A Class-Based Object Calculus of Dynamic Binding: Reduction and Properties

Paweł T. Wojciechowski

Poznań University of Technology
60-965 Poznań, Poland
`ptw@cs.put.poznan.pl`

Abstract. To be able to compose and decompose software components at run time, some form of *dynamic rebinding* between components (or objects) is needed. In this paper, we identify basic properties of dynamic object (re)binding, and propose a class-based object calculus that gives precise meaning to these properties. We also define two example semantic properties that are characteristic for many concurrent programs with low-level bind/unbind operations. Our calculus has a built-in construct `atomic` that can be used to implement one of the semantic properties.

Key-words: lambda and object calculi, dynamic binding, atomicity

1 Introduction

What do we mean by *dynamic object rebinding*? Consider a construct `bind` X a that binds a name X to an object a. The effect of binding name X to a is that we can refer to a via name X, e.g. a method m of object a can be invoked either via $a.m$ or $X.m$. The crucial point here is that the object a can be later unbound from X (using a construct `unbind` X) and another object b can be rebound to X at runtime. By the alias change, any concurrent object c that knows name X, has been therefore unbound from a and bound to b.

We must ensure that types of objects a and b that are dynamically bound to X, match the corresponding field accesses and method calls via name X. For this, X is not a pure name but it is a *signature* that declares types of fields and methods of objects that are bindable to X. Objects are defined by classes, which define fields and methods with their types. Checking the match between signatures and classes is mostly standard; for clarity, we leave therefore our calculus untyped, focusing on the operational semantics. Note that an object c invoking a method $X.m$ may not even know the object on which method m is invoked. This simple mechanism can be used to implement software components (or objects with a predefined interface) that can be *composed dynamically*.

In our previous work, we developed SAMOA [RWS06a,WRS04] – a software framework for implementing network protocols from reusable components, that provide services (a *service* corresponds to signatures presented in this paper, extended with requirement declarations). The programmers can easily encode dynamic replacement of components, using high-level abstractions that are built on top of the dynamic binding feature described in this paper. A software framework, such as SAMOA, can be used for implementing dynamically composable systems. For instance, we have used our framework to design and implement an *Adaptive Group Communication (AGC)* middleware [RWS06b], in which network protocols can be replaced on-the-fly. For this, we have designed various algorithms for *Dynamic Protocol Update (DPU)*, i.e. a synchronous replacement of protocols in a distributed system [WR05].

In [RWS06a], we described the high-level architecture of a software framework for building dynamically composable systems, such as ours. In this paper, we take a more fundamental view, and investigate a small set of low-level language constructs that can be used to reason formally about dynamic object rebinding. In particular, we have used our language to give precise meaning to basic properties of dynamic object rebinding. We also define two example semantic properties that are characteristic for many concurrent programs with low-level bind/unbind operations. Our calculus has a built-in construct `atomic` that can be used to implement one of the semantic properties.

The paper is organized as follows. Section 2 introduces basic notions and defines the syntax of our calculus. Section 3 presents a set of language properties of dynamic object rebinding, and example semantic properties of programs that use the dynamic rebinding feature. To illustrate one property, Section 4 shows an example erroneous program and its fix-up. Section 5 formalizes the operational semantics of our language, thus giving precise meaning to the properties defined earlier. Section 6 presents related work. Finally, we conclude and discuss future work in Section 7.

2 The Class-Based Object Calculus

We define our language as the call-by-value λ-calculus, extended with signatures, objects, object binding/unbinding, exceptions, threads and atomic tasks. The abstract syntax of the language is in Figure 1. The main syntactic categories are signatures, classes, values and expressions. For convenience, we differentiate names: X, Y range over signature names; A, B range over class names; f ranges over object field names, and m ranges over method names. We write $\overline{x}$ as shorthand for a possibly empty sequence of variables $x_1, ..., x_n$ (and similarly for $\overline{t}$, $\overline{v}$, and $\overline{e}$). We abbreviate operations on pairs of sequences in the obvious way, writing e.g. $\overline{x} : \overline{t}$ as shorthand for $x_1 : t_1, ..., x_n : t_n$ (and similarly for $\overline{f = v}$). Sequences of parameter names in functions and class methods are assumed to contain no duplicate names. We write $\overline{M}$ as shorthand for a (non-empty) sequence of methods $M_1, \ldots, M_n$ in a class. Methods of the same class must contain no duplicate names; similarly, field names are unique per class.

Types Types include the base type `Unit` of unit expressions, which abstracts away from concrete ground types for basic constants (integers, Booleans, etc.), the type `Sig` of object signatures, the type `Obj` of objects, and the type $t \to t'$ of functions and class methods.

Variables	$x, y, a, b \in Var$	
Signature names	$X, Y \in Sig$	
Class names	$A, B \in Lab$	
Field names	f	
Method names	m	
Interface names	$n \in Sel$	$::= f \mid m$
Types	t	$::= \texttt{Unit} \mid \texttt{Sig} \mid \texttt{Obj} \mid \bar{t} \to t'$
Signatures	s	$::= \texttt{sig}\ X\ \{f_1 : t_1, \ldots, f_k : t_k,$ $m_1 : \bar{t}_1 \to t'_1, \ldots, m_n : \bar{t}_n \to t'_n\}$
Fun. abstractions	F	$::= \overline{x} : \bar{t} = \{e\}$
Methods	M	$::= t\ m\ F$
Classes	$C \in Class$	$::= \texttt{class}\ A\ \{f_1 = v_1, \ldots, f_k = v_k, M_1, \ldots, M_n\}$
Values	$v, w \in Val$	$::= () \mid X \mid \texttt{new}\ A \mid F$
Expressions	$e \in Exp$	$::= x \mid v \mid e.n \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid e := e$ $\mid \texttt{bind}\ e\ e \mid \texttt{unbind}\ e \mid \texttt{try}\ e\ \texttt{catch}\ e \mid \texttt{escape}$ $\mid \texttt{fork}\ e \mid \texttt{atomic}\ e$

We work up to alpha-conversion of expressions throughout, with $\overline{x}$ binding in e in an expression $\overline{x} : \bar{t} = \{e\}$, and x in e' in an expression $\texttt{let}\ x = e\ \texttt{in}\ e'$. Names do not bind, and so are not subject to alpha-conversion.

