UNIVERSITY OF OSLO
Department of Informatics

Dynamic Structural Operational Semantics (long version)

Research Report No. 426

Cristian Prisacariu and Olaf Owe

December 2012

Dynamic Structural Operational Semantics (long version)

Cristian Prisacariu[*] Olaf Owe[†]

December 2012

Abstract

We introduce Dynamic SOS as a framework for describing semantics of programming languages that include dynamic software upgrades. Dynamic SOS is built on top of the Modular SOS of P. Mosses, with an underlying category theory formalization. Dynamic SOS wants to bring out the essential differences between dynamic upgrade constructs and execution constructs. The important feature of Modular SOS that we exploit in our framework is the sharp separation of the program code from the additional data structures needed at run-time. We exemplify Dynamic SOS on the C-like Proteus language and the concurrent object-oriented Creol language. On the way we introduce a construction on Modular SOS useful in the setting of the concurrent objects of Creol, where the executing code is running inside the object. This "encapsulating construction" may be used in any situation where a form of encapsulation of the execution is needed.

1 Introduction

We are interested in dynamic aspects of programming languages, like dynamic software updates for imperative languages such as the C-like PROTEUS [27, 28] or dynamic class upgrades for object-oriented languages such as the concurrent Creol [17, 16]. The nature of such dynamic aspects is different from normal control flow and program execution constructs of a language. Yet the interpretation of these dynamic operations is given using the same style of structural operational

[*]Dept. of Informatics – Univ. of Oslo, P.O. Box 1080 Blindern, N-0316 Oslo, Norway. E-mail: cristi@ifi.uio.no

[†]Dept. of Informatics – Univ. of Oslo, P.O. Box 1080 Blindern, N-0316 Oslo, Norway. E-mail: olaf@ifi.uio.no

semantics (SOS) as for the other language constructs, often employing elaborate SOS definitions.

We want the differences between the dynamic constructs and the standard execution constructs to be apparent in the SOS descriptions. For this reason we develop *Dynamic SOS* (DSOS). We are taking a *modular* approach to SOS, following the work of P. Mosses [22], thus building on *Modular SOS* (MSOS). This later formalism uses notions of category theory, but we try not to depart from the established notations and terminology used in programming language semantics. Dynamic SOS aims to be notation-free, where the notions and concepts underlying our work can be described using various notational conventions for operational semantics.

With software evolution [18] becoming a central paradigm, interest has increased over the past few years in dynamic software upgrades [19, 9, 8, 6, 7, 15]. But these various approaches are different in presentation and formalization, making it difficult to compare or combine them. Each of these approaches concentrate on some particular programming language or kind of system. The work that we undertake here is to extract the essentials of the operational semantics for dynamic upgrading constructs regardless of the programming language or the kind of system paradigm. The Dynamic SOS is intended as a semantic framework for studying future (and existing) dynamic upgrade programming constructs, where any of the existing works should be easily represented; we exemplify only two of them: [27] and [17]. Since existing works mainly concentrate on type systems and type safety, and since their results can be readily done over the Dynamic SOS, we concentrate on the semantical aspect alone. Moreover, the way we propose to give semantics makes it easier to derive the typing system, once having the Dynamic SOS semantic rules.

1.1 Dynamic SOS

The dynamic software updates are a method for updating a program at runtime, during its execution, by changing various definitions the program uses, where examples include definitions of types, or of methods and classes, or values of variables. These updates are essentially different than the normal programming constructs because they are external to the program, using information that is not produced by the program, but is provided at runtime by an external entity.

At the semantic level the characteristic of the dynamic updates comes from the fact that they are working only at the level of the data structures that the program uses, changing them using update information provided externally. In consequence, the semantics of the updates should be given only in terms of these two data structures: the update data and the program data. This is in contrast with the semantics for programming constructs which are given in terms of the structure of the program code.

In consequence, a sharp separation of the program code from the additional data structures that are manipulated at runtime is needed. The *additional data structures* are considered to be everything that is not a program term (code that can be generated using the grammar of the language). Additional data includes stores of variable values, but also the various definitions of functions or of classes and methods, or message pools and thread pools. Thus, the additional data is everything that is not being executed, but is being stored for some later use in the execution; be that values or even other program terms e.g. in the thread pool, which are supposed to be executed later.

This separation can be easily made possible by the Modular SOS, and its underlying category theory. Complex features like abrupt termination and error propagation can be nicely handled by MSOS, as well as combinations of big-step and small-step, or rewriting based, semantic styles. Therefore, we are not constrained in any way by building our Dynamic SOS on MSOS. On the contrary, MSOS is only about mathematical concepts and thinking style about the semantics, and is in no way binding the language designer to a notational style. The notation can be the same (or similar) to existing ones, as soon as the concepts and style of MSOS and DSOS are understood.

Another idea that we build into DSOS is that *upgrade points* must be identified and marked accordingly in the program code. The marking should be done with special upgrade programming constructs. Here we are influenced by the work on PROTEUS [28]. But opposed to a single marker as in PROTEUS, one could use multiple markers. This would allow also for incremental upgrades. The purpose of identifying and marking such upgrade points is to ensure type safety after upgrades.

Compared to the normal flow of control and change of additional data that the execution of the program does, we view a dynamic upgrade as a *jump* to a possibly completely different data content. This, in consequence, can completely alter the execution of the program. Moreover, these jumps are strongly knit to the upgrade information, which is regarded as outside the scope of the executing program, as externally provided.

1.2 Modular Semantics for Concurrent Object-Orientation

We take a *modular* approach to giving semantics to programming languages, as advocated by P. Mosses [22, 21]. A side contribution of the present work is that we give a modular SOS semantics to concurrent object-oriented programming concepts. Object-orientation has not been treated before in the MSOS style. Concurrent ML was treated in [20]. But the concurrency model in the object-oriented setting differs from that in [20]. The concurrency model that we treat is that of the *actors model* [12, 4, 5] where each concurrent entity is standalone, thought as running on one dedicated machine or processor. Therefore, the auxiliary data structures that the standard SOS employs are now localized, for each object. We capture this localization mechanism in a general manner, yet staying in the framework of MSOS. For this we employ a construction on the category theory of MSOS, which we call the *encapsulating construction*, and show it to be in complete agreement with the other category notions of MSOS.

A main goal of the modular approach is to ensure that once the semantics has been given to one programming construct it will not be changed in the future. A new programming construct is given semantics independent of the existing constructs. This will be illustrated all throughout our development. Because of this, we can easily work within different notation conventions. Translations between these notations is possible because of the common underlying mathematical structure given by the MSOS and its category theory foundations. Nevertheless, these categorical foundations are transparent to the scientists giving semantics to programming languages. Standard notational conventions can be adopted for MSOS, but the working methodology changes to a modular way of thinking about the semantics. This notion of independence of notation is also seen in [23] which presents new notation conventions called IMSOS, intended to be more attractive to the developers of programming languages.

A theoretical motivation for our developments is the close similarity of the transition systems we obtain, with the labeled transition systems obtained by the SOS of process algebras. There is a great wealth of general results in the process algebra community on SOS rule formats [3], some of which we think can be translated to the theory we develop here. In particular, the similarity with our approach is the fact that the states/configurations of the transition systems we obtain are only program terms, whereas the rest of auxiliary notions are flowing on the transitions as labels. This is the same as in process algebras, only that we have more complex labels here. But many of the results surveyed in [3] do not make use of the structure of the labels. Such general results would mean that any programming language that is developed within the restrictions of the rule format will get the specific results for free. General results that could be investigated by following the work presented in [3] are: (i) generating algebraic semantics [2] from specific forms of the transition rules; (ii) compositional reasoning results wrt. dynamic logic [26, 11] using specific forms of transition rules in the style of [10]; or (iii) expressiveness results of the programming constructs specified within various rule formats.

Structure of the paper: We first give a short listing of some simple notions of category theory that will be used throughout the paper. Then Section 3 introduces modular SOS and exemplifies it on the PROTEUS language. In Section 4 we define the encapsulating construction and use it to give modular semantics for the concurrent object-oriented CREOL. Both languages have dynamic upgrading constructs which are given semantics in Section 5 where we develop the Dynamic SOS theory, our main contribution.

2 Preliminaries

We recall some standard technical notions that will be used throughout this paper (maybe in slight variations). We try to stay close with our notation to that in [22] for the MSOS related notions, and to that in [24] for other notions of category theory.

Definition 2.1 (category) *A* category *(which we denote by capital letters of the form $\mathbb{A}$) consists of a set of* objects *(which we denote by $|\mathbb{A}|$ with usual representatives o, o', o_i) and a set of* morphisms*, also called* arrows*, between two objects (which we denote by $Mor(\mathbb{A})$ with usual representatives α, β, possibly indexed). A morphism has a* source *object and a* target *object (also called the domain and codomain, respectively) which we denote by α^s and α^t. A category is required (i) to have* identity *morphisms for each object, satisfying an identity law for each morphism with source in that object, and (ii) that composition of any two morphisms α and β, with $\alpha^t = \beta^s$, exists (denoted $\beta \circ \alpha$, in mathematics notation, or just $\alpha\beta$, as in computer science) and is associative.*

Definition 2.2 (functors) *Consider two arbitrary categories $\mathbb{A}$ and $\mathbb{B}$. A* functor *$F : \mathbb{A} \to \mathbb{B}$ is defined as a map that takes each object of $|\mathbb{A}|$ to some object of $|\mathbb{B}|$, and takes each morphism $\alpha \in Mor(\mathbb{A})$ to some morphism $\beta \in Mor(\mathbb{B})$ s.t. $o \xrightarrow{\alpha} o'$ is associated to some $F(o) \xrightarrow{\beta} F(o')$, and the following hold:*

1. *$F(id_o) = id_{F(o)}$;*
2. *$F(\alpha\beta) = F(\alpha)F(\beta)$.*

A functor $F : \mathbb{A} \to \mathbb{A}$ is called an endofunctor *applied to $\mathbb{A}$ (or on $\mathbb{A}$).*

Definition 2.3 (natural transformations) *Consider two arbitrary categories $\mathbb{A}$ and $\mathbb{B}$ and two functors F, G from $\mathbb{A}$ to $\mathbb{B}$. A* natural transformation *$\eta : F \to G$, from the functor F to G, is defined as a function that associates to each object o of $|\mathbb{A}|$ a morphism β of $Mor(\mathbb{B})$ with $\beta^s = F(o)$ and $\beta^t = G(o)$ s.t. for any* *morphism α of $Mor(\mathbb{A})$, with $\alpha^s = o$, the following diagram commutes:*

$$\begin{tikzcd}
F(o) \arrow[r, "\eta(o)"] \arrow[d, "F(\alpha)"'] & G(o) \arrow[d, "G(\alpha)"] \\
F(o') \arrow[r, "\eta(o')"] & G(o')
\end{tikzcd}$$

3 Modular Semantics and PROTEUS

Modular SOS is defined wrt. a more general form of transition systems which in [22] are called *general transition systems*, but here we adopt the terminology from [21] of *arrow-labeled transition systems* because, essentially, the transitions are labeled with morphisms (also called arrows) from a category.

Definition 3.1 (arrow-labeled transition systems) *An* arrow-labeled transition system *(ATS) is like a classic transition system* $(\Gamma, Mor(\mathbb{A}), \rightarrow, T)$ *which has a set of configurations (or states)* Γ*, a set of transitions between these configurations which are labeled by morphisms from a category* $\mathbb{A}$ *(*$\rightarrow \subseteq \Gamma \times Mor(\mathbb{A}) \times \Gamma$*), and a designated set of final configurations* $T \subseteq \Gamma$*.*

A computation *in an ATS is a sequence of transitions s.t. if a transition labeled by α is followed by a transition labeled by β then the two morphisms are composable in $\mathbb{A}$ (i.e., $\alpha \circ \beta \in Mor(\mathbb{A})$).*

Since in an ATS transitions $\xrightarrow{\alpha}$ are labeled with morphisms from $\mathbb{A}$, we also have a grip on the underlying objects involved in the transition, i.e., α^s and α^t. Because of this, where we need to talk explicitly in a transition about the objects of the morphism label we may write $\xrightarrow{\{\alpha^s,\alpha^t\}}$.

