not only to improve understanding of the two formalizations, but also to translate certain useful properties between the systems. In particular, such a translation might be useful to develop a sound type system for φ-calculus in the future.

6 Extensions

Bugayenko [8] introduces a calculus that reflects capabilities of his EO programming language. As such it is richer than φ-calculus we have presented in Section 2. In this section, we give examples of possible syntactic extensions to our calculus closing the gap between the two presentations. We leave out the more complicated extensions, such as mutable memory, primitive data types, or modelling input/output for future work.

6.1 Attribute-variables

Locators are often used in combination with attribute access: $\rho^n.a$. In practice, though, attribute names can be descriptive and unique (at least in a certain scope or subterm) so that a person can understand easily which object this attribute belongs to. Such practice prompts a version of the syntax where locators are optional and can be omitted. For example, instead of $[\![x \mapsto \rho^0.y, y \mapsto [\![z \mapsto \rho^1.x]\!]]\!]$ one could omit both locators and it would still be clear where attributes should come from: $[\![x \mapsto y, y \mapsto z \mapsto x]\!]$.

More formally, we extend syntax of φ-terms with *attribute-variables*:

Definition 6.1. A set of φ-terms with attribute-variables T_a is defined inductively as follows:

1. if $t \in T$ (t is a φ-term) then $t \in T_a$;
2. if $a \in \mathcal{L}$ then $a \in T_a$.

As long as all attributes are defined in some enclosing object, we can restore locators. To do so we, have to traverse the term while keeping track of locators for known attributes. For the latter we will use a context represented by a mapping $\Gamma : \mathcal{L} \to \mathbb{N} \cup \{\bot\}$. For convenience we will write $\Gamma, a \in \rho^n$ to mean context Γ' defined as follows:

$$\Gamma'(a) := n$$
$$\Gamma'(x) := \Gamma(x) \quad \text{when } x \neq a$$

We will also define an increment operation on the context:

$$\Gamma \uparrow (a) := \Gamma(a) + 1$$

Translation from φ-terms with attribute-variables to regular φ-terms can be summarized with the following rules:

$$\frac{}{\Gamma, a \in \rho^n \vdash a \longrightarrow \rho^n.a} \qquad \frac{}{\Gamma \vdash \rho^n \longrightarrow \rho^n}$$

$$\frac{\Gamma \vdash t \longrightarrow t'}{\Gamma \vdash t.a \longrightarrow t'.a} \qquad \frac{\Gamma \vdash t \longrightarrow t' \qquad \Gamma \vdash u \longrightarrow u'}{\Gamma \vdash t(a \mapsto u) \longrightarrow t(a \mapsto u')}$$

$$\frac{\Gamma \uparrow, a_0 \in \rho^0, \ldots, b_1 \in \rho^0, \ldots \vdash t_j \longrightarrow t'_j \quad \text{for all } j \in \{1, \ldots, n\}}{\Gamma \vdash [\![a_1 \mapsto \varnothing, \ldots, b_1 \mapsto t_1, \ldots]\!] \longrightarrow [\![a_1 \mapsto \varnothing, \ldots, b_1 \mapsto t'_1, \ldots]\!]}$$

Note that, by reversing the first rule, we can similarly erase unnecessary locators, yielding translation in the other direction.

6.2 Global object

Sometimes tracking nested objects might be inconvenient, and it might be easier to reference objects "from the top-level". This statement is especially true in an actual programming language. One can extend calculus with explicit names for objects to use instead of locators, but Section 6.1 already provides a clean solution to provide a name for all terms, except for those at the top-level.

To reference top-level object by name, we may extend syntax with *global object locator* Φ. Similarly to attribute-variables, this extension is purely syntactic and requires no extension of evaluation rules as a proper locator can safely replace each occurrence of Φ.

6.3 Positional arguments

Void attributes often serve as method arguments. To emphasize this role, we extend the syntax with positional arguments and nameless application.