Fig. 1. A concurrent language of dynamic object (re)binding

Signatures A *signature* describes an *object interface*, i.e. a declaration of object fields and methods that can be accessed or called upon an object via the signature. Syntactically, a signature is a keyword `sig`, followed by the name of the signature, and a sequence of field and method names, accompanied with their types.

Methods A *method* of the form $t\ m\ F$ has declarations of a type t of the values that it returns, its name m, and its body F. Access control is not modelled (all fields and methods are public). Objects can refer to their own methods with *self.m*, where *self* is a variable. A method's body is a function abstraction of the form $\overline{x} : \bar{t} = \{e\}$ (we adopted the C++ or Java notation, instead of the usual $\lambda\overline{x} : \bar{t}.e$ from the λ-calculus).

Classes A *class* has declarations of its name (e.g. `class` A) and the class body $\{\overline{f} = \overline{v}, \overline{M}\}$, where $\overline{f} = \overline{v}$ is a sequence of fields (data containers) accessible via names $\overline{f}$ and instantiated to values $\overline{v}$, and $\overline{M}$ is a sequence of object methods. Classes do not explicitly declare their superclass with `extends` since we do not model class inheritance. Class inheritance and object constructor methods can be easily added to the calculus definition, in the style of Featherweight Java (FJ) [IPW99]. We assume that every class implicitly extends a special class `Object`, like in FJ. The class `Object` does not define any fields nor methods.

Values A *value* is either an empty value () of type `Unit`, a signature name, e.g. X, an object instance, e.g. `new` A, or function abstraction $\overline{x} : \overline{t} = \{e\}$. Values are first-class, they can be passed as arguments to functions and methods, and returned as results or extruded outside objects. (Typing could be used to forbid extruding functions that contain object *self* references).

Basic expressions Basic expressions e are mostly standard and include variables, values, field/method selectors, function/method applications, `let` binders, and field assignment $e := e$. The `let`-binder is a construct of ML-like languages, that can be used to define functions, and to bind object and immutable data to variables. For instance, `let` $x =$ `new` A `in` e creates a new object of class A that is bound to a variable x (where x binds in e). Then, we can write e.g. $x.f := v$ to overwrite a field f of object x with a value v, or we can write e.g. $x.m\ v$ to call a method m of object x. We use syntactic sugar $e_1; e_2$ (sequential execution) for `let` $x = e_1$ `in` e_2 (for some x, where x is fresh).

Dynamic binders and exceptions Execution of `bind` $X\ a$ binds a signature X to an object a; any previous binding of signature X disappears. Execution of `unbind` X unbinds a signature X from any object bound to X, or raises an exception if no object is bound to X.

To catch exceptions, we have an expression `try` e `catch` e', which is similar to the one found in ML-like languages. If there was an exception thrown in e then the execution of e terminates and e' commences. Execution of `try` e `catch` e' returns either the result of e, if no exception occurred, or the result of e', if there was an exception thrown in e and no exception in e'. Exceptions can be thrown explicitly using `escape`, or implicitly (as in `unbind`). If there is no expression to catch an exception, the execution of `escape` blocks its thread of execution.

Threads and atomic tasks The language allows multithreaded programs by including an expression `fork` e, which spawns a new thread for the evaluation of expression e. This evaluation is performed only for its effect; the result of e is never used.

Execution of `atomic` e creates a new concurrent thread to evaluate an expression e *atomically*; we call such expressions *tasks*. Concurrent execution of atomic tasks can be interleaved but the following property holds.

Property 1 (Isolation Property). Consider all atomic tasks in a program P, and a set N of all signatures that the tasks may refer to. A non-terminating execution of P satisfies the *isolation property*, if given any signature name X in N, the order of accessing fields or calling methods via X by the atomic tasks is the same as in an ideal execution of P in which the tasks would be executed sequentially.

An atomic task in our language can itself be multithreaded since its execution can spawn new threads using `fork`. The operational semantics of tasks and the `atomic` construct ensuring isolation will be given in Section 5.

In our previous work [Woj05], we have presented an example implementation of tasks, but for a different, more restrictive definition of isolation that considers modifications of data stores. The implementation is based on static typing and runtime *versioning*. In [WRS04], we have proposed several optimizations of the concurrency control algorithm implementing versioning.

Programs A *program* is a pair $(\textsc{ct}, e)$ of a class table $\textsc{ct}$ and a main expression e, where the class table $\textsc{ct}$ is a mapping from class names to class declarations. To lighten the notation, we always assume a *fixed* class table $\textsc{ct}$. To avoid uncaught exceptions we syntactically restrict the program's main expression e to have the form `try` e' `catch` v, where v is a value. We assume that a class table satisfies some sanity conditions: (1) $\textsc{ct}(A) = \texttt{class}\ A \ldots$; (2) $\texttt{Object} \notin \textsc{ct}$; and (3) for every class name A (except `Object`) appearing anywhere in $\textsc{ct}$, we have $A \in \mathit{dom}(\textsc{ct})$. Given these conditions, a class table can be easily identified with a sequence of class declarations.