A main motivation for arrow-labeled transition systems and MSOS is to have as configurations *only program terms*, without the additional data structures that the program may manipulate (like stores or heaps). This goal is related to what one usually seas in e.g.: (i) typing systems where the program syntax alone is under analysis; or in (ii) Hoare logic style of reasoning about programs where the Hoare rules are defined for program terms only (with the pre- and post-conditions being the ones talking about the stores/heaps); or in (iii) process algebras where the semantics of processes is given as a labeled transition system with process terms as the states and their observable behavior as labels on transitions. In MSOS the additional data structures that are manipulated by the program will be present on the transitions, part of the morphisms that label them, as we see further.

In current practice the following three kinds of label categories are enough. But this does not restrict the modular SOS framework, and future kinds of data could be accommodated also.

Definition 3.2 (label categories) *There are three kinds of categories we may use to build more complex label categories:*

- ***discrete category:*** *A* discrete category *is a category which has only identity morphisms. No other morphisms are allowed.*
- ***pairs category:*** *A* pair category *is a category which has one unique morphism between every two objects.*
- ***monoid category:*** *A* monoid category *is a category that has a single object and the morphisms are elements from some predefined set Act.*

Intuitively, discrete categories correspond to additional information that is of a read-only type, like heaps. Pairs categories correspond to additional data of a type read/write, like stores. Each store is one object in the category. The morphisms between two stores represent how this store may be modified by the program when executed. We take a most general view where a program may change a store in very radical ways, therefore, we have morphisms between every two store objects. Monoid categories correspond to write-only type of data, like observable information that is emitted during the execution of the program, or messages sent between components in a concurrent program.

Example 3.3 *The following examples of categories are used in this paper. We can build a discrete or a pair category by taking some underlying set of objects. Different underlying sets make different categories.*

- *A discrete category can be seen as a set. One discrete category $\mathbb{CN}$ may contain as objects class names. Another standard example of a discrete category $\mathbb{H}$ has as objects heaps: $|\mathbb{H}| = IdVar \rightharpoonup Val$, i.e. all partial functions from some set $IdVar$ of variable identifiers to some set Val of values.*
- *In a pairs category every two objects are related to each other in a symmetric way (more than in a total preorder). An example of a pairs category $\mathbb{S}$ has as objects stores, i.e. partial mappings from a set of variable identifiers to a set of values. Pairs categories appear almost everywhere in this paper because of the rewriting style that we take to giving semantics (even what some works consider as heaps, like for holding definitions, we consider as stores).*

When several additional data are needed to define the semantics then we use complex label categories obtained by making product of such basic label categories as above. It is recommended to use as many data components as needed to get a more clean view of the semantics for each programming construct. An implementation may choose to put several data structures together, if no clashes can appear.

To have a clear grip on each component of such a product we use a special construction that returns product categories and attaches an index to each label component, thus offering the possibility to uniquely extract each component from a complex label by using the associated index. This procedure makes it possible to have the modularity of the semantics, as we explain further.

Definition 3.4 (label transformers) *Let $Index$ be a countable set of indexes. Let $\mathbb{B}$ be a category of one of the three kinds from Definition 3.2. The* label transformer $\mathbf{LabTrans}(i, \mathbb{B})$ *maps any category $\mathbb{A}$, which is either the trivial category $\mathbf{TrivCat}$ or is obtained using the label transformer itself, to the category $\mathbb{A} \times \mathbb{B}$, and associates a* get operation $get : Mor(\mathbb{A} \times \mathbb{B}) \times Index \to (\cup_j Mor(\mathbb{A}_j)) \cup Mor(\mathbb{B})$ *which for each composed morphism of the new $\mathbb{A} \times \mathbb{B}$ associates a morphism in one of the component categories of the product, as follows:*

$$get((\alpha_{\mathbb{A}}, \beta_{\mathbb{B}}), k) = \begin{cases} \beta_{\mathbb{B}}, & \textit{if } i = k \\ get(\alpha_{\mathbb{A}}, k), & \textit{otherwise.} \end{cases}$$

Notation: For a composed morphism α of a product category obtained using the label transformer we may denote the get operation using the dot-notation (well established in object-oriented languages) to refer to the respective component morphism; i.e., $\alpha.i$ for $get(\alpha, i)$, with i being one of the indexes used to construct the product category. Since $\alpha.i$ is a morphism in a basic category, we may also refer to its source and target objects (when relevant, like in the case of discrete or pair categories) as $\alpha.i^s$ and $\alpha.i^t$.

Henceforth, the labels that we allow in arrow-labeled transition systems come only from categories built using $\mathbf{LabTrans}(,)$. In consequence, we may have complex labels, formed as tuples of morphisms from basic categories; and we may access each of the components of a label using the dot-notation to stand for the precisely defined get operation. Now we proceed to define what the SOS rules look like in this setting.

Definition 3.5 (program terms) *Consider a set of (meta)variables* Var. *A* multi-sorted signature Σ *is a set of function symbols, together with an* arity *mapping $ar()$ that assigns a natural number to each function symbol, and a family of* sorts S_i. *Each function symbol has a sort definition which specifies what sorts correspond to its inputs and output. A function of arity zero is called a* constant.

The set of terms *over a signature Σ is denoted* $\mathsf{Terms}(\Sigma)$ *and is defined as:*

- *a metavariable from* Var *is a term;*
- *for some function symbol f and set of terms $t_1, \ldots, t_{ar(f)}$, of the right sort, then $f(t_1, \ldots, t_{ar(f)})$ is a term.*

Definition 3.6 (rules) *A* literal *is $t \overset{\alpha}{\to} t'$, with t, t' program terms, possibly containing meta-variables (i.e., these are program schemes). A literal is* closed *if t, t' do not contain meta-variables. A* transition rule *is of the form H/l with H a set of literals, called the* premises*, and l one literal, called the* conclusion.

Notation: When writing literals we use the following notation for the labels. We write $t \xrightarrow{\{\alpha.i^s \ldots \alpha.i^t\}} t'$ to mean that the morphism α is a tuple where the component indexed by i is the one given on the transition, and all other components are the identity morphism, symbolized by the *three dots*. We write sources of morphisms on the left of the three dots, and targets on the right. In one transition we may refer to several components, e.g.: $t \xrightarrow{\{\alpha.i^s, \alpha.j \ldots \alpha.i^t\}} t'$. In this example the j index is associated with a discrete category and therefore, we do not write the target of it on the right because it is understood as being the same. Moreover, because of the right/left convention we omit the superscripts denoting sources and targets. An even more terse notation may simply drop all references to α and keep only the indexes, thus the last example becomes $t \xrightarrow{\{i=o, j=h \ldots i=o'\}} t'$. The objects o, o' may be stores, and thus the transition says that the store o is changed to the o', whereas the component j may only be inspected, like with a heap h.

Rules are written as in proof systems, with the premises on top of a line and the conclusion below the line. When side condition should be mentioned, we write these also on top of the line.

Modularity refers to the fact that once a rule is defined, using labels from some label category (i.e., referring to some needed additional data), this rule is not changed when new additional data is required for defining some other new rules for a new programming construct. This is made precise by the essential result in [21, Corollary 1].

Intuitively, the result says that any transition defined by an old rule, i.e., labeled with some α from some category $\mathbb{A}$, is found in the new arrow-labeled transition system, over a new category $\mathbf{LabTrans}(i, \mathbb{B})(\mathbb{A})$, using an embedding functor which just attaches an identity morphism to the old morphism, i.e., (α, id_b), for the current object $b \in |\mathbb{B}|$. Moreover, for any transition defined in terms of the new composed labels from $\mathbb{A} \times \mathbb{B}$, if it comes from one of the old rules then the projection from $\mathbb{A} \times \mathbb{B}$ to $\mathbb{A}$ gives an old label morphism by forgetting the identity morphism on $\mathbb{B}$. This is the case because the old transition is defined with the dots notation and refers only to components in $\mathbb{A}$, making all other components contribute only with the identity morphism.

Theorem 3.7 ([21, Corollary 1]) *Let $\mathbb{A}$ be a category constructed using the label transformers $\mathbf{LabTrans}(j, \mathbb{B}_j)$ for some basic categories $\mathbb{B}_j$ of the three kinds defined before and with $j \in J \subset Index$. Consider a set of rules R which specifies an arrow-labeled transition system over $\mathbb{A}$, where the rules in R refer to only indexes from J (i.e., the get operations in the morphism specifications of labels). Let the category $\mathbb{A}' = \mathbf{LabTrans}(i, \mathbb{B}_i)(\mathbb{A})$, where $i \notin J$, and let $\rightarrow'$ be the transition relation specified by the same set of rules R but having labels from $\mathbb{A}'$.*

We have for each computation $\xrightarrow{\alpha}\xrightarrow{\beta}\ldots$ specified by R over $\mathbb{A}$ a corresponding computation $\xrightarrow{\alpha'}{}'\xrightarrow{\beta'}{}'\ldots$ over $\mathbb{A}'$, and vice versa.

Proof: We give only an idea of the proof. The label transformer $\mathbf{LabTrans}(i, \mathbb{B}_i)$ forms a projection functor from $\mathbb{A} \times \mathbb{B}_i$. This functor is used to get the vice versa direction of the statement, by forgetting the structure of $\mathbb{B}_i$. This is possible because the rules in R do not refer to this index i, hence to morphisms in $\mathbb{B}_i$, which means these are just the identity morphisms. The label transformer also forms a family of embedding functors from $\mathbb{A}$ into $\mathbb{A} \times \mathbb{B}_i$ (for each object of $\mathbb{B}_i$). These functors are used to obtain the first direction of the statement. Depending on the current object of $\mathbb{B}_i$ we use the corresponding embedding functor to add to the label specified by the rules R on $\mathbb{A}$ an identity functor on $\mathbb{B}_i$, thus obtaining a corresponding transition with label morphism from $\mathbb{A} \times \mathbb{B}_i$. $\square$

Henceforth we consider various programming constructs that are normally used in languages, and do not concern ourselves with their redundancy. In particular, we will treat those constructs found in the two programming languages PROTEUS [27] and CREOL [17].

Example 3.8 (no labels) *Consider a multi-sorted signature $\Sigma^{3.8}_{exe}$ consisting of the following programming constructs:*

$$S \ ::= \ \mathbf{skip} \mid S\,;S$$

where **skip** *is a constant, standing for the program that does nothing, being the identity element for _ ; _ which is a binary function symbol, standing for sequential composition of two programs. For now we have one sort, that of Statement, where the constant* **skip** *is the trivial statement, and sequence is defined over statements.*

We define the following transition rules:

$$\frac{}{\mathbf{skip} \xrightarrow{U} \mathsf{nil}} \qquad \frac{S_1 \xrightarrow{X} S_1'}{S_1\,;S_2 \xrightarrow{X} S_1'\,;S_2} \qquad \frac{}{\mathsf{nil}\,;S_2 \xrightarrow{U} S_2}$$

where the special label X stands for any morphism, and U stands for an unobservable label, which are exactly the identity morphisms. Note that for these rules *we do not specify the label categories because they can use any category, even the trivial one. This means that no additional data is needed by the respective two programming constructs. The second rule has one premise, and assumes nothing about the morphism of the transition. It only says that the label is carried along from the statement S_1 to the whole sequence statement $S_1\,;S_2$. The first rule is an axiom because it contains no premises, and says that the* **skip** *program reduces to the value* nil *by an unobservable transition; i.e., by the identity morphism on the current object in the current category of labels, whichever these may be.*

Throughout the paper we work with, what is called, *value added syntax*, where the values that the program may take are included in the syntax as constant symbols. Denote these generally as $v \in \mathit{Val}$, with $n \in \mathbb{N} \subseteq \mathit{Val}$ and $b \in \{\mathbf{true}, \mathbf{false}\} \subseteq \mathit{Val}$. The $\mathsf{nil} \in \mathit{Val}$ is seen as the special value that statements take when finished executing. The values are considered to have sort *Expressions*, denoted usually by $e \in E$. We also consider to have the standard arithmetic and Boolean operators which take expressions and return expressions. (See Appendix for a detailed example.)