We denote by

$$[\![\ldots, f(a_1, \ldots, a_k) \mapsto [\![\ldots]\!], \ldots]\!]$$

an object where attribute f is mapped to object with attributes $a_1, \ldots, a_k$ that are mapped to special void *positional attributes* $\pi_1, \ldots, \pi_k$:

$$[\![\pi_1 \mapsto \varnothing, \ldots, \pi_k \mapsto \varnothing, a_1 \mapsto \rho^0.\pi_1, \ldots, a_k \mapsto \rho^0.\pi_k, \ldots]\!]$$

We denote by

$$t\ t_1\ \ldots\ t_n$$

an application using positional attributes:

$$t(\pi_1 \mapsto t_1, \ldots, \pi_n \mapsto t_n)$$

With this syntax abstract objects can be more easily identified as methods:

Example 6.2. Using extended syntax, we can rewrite Example 2.8 as follows

$$\text{Base} := ⟦\text{new}(\text{this}) \mapsto ⟦\varphi \mapsto \rho^0.\text{this},\ g \mapsto 3⟧⟧$$
$$\text{Derived} := ⟦\text{new}(\text{this}) \mapsto ⟦\varphi \mapsto \text{Base.new}(\text{this} \mapsto \rho^0.\text{this}),\ f \mapsto 2.\text{add}(n \mapsto \rho^0.g)⟧⟧$$
$$d := \text{Derived.new}(\text{this} \mapsto ⟦⟧)$$

7 Related work

Systems based on row types and row polymorphism [26] were originally introduced to model inheritance. Row types combine structural typing for records and variants with parametric polymorphism, which simplifies type inference. Rows can be extended (by adding new entries to the existing row) and concatenated (by combining several rows), which can be challenging for adoption in different typing settings and require new approaches [10]. The last one introduces Rose language, based on row types and supporting record concatenation through its monoidal nature of row extension. Rose uses qualified types to bind records to the rows and to abstract them from each other and allow them to evolve independently.

Some of these theoretical attempts define only the restricted set of language features or focus on language particular aspects to simplify the formal reasoning and proofs. In [19], Igarashi et al. have introduced Featherweight Java (FJ), a minimal core calculus for Java and generics. This model supports classes (including generics), methods, fields, inheritance, and dynamic typecasts, but omits complex features, such as interfaces, concurrency, reflection, etc. Later, Bettini et al. has extended Featherweight Java [5] with a new core calculus including interfaces, multiple inheritance, lambda-expressions and intersection types. The intention behind the FJ&λ tool was to add the support of a Java 8 subset to a FJ. Bettini et al. used intersection types to typecast lambda-expressions and conditional expressions. In [9], Cardelli presents a semantics of multiple inheritance. In [18], Hailpern and Nguyen presented a model for object-based inheritance. In [3], Attardi and Simi introduced Semantics of Inheritance and Attributions for the Description System Omega. In [7], Bono et al. have introduced a calculus for classes and mixins, and a calculus for classes and objects.

In [21] Nierstrasz criticized the lack of formal semantics for describing the concurrent-based programming languages. To address this issue he has created the OC language which combines agents of process calculi with functions of lambda calculi. An object is presented as a function (or agent) with state and by making functional composition (or combining agents) software composition is gained.

8 Conclusion and future work

In this paper, we have introduced φ-calculus, a calculus of objects with decoration as a primary mechanism of object extension. We have shown that even though our variant of φ-calculus is not based on λ-calculus, it possess the important properties, such as confluence (Church-Rosser property) and completeness of normal order evaluation.

Then we introduced an abstract machine for call-by-name evaluation of φ-terms. This machine can serve as reasoning tool for compilers and interpreters of φ-calculus and languages based on it, such as EO programming language.

We have also provided a sound translation from φ-calculus to λ-calculus with records. This translation emphasizes the differences between decoration and object extension using **with**-expression. Finally, we discussed some syntactic extensions to the calculus, closing the gap between our presentation and that of Bugayenko [8].

We expect two main departures for future work. First, we could add type system for the calculus, probably based on row types to facilitate type inference. Second, we could extend the calculus with the ability to decorate or compose multiple objects, enabling simpler models for languages multiple inheritance.

Acknowledgments

This research has been generously funded by Huawei in the framework of Polystat project. We thank Yegor Bugayenko for taking his time to explain ideas behind EO, his version of φ-calculus, and especially his vision regarding decorator and parent object locators. We thank Bertrand Meyer for giving his feedback on the early version of the calculus and suggesting terminology for "void" and "attached" attributes. We also thank Nickolay Shilov and Larisa Safina for their feedback on the paper and different versions of the calculus. Finally, we thank Georgii Gelvanovskii for proofreading the paper.

References

[1] M. Abadi and L. Cardelli. 1994. A semantics of object types. In *Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science*. IEEE Comput. Soc. Press, Paris, France, 332–341. https://doi.org/10.1109/LICS.1994.316056

[2] Martín Abadi and Luca Cardelli. 1996. A Theory of Primitive Objects: Untyped and First-Order Systems. *Information and Computation* 125, 2 (1996), 78–102. https://doi.org/10.1006/inco.1996.0024

[3] Giuseppe Attardi, Artificial Intei I Igence Iaboratory, and Maria Shni. 1982. Semantics of Inheritance and Attributions in the Description System Omega.

