Dynamic Structural Operational Semantics*

Christian Johansen
Institute for Technology Systems, University of Oslo
cristi@ifi.uio.no

Olaf Owe
Department of Informatics, University of Oslo
olaf@ifi.uio.no

We introduce Dynamic SOS as a framework for describing semantics of programming languages that include dynamic software upgrades. Dynamic SOS (DSOS) is built on top of the Modular SOS of P. Mosses, with an underlying category theory formalization. The idea of Dynamic SOS is to bring out the essential differences between dynamic upgrade constructs and program execution constructs. The important feature of Modular SOS (MSOS) that we exploit in DSOS is the sharp separation of the program execution code from the additional (data) structures needed at run-time. In DSOS we aim to achieve the same modularity and decoupling for dynamic software upgrades. This is partly motivated by the long term goal of having machine-checkable proofs for general results like type safety.

We exemplify Dynamic SOS on two languages supporting dynamic software upgrades, namely the C-like PROTEUS, which supports updating of variables, functions, records, or types at specific program points, and CREOL, which supports dynamic class upgrades in the setting of concurrent objects. Existing type analyses for software upgrades can be done on top of DSOS too, as we illustrate for PROTEUS.

A second contribution is the definition of a general encapsulating construction on Modular SOS useful in situations where a form of encapsulation of the execution is needed. We show how to apply this in the setting of concurrent object-oriented programming with active objects and asynchronous method calls.

Contents

*This work was partially supported by the project DiversIoT – Diversification for Resilient and Secure IoT-services, with number 270933/O70, part of the IKTPLUSS program funded by the Norwegian Research Council.

Submitted to:
Journal of Logical and Algebraic Methods in Programming

1 Introduction

With renewed focus on software evolution [25, 27], the interest in dynamic software upgrades has increased over the past few years [26, 13, 8, 23, 6, 40, 7]. Approaches for dynamic upgrades are different in presentation and formalization, making it difficult to compare or combine them, especially since each of these approaches concentrates on some particular programming language or paradigm. The work that we undertake here is to extract the essentials of the operational semantics for dynamic upgrading constructs independent of the programming language or the kind of system paradigm.

Dynamic software upgrades provide mechanisms for upgrading a program at runtime, during its execution, by changing essential definitions used in executing the program, typically by adding or changing definitions of classes, interfaces, types, or methods, as well as modifying or resetting values of variables. Upgrades may be restricted, semantically or syntactically, so that they may only occur in certain states, called *upgrade points*, where upgrading is meaningful or safe. Dynamic upgrades allow a program to be corrected, improved, maintained or integrated with other programs, without stopping and restarting the execution. Dynamic upgrades are inherently different from normal programming mechanisms 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 or programmer.

Semantically, dynamic upgrades change *static data structures*, i.e, the data structures established at the start of runtime such as class tables, function definitions and static typing information. This is in contrast to the semantics for normal programming constructs, which change the *dynamic data structures* (also referred to as the program state), such as the binding of values to program variables (the program store), heaps, message pools, or thread pools.

Thus at runtime we distinguish between (i) the code being executed, (ii) the dynamic data structures, and (iii) the static data structures. Standard operational semantics for programming languages is concerned with the runtime changes of the two former in the context of a given static data structure. The complexity of the program state depends on the complexity of the language, for instance, recursion requires a stack-based store. Thus the operational semantics of a given code construct, such as assignment, may need to be reformulated when the language is enriched. Modular SOS [30] solves this problem by separating the structural layers of a program state.

In particular, Modular SOS (MSOS) promotes a sharp separation of the program code from the *additional data structures*[1] that are manipulated by the semantics. Moreover, complex features like abrupt termination and error propagation can be nicely handled by MSOS, as well as combinations of big-step and small-step semantic styles. We are not constrained in any way by building Dynamic SOS on MSOS. On the contrary, MSOS is not binding

[1] Other works use the term *auxiliary entities*, which we also use interchangeably throughout this paper to refer to the same concept.

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 adopted. The independence of notation is also seen in [31] which presents new notational conventions called IMSOS, intended to be attractive for the developers of programming languages.

We are interested in *dynamic software updates* for imperative languages such as the sequential C-like PROTEUS [40] and *dynamic class upgrades* for object-oriented languages such as the concurrent CREOL [23, 21]. 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 in the literature is given using the same style of structural operational semantics (SOS) as for the other language constructs, often employing elaborate SOS definitions, affecting the basic language elements as well as advanced ones.

Since the nature of dynamic upgrade constructs is different from normal control flow and program execution constructs of a language, we would like these differences to be apparent in the SOS descriptions. For these reasons we develop Dynamic SOS (DSOS).

1.1 An illustrative example

We give a simple example to illustrate some aspects of dynamic software upgrades. More complex examples can be found in e.g., [40, Fig.3&4] from the Linux kernel, [23, Sec.3] for complex class upgrades, or [7, Sec.3].

Consider a class for keeping track of temperatures. The class implements a simple interface for setting and getting the (latest) temperature. With Java-like syntax it could look like

```
interface Temp {
 void setTemp(int t)
 int getTemp()
}
```

```
class TEMP implements Temp {
 int temp;
 void setTemp(int t){temp = t;}
 int  getTemp(){return temp;}
}
```

Assume we would like to update a running system that uses this class such that it can log the history of past *temp* values and be able to calculate the average temperature value. We would like the update to happen without restarting (and recompiling) the system. In CREOL this is done by inserting into the message pool a runtime upgrade message containing upgrade information (using the keyword **upgrade**), which may redefine one or more classes or add new classes and interfaces. With high-level Java-like syntax the upgrade could look like

```
upgrade {
  interface TempStat extends Temp {int avgTemp()}

  class TEMP implements TempStat{
  int[] log = empty;

  void setTemp(int t){temp = t; log.append(t);}

  int avgTemp(){int avg=0;
    for all x in log
      {avg = avg + x; }
    return avg; }
}}
```

The upgrade introduces a new interface `TempStat` and a new version of class `TEMP` augmented with a `log` variable, meant to store the sequence of temperature readings, as well as a new method `avgTemp` for finding the average temperature. The actual logging is done in a changed version of the original `setTemp` method. The `getTemp` method is unchanged. (Note that names are case sensitive, class names are written in upper case and interface names start with an upper case character, while methods and variables start