Example 3.9 (read-only label categories) *We add a set of variable identifiers as constant symbols, denote these by $\mathbf{x} \in \mathit{IdVar}$. Variable identifiers have sort* Expressions *and can be used to form program terms in combination with the other operators from before that take expressions as input. Let these make a signature $\Sigma^{3.9}_{exe}$, which can be added to any other signature.*

$$E \ ::= \ \mathbf{x}\,(\mathbf{x} \in \mathit{IdVar}) \mid \ldots$$

The interpretation of variable identifiers is given wrt. an additional data structure called store, which keeps track of the values associated to each variable identifier. In consequence, we define a label category $\mathbb{S}$, having as objects $|\mathbb{S}| = \mathit{IdVar} \rightharpoonup \mathit{Val}$ the set of all partial functions from variable identifiers to values, denoting stores. Define $\mathbb{S}$ as a discrete category, i.e., only with identity morphisms, since in the case of variable identifiers alone, the store is intended only to be inspected by the program. The label category to be used for defining the transitions is formed by applying the label transformer $\mathbf{LabTrans}(1, \mathbb{S})$ to some category of labels, depending on the already chosen programming constructs and transition rules.

The transition rule corresponding to the variable identifiers is:

$$\frac{\rho(\mathbf{x}) = v}{\mathbf{x} \xrightarrow{U\{1=\rho\ldots\}} v}$$

The rule defines a transition, between terms $\mathbf{x}$ and v, labeled with a morphism satisfying the condition that the label component with index 1 has as source an *object $\rho \in |\mathbb{S}|$ that maps the variable identifier to the value v. Any other possible label components, if and when they exist, contribute with an identity morphism. Moreover, this transition is unobservable. Note that because the category associated to the label component with index 1 is a discrete category, we do not specify the target object explicitly since it is the same as the source object that is specified in the label.*

Naturally, if one chooses to give semantics to variable identifiers using a division of this auxiliary data into one heap and one store, the above rule has to be changed. Otherwise, we will not change this rule in the future.

Example 3.10 (changing label categories from read-only to read/write) *Having variable identifiers we may add* assignment *statements and* variable declarations *as $\Sigma^{3.10}_{exe}$, which includes $\Sigma^{3.9}_{exe}$.*

$$D \ ::= \ \mathbf{var}\,\mathbf{x} := e \mid d_0 \,;d_1\,(d_0, d_1 \in D)$$

$$S \ ::= \ \mathbf{x} := e \mid d\,(d \in D) \mid \ldots \qquad\qquad E \ ::= \ \mathbf{let}\,d\,\mathbf{in}\,e\ (d \in D) \mid \ldots$$

Both assignments and declarations (which are a subsort of statements) allow the program to change the store data structure that we used before for evaluating variable identifiers. Therefore, here we need $\mathbb{S}$ to be a pairs category so to capture that a program can also change a store, besides inspecting it. Important in Modular SOS is that rules which use read-only discrete categories are not affected if we change these label components to be read/write pairs categories (with the same objects). Indeed, the syntax used in the rules refers only to the source objects of the morphisms. Henceforth, whenever in a rule we mention only the source of a morphism component it means that the target is the same, i.e., we specify a particular identity morphism.

To exemplify the notation-freedom of the theory that we use, we will switch from now on to indexes which are more meaningful for the reader (e.g., the rewriting logic semantics of CREOL *from [16] uses notations like PrQ for "process queue", or A for "attributes"). In consequence, using now the label category* $\mathbf{LabTrans}(S, \mathbb{S})$*, with $\mathbb{S}$ a pairs category of stores, we add new rules which refer to both the source and the target stores of the morphisms. (See Appendix for the complete set of rules.)*

$$\frac{\mathbf{x} \in \rho}{\mathbf{x} := v \xrightarrow{\{S=\rho\,\ldots\,S=\rho[\mathbf{x}\mapsto v]\}} \mathsf{nil}} \qquad\qquad \frac{d \xrightarrow{\{S=\rho\,\ldots\,S=\rho'\}} \mathsf{nil} \qquad e \xrightarrow{\{S=\rho'\,\ldots\}} v}{\mathbf{let}\,d\,\mathbf{in}\,e \xrightarrow{\{S=\rho\,\ldots\}} v}$$

*Note that the rule for the **let** construct is given in a big-step SOS style. This is intended to illustrate that MSOS is amenable to mixed-step style of semantics. The requirement above the line in the first rule is normally ensured by the typing system, and thus can be removed. This is even desired so that we get a transition rule that is more close to the standard rule formats [3].*

Example 3.11 (functions) *We consider* function identifiers *as constants denoted by* $\mathbf{f} \in \mathit{IdFun}$*, and* function definitions *and* function applications*, in a signature* $\Sigma_{exe}^{3.11}$ *which may be added to any signature that includes variable identifiers.*

$$D \ ::= \ \mathbf{fun}\,\mathbf{f}(\mathbf{x})\,\{s\} \ \ (s \in S) \mid \ldots \qquad\qquad S \ ::= \ \mathbf{f}\,e\;(e \in E) \mid \ldots$$

Function declarations are stored in a new label component which is a pairs category containing objects which associate function identifiers with lambda terms in a way known from functional languages. Denote this category by $\mathbb{F}$ *and its objects as* $\rho_f \in |\mathbb{F}|$*. Add this as a label component using the label transformer* $\mathbf{LabTrans}(F, \mathbb{F})\mathbf{LabTrans}(S, \mathbb{S})$*. Since variable identifiers are needed, the stores component is present also. Another semantics may consider these two as a single store-like data structure. In this paper we prefer to use more disjoint structures when possible.*

The transition rules below are as in PROTEUS*, using a functional languages style. We are using the notation* $s[v/\mathbf{x}]$ *for capture avoiding substitution of all occurrences of a variable in a program statement.*

$$\frac{e \xrightarrow{X} e'}{\mathbf{f}\,e \xrightarrow{X} \mathbf{f}\,e'} \qquad \frac{\rho_f(\mathbf{f}) = \lambda(\mathbf{x}).s}{\mathbf{f}\,v \xrightarrow{\{F=\rho_f\,\ldots\}} s[v/\mathbf{x}]} \qquad \frac{}{\mathbf{fun}\,\mathbf{f}(\mathbf{x})\,\{s\} \xrightarrow{\{F=\rho_f\,\ldots\,F=\rho_f[\mathbf{f}\mapsto\lambda(\mathbf{X}).s]\}} \mathsf{nil}}$$

With the exception of types,[1] we have reached by now the language PROTEUS of [27, 28] (see in Appendix the conditional construct and the records, which add a third label component $\mathbf{LabTrans}(R, \mathbb{R})$). We will add the update construct in Section 5. We have used single variable identifiers above, but this can be easily generalized to lists. Moreover, since we investigate only semantics aspects in this paper (i.e., no typing systems), we assume only syntactically correct programs, including static typing.

4 Encapsulation for Concurrent Object-Orientation

When adding the notion of concurrent objects we require that some constructs be run *inside* an object. This notion of encapsulation of the execution must be captured in the category theory of the labels. We provide for this an *encapsulating construction*. The term "encapsulate" has specific meaning in object-oriented languages. Our categorical construction has a similar intuition, therefore we prefer the same terminology.

[1]Types may be added to programming constructs, and also a type environment as one new label component.

Not only the code is encapsulated in an object, but also the auxiliary data that is used to give semantics to the code. These data components are now private to the specific object. We want to keep the modularity in defining semantics for object-oriented constructs, similar to what we had until now. We want that the definition of new semantic rules not change the definitions of the old rules. On the contrary, we may use the old transition relation to define new transition relations. Essentially, we will encapsulate old transitions into transitions that are localized to one object. In the concurrent setting, we even see how more objects may perform transitions localized to each of them, thus making a global transition, changing many of the local data.

Definition 4.1 (encapsulating construction) *Let $\mathbb{O}$ be a discrete category, and $\mathbb{A}$ a label category. The* encapsulating construction $\mathbf{Enc}(\mathbb{O}, \mathbb{A})$ *returns a category $\mathbb{E}$ with all the functors $F : \mathbb{O} \to \mathbb{A}$ as objects, and natural transformations between these functors as morphisms.*

The discrete category $\mathbb{O}$ captures programming objects identifiers (i.e., each object of the category is a unique identifier for a programming object).

There are various properties that we need of this construction. One is that the resulting category is similar to the product categories that the label transformer generates. Once having such properties we may use the encapsulation to define labels on the transitions, similar to any of the three kinds of basic categories.

Proposition 4.2 *Properties for basic categories.*

1. *In discrete or pair categories morphisms are uniquely defined by the objects.*

2. *In categories obtained using the label transformer applied to discrete or pair categories, the morphisms are uniquely defined by the objects.*

3. *When monoid categories are used in the label transformers, then one morphism is uniquely determined by the objects up to the morphism components coming from the monoid categories; i.e., when the monoid components are projected away.*

Proposition 4.3 *When the encapsulating construction is applied to $\mathbb{A}$ which is built only with discrete or pair categories, then the morphisms of $\mathbb{E} = \mathbf{Enc}(\mathbb{O}, \mathbb{A})$ are uniquely defined by the objects.*

Proof: The objects of $\mathbb{E}$ are functors $F : \mathbb{O} \to \mathbb{A}$. Take two such functors F, F'; a morphism between them is a natural transformation η which for each object of $\mathbb{O}$ associates one morphism of $\mathbb{A}$, i.e., $\eta(o) \in Mor(\mathbb{A})$, with the following property:

for some $o \in |\mathbb{O}|$ and some morphism $\alpha \in Mor(\mathbb{O})$ with source o and target o', the following diagram commutes

$$\begin{tikzcd}
\text{F(o)} \arrow[r, "\eta(o)"] \arrow[d, "F(\alpha)"'] & \text{F'(o)} \arrow[d, "F'(\alpha)"] \\
\text{F(o')} \arrow[r, "\eta(o')"] & \text{F'(o')}
\end{tikzcd}$$

But in our case this diagram becomes more simple because in $\mathbb{O}$ the only morphisms are the identities, which means that α is in fact id_o and thus the o' in the diagram above is just o. Moreover, the functors take identities to identities, so $F(\alpha)$ becomes $id_{F(o)}$. Now the diagram would look like the one below, which clearly commutes for any natural transformation.

$$\begin{tikzcd}
\text{F(o)} \arrow[r, "\eta(o)"] \arrow[d, "id_{F(o)}"'] & \text{F'(o)} \arrow[d, "id_{F'(o)}"] \\
\text{F(o)} \arrow[r, "\eta(o)"] & \text{F'(o)}
\end{tikzcd}$$

The natural transformation η assigns the morphism $\eta(o)$ between $F(o)$ and $F'(o)$ in $\mathbb{A}$. Since $\mathbb{A}$ is one of our special label categories then there is a unique morphism which is the pair $(F(o), F'(o))$, as it is uniquely determined by the objects on which it acts. The same for any $o' \in |\mathbb{O}|$ the $\eta(o')$ is unique.

The morphism (natural transformation) η is composed of many morphisms from $\mathbb{A}$, one for each object of $\mathbb{O}$. All of these are unique, determined by the application of the two functors on the specific object of $\mathbb{O}$. In consequence, the η is uniquely defined by the two functors on which it is applied. We can either write η as a pair of functors (F, F') or we can write it as a set of morphisms from $\mathbb{A}$ indexed by the objects $o \in \mathbb{O}$, i.e., $\eta = \{(F(o), F'(o)) \mid o \in |\mathbb{O}|\}$. □

The intuition is that each functor attaches one data object to each programming object identifier (i.e., object of $\mathbb{O}$). So each functor captures one snapshot of the private data of each object. We have access to these encapsulated data by taking the appropriate index, i.e., $F(o)$ is the data encapsulated in the programming object identified by o.

But we are interested in the morphisms, which capture how the data is changed by the program. The fact that the morphisms are identified by the objects on which they act says that it is equally good to mention the morphism or to mention the pair of objects of interest. In the encapsulating construction we may refer to the morphisms also, and these are also indexed by the programming object identifiers, i.e., there is one local morphism associated to each o, it is $\eta(o)$. Therefore, morphisms are localized also, and we can refer to each local morphism by its corresponding object identifier.

All these intuitions hold also when monoid categories are part of the labels. In this case there are multiple natural transformations between two functors (and also from one functor to itself), one for each morphism in the underlying monoid category. When more monoid categories are used, the natural transformations multiply as expected. The same indexing exists and we have access to the unique local morphism for every object identifier.