3 Properties of Dynamic (Re)binding

Below we present basic properties of language constructs for binding/unbind objects in our calculus, together with some discussion of higher-level rebinding constructs that could be built on top of our calculus.

Then, we give two example semantic properties of *programs*, in which objects can be rebound dynamically. The untyped calculus presented in this paper does not have language support to declare and verify if such semantic properties hold. We leave this for future work.

3.1 Language Properties

Below are runtime properties of the language constructs. After each property, we provide a short justification of our design choice.

Property 2 (Binding Uniqueness). At run time, a signature X has two possible states: it either binds to some object or not.

This is due to the fact that we decided to have *two* language constructs: `bind` X v that binds a signature X to an object v, and `unbind` X that unbinds the signature. Our intention was to model these two operations. At the higher-level of abstraction, however, the programmers may want to have a single construct that e.g. replaces software components in one atomic step.

Property 3 (Binding Restriction). At most one object can be bound to a signature X at a time.

If more than one object could be bound to a signature X, then a method call $X.m$ would not know which object to call; similarly, a field access $X.f$ would not know which object to select. (In our language, the same field or method names can appear in different classes.) At the higher-level of abstraction, however, overwriting bindings of X could be encoded; the higher-level `unbind` construct could then remove the current binding and deactivate any previous binding if it exists.

Property 4 (Object Aliasing). An object can be bound to many signatures.

We allow this for expressiveness at the operational semantics. Note that $X.m$ and $Y.m$ mean something different in programs with atomic tasks, event if X and Y may bind the same object; to understand why, see the definition of the isolation property. We think that object aliasing could be useful for programmers. If any restriction is required, then it should be declared by programmers, and enforced via a type system.

Property 5 (Failures). If no object is bound to X, then `unbind` X fails, field access $X.f$ fails for any f, and method call $X.m$ fails for any m.

The above property with an exception mechanism built into the calculus allows for more expressiveness. We can express alternative actions on failure at the higher level of abstraction, e.g. "wait till some object is bound".

Property 6 (Concurrency). The operations of binding/unbinding a signature X, and the object field accesses or method calls via X can be concurrent.

Dynamic *re*-binding of objects in a sequential program seems to be a rarely needed feature (e.g. dynamic class loading usually occurs only on object construction). On the other hand, new emerging applications that depend on dynamic object rebinding, such as dynamic protocol updating and *adaptive systems* are often concurrent. Concurrency in these applications stems from various reasons: the old and updated protocol components may need to coexist for some time [WR05], the protocol components are themselves concurrent with the *protocol updater* [RWS06a] that dynamically rebinds the components, etc.

3.2 Semantic Properties

Below are two example properties that may be required by programs with object rebinding.

Property 7 (Reference Consistency). A set of object references $R = \{X_i.n_j : i = 1..k, j = 1..l\}$ is *consistent* in an expression e, if exists object a such that any method call or field access $X_i.n_j$ in R, as part of evaluation of e, refers to a.

In Section 4, we present an example program that requires this property. In the program, e.g. if a method call `X.put` has been executed upon some object, then another reference to `X` (a field access `X.getn`) in the same round of the protocol should also be executed upon the same object.

Property 8 (Signature Linearity). A signature X is *linear* in a program, if it is either unbound, or it binds the same object v during whole program execution; object v that was bound to X cannot be rebound to other signature.

If a linear signature X has been bound to some object, then it cannot be rebound to another object, and vice versa. This property could be useful in programs in which dynamic object rebinding is not a feature to mask implementations of a given signature, but to authenticate an object via a signature. If objects are communicated between machines (as part of some protocol), it may be useful to use for this an abstract signature of an object, rather than its concrete name.

4 Example of the Reference Consistency Requirement

In this section, we give a small example program to explain the need for the Reference Consistency (Property 7 in Section 3), and the use of the `atomic` construct (with the isolation property) to ensure reference consistency. The program implements a simple protocol involving the exchange of messages between a client and an anonymous server, accessible via a signature `X`.

The protocol uses *public key cryptography*, which can be explained as follows. The client encrypts a message m using server's public key to produce an encrypted message; only the server can decrypt this message, so this ensures secrecy. The server can sign a message m by encrypting it with its secret key (which is the inverse of the public key); any client in possession of server's public key can then decrypt this message. Public key cryptography is used, e.g. in an *authentication protocol* [Low96]).

A client obtains server's public key from a trusted key store `keyStore`, using a method `keyStore.publicKey`; the method accepts as its argument the server's name `X.getn` (see in the end of the program). The key store (omitted here) returns a public key that corresponds to this name. To send a message (a value 100) encrypted using the public key, the client invokes server's method `X.put`. Execution of `X.put` (see class `A` or class `B`) decrypts the message using server's secret key, which is stored in the object field `secretKey`.

```
sig X
{
  getn : Obj
  put : Int -> Int
}
class A
{
  getn = self       (* an object name *)
  secretKey = 1     (* a secret key of A *)
  Int put (v : Int) = { decrypt (v, self.secretKey) }
}
class B
{
  getn = self       (* an object name *)
  secretKey = 2     (* a secret key of B *)
  Int put (v : Int) = { decrypt (v, self.secretKey) }
}

class Updater
{
  Unit update (x : Sig, o : Obj) =
  {
    unbind x;   (* unbind signature x from any object *)
    bind x o;   (* bind signature x to object o *)
  }
}
```

```
let a = new A in    (* create object a *)
bind X a;           (* and binds sig X to a *)
let b = new B in    (* create object b *)
fork (new Updater).update(X, b);  (* rebind X to b *)
try
  X.put (encrypt(100, keyStore.publicKey(X.getn)))  (* The client *)
catch
  0
```

Exchange of an encrypted message between server X and the client occurs in parallel with *dynamic replacement* of the actual object implementing X. For this, we have an *updater* object `Updater`, with a single method `update` that implements a simple handover protocol: it takes as arguments a signature and an object, unbinds anything bound to the signature and binds the object. (For simplicity, we require that X is initially bound.)