[4] Henk (Hendrik) Barendregt and E. Barendsen. 1984. Introduction to lambda calculus. *Nieuw archief voor wisenkunde* 4 (01 1984), 337–372.

[5] Lorenzo Bettini, Viviana Bono, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri. 2018. Java & Lambda: a Featherweight Story. *Logical Methods in Computer Science ; Volume 14* (2018), Issue 3 ; 18605974. https://doi.org/10.23638/LMCS-14(3:17)2018 arXiv: 1801.05052.

[6] Andrew P. Black and Jens Palsberg. 1994. Foundations of Object-Oriented Languages - Workshop Report. *ACM SIGPLAN Notices* 29, 3 (1994), 3–11. https://doi.org/10.1145/181587.181588

[7] Viviana Bono and Amit Patel. 1970. A Core Calculus of Classes and Mixins. https://doi.org/10.1007/3-540-48743-3_3

[8] Yegor Bugayenko. [n.d.]. *EOLANG and φ-calculus*. https://www.eolang.org/eolang-paper.pdf

[9] Luca Cardelli. [n.d.]. A Semantics of Multiple Inheritance. ([n. d.]), 25.

[10] Adam Chlipala. 2010. Ur: statically-typed metaprogramming with type-level record computation. In *PLDI '10*.

[11] Alberto Ciaffaglione, Pietro Di Gianantonio, Furio Honsell, and Luigi Liquori. 2021. A prototype-based approach to object evolution. *The Journal of Object Technology* 20 (01 2021), 4:1. https://doi.org/10.5381/jot.2021.20.2.a4

[12] David G. Clarke, James Noble, and John M. Potter. 2001. Simple Ownership Types for Object Containment. In *ECOOP 2001 — Object-Oriented Programming*, Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Jørgen Lindskov Knudsen (Eds.). Vol. 2072. Springer Berlin Heidelberg, Berlin, Heidelberg, 53–76. https://doi.org/10.1007/3-540-45337-7_4 Series Title: Lecture Notes in Computer Science.

[13] N.G de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381–392. https://doi.org/10.1016/1385-7258(72)90034-0

[14] E. Durr and J. van Katwijk. 1992. VDM++, a formal specification language for object-oriented designs. In *CompEuro 1992 Proceedings Computer Systems and Software Engineering*. IEEE Comput. Soc. Press, The Hague, Netherlands, 214–219. https://doi.org/10.1109/CMPEUR.1992.218511

[15] Kathleen Fisher, Furio Honsell, and John C. Mitchell. 1993. A lambda calculus of objects and method specialization. *[1993] Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science* (1993), 26–38.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA.

[17] Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and Types*. Cambridge University Press, USA.

[18] B. Hailpern and Van Nguyen. 1987. A Model for Object-Based Inheritance. *undefined* (1987). https://www.semanticscholar.org/paper/A-Model-for-Object-Based-Inheritance-Hailpern-Nguyen/6a32a53fa7184c1368b9ec36d5f749fd9d1ec961

[19] Atsushi Igarashi. 2002. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems* 23, 3 (2002), 55.

[20] J. L. Krivine. 1993. *Lambda-Calculus, Types and Models*. Ellis Horwood, USA.

[21] Markus Lumpe, Franz Achermann, and Oscar Nierstrasz. 2000. A Formal Language for Composition. (07 2000).

[22] Semantics Of Object-Oriented, Mario Wolczko, and Mario I. Wolczko. [n.d.]. Mario Wolczko Semantics of Object-Oriented Languages.

[23] Benjamin C. Pierce and David N. Turner. 1993. Simple Type-Theoretic Foundations for Object-Oriented Programming.

[24] Graeme Smith. 2000. Concrete Syntax. In *The Object-Z Specification Language*, Graeme Smith (Ed.). Springer US, Boston, MA, 133–142. https://doi.org/10.1007/978-1-4615-5265-9_6

[25] M. Takahashi. 1995. Parallel Reductions in λ-Calculus. *Information and Computation* 118, 1 (1995), 120–127. https://doi.org/10.1006/inco.1995.1057

[26] Mitchell Wand. 1987. Complete Type Inference for Simple Objects. In *LICS*.

[27] Mitchell Wand. 1991. Type inference for record concatenation and multiple inheritance. *Information and Computation* 93, 1 (1991), 1–15. https://doi.org/10.1016/0890-5401(91)90050-C Selections from 1989 IEEE Symposium on Logic in Computer Science.