Since for an object identifier o the $\eta(o)$ is a morphism in the label category $\mathbb{A}$, we are free to use the get operation to refer to a particular morphism component, i.e., we may write $\eta(o).i$ to access the i^{th} component local to the object o.

Once the encapsulation concept above is understood one can choose any preferred notation for it; e.g.: one may write $o.i$ or $o \mapsto i$ or $\langle o \mid i \rangle$ or $o : i$.

The encapsulating construction is used to give semantics to concurrent object-oriented programming languages where code is executed locally, in each object, and the objects are running in parallel, maybe communicating with each other. The modularity is obtained by defining the localized transitions in terms of the transitions defined for the individual executing programming constructs, as given by the rule in Example 4.4.

The category built by the encapsulating construction can be used with the label transformer to attach more global data structures. Therefore, the encapsulating construction is modular in the sense that new global programming constructs and rules may be added without changing the rules for encapsulation. The reference mechanism provided by the label transformer is used as normal. We see this in Example 4.8 on asynchronous method calls where additional global structures are needed for keeping track of the messages being passed around.

Moreover, we may encapsulate this category again, wrt. a new discrete category, giving us a different set of identifiers. This can be seen applicable in languages for components, i.e., where several objects may execute in parallel inside one component, and many components may execute in parallel also.

The encapsulating construction preserves modularity also in the sense that new programming constructs may be added to run localized (inside objects), and thus the encapsulated category may need to be extended to include new auxiliary data components. The encapsulation is not affected, in the sense that the rules for encapsulation, or which were defined referring to some encapsulated data, need no change. The reference mechanism (with the get operation provided by the label transformer) used in defining the localized rules is independent of the new local categories added. We see this in Example 4.6 treating *threads*, which are a programming notion independent of the encapsulation. Henceforth we denote the encapsulated (or local or internal) category by $\mathbb{I}$ when its components are irrelevant.

Example 4.4 (objects) *We add object identifiers as constants denoted* $\mathbf{o} \in IdObj$. *We add one programming construct of a new sort called* Objects, *denoted* O, *which localizes a term of sort statement wrt. an object identifier.*

$$O \ ::= \ \langle\, \mathbf{o} \mid s \,\rangle \ (s \in S)$$

This signature $\Sigma^{4.4}_{exe}$ *should include some signature defining statements; any of the constructs before can run inside the object construction, but which exactly is irrelevant for the transition rules below. The semantics of object programs is given using transitions labeled from a category constructed using the encapsulating construction applied to some appropriate* $\mathbb{I}$*:* $\mathbb{E} = \mathbf{Enc}(\mathbb{O}, \mathbb{I})$, *where* $|\mathbb{O}| = IdObj$. *We give one transition rule that encapsulates any transition at the level of the statements inside the objects.*

$$\frac{s \xrightarrow{X} s'}{\langle\, \mathbf{o} \mid s \,\rangle \xrightarrow{\mathbf{o}:X} \langle\, \mathbf{o} \mid s' \,\rangle} \quad (\text{ENC})$$

The label X *stands, as before, for any morphism in the local category* $\mathbb{I}$. *The label of the conclusion is taken as a morphism in the encapsulation category* $\mathbb{E}$. *The notation* $o : X$ *specifies only part of the morphism, whereas the rest of the natural transformation may be any identity morphism. This says that we specify that the data for the object* o *is known before and after the local execution, wheres the local data of any other objects are irrelevant and may be anything, but is not changed in any way. Therefore, any functors* F, F' *that respect the fact that they assign to* o *the source and target objects of* X, *and may assign anything to all other objects, are good. Moreover, the monoid labels that may appear in* X *are part of the specific natural transformation that we choose between the two functors* F, F'*; i.e., it is exactly the natural transformation assigning to* o *the morphism* $X \in Mor(\mathbb{I})$.

Example 4.5 (systems of objects) *Objects may run in parallel, thus forming systems of distributed objects, in each object running some code. For this we add a parallel construct* $\|$ *of sort* Objects, *with all object identifiers different:*

$$O \ ::= \ obj_1 \parallel obj_2 \ (obj_1, obj_2 \in O) \mid \ldots$$

We choose an interleaving semantics for our parallel operator, hence the rule:

$$\frac{obj_1 \xrightarrow{X} obj_1'}{obj_1 \parallel obj_2 \xrightarrow{X} obj_1' \parallel obj_2}$$

Note that the X in this rule stands for any morphism in the encapsulating category, whereas in the previous rule it was standing for morphisms in the local category.

We may easily specify non-interleaving concurrency by specifying more precisely the label components:

$$\frac{\langle\, \mathbf{o_1} \mid s_1 \,\rangle \xrightarrow{\mathbf{o_1}:X} \langle\, \mathbf{o_1} \mid s_1' \,\rangle \qquad obj_2 \xrightarrow{\eta} obj_2'}{\langle\, \mathbf{o_1} \mid s_1 \,\rangle \parallel obj_2 \xrightarrow{\eta[\mathbf{o_1}:X]} \langle\, \mathbf{o_1} \mid s_1' \,\rangle \parallel obj_2'}$$

The label of the conclusion specifies the morphism which is the natural transformation η changed so that it incorporates the specified local morphism of $\mathbf{o_1}$. In this way any number of objects may execute local code and the local changes to their data is visible in the global label.

Example 4.6 (threads) *We take the model of threads studied in [1] and consider the following programming constructs of sort statement in a signature $\Sigma_{exe}^{4.6}$ which normally would include also other constructs for statements from before:*

$$S \;\; ::= \;\; \mathbf{yield} \mid \mathbf{async}\,(s)\,(s \in S) \mid \ldots$$

Threads need an additional data component called thread pool. We build a pairs category $\mathbb{T}$ which has as objects thread pools. The internal label category $\mathbb{I}$ (chosen depending on the other constructs) is extended with $\mathbf{LabTrans}(T, \mathbb{T})$.

We need more algebraic structure for the thread pools objects, which is used when defining the transition rules. A thread pool may be implemented in multiple ways (e.g., as sets or lists); here we only require two operations on a thread pool, an insertion $\oplus$ and a deletion $\ominus$ operation. Take ρ_t to be a thread pool and s a program term, then $\rho_t \oplus s$ is also a thread pool containing s; and when $s \in \rho_t$ then $\rho_t \ominus s$ is also a thread pool that is the same as ρ_t but does not contain s.

Because of the $\mathbf{yield}$ *which needs the whole program term that follows it, we give semantics to threads using evaluation contexts. The modular SOS is perfectly suited for describing semantics using evaluation contexts.*

Evaluation contexts are statements with a hole $[\,]$:

$$Ev \;\; ::= \;\; [\,] \mid Ev\,;S$$

Placing a program term s in the whole of a context Ev is denoted $Ev[s]$ and results in a normal program term (i.e., without the hole). It is essential to prove that any statement in the language can be uniquely decomposed into an evaluation context Ev and a program term s so that the choice of transition rules is unambiguous. For the simple contexts that we defined above, this result is easy.

Instead of changing all the rules from before using evaluation contexts, we choose to give the following rule, and remove the two rules for sequential composition from Example 3.8. A second rule is required when object terms are present.

The X label on the left comes from an encapsulated $\mathbb{I}$ whereas on the right comes from a global label.

$$\frac{s \neq \mathsf{nil} \qquad s \xrightarrow{X} s'}{Ev[s] \xrightarrow{X} Ev[s']} \qquad \frac{s \neq \mathsf{nil} \qquad \langle\, \mathbf{o} \mid s \,\rangle \xrightarrow{X} \langle\, \mathbf{o} \mid s' \,\rangle}{\langle\, \mathbf{o} \mid Ev[s] \,\rangle \xrightarrow{X} \langle\, \mathbf{o} \mid Ev[s'] \,\rangle}$$

Now we can give the rules for the new programming constructs, which may be compared to the ones given in [1, Fig.4].

$$\frac{}{\mathbf{async}\,(s) \xrightarrow{\{T=\rho_t \,\ldots\, T=\rho_t \oplus s\}} \mathsf{nil}} \qquad \frac{}{Ev[\mathbf{yield}] \xrightarrow{\{T=\rho_t \,\ldots\, T=\rho_t \oplus Ev[\mathsf{nil}]\}} \mathsf{nil}}$$

$$\frac{s \in \rho_t}{\mathsf{nil} \xrightarrow{\{T=\rho_t \,\ldots\, T=\rho_t \ominus s\}} s}$$

Example 4.7 (classes) *It is common in the setting of object-orientation to have method definitions part of class definitions, where objects are instances of such classes and can be created anytime with the* **new** *programming construct. Inheritance and interfaces are normally part of class definitions, but we do not complicate this example with them; these can be easily added. The details of this example are already too complicated for the purposes of this paper.*

Class identifiers are introduced from a set $IdClass$, and denoted $\mathbf{C}$. *Class definitions include method definitions and attribute definitions:*

$$At \;::=\; \mathbf{var}\,\mathbf{x} \mid At\,;At \qquad M \;::=\; \mathbf{mtd}\,\mathbf{m}(\mathbf{x})\,\{s\} \;\; (s \in S) \mid M\,;M$$

$$D \;::=\; \mathbf{class}\,\mathbf{C}\,\{At\,;M\} \mid \ldots \qquad S \;::=\; \mathbf{x} := \mathbf{new}\,\mathbf{C} \mid \mathbf{m}(e) \;(e \in E) \mid \ldots$$

For the semantics we need two global category components (i.e., not local to the objects) which keep definitions of methods for each class and another to keep the attributes. Denote these by $\mathbb{C}$ and $\mathbb{A}$, and associate using the label transformer the indexes C and A. The objects $\rho_c \in |\mathbb{C}|$ are mappings from class identifiers to definitions of methods; i.e., $\rho_c : IdClass \rightharpoonup (IdMethods \rightharpoonup MtdDef)$. Objects $\rho_a \in |\mathbb{A}|$ are mappings $IdClass \rightharpoonup At$. The encapsulation is a global component of its own, to which the label transformer associates index E. The transition rule for class definitions is:

$$\frac{\rho'_a = \rho_a[\mathbf{C} \mapsto At] \qquad \rho'_c = \rho_c[\mathbf{C} \mapsto \{\mathbf{m} \mapsto \lambda(\mathbf{x}).(s) \mid \mathbf{m} \in M\}]}{\langle\, \mathbf{o} \mid \mathbf{class}\,\mathbf{C}\,\{At\,;M\} \,\rangle \xrightarrow{\{A=\rho_a, C=\rho_c \,\ldots\, A=\rho'_a, C=\rho'_c\}} \langle\, \mathbf{o} \mid \mathsf{nil} \,\rangle}$$

Each object is an instance of a class. In consequence we associate to each object the name of the class it belongs to, and where method definitions can be *retrieved from.*[2] *Normally this class name information is held in a special variable of the object, but here we will use a category component, to keep with the modular style. Therefore, to the internal category $\mathbb{I}$ we add one more category $\mathbb{CN}$ to which the label transformer will associate the index CN. The objects $|\mathbb{CN}| = \mathit{IdClass}$ are just class identifiers. The rule for object creation is:*

$$\frac{\mathit{fresh}(\mathbf{o}') \qquad \forall i \neq CN \;\; \mathbf{o}' : i = \emptyset \qquad \rho_a(\mathbf{C}) = At}{\langle \mathbf{o} \,|\, Ev[\,\mathbf{x} := \ \mathbf{new}\,\mathbf{C}\,]\rangle \xrightarrow{\{A=\rho_a, C=\rho_c, \mathbf{o}:S=\rho \ldots \mathbf{o}':CN=\mathbf{C}, \mathbf{o}:S=\rho[\mathbf{x}\mapsto\mathbf{o}']\}} \langle \mathbf{o} \,|\, Ev[\mathsf{nil}]\rangle \,\|\, \langle \mathbf{o}' \,|\, At\rangle}$$

There are ways of ensuring freshness of the object identifiers. Also there are various styles of creating new objects, where some use a constructor which initializes the attributes, instead of just running the list of attribute definitions as we did. Also, Creol *keeps the attributes in a special data structure, opposed to how we put them in the store of the object. All these are readily definable. But for the purpose of our example these details would only clutter the presentation.*

The transition rule for method application must include the object because it needs the global class definitions where the method definitions are found. (See Appendix for methods defined inside objects.)

$$\frac{e \xrightarrow{X} e'}{\mathbf{m}(e) \xrightarrow{X} \mathbf{m}(e')} \qquad\qquad \frac{\mathbf{C} \in \rho_c \qquad \mathbf{m} \in \rho_c(\mathbf{C}) \qquad \rho_c(\mathbf{C})(\mathbf{m}) = \lambda(\mathbf{x}).(s)}{\langle\, \mathbf{o} \mid \mathbf{m}(v)\,\rangle \xrightarrow{\{\mathbf{o}:CN=\mathbf{C}, C=\rho_c \ldots\}} \langle\, \mathbf{o} \mid s[v/\mathbf{x}]\,\rangle}$$

Example 4.8 (asynchronous method calls) *We take the model of asynchronous method calls from [16] and consider two programming constructs for calling a method and reading the result of the completion of a call:*

$$S \;\; ::= \;\; \mathbf{t}!\mathbf{o}.\mathbf{m}(e) \;\mid\; \mathbf{t}?(\mathbf{x}) \;\mid \mathbf{return}\, e \mid \ldots$$

where $\mathbf{t} \in \mathit{IdFut}$ are special identifiers used for retrieving the result of the method call. Denote this signature $\Sigma^{4.8}_{exe}$, which can be added to any previous signature.