In the main expression, a concurrent thread (created with `fork`) calls a method `update` that unbinds a server object a (bound to X) and binds server object b to X. The client does not know if it calls a or b; it is not aware of the hot-swapping done by the updater. The program is however problematic in twofold ways. Firstly, the client may call a server using a signature X that has been unbound by the `update` method and not rebound yet, thus leading to an exception error. Secondly, the following property is not true:

Property 9 (Safety). A message encrypted with a public key of object x is also received by x (for any x).

We would like this property to hold during program execution. Otherwise, the client may encrypt and send a message to the server using a public key of another server, which is like an attack on a protocol using public key cryptography.

To fix up our program, we can use the `atomic` construct to encode the message exchange protocol (initiated by the client) and the update protocol (in the `update` method) as two parallel atomic tasks. Below is an example code:

```
class Updater
{
  Unit update (x : Sig, o : Obj) =
  {
    atomic
      (unbind x;   (* unbind signature x from any object *)
       bind x o;)  (* and bind signature x to object o atomically *)
  }
}
let a = new A in    (* create object a *)
bind X a;           (* and binds sig X to a *)
let b = new B in    (* create object b *)
fork (new Updater).update(X, b);  (* rebind X to b *)
try
  atomic X.put (encrypt(100, keyStore.publicKey(X.getn))) (* The client *)
catch
  0
```

The advantage of `atomic` with respect to coarse-grain locking is that the client-server protocol and server updating can be executed concurrently. Moreover possible deadlocks are avoided, which simplifies programming. However, isolation ensured by `atomic` is actually a stronger property than reference consistency – atomic tasks that do not do object rebinding may also be mutually isolated, even if they cannot themselves invalidate reference consistency.

The use of `atomic` in protocols depends on its implementation. Protocols have various side effects (I/O actions, network communication, etc.); these side-effects are not always revocable. The implementations of `atomic` (we give examples in Section 6) usually restrict I/O actions in atomic blocks, e.g. due to rollback support. This restriction should not be a problem if `atomic` is used to protect only short code fragments, as in our example program. Alternatively, we proposed in [Woj05] an implementation of `atomic` that does not depend on rollback-recovery of tasks. (We do not have an explicit rollback construct in our language.)

5 Operational Semantics

We specify the operational semantics of our language using the abstract machine defined in Figures 2 and 3. The machine evaluates a program by stepping through a sequence of states. A state S consists of four components: an object store Δ, a counter α of fresh atomic blocks, a bind store β, and execution threads T, organized as a sequence $T_0, ..., T_n$.

The *object store* Δ is a finite map from object field selectors to values stored in the fields, where a *field selector*, denoted $o_A.f$, is an object location o_A indexed by a field name f.

The *bind store* β is a set of pairs (X, o^A) of a signature name X and an object location o^A bound to the signature. The set difference $\beta \setminus \beta'$ is the set of elements found in β but not found in β'; the union of sets $\beta \cup \beta'$ is the set consisting of the elements of both sets, with no duplicate elements.

The expressions g in a sequence of threads T are written in the calculus presented in Section 2, extended with a new construct $\texttt{task}\ i\ N\ T$. The construct is not part of the language to be used by programmers; its meaning will be explained below.

We define a small-step evaluation relation $\Delta, \alpha, \beta \mid g \longrightarrow \Delta', \alpha', \beta' \mid g'$, read "expression g reduces to expression g' in one step, with Δ, α, β being transformed to Δ', α', β'". We also use $\longrightarrow^*$ for a sequence of small-step reductions. By *concurrent execution*, we mean a sequence of small-step reductions in which the reduction steps can be taken by different threads with possible interleaving.

Reductions are defined using evaluation context $\mathcal{E}$ for expressions e and g. The evaluation context ensures that the left-outermost reduction is the only applicable reduction for each individual thread in the entire program. Context application is denoted by $[]$, as in $\mathcal{E}[\,e\,]$. Structural congruence rules allow us to simplify reduction rules by removing the context whenever possible.

Evaluation of a program $(\textsc{ct}, e)$, where $\textsc{ct}$ is constant, starts in an initial state with empty stores $\emptyset$, a null counter 0, and with a single thread that evaluates

State Space:

$$
\begin{array}{rcl}
S \in \mathit{State} & = & \mathit{ObjStore} \times \mathit{TaskId} \times \mathit{BindStore} \times \mathit{ThreadSeq} \\
\Delta \in \mathit{ObjStore} & = & \mathit{ObjLoc.Sel} \rightarrow \mathit{Val} \\
\alpha \in \mathit{TaskId} & = & \mathbf{Nat} \\
\beta \in \mathit{BindStore} & = & \mathit{Sig} \times \mathit{ObjLoc} \\
o^A \in \mathit{ObjLoc} & \subset & \mathit{Var} \\
T \in \mathit{ThreadSeq} & ::= & g \mid T, T \\
g \in \mathit{Exp}_{ext} & ::= & x \mid v \mid e.n \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid e := e \mid \texttt{bind}\ e\ e \mid \texttt{unbind}\ e \\
& & \mid \texttt{try}\ e\ \texttt{catch}\ e \mid \texttt{escape} \mid \texttt{fork}\ e \mid \texttt{atomic}\ e \mid \texttt{task}\ i\ N\ T
\end{array}
$$