The asynchronous method calls, as discussed in [16], work with asynchronous message passing, as in the Actors model [5]. In consequence we need a global data component to keep track of the messages in the system. We consider each object having a pool of messages. Since the message pools will be manipulated by the distributed objects of the system we use a pairs category $\mathbb{M}$ with objects $|\mathbb{M}| = \mathit{IdObj} \to 2^{\mathit{MsgTerm}}$ being mappings from object identifiers to message sets. The label transformer $\mathbf{LabTrans}(M, \mathbb{M})$ is applied at least to an encapsulating

[2]This is the *dynamic binding* notion (also known as late binding, or dynamic dispatch) where the method definitions are retrieved when they are needed. This is especially useful in the presence of inheritance and dynamic class upgrades, as in Section 5.2; otherwise we could do without, and use the method definitions local to objects as in Example A.5 from Appendix.

category. Similarly to the thread pools, define set operations $\oplus$ *and* $\ominus$ *to add and remove messages from any set* $\mathcal{MS} \in 2^{MsgTerm}$*. For our exemplification purposes the messages are of the form:* $invoke(\mathbf{o}, n, \mathbf{m}(v))$ *and* $comp(n, v)$*, where* $\mathbf{o}$ *is an object identifier,* $n \in \mathbb{N}$ *is a natural number, and* $\mathbf{m}(v)$ *represents the method named* $\mathbf{m}$ *and* v *a value term. Because of the asynchronous method calling scheme, the method declarations are particular in the sense that the first two parameters are predefined for all methods as being* $caller$ *and* $label$*, and the statements may end with a* $\mathbf{return}$*:* $\mathbf{mtd}\ \mathbf{m}(caller, label, \mathbf{x})\,\{s\,;\mathbf{return}\,e\}$.

The special identifiers $\mathbf{t}$ *can be seen as variables which may hold only natural numbers and cannot be modified by the program constructs, but only by the semantic rules. Since identifiers* $\mathbf{t}$ *are local to the objects, we extend the category* $\mathbb{I}$ *by attaching another data component* $\mathbf{LabTrans}(L, \mathbb{L})$*. The category* $\mathbb{L}$ *is a pairs category with objects* $|\mathbb{L}|$ *being mappings* $IdFut \rightharpoonup \mathbf{Nat}$.

$$\frac{fresh(n,\rho) \qquad \rho_m(\mathbf{o}') = \mathcal{MS}}{\langle \mathbf{o}|\ \mathbf{t}!\mathbf{o}'.\mathbf{m}(v)\ \rangle \xrightarrow{\mathbf{o}:L=\rho,M=\rho_m \,\ldots\, M=\rho_m[\mathbf{o}'\mapsto\mathcal{MS}\oplus invoke(\mathbf{o},n,\mathbf{m}(v))],\mathbf{o}:L=\rho[\mathbf{t}\mapsto n]} \langle \mathbf{o}|\,\mathsf{nil}\,\rangle}$$

$$\frac{\rho_m(\mathbf{o}) = \mathcal{MS} \qquad invoke(\mathbf{o}', n, \mathbf{m}(v)) \in \mathcal{MS}}{\langle\, \mathbf{o} \mid s\,\rangle \xrightarrow{M=\rho_m \,\ldots\, M=\rho_m[\mathbf{o}\mapsto\mathcal{MS}\ominus invoke(\mathbf{o}',n,\mathbf{m}(v))]} \langle\, \mathbf{o} \mid \mathbf{async}\,(\mathbf{m}(\mathbf{o}', n, v))\,;s\,\rangle}$$

$$\frac{\rho(caller) = \mathbf{o}' \qquad \rho(label) = n \qquad \rho_m(\mathbf{o}') = \mathcal{MS}}{\langle\, \mathbf{o} \mid Ev[\mathbf{return}\,v]\,\rangle \xrightarrow{\mathbf{o}:S=\rho,M=\rho_m \,\ldots\, M=\rho_m[\mathbf{o}'\mapsto\mathcal{MS}\oplus comp(n,v)]} \langle\, \mathbf{o} \mid \mathsf{nil}\,\rangle}$$

$$\frac{\rho(\mathbf{t}) = n \qquad \rho_m(\mathbf{o}) = \mathcal{MS} \qquad comp(n,v) \in \mathcal{MS}}{\langle\, \mathbf{o} \mid \ \mathbf{t}?(\mathbf{x})\,\rangle \xrightarrow{\mathbf{o}:L=\rho,M=\rho_m \,\ldots\, M=\rho_m[\mathbf{o}\mapsto\mathcal{MS}\ominus comp(n,v)]} \langle\, \mathbf{o} \mid \ \mathbf{x} := v\,\rangle}$$

Essential to the above rules is that in each rule only one object term is present, thus capturing the asynchronous method call aspect. Moreover, one can clearly see the production and consumption of the messages.

Rules two and four are dependent on additional program constructions, and thus on their semantics. This is not in the modular spirit. We would achieve the same effect by simulating the two corresponding transition rules (for ***async*** *and assignment) and modify the required local data components directly in the rule above; this means that the second rule would involve the* $\mathbb{T}$ *local category and the last rule would involve* $\mathbb{S}$*. In this way dependency on program constructs is removed, but still the rules depend on the two local label components. This is more preferred in the modular SOS.*

We did not complicate the examples because our aim was only to show how such a semantics is written in the modular SOS style that we take in this work. Essential is the way in which our rules above clearly separate the data from the program code. Just from the rules we can see exactly which data components are manipulated (which are read which are written) by the program at runtime. Moreover, we can clearly identify which data components can be used by dynamic upgrade constructs.

5 Dynamic SOS

Dynamic SOS enriches the modular approach from before in the following ways. The program syntax is enriched with programming constructs denoting update points of various kinds (to be defined by the language developer). The arrow-labeled transition system is enriched by adding new kinds of transitions labeled not with morphisms, but with endofunctors. Thus, the syntax for writing transition rules is enriched to use endofunctors. The label transformer is enriched, and also the label categories that we use. Defining the endofunctors, though, is something we are already familiar with, as we shortly see.

Definition 5.1 (update transition systems) *An* update transition system *(UTS) is a classic transition system* $(\Gamma, L, \rightarrow, T)$ *as in Definition 3.1 where the set of labels is* $L = Mor(\mathbb{A}) \cup Mor(End(\mathbb{A}))$*, with* $End(\mathbb{A})$ *the category of endofunctors on* $\mathbb{A}$*, having* $\mathbb{A}$ *as the single object and endofunctors on* $\mathbb{A}$ *as morphisms.*

A computation *in a UTS is a sequence of transitions s.t. two transitions follow each other* $\xrightarrow{\alpha}\xrightarrow{\beta}$ *only if both the following are respected*

1. *if* α *and* β *are both morphisms in* $\mathbb{A}$ *then they are composable in the same order,* $\alpha;\beta \in Mor(\mathbb{A})$*; or*

2. *if* α *is an endofunctor on* $\mathbb{A}$ *and* β *is a morphism in* $\mathbb{A}$ *then the source* β^s *is the same as the output of the endofunctor* α*.*[3]

We call the transitions labeled with endofunctors, *jumps*.

Definition 5.2 (update label transformers) *Consider a second indexing set* $IndexU$ *disjoint from* $Index$*. The* update label transformer *is defined the same as the label transformer from Definition 3.4, but using the update indexes* $j \in IndexU$*. The* $\mathbf{ULabTrans}(j, \mathbb{U})$ *maps a category* $\mathbb{A}$ *to a product category* $\mathbb{A} \times \mathbb{U}$*, where* $\mathbb{U}$ *is either the trivial category or a pairs category.*

[3] When the transition immediately before an endofunctor α is labeled with a morphism then α is applied to the target of this morphism, otherwise, when it is labeled also with an endofunctor, then α is applied to the output of this endofunctor.

The $\mathbb{U}$ categories are the update components of the labels. Because of the disjointness of the indexing sets, the same get operation from before is still applicable, and existing transition rules are not affected by the addition of an update component. Intuitively, the update label transformer is the label transformer applied in a special way, i.e., to a disjoint set of indexes and to pairs categories.

Modularity is not disturbed, and new data categories may be added with the label transformer in the same way, without any interference with the update components.

The semantics of dynamic software upgrades is given in terms of endofunctors on the big product category. These endofunctors are obtained from combining *basic endofunctors*, which are defined in terms of only some of the data and the update components. To understand how the endofunctors are obtained and how the basic ones should be defined we first give some properties specific to the kinds of categories that we use.

Since the write-only types of data categories (i.e., the monoid categories) are of no use to the program (i.e., the program does not read them when executing) we see no reason to update these at runtime. Therefore, the update functors are to be defined only in correlation with discrete or pairs categories. For such categories the endofunctors have a special property, they are completely defined by their application to the objects of the category only.

Proposition 5.3 *An endofunctor over a discrete or a pair category is completely defined by its application to the objects of the category.*

Proof: Consider a discrete or pair category $\mathbb{A}$ and an endofunctor F for which we know how it is applied to objects in $|\mathbb{A}|$. Consider one morphism $o \overset{\alpha}{\to} o'$, which is uniquely defined by the two objects o, o' (which may also be the same object). The the functor associates to this morphism the following morphism from $\mathbb{A}$: $F(\alpha) = (F(o), F(o'))$ which is the unique morphism from $F(o)$ to $F(o')$, hence respecting the requirements from Definition 2.2 of being a functor. $\square$

This property of our endofunctors essentially says that in order to define one basic endofunctor for a label component we only need to define how each pair of data and update information are to be changed to a new pair of data and update. Essentially, each update functor specifies how the update information from the update component acts on the data information from the data component, changing this and also the information in the update component (e.g., when doing incremental updates using only part of the update information, which disappears after the update operation). All these become clear in the examples below, when we see the specific structure of the objects of the data and update component categories. But abstracting away from this structure, an update endofunctor defines a correspondence between the data before and after some update, for any update information. Therefore we may have many update endofunctors, and in defining the semantics of the update constructs we specify which of these endofunctors to be used.

It remains to understand how to combine such endofunctors from acting on the basic categories to an endofunctor that acts on the composed label category, as given by the label transformers. We essentially make pairs of endofunctors over the products of categories, i.e., product of endofunctors.

Proposition 5.4 (endofunctors as morphisms) *Consider two categories $\mathbb{A}$ and $\mathbb{B}$ with the property of Proposition 4.2(2), like discrete or pair categories and their products. Denote by $\mathbb{E}nd(\mathbb{A})$ the category which has one object $\mathbb{A}$ and as morphisms all the endofunctors on $\mathbb{A}$ with the identity endofunctor as the identity morphism. Define the product of two such categories $\mathbb{E}nd(\mathbb{A}) \times \mathbb{E}nd(\mathbb{B})$ to have one object $(\mathbb{A}, \mathbb{B})$ and morphisms the pairs of morphisms from the two categories.*

Then any morphism $(F_{\mathbb{A}}, F_{\mathbb{B}})$ in the product is an endofunctor on $\mathbb{A} \times \mathbb{B}$ which takes any object $(a, b) \in |\mathbb{A} \times \mathbb{B}|$ to an object $(F_{\mathbb{A}}(a), F_{\mathbb{B}}(b))$. These endofunctors are also completely defined by their application on the objects.

Therefore, the paired endofunctors have the same properties as the basic endofunctors, and their behavior is defined by their component endofunctors.

For the basic update categories that we will use we consider that their underlying set of objects has a special information-less object, which we denote $o_\perp$. The categories that we encountered until now all have such an object, e.g., when the underlying objects are sets then $o_\perp$ is the $\emptyset$; when the underlying objects are partial functions (like with heaps or stores) then $o_\perp$ is the minimal partial function completely undefined. For a category with a single object, like the trivial category, then this is considered to be the $o_\perp$. For a product of categories then the pairing of all the corresponding $o_\perp$ is the minimal object.

The only requirement that we ask of the endofunctors is the following, which intuitively says that once the information-less object is reached then no more change of data objects can be performed; i.e., the endofunctor returns the same object. This is a kind of termination condition where inaction from the functor is required.

Definition 5.5 (no sudden jumps) *An endofunctor F on $\mathbb{D} \times \mathbb{U}$ is said to* have no sudden jumps *iff $F((d, u_\perp)) = (d, u_\perp)$, where $u_\perp \in |\mathbb{U}|$ is the information-less object.*

Notation: For some indexing set $I \subset \mathit{Index}$ (or $I \subset \mathit{IndexU}$) denote by $\mathbb{D}_I$ (respectively $\mathbb{U}_I$) the product category $\times_{i \in I} \mathbb{D}_i$ obtained using the (update) label transformer using the indexes from I attached to the respective category component.

Definition 5.6 (defining update endofunctors) *For some complex product category $\mathbb{D}_I \times \mathbb{U}_K$, define a* basic update endofunctor E *wrt. at least one data component and at least one update component, as a total function over the objects of the corresponding product category $|\mathbb{D}_{I'} \times \mathbb{U}_{K'}|$, with $\emptyset \neq I' \subseteq I$ and $\emptyset \neq K' \subseteq K$. This endofunctor must have no sudden jumps. Extend this basic endofunctor to the whole product category by pairing it with the identity endofunctor on the remaining component categories, as in Proposition 5.4.*

Proposition 5.3 talks only about pairs or discrete categories. Note that product of pairs categories is a pairs category and product of discrete categories is a discrete category. The above definition of basic endofunctors is easy to obtain for such categories. But product of a discrete and a pairs category is special in the sense that there may be pairs of objects with no morphism between them. This issue may appear only when choosing one data component which is pairs and one which is discrete; because the update components are always pairs. Therefore, in the case when in the above definition, for some $i \in I'$, $\mathbb{D}_i$ is a discrete category then the definition of the function must take care that it maps all tuple objects that have o_i to tuple objects which have the same object on the i position; e.g., if $F((o_i, o_{i'}, u_k)) = (o'_i, o'_{i'}, u'_k)$ then $F((o_i, o''_{i'}, u''_k)) = (o'_i, o'''_{i'}, u'''_k)$.

Proposition 5.7 (composing update endofunctors) *For two basic endofunctors when defined on disjoint sets of indexes, their extensions can be composed in any order, and the result of the composition is an endofunctor on the union of the indexing sets, which behaves the same as the product of the two basic endofunctors.*

Proof : Consider a product category $\mathbb{D}_I \times \mathbb{U}_K$ built with the label transformer over the index sets $I \cup K$. Consider two endofunctors E', E'' build over subsets of these indexes, respectively $I' \subset I$, $K' \subset K$ and $I'' \subset I$, $K'' \subset K$. Assume these two subsets are disjoint: $(I' \cup K') \cap (I'' \cup K'') = \emptyset$. Denote $I \setminus I'$ by $\hat{I}'$ and $K \setminus K'$ by $\hat{K}'$, and the same for $\hat{I}''$ and $\hat{K}''$. Without loss of generality, consider the above I', I'', K', K'' to be singleton sets. One endofunctor E' maps objects from $\mathbb{D}_{I'} \times \mathbb{U}_{K'}$, and for two objects $(d'_1, u'_1), (d'_2, u'_2) \in |\mathbb{D}_{I'} \times \mathbb{U}_{K'}|$ with a morphism α' between them, the endofunctor maps $E'(\alpha')$ to some α'_1 between $E'((d'_1, u'_1))$ and $E'((d'_2, u'_2))$.

The basic endofunctors E', E'' are defined only over products of pairs and discrete categories (there may be morphism categories in the big product $\mathbb{D}_I \times \mathbb{U}_K$, but not in the subset product).

Extend each endofunctor from above to the whole category $\mathbb{D}_I \times \mathbb{U}_K$ as in Definition 5.6 by pairing it with the identity endofunctor on the remaining category $\mathbb{D}_{\hat{I}'} \times \mathbb{U}_{\hat{K}'}$; e.g., for E' denote its extension as $\tilde{E}'$ to be the product $E' \times ID_{\hat{I}' \cup \hat{K}'}$. The similar extension for E'' is $\tilde{E}'' = E'' \times ID_{\hat{I}'' \cup \hat{K}''}$. We explicit the indexes a little more: $\tilde{E}' = E'_{I' \cup K'} \times ID_{\hat{I'} \cup \hat{K'}}$ and $\tilde{E}'' = E''_{I'' \cup K''} \times ID_{\hat{I''} \cup \hat{K''}}$. Because of the disjointness condition we know that $I' \subseteq \hat{I}''$, $K' \subseteq \hat{K}''$, $I'' \subseteq \hat{I}'$, and $K'' \subseteq \hat{K}'$. Since the identity endofunctors can be easily seen as products of smaller identity endofunctors, we can rewrite the above endofunctors to: $\tilde{E}' = E'_{I' \cup K'} \times ID_{I'' \cup K''} \times ID_{\hat{I'''} \cup \hat{K'''}}$, with $\hat{I'''} = \hat{I}' \setminus I'' = (I \setminus I') \setminus I''$ which because of the disjointness of I' and I'' we have $\hat{I'''} = I \setminus (I' \cup I'')$, and $\tilde{E}'' = E''_{I'' \cup K''} \times ID_{I' \cup K'} \times ID_{\hat{I'''} \cup \hat{K'''}}$. We had been relaxed with the notation for the products, but care must be taken for the order of the arguments, so one would write $\tilde{E}''$ as $ID_{I' \cup K'} \times E''_{I'' \cup K''} \times ID_{\hat{I'''} \cup \hat{K'''}}$. Since both the primed and double primed indexes do not contain indexes of the monoid categories, all these categories enter under the application of the identity endofunctor $ID_{\hat{I'''} \cup \hat{K'''}}$.

We now make the composition of the two endofunctors $E'' \circ E'$. Pick now two objects from the big category $|\mathbb{D}_I \times \mathbb{U}_K|$:

$$(d'_1, u'_1, d''_1, u''_1, d'''_1, u'''_1) \text{ and } (d'_2, u'_2, d''_2, u''_2, d'''_2, u'''_2).$$

The number of objects in the tuples is not relevant. The morphism between there tuple objects is also tuple of respective morphisms $(\alpha', \beta'', \gamma''')$. Apply now the endofunctor $\tilde{E}'$ which is $E'_{I' \cup K'} \times ID_{I'' \cup K''} \times ID_{\hat{I'''} \cup \hat{K'''}}$, to obtain tuples of objects $(E'_{I' \cup K'}(d'_1, u'_1), d''_1, u''_1, d'''_1, u'''_1)$ and $(E'_{I' \cup K'}(d'_2, u'_2), d''_2, u''_2, d'''_2, u'''_2)$, and morphism $(E'_{I' \cup K'}(\alpha'), \beta'', \gamma''')$. To this one applies the second endofunctor, which is $ID_{I' \cup K'} \times E''_{I'' \cup K''} \times ID_{\hat{I'''} \cup \hat{K'''}}$, to obtain objects $(E'_{I' \cup K'}(d'_1, u'_1), E''_{I'' \cup K''}(d''_1, u''_1), d'''_1, u'''_1)$ and $(E'_{I' \cup K'}(d'_2, u'_2), E''_{I'' \cup K''}(d''_2, u''_2), d'''_2, u'''_2)$, and morphism

$$(E'_{I' \cup K'}(\alpha'), E''_{I'' \cup K''}(\beta''), \gamma''').$$

It is easy to see that for the other composition $E' \circ E''$ we would obtain the same objects and morphism. Moreover, these are independent of the monoid categories that are subject only to the identity endofunctor $ID_{\hat{I'''} \cup \hat{K'''}}$.

From the above one can easily see how one could first make product of the two endofunctors E' and E'' and only the extend this to the whole category, and the result of the application of this product of basic endofunctors results in the same objects and morphisms as the compositions above. □

This last result ensures modularity of the Dynamic SOS. One defines a basic endofunctor for some dynamic upgrade construct, and this is never changed upon addition of other dynamic upgrade constructs and their update categories and related endofunctors. Moreover, the method of *extending* the basic endofunctors with the identity functor on the rest of the indexes ensures modularity for when new data or update components are added by the label transformers.

Much of the work in [28] is concerned with analyzing the program term to automatically insert update statements at the appropriate points in the program where some update is meaningful and would not cause errors. The same analyses can be carried also when the language is given a modular semantics.

Definition 5.8 (updates) *Add to the programming constructs signature Σ_{exe} a disjoint signature Σ_{upd}.*

The exact function symbols of Σ_{upd} will appear later when we treat the two languages PROTEUS and CREOL. Σ_{upd} contains programming constructs intended to be used for marking dynamic upgrade points. Each endofunctor that we define, as some kind of upgrade mechanism, should be matched (using a transition rule) by an update construct.

5.1 Exemplifying DSOS for PROTEUS

The transition rules that we gave for PROTEUS used a label category formed of three components: (1) $\mathbb{S}$ with objects mapping variable identifiers to values, (2) $\mathbb{F}$ with objects mapping function names to definitions of functions as lambda abstractions, (3) $\mathbb{R}$ with objects mapping record identifiers to definitions of records. All tree are pairs categories. In [28] the semantics of PROTEUS keeps all these information in one single structure called *heap*. The separation of this structure that we took does not impact on the resulting semantic object, as one can check against [28, Fig.12].

Four kinds of update information are present in PROTEUS. In this exemplification we treat only two: the updating of bindings and the addition of new bindings to the heap. In [28, Fig.11] this information comes in the form of a partial mapping from top-level identifiers to values (we omit the types). This update information follows the same structure as the heap. At any time point, in the heap we can see the identifiers separated into variables, function names, or record names; the values being either basic values for variables, lambda abstractions containing the function body, and record definitions. It is easy to see that we get the corresponding structures as the objects in our categories $\mathbb{S}$, $\mathbb{F}$, respectively $\mathbb{R}$. In consequence it it clear what the corresponding update categories should be: $\mathbb{U}_{\mathbb{S}}$, $\mathbb{U}_{\mathbb{F}}$, and $\mathbb{U}_{\mathbb{R}}$ are pairs categories containing the same objects as respectively $\mathbb{S}$, $\mathbb{F}$, and $\mathbb{R}$.

In general, we observed that the objects of the update categories are the same as the objects in the corresponding data categories. But this need not always be the case. One example is the information for updating types in PROTEUS which differs from the type environment (which maps type names to types) in the fact that the update data comes as a mapping from type names to pairs of type and type transformer. We are not concerned with types though.

PROTEUS uses a single update construct, which marks points in the program where updates can take place. An ingenious analysis of the program code of PROTEUS can establish at each program point which type names can be upgraded without breaking the type safety. This preliminary analysis of the code labels each program point with a set of *capabilities*, which are the set of type names that cannot be safely upgraded at that point (wrt. the typing system). The semantics then checks if the upgrade information contains any of these types, and the upgrade fails if this is the case. One could use the same information to have *incremental upgrades*, where at each point the upgrade is made only for those type names which are safe, when possible (dependencies between the names in the upgrade information may not allow for such splitting of the upgrade).