Evaluation Contexts:

$$
\begin{array}{l}
\mathcal{E} = [\,] \mid \mathcal{E}.n \mid \mathcal{E}\ e \mid v\ \mathcal{E} \mid \texttt{let}\ x = \mathcal{E}\ \texttt{in}\ e \mid \mathcal{E} := e \mid o^A.f := \mathcal{E} \mid \texttt{bind}\ \mathcal{E}\ e \mid \texttt{bind}\ X\ \mathcal{E} \\
\quad \mid \texttt{try}\ \mathcal{E}\ \texttt{catch}\ e \mid \texttt{task}\ i\ N\ \mathcal{E} \mid \mathcal{E}, T \mid T, \mathcal{E}
\end{array}
$$

Structural Congruence

$$T, T' \equiv T', T \qquad T, () \equiv T$$

$$\frac{\Delta, \alpha, \beta \mid g \longrightarrow \Delta', \alpha', \beta' \mid g'}{\Delta, \alpha, \beta \mid \mathcal{E}[\,g\,] \longrightarrow \Delta', \alpha', \beta' \mid \mathcal{E}[\,g'\,]} \qquad \frac{g \longrightarrow g'}{\Delta, \alpha, \beta \mid g \longrightarrow \Delta, \alpha, \beta \mid g'}$$

Transition Relation

$$\mathit{eval} \subseteq ((\mathit{Lab} \rightarrow \mathit{Class}) \times \mathit{Exp}) \times \mathit{Val}$$

$$\mathit{eval}((\textsc{ct}, e), v_0) \Leftrightarrow \emptyset, 0, \emptyset \mid e \longrightarrow^* \Delta, \alpha, \beta \mid v_0, (), \cdots, ()$$

Method Body Lookup:

$$\frac{\begin{array}{c}\textsc{ct}(A) = \texttt{class}\ A\ \{\overline{f} = \overline{v}, \overline{M}\} \\ t\ m\ F \in \overline{M}\end{array}}{\mathit{mbody}(m, A) = F}$$

Fig. 2. Reduction semantics - Part I

the expression e. Evaluation then takes place according to the machine's rules in Figure 3. The evaluation terminates once all threads have been reduced to values, in which case the value v_0 of the initial, first thread T_0 is returned as the program's result. Subscripts in values reduced from threads denote the sequence number of the thread, i.e. v_i is reduced from i's thread, denoted T_i $(i = 0, 1, ..)$. The execution of threads can be arbitrarily interleaved.

5.1 Reduction rules

Below we describe reduction rules in Figure 3. The first two evaluation rules are the standard rules of a call-by-value λ-calculus [Plo75]. We write $e\{\overline{v}/\overline{x}\}$ to denote the capture-free substitution of v_i for x_i in the expression e $(i = 1, .., n)$. Function application $\overline{x}\ :\ \overline{t} = \{e\}\ \overline{v}$ in (R-App) reduces to the function's body e in

$$\overline{x} : \bar{t} = \{e\}\ \overline{v} \longrightarrow e\{\overline{v}/\overline{x}\} \qquad \text{(R-App)}$$

$$\texttt{let}\ x = v\ \texttt{in}\ e \longrightarrow e\{v/x\} \qquad \text{(R-Let)}$$

$$\frac{o^A \notin dom(\Delta) \quad \textsc{ct}(A) = \texttt{class}\ A\ \{f_1 = v_1, ..., f_k = v_k, \overline{M}\} \quad \Delta' = (\Delta, o^A.f_1 \mapsto v_1, ..., o^A.f_k \mapsto v_k)}{\Delta, \alpha, \beta \mid \texttt{new}\ A \longrightarrow \Delta', \alpha, \beta \mid o^A} \qquad \text{(R-New)}$$

$$\Delta, \alpha, \beta \mid o^A.f := v \longrightarrow \Delta[o^A.f \mapsto v], \alpha, \beta \mid () \qquad \text{(R-Assign)}$$

$$\Delta, \alpha, \beta \mid o^A.f \longrightarrow \Delta, \alpha, \beta \mid v\{o^A/self\} \quad \text{if} \quad \Delta(o^A.f) = v \qquad \text{(R-Field)}$$

$$\frac{mbody(m, A) = F}{o^A.m\ \overline{v} \longrightarrow F\{o^A/self\}\ \overline{v}} \qquad \text{(R-Invk)}$$

$$\texttt{try}\ v\ \texttt{catch}\ e \longrightarrow v \qquad \text{(R-Try)}$$

$$\frac{\texttt{try..catch} \notin \mathcal{E}'}{\texttt{try}\ \mathcal{E}'[\,\texttt{escape}\,]\ \texttt{catch}\ e \longrightarrow e} \qquad \text{(R-Esc)}$$

$$\Delta, \alpha, \beta \mid \texttt{bind}\ X\ o^A \longrightarrow \Delta, \alpha, (\beta \setminus \{(X, \cdot)\}) \cup \{(X, o^A)\} \mid () \qquad \text{(R-Bind)}$$

$$\Delta, \alpha, \beta \mid \texttt{unbind}\ X \longrightarrow \Delta, \alpha, \beta \setminus \{(X, o^A)\} \mid () \quad \text{if} \quad (X, o^A) \in \beta \qquad \text{(R-Unbind1)}$$

$$\Delta, \alpha, \beta \mid \texttt{unbind}\ X \longrightarrow \texttt{escape} \quad \text{if} \quad (X, \cdot) \notin \beta \qquad \text{(R-Unbind2)}$$

$$\Delta, \alpha, \beta \mid X.n \longrightarrow \Delta, \alpha, \beta \mid o^A.n \quad \text{if} \quad (X, o^A) \in \beta \qquad \text{(R-Lookup1)}$$

$$\Delta, \alpha, \beta \mid X.n \longrightarrow \Delta, \alpha, \beta \mid \texttt{escape} \quad \text{if} \quad (X, \cdot) \notin \beta \qquad \text{(R-Lookup2)}$$

$$\frac{N = \{X \in Sig\ :\ X \in e\}}{\Delta, \alpha, \beta \mid \mathcal{E}[\,\texttt{atomic}\ e\,] \longrightarrow \Delta, \alpha + 1, \beta \mid \mathcal{E}[\,()\,], \texttt{task}\ \alpha + 1\ N\ e} \qquad \text{(R-Atomic)}$$

$$\mathcal{E}[\,\texttt{fork}\ e\,] \longrightarrow \mathcal{E}[\,()\,], e \qquad \text{(R-Fork1)}$$

$$\texttt{task}\ i\ N\ \mathcal{E}[\,\texttt{fork}\ e\,] \longrightarrow \texttt{task}\ i\ N\ (\mathcal{E}[\,()\,], e) \qquad \text{(R-Fork2)}$$

$$\frac{\texttt{task}\ i\ N\ e \in \mathcal{E} \quad i < j \quad X \in N \cap M \quad X \notin e \quad (X, o^A) \in \beta}{\Delta, \alpha, \beta \mid \mathcal{E}[\,\texttt{task}\ j\ M\ \mathcal{E}'[\,X.n\,]\,] \longrightarrow \Delta, \alpha, \beta \mid \mathcal{E}[\,\texttt{task}\ j\ M\ \mathcal{E}'[\,o^A.n\,]\,]} \qquad \text{(R-Task1)}$$

$$\texttt{task}\ i\ N\ v \longrightarrow () \qquad \text{(R-Task2)}$$

$$v_i, v'_j \longrightarrow v_i \quad \text{if} \quad i < j \qquad \text{(R-Thread)}$$

Fig. 3. Reduction semantics - Part II

which formal arguments $\overline{x}$ are replaced with the actual arguments $\overline{v}$. Execution of `let` $x = v$ `in` e in (R-Let) reduces the whole expression to the expression e in which variable x is replaced by value v.

Execution of `new` A creates a new object of class A. The object is identified by a fresh object location o_A, and represented by a new record of object fields $f_1, ..., f_k$ in the object store Δ; see the (R-New) rule. The notation $(\Delta, \overline{o^A.f} \mapsto \overline{v})$ means "the store that maps $\overline{o^A.f}$ to $\overline{v}$ and maps all other selectors to the same thing as Δ". The object fields $f_1, .., f_k$ are accessible via the object location o_A, e.g. $o_A.f_i$ $(i = 1..k)$ refers to a field f_i of object o_A. The object fields in the object record are initialized with field values $v_1, .., v_k$ defined by class A.

Rules (R-Assign) and (R-Field) correspondingly, assign a new value v to the field f of an object o^A, and read the current value stored in an object field $o^A.f$. For instance, let us look at the rule (R-Assign). We use the notation $\Delta[o^A.f \mapsto v]$ to denote update of map Δ at $o^A.f$ to v. Note that the term resulting from this evaluation step is just (); the interesting result is the updated store. The (R-Assign) rule must be applied first, if not possible then we try (R-Field).

Similarly to FJ, the invocation $o^A.m\ \overline{v}$ of a method m of an object o^A applies the beta-reduction rule from the call-by-value λ-calculus; see the (R-Invk) rule. The rule first looks up in the class table CT a method body F of the form $\overline{x} : \bar{t} = \{e\}$ (using a function $mbody(m, A)$ defined in the bottom of Figure 2); then, it reduces to the method body in which *self* is replaced by the receiver o^A. Then, the application rule (R-App) (described earlier) can be used, which applies the arguments $\overline{v}$ to the method m.

Exceptions are defined using two rules. The (R-Try) rule defines the case when no exception was thrown; it simply reduces the whole expression `try` ... `catch` with the body reduced to a value v to the value v; the `catch` clause is discarded. To throw an exception, the `escape` construct is used. If `escape` is in the redex position of the expression e' in the body of the innermost `try` e' `catch` e, the (R-Esc) rule reduces `try` e' `catch` e to the exception handler e.

Dynamic binder `bind` $X\ o^A$ in rule (R-Bind) removes from store β any previous binding $(X, \cdot)$ of a signature X, and extends β with a new element of X paired with an object location o^A. The whole expression reduces to the empty value (). Dynamic unbinder `unbind` X in rules (R-Unbind1) and (R-Unbind2) respectively, removes the binding $(X, \cdot)$ from store β and reduces to the empty value (), or throwns an exception with `escape` if no binding of X exists.

Dynamic resolver $X.n$ in rules (R-Lookup1) and (R-Lookup2) respectively, returns the field/method selector $o^A.n$, where o^A is the object location currently bound to a signature X, or throwns an exception if no binding of X exists.