In our situation we can apply the same analysis of the program and use upgrade constructs which are labeled with the set of identifiers that can be safely upgraded at that point. Moreover, we separate these upgrade constructs into three kinds, each dealing with variables, functions, or records. Thus, our upgrade signature Σ_{upd} contains:

$$S \ ::= \ \mathbf{update}^{v}\Delta \mid \mathbf{update}^{f}\Delta \mid \mathbf{update}^{r}\Delta \mid \dots$$

where Δ is a set of respectively variables, functions, or record identifiers.

Having identified the upgrade categories above, it remains to define the corresponding endofunctors. Since the endofunctors for our special categories can be given solely by their application on the set of objects, we define one endofunctor for each update category as a function applied to pairs of data/update objects, e.g., from $|\mathbb{S}| \times |\mathbb{U}_{\mathbb{S}}|$. This definitions are readily extracted from the definition of the updating operation in PROTEUS from [28, Fig.13]. Specific to us is that we divided the single update construct of PROTEUS into many, one for each set of identifiers. For each set of identifiers we give a different endofunctor by restricting the definition from [28, Fig.13] to consider only those identifiers and remove them from the update objects. Thus, both the data object and the update object may be changed by an endofunctor. Define an update transition rule as:

$$\frac{}{\langle\, \mathbf{o} \mid \mathbf{update}^{v}\Delta \,\rangle \xrightarrow{E^{v}_{\Delta}} \langle\, \mathbf{o} \mid \mathsf{nil} \,\rangle}$$

with $E^{v}_{\Delta} \in Mor(\mathbb{E}nd(\mathbb{S} \times \mathbb{U}_{\mathbb{S}}))$ an endofunctor on the product category $\mathbb{S} \times \mathbb{U}_{\mathbb{S}}$, defined as in [28, Fig.13] but restricted to only the variable identifiers specified in Δ. For one store object ρ of $|\mathbb{S}|$ and one update object ρ_u of $|\mathbb{U}_{\mathbb{S}}|$ the endofunctor E^{v}_{Δ} changes ρ_u to ρ'_u by removing all the mappings from the variable identifiers appearing in Δ; and changes ρ to ρ' by replacing all mappings from variable identifiers appearing in Δ with the corresponding ones from ρ_u.

The definition of the endofunctors is outside the category theory framework of Dynamic SOS because these depend solely on the objects of the data and update categories and their underlying algebraic structure. In consequence, defining endofunctors requires standard methods of defining functions. This is also the reason for which it was immediate to take the definition from [28, Fig.13] into our setting. The contribution of DSOS is not at this level, but it consists of the general methodological framework that DSOS provides, which gives a unified approach to defining dynamic software upgrades in tight correlation with the normal programming constructs.

5.2 Exemplifying DSOS for dynamic class upgrades in CREOL

A loose correlation between the updating styles of PROTEUS and CREOL can be the following: upgrading of variable names corresponds to upgrading class attributes, upgrading functions to upgrading methods of a class, upgrading type definitions to upgrading classes and their interfaces. The updates in CREOL support all the above, but here we stick to our simple examples where classes have only methods and attributes.

The works on class upgrades [17, 15] introduce extra complexity in the form of *dependencies between upgrades*. In consequence, classes have associated an *upgrade number*. The upgrade numbers are not manipulated by the programming constructs, but only by the update constructs. Add a pairs category $\mathbb{UN}$, which has objects $|\mathbb{UN}| = \mathit{IdClass} \rightharpoonup \mathbf{Nat}$ mappings from class identifiers to natural numbers, as a global component $\mathbf{LabTrans}(UN, \mathbb{UN})$. Thus we have identified the data components which are subject to the upgrade: $\mathbb{C} \times \mathbb{A} \times \mathbb{UN}$ with $\mathbb{C}$ and $\mathbb{A}$ from Example 4.7 holding the methods and attributes from class definitions. Denote this product as $\mathbb{D}$.

We take the same steps as for PROTEUS and identify first the upgrade information. From [17, 15] we identify three components: two holding the actual new code for methods and attributes, and another holding the dependencies. Thus, define one pairs category $\mathbb{UC}$ which has the same objects as the category $\mathbb{C}$: $|\mathbb{UC}| = \mathit{IdClass} \rightharpoonup (\mathit{IdMethods} \rightharpoonup \mathit{MtdDef})$. One such objects ρ_{uc} holds information about which class names need to be upgraded and what is the new information to be used. Another pairs category $\mathbb{UA}$ has objects $\mathit{IdClass} \rightharpoonup A$. We also define one pairs category $\mathbb{UD}$ which has as objects: $|\mathbb{UD}| = \mathit{IdClass} \rightharpoonup (\mathit{IdClass} \rightharpoonup \mathbf{Nat})$. One such object holds upgrade information about which class depends on which versions of which classes. Denote the upgrade categories as $\mathbb{U}_{\mathbb{D}} = \mathbb{UC} \times \mathbb{UA} \times \mathbb{UD}$. Thus, the endofunctors are defined on $\mathbb{D} \times \mathbb{U}_{\mathbb{D}}$, i.e., on tuples of six objects.

We now decide on the update constructs and we take the same approach as for PROTEUS to have incremental upgrades. This is in contrast to [17, 15] where there is no actual update construct, but only update messages floating in the distributed system and holding the upgrade information. We achieve the same results but using the update constructs. Essentially the technique of [17, 15] corresponds to a single update construct, as in PROTEUS, which appears at every point in the program. For incremental upgrades we take $\quad S \ ::= \ \mathbf{update}^c\Delta, \quad$ where Δ

is a set of class identifiers.

The corresponding update transition rule is as before:

$$\frac{}{\langle \mathbf{o} \mid \mathbf{update}^c\Delta \rangle \xrightarrow{E^c_\Delta} \langle \mathbf{o} \mid \mathsf{nil} \rangle}$$

with $E^c_\Delta \in Mor(\mathbb{E}nd(\mathbb{D} \times \mathbb{U}_\mathbb{D}))$ an endofunctor on the product category from above, which is defined following the work in [17, 15]. We need some notation first.

Definition 5.9 (dependencies check) *We define a binary relation* $\subseteq$ *on partial mappings* $\rho, \rho' \in IdClass \rightharpoonup \mathbb{N}$ *as:*
$\rho \subseteq \rho'$ *iff* $\forall \mathbf{C} \in IdClass : \mathbf{C} \in \rho \Rightarrow \mathbf{C} \in \rho' \ \wedge \ \rho(\mathbf{C}) \leq \rho'(\mathbf{C})$.

Define the endofunctor E^c_Δ on $\mathbb{C} \times \mathbb{A} \times \mathbb{UN} \times \mathbb{UC} \times \mathbb{UA} \times \mathbb{UD}$ as follows.

$$E^c_\Delta(\rho_c, \rho_a, \rho_{un}, \rho_{uc}, \rho_{ua}, \rho_{ud}) =$$

$$\begin{cases} \begin{pmatrix} \rho_c[\mathbf{C} \mapsto \rho_c(\mathbf{C})[\rho_{uc}(\mathbf{C})] \mid \forall \mathbf{C} \in \Delta \cap \rho_{uc}], \\ \rho_a[\mathbf{C} \mapsto \rho_{ua}(\mathbf{C}) \mid \forall \mathbf{C} \in \Delta \cap \rho_{ua}], \\ \rho_{un}[\mathbf{C} \mapsto \rho_{un}(\mathbf{C}) + 1 \mid \forall \mathbf{C} \in \Delta \cap (\rho_{uc} \cup \rho_{ua})], \\ \rho_{uc} \ominus \Delta, \\ \rho_{ua} \ominus \Delta, \\ \rho_{ud} \ominus \Delta \end{pmatrix} & \begin{array}{l} \text{if } \forall \mathbf{C} \in \Delta \cap \rho_{ud} : \\ \rho_{ud}(\mathbf{C}) \subseteq \rho_{un} \end{array} \\ \\ (\rho_c, \rho_a, \rho_{un}, \rho_{uc}, \rho_{ua}, \rho_{ud}) & \text{otherwise} \end{cases}$$

The update message used by CREOL is the case here where Δ contains one class identifier and the three update objects also contain this single class identifier. The apparent complication in the definition of the endofunctor comes from the complicated update information that must be manipulated. This has nothing to do with the category theory, but only with the algebraic structures of the underlying objects. It is easy to check that the above endofunctor has no sudden jumps.

We treat the dynamic upgrades of CREOL somehow superficial if we only upgrade the global class definitions (i.e., the types of the objects). Compared to PROTEUS, challenging in the dynamic upgrading mechanism of CREOL is the fact that the distributed and concurrent objects must be upgraded also (i.e., their local attributes), where things become more interesting in the setting of inheritance, i.e., when a super-class is upgraded in a class hierarchy, and objects of a sub-class must be aware of this upgrade. A full analysis of the CREOL upgrades is part of our further work.

Objects are the active unit of computation in a distributed object-oriented setting. In CREOL also the classes are active (and the messages which are terms at the same level as the objects and classes). The upgrade numbers that the classes keep in the category component $\mathbb{UN}$ are used by the objects to upgrade themselves; also objects keep an upgrade number so to be able to detect when their class type has been upgraded. In [17, 15] upgrading of the objects, by getting the new attributes, is done in the rewriting logic implementation through equations, which are unobservable, many steps, and considered atomic (all at once). This implies that all the objects in the system are upgraded at once, and at any single step of computation (i.e., before any rewrite rule application).

6 Conclusion and Further Work

We have built on the modular SOS of [22, 21] a Dynamic SOS framework which is intended to be used for defining the semantics of dynamic software upgrades. At the same time we have given modular SOS definitions for concurrent object-oriented programming constructs, where we defined an encapsulating construction on the underlying category theory of MSOS. The encapsulation can be used also in other places where a notion of localization of the program execution is needed. We have considered two examples of languages with dynamic software upgrades: the C-like PROTEUS, and the concurrent and distributed object-oriented CREOL.

We have ignored typing aspects in this paper, and concentrated only on the semantic definition. The cited papers that investigate forms of dynamic upgrade do thorough investigations into typing issues. These investigations can be readily adopted in Dynamic SOS because they are usually based only on the program terms and on the upgrade informations. Nevertheless, it is interesting to further investigate such typing systems in the setting of DSOS, and we plan to do this for CREOL, where the combination of distributed objects with concurrency and asynchronous method calls with futures, interfaces and inheritance, dynamic binding and behavior types, make the example non-trivial.

The modular aspect of Dynamic SOS (and MSOS) is a good motivation for undertaking a more practical challenge of building a database of programming constructs together with their respective (D)MSOS transition rules. A new programming language would then be built by choosing the needed constructs and their preferred semantics, when more exist (e.g., variables implemented with a single store or with a heap and store). The language developer would then only concentrate on the new programming feature/construct that is under investigation. A few requirements of this database are in order. One is a ready integration of the (D)MSOS rules with a proof assistant like Coq, where the work in [25] is a good inspiration point. Another is the use of a notation format with the possibility of extensible notation style overlays, which would allow the developer to view the semantics in the preferred notation. Also, such a database needs to be maintainable by the community, as with a wiki and a web interface.

References

[1] M. Abadi and G. D. Plotkin. A model of cooperative threads. In *POPL'09*, pages 29–40. ACM, 2009. (see also the LMCS journal version 2010-6(4)).

[2] L. Aceto, B. Bloom, and F. W. Vaandrager. Turning SOS Rules into Equations. *Information and Computation*, 111(1):1–52, 1994.

[3] L. Aceto, W. J. Fokkink, and C. Verhoef. Structural Operational Semantics. In *Handbook of Process Algebra*, chapter 3. Elsevier, 2001.

[4] G. Agha and C. Hewitt. Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming. In *Research Directions in Object-Oriented Programming*, pages 49–74. 1987.

[5] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A Foundation for Actor Computation. *J. Funct. Program.*, 7(1):1–72, 1997.

[6] S. Ajmani, B. Liskov, and L. Shrira. Modular Software Upgrades for Distributed Systems. In *ECOOP'06*, volume 4067 of *LNCS*, pages 452–476. Springer, 2006.