5.2 Concurrent and atomic evaluations

Execution of an expression `atomic` e creates a new thread for evaluation of a *task* e with the isolation property, defined in Section 2. The task has the syntactic form `task` $i\ N\ e$, where i is the sequence number of the task, and N is a set of all signatures X that *may* be referred to by expression e. The (R-Atomic) rule reduces an expression $\mathcal{E}[\,\texttt{atomic}\ e\,]$ to the context $\mathcal{E}$ with the empty value () in the redex position, and a new thread evaluating a task $\texttt{task}\ \alpha+1\ N\ e$; the rule also increments the task counter α.

Execution of an expression $\texttt{fork}\ e$ in (R-Fork1) creates a new thread which evaluates e; the result of evaluating expression e will be discarded by rule (R-Thread); threads may however have side-effects, e.g. modification of object fields. Tasks can spawn their own threads using `fork`; see rule (R-Fork2).

The (R-Task1) rule specifies evaluation of concurrent tasks that satisfies the isolation property. Consider evaluation of some task $\texttt{task}\ j\ M\ e'$ in the context $\mathcal{E}$, where the redex position of expression e' is a field or method access via a signature X, i.e. $e' = \mathcal{E}'[\,X.n\,]$ for some context $\mathcal{E}'$ and an interface name n. If context $\mathcal{E}$ is such that there is some *older* concurrent task $\texttt{task}\ i\ N\ e$ (i.e. $i < j$) that evaluates some expression e and may refer to X (since X is declared in set N), then the rule (R-Task1) applies. It reduces the expression $\texttt{task}\ j\ M\ e'$ by replacing X by a concrete object location o^A if two conditions hold: (1) e cannot refer to X anymore (i.e. $X \notin e$), and (2) there is actually some binding of X in bind store β. If X is in e then the rule does not apply, and the other task may be evaluated. If no binding of X exists, the rule (R-Lookup2) applies.

Once evaluation of an expression e of task $\texttt{task}\ i\ N\ e$ yields a value, the rule (R-Task2) returns the empty value as the result of the whole thread. The results of evaluating threads (except of the initial thread) are discarded by (R-Thread).

6 Related Work

Object calculi There have been many proposals of various object calculi; we sketch some of the most known examples below.

Abadi and Cardelli [AC95] have developed an imperative calculus of objects, equipped with an operational semantics and typing (and subtyping); with addition of polymorphism, the calculus can express classes and inheritance. The object calculus of Gordon and Hankin [GH98] extends Abadi and Cardelli's imperative object calculus with operators for concurrency from the π-calculus and operators for synchronization based on mutexes. Our calculus also has a synchronization abstraction built-in (the `atomic` construct), albeit semantically richer than mutexes; we discuss the related work on atomicity below.

Igarashi, Pierce and Wadler [IPW99] have proposed a small calculus, Featherweight Java (FJ), that provides classes, methods, fields, inheritance, and dynamic typecasts, with semantics closely following Java's. The design of our calculus has been inspired by FJ, e.g. we have the same rule for method calls, which uses the call-by-value principle of the λ-calculus. However, their calculus omits interfaces and even assignment, while we have assignment and also signatures (which are similar to Java interfaces). On the other hand, we do not model typing and class inheritance in this paper since our focus is on the reduction semantics.

The above calculi have been developed mainly to reason about the implementation of objects, object encodings, typing, class inheritance, etc. We are not aware of concurrent object calculi that would have constructs for dynamic object rebinding similar to ours. We discuss some examples of (non-object) calculi with dynamic binding in the next paragraph.

Dynamic rebinding A lot of work on dynamic rebinding appeared in the context of functional languages (see, e.g., work of Moreau [Mor98]), focusing either on *dynamic scoping*, in which variable occurrences are resolved with respect to their dynamic environment, or *static scoping with explicit rebinding*, where variables are resolved with respect to their static environment, but additional primitives can be used to explicitly modify these environments.

Dynamic scoping exists in most modern dialects of Lisp, e.g. MIT Scheme's `fluid-let` [MIT] construct performs dynamically-scoped rebinding of local and global variables; once the construct's expression has been evaluated, the values of the variables are restored. The *quasi-static scoping* Scheme extension of Lee and Friedman [LF93] has a class of variables, which are initially unresolved. The programmer can use a rebinding primitive to specify new bindings for individual variables. The above work is different from ours; we bind whole objects to typed signatures, while the above work is on dynamic binding of variables in functional languages, with a correspondingly different semantics of rebinding.

Dynamic linking of objects in object languages such as Java, refers to resolving object components at runtime. However, once bound the code usually cannot be rebound, which is different from our approach, which aims at studying object *re*-binding. Different dynamic linking models have been described in [DLE03].

There are different applications of dynamic rebinding. For instance, Bierman *et al.* [BHS$^+$03] proposed abstraction-safe *marshalling* and *unmarshalling* (or rebinding) values between separate programs in the λ-calculus; see also the Acute programming language [LPSW03]. An extension of Smalltalk with dynamic method redefinition in the scope of *classboxes* is described in [BDW03]; the dynamic rebinding feature is used here to support *software evolution*.

We are not aware of much discussion of concurrency issues in the context of dynamic rebinding. The existing implementations are often not satisfactory, e.g. the runtime support of type-safe dynamic Java classes in [MPG$^+$00] aborts a thread if a class update is attempted while the thread is executing a method of that class. Our solution to this problem is to execute rebindable code fragments and code fragments that do rebinding, as concurrent (possibly multithreaded) atomic tasks, using the `atomic` construct. The semantics of the construct given in this paper eliminates the need to abort threads while doing an update.

Atomicity Below we sketch some work on formalizing the isolation property (also known as *atomicity* in the programming language research community), with the semantics as in transactional systems; such semantics is slightly different than the one presented in this paper. We are not aware of any formal work on using isolation (or atomicity) in the context of dynamic binding.

Vitek *et al.* [VJWH04] have recently proposed a calculi-based model of standard database transactions. They have formalized the optimistic and two-phase locking concurrency control strategies. Their approach to formalization of the isolation property is similar to ours, in the sense that both specifications refer to *order* (or scheduling) of concurrent actions.

There have recently been a lot of interest in developing language support for atomicity. For example, Flanagan and Qadeer [FQ03] presented a type system for specifying and verifying atomicity of (single threaded) methods in multithreaded Java programs. The type system is a synthesis of Lipton's theory of left and right movers (for proving properties of parallel programs) and type systems for race detection.

Harris and Fraser [HF03] have been investigating an extension of Java with (again, sequential only) atomic code blocks that implement conditional critical regions (CCRs). The programmer can guard a conditional region by an arbitrary boolean condition, with calling threads blocking until the guard is satisfied. It is also possible to terminate an execution of an atomic block and rollback, if some condition is not satisfied.

In [Woj05], we have discussed the above implementation work in more detail, including comparison with our approach to atomicity.

7 Conclusion

In this paper, we proposed a class-based object calculus with constructs for dynamic rebinding of objects to signatures; signatures describe types of object fields and methods, and can be used to call the objects. We have also discussed properties of the bind/unbind constructs.

Dynamic object binding enables developing novel applications, such as dynamic service update (as in our example). However, it also makes programming more difficult, since additional *semantic* properties may be required by programs. We have discussed an example semantic property, called reference consistency, and showed how it can be encoded using the `atomic` construct of our calculus that ensures isolation.

In the future work, we would like to develop tools for automatic verification of certain properties of dynamic binding/unbinding, based on the typed variant of the calculus presented in this paper.

Acknowledgments The author would like to thank Olivier Rütti and Sophia Drossopoulou (and other members of the SLURP group) for discussions and comments. This work was supported in part by the State Committee for Scientific Research (KBN), Poland, under KBN grant 3 T11C 073 28.

References

[AC95] Martin Abadi and Luca Cardelli. An imperative object calculus. In *Proc. TAPSOFT '95: Theory and Practice of Software Development, the 6th International Joint Conference CAAP/FASE*, LNCS 915, May 1995.

[BDW03] Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Proc. JMLC '03: the Joint Modular Languages Conference*, LNCS 2789. Springer, August 2003.

[BHS+03] Gavin Bierman, Michael Hicks, Peter Sewell, Gareth Stoyle, and Keith Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time lambda. In *Proc. ICFP '03*, August 2003.

[DLE03] Sophia Drossopoulou, Giovanni Lagorio, and Susan Eisenbach. Flexible models for dynamic linking. In *Proc. ESOP '03*, April 2003.

[FQ03] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proc. PLDI '03*, June 2003.

[GH98] Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. In *Proc. HLCL'98: the 3rd Int'l Workshop on High-Level Concurrent Languages*, Elsevier ENTCS 16(3), 1998.

[HF03] Timothy Harris and Keir Fraser. Language support for lightweight transactions. In *Proc. OOPSLA '03*, 2003.

[IPW99] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proc. OOPSLA '99*, Nov. 1999.

[LF93] Shinn-Der Lee and Daniel P. Friedman. Quasi-static scoping: Sharing variable bindings across multiple lexical scopes. In *Proc. POPL '93*, Jan 1993.

[Low96] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. TACAS '96: Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1055, March 1996.

[LPSW03] James Leifer, Gilles Peskine, Peter Sewell, and Keith Wansbrough. Global abstraction-safe marshalling with hash types. In *Proc. ICFP '03*, 2003.

[MIT] MIT. *Scheme*. http://www.swiss.ai.mit.edu/projects/scheme/.

[Mor98] Luc Moreau. A syntactic theory of dynamic binding. *Higher-Order and Symbolic Computation*, 11(3):233–279, December 1998.

[MPG+00] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic Java classes. In *Proc. ECOOP 2000*, LNCS 1850, June 2000.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[RWS06a] Olivier Rütti, Paweł T. Wojciechowski, and André Schiper. Service Interface: A new abstraction for implementing and composing protocols. In *Proc. SAC '06: the 21st ACM Symposium on Applied Computing, Track on Dependable and Adaptive Distributed Systems*, April 2006.

[RWS06b] Olivier Rütti, Paweł T. Wojciechowski, and André Schiper. Structural and algorithmic issues of dynamic protocol update. In *Proc. IPDPS '06: the 20th IEEE Int'l Parallel and Distributed Processing Symposium*, April 2006.

[VJWH04] Jan Vitek, Suresh Jagannathan, Adam Welc, and Antony L. Hosking. A semantic framework for designer transactions. In *Proc. ESOP '04*, LNCS 2986, March/April 2004.

[Woj05] Paweł T. Wojciechowski. Isolation-only transactions by typing and versioning. In *Proc. PPDP '05: the 7th ACM-SIGPLAN Int'l Symposium on Principles and Practice of Declarative Programming*, July 2005.

[WR05] Paweł T. Wojciechowski and Olivier Rütti. On correctness of dynamic protocol update. In *Proc. FMOODS '05*, LNCS 3535, June 2005.

[WRS04] Paweł T. Wojciechowski, Olivier Rütti, and André Schiper. SAMOA: A framework for a synchronisation-augmented microprotocol approach. In *Proc. IPDPS '04: the 18th IEEE Int'l Parallel and Distributed Processing Symposium*, April 2004.