[7] G. M. Bierman, M. J. Parkinson, and J. Noble. UpgradeJ: Incremental Typechecking for Class Upgrades. In *ECOOP'08*, volume 5142 of *LNCS*, pages 235–259. Springer, 2008.

[8] C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In *OOPSLA'03*, pages 403–417. ACM, 2003.

[9] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More Dynamic Object Re-classification: Fickle$_{II}$. *ACM Trans. Program. Lang. Syst.*, 24(2):153–191, 2002.

[10] W. Fokkink, R. J. van Glabbeek, and P. de Wind. Compositionality of Hennessy-Milner logic by structural operational semantics. *Theoretical Computer Science*, 354(3):421–440, 2006.

[11] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[12] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCIA'73*, pages 235–245. William Kaufmann, 1973.

[13] G. J. Holzmann. *The Spin Model Checker*. Addison-Wesley, 2003.

[14] H. Hüttel. *Transitions and Trees: An Introduction to Structural Operational Semantics*. Cambridge Univ. Press, 2010.

[15] E. B. Johnsen, M. Kyas, and I. C. Yu. Dynamic Classes: Modular Asynchronous Evolution of Distributed Concurrent Objects. In *FM'09*, volume 5850 of *LNCS*, pages 596–611. Springer, 2009.

[16] E. B. Johnsen and O. Owe. An Asynchronous Communication Model for Distributed Concurrent Objects. *Software and System Modeling*, 6(1):39–58, 2007.

[17] E. B. Johnsen, O. Owe, and I. Simplot-Ryl. A Dynamic Class Construct for Asynchronous Concurrent Objects. In *FMOODS'05*, volume 3535 of *LNCS*, pages 15–30. Springer, 2005.

[18] M. M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of The IEEE*, 68(9):1060–1076, 1980.

[19] S. Malabarba, R. Pandey, J. Gragg, E. T. Barr, and J. F. Barnes. Runtime support for type-safe dynamic java classes. In *ECOOP'00*, volume 1850 of *LNCS*, pages 337–361. Springer, 2000.

[20] P. D. Mosses. A modular SOS for ML concurrency primitives. Technical report, Dept. of Computer Science, Univ. of Aarhus, 1999.

[21] P. D. Mosses. Foundations of Modular SOS. In *Mathematical Foundations of Computer Science (MFCS'99)*, volume 1672 of *LNCS*, pages 70–80. Springer, 1999.

[22] P. D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

[23] P. D. Mosses and M. J. New. Implicit Propagation in Structural Operational Semantics. *Electr. Notes Theor. Comput. Sci.*, 229(4):49–66, 2009.

[24] B. C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[25] B. C. Pierce, C. Casinghino, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey. *Software Foundations*. e-book (`http://www.cis.upenn.edu/~bcpierce/sf/`), July 2012.

[26] V. R. Pratt. Semantical considerations on floyd-hoare logic. In *IEEE Symposium On Foundations of Computer Science (FOCS'76)*, pages 109–121, 1976.

[27] G. Stoyle, M. W. Hicks, G. M. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: safe and predictable dynamic software updating. In *POPL'05*, pages 183–194. ACM, 2005.

[28] G. Stoyle, M. W. Hicks, G. M. Bierman, P. Sewell, and I. Neamtiu. *Mutatis Mutandis*: Safe and predictable dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 29(4), 2007.

A Examples, Proofs and Additional Results

A.1 Further Examples for Modular SOS of PROTEUS and CREOL

In this section we give the examples for programming constructs that make up the PROTEUS language. Some of these were already considered in the paper, but here they contain some more explanatory text.

Example A.1 (arithmetic and Boolean expressions) *We may add to $\Sigma_{exe}^{3.8}$ arithmetic operators like $+$ and Boolean operators like $=$. Denote these generally as* $\mathbf{op}$. *We may also add as constants the set of natural numbers and the truth values; having a so called* value added syntax*, where the values that the program may take are included in the syntax. Denote these generally as $v \in Val$, with $n \in \mathbb{N} \subseteq Val$ and $b \in \{\mathbf{true}, \mathbf{false}\} \subseteq Val$. The constant values are considered to have sort* Expressions*, denoted usually by $e_i \in E$, and the arithmetic and Boolean operators take expressions and return expressions. Denote this new signature as $\Sigma_{exe}^{A.1}$.*

$$E \ ::= \ v\,(v \in Val) \mid E\,\mathbf{op}\,E$$

We may now add transition rules in the style of small-step SOS, which, using the theory developed thus far, would be presented as:

$$\frac{e_0 \xrightarrow{X} e_0'}{e_0\,\mathbf{op}\,e_1 \xrightarrow{X} e_0'\,\mathbf{op}\,e_1} \qquad \frac{e_1 \xrightarrow{X} e_1'}{v\,\mathbf{op}\,e_1 \xrightarrow{X} v\,\mathbf{op}\,e_1'} \qquad \frac{\mathbf{op} = + \qquad v = v_0 + v_1}{v_0\,\mathbf{op}\,v_1 \xrightarrow{U} v}$$

Example A.2 (assignment and variable declaration) *Having variable identifiers we may add* assignment *statements and* variable declarations*:*

$$S \ ::= \ \mathbf{x} := e \mid d\,(d \in D) \mid \ldots$$

$$E \ ::= \ \mathbf{let}\,d\,\mathbf{in}\,e\ (d \in D) \mid \ldots$$

$$D \ ::= \ \mathbf{var}\,\mathbf{x} := e \mid d_0\,;d_1\,(d_0, d_1 \in D)$$

Using the label category $\mathbf{LabTrans}(1, \mathbb{S})(\mathbf{TrivCat})$, with $\mathbb{S}$ a pairs category of stores, we consider the following rules for the new constructs:

$$\frac{e \xrightarrow{X} e'}{\mathbf{x} := e \xrightarrow{X} \mathbf{x} := e'} \qquad \frac{\mathbf{x} \in \rho}{\mathbf{x} := v \xrightarrow{\{1=\rho\ldots 1=\rho[\mathbf{x}\mapsto v]\}} \mathsf{nil}}$$

$$\frac{e \xrightarrow{X} e'}{\mathbf{var}\,\mathbf{x} := e \xrightarrow{X} \mathbf{var}\,\mathbf{x} := e'} \qquad \frac{\mathbf{x} \notin \rho}{\mathbf{var}\,\mathbf{x} := v \xrightarrow{\{1=\rho\ldots 1=\rho[\mathbf{x}\mapsto v]\}} \mathsf{nil}}$$

$$\frac{d_0 \xrightarrow{X} d_0'}{d_0\,;d_1 \xrightarrow{X} d_0'\,;d_1} \qquad \frac{}{\mathsf{nil}\,;d_1 \xrightarrow{U} d_1} \qquad \frac{d \xrightarrow{\{1=\rho\,...\,1=\rho'\}} \mathsf{nil} \qquad e \xrightarrow{\{1=\rho'\,...\}} v}{\mathbf{let}\,d\,\mathbf{in}\,e \xrightarrow{\{1=\rho\,...\}} v}$$

*Note that the two rules on the left of the last row are in fact redundant because they are subsumed by the same ones for the statements, since declarations are also statements. These are necessary when we do not have the subsorting information. Note also that the last rule for the **let** construct is given in a big-step SOS style. This is intended to illustrate that MSOS is amenable to mixed-step style of semantics.*

Example A.3 (conditional construct) *The conditional construct, of sort* Statement*, taking as parameters a term of sort expression and two terms of sort statement, can be added to any of the signatures from before; here* $\Sigma^{3.10}_{exe} \subset \Sigma^{A.3}_{exe}$.

$$S \;::=\; \mathbf{if}\,e\,\mathbf{then}\,s_1\,\mathbf{else}\,s_2\ \ (e \in E,\ s_1, s_2 \in S) \mid \ldots$$

The semantics does not rely on any particular form of the label categories.

$$\frac{e \xrightarrow{X} \mathbf{true}}{\mathbf{if}\,e\,\mathbf{then}\,s_1\,\mathbf{else}\,s_2 \xrightarrow{X} s_1} \qquad \frac{e \xrightarrow{X} \mathbf{false}}{\mathbf{if}\,e\,\mathbf{then}\,s_1\,\mathbf{else}\,s_2 \xrightarrow{X} s_2}$$

Example A.4 (records) *We add to* $\Sigma^{3.11}_{exe}$ *(any other signature could also be used) a set of record names as constants* $\mathbf{r} \in \mathit{IdRec}$ *and a set of record labels as constants* $\mathbf{l} \in \mathit{IdRecLab}$*, together with a record definition construction and a record projection, thus making* $\Sigma^{A.4}_{exe}$*:*

$$D \;::=\; \mathbf{record}\ \mathbf{r}\,\{\mathbf{l_i} = e_i\}\ \ (\mathbf{r} \in \mathit{IdRec}, \mathbf{l_i} \in \mathit{IdRecLab}, e_i \in E) \mid \ldots$$

$$E \;::=\; \mathbf{r}.\mathbf{l}\,(\mathbf{r} \in \mathit{IdRec}, \mathbf{l} \in \mathit{IdRecLab}) \mid \ldots$$

Record declarations are stored in a new label component $\mathbb{R}$ *which is a pairs category containing objects mapping record identifiers to record terms (where a record term is* $\{\mathbf{l_i} = e_i\}$*). Extend the previous labels category with:*

$$\mathbf{LabTrans}(R, \mathbb{R}).$$

The transition rules for the two new programming constructs are:

$$\frac{\mathbf{r} \notin \rho_r}{\mathbf{record}\ \mathbf{r}\,\{\mathbf{l_i} = e_i\} \xrightarrow{\{R=\rho_r\,...\,R=\rho_r[\mathbf{r} \mapsto \{\mathbf{l_i}=e_i\}]\}} \mathsf{nil}}$$

$$\frac{\rho_r(\mathbf{r}) = \{\mathbf{l_i} = e_i\} \qquad \exists i : \mathbf{l_i} = \mathbf{l} \qquad e_i = e}{\mathbf{r}.\mathbf{l} \xrightarrow{\{R=\rho_r\,...\}} e}$$

The rules above give a "lazy" semantics for records, where the evaluation of the expressions is postponed until the moment of the record label reference. This is similar to the inlining constructs in some programming languages like the Promela [13, Chp.3]. Moreover, the above rules implement a small-step semantics. We leave it as an exercise to give big-step semantics, or to give an eager semantics, where the labels of $\mathbb{R}$ will be guaranteed to associate values to the record labels.

The choice of syntax for the records in Example A.4 is biased by our goal to reach the PROTEUS. Nevertheless, using the theory we presented, one may give semantics to more complex records as those in e.g. [14, Chap.9].

Example A.5 (methods inside objects) *Methods are like functions only that they have a **return** statement which is treated specially.*[4] *We thus add method definitions to $\Sigma_{exe}^{4.8}$:*

$$\begin{aligned} D &::= \ \mathbf{mtd}\,\mathbf{m}(\mathbf{x})\,\{s\} \ \mid \ldots \\ S &::= \ \mathbf{return}\,e \mid \mathbf{m}(e) \mid \ldots \end{aligned}$$

The transition rules for method definitions are simple and use another pairs-like component category $\mathbb{MD}$ for storing method definitions (the same as was done for function definitions) which is added by the label transformer, identified by the index MD, to the local labels category $\mathbb{I}$ that is encapsulated:

$$\frac{}{\mathbf{mtd}\,\mathbf{m}(\mathbf{x})\,\{s\} \xrightarrow{\{MD=\rho_m \ldots MD=\rho_m[\mathbf{m}\mapsto\lambda(\mathbf{x}).(s)]\}} \mathsf{nil}}$$

$$\frac{e \xrightarrow{X} e'}{\mathbf{return}\,e \xrightarrow{X} \mathbf{return}\,e'} \qquad \frac{e \xrightarrow{X} e'}{\mathbf{m}(e) \xrightarrow{X} \mathbf{m}(e')} \qquad \frac{\rho_m(\mathbf{m}) = \lambda(\mathbf{x}).(s)}{\mathbf{m}(v) \xrightarrow{\{MD=\rho_m \ldots\}} (s)[v/\mathbf{x}]}$$

[4]Other programming options are very well possible like having functions evaluate to a value, and thus not use the return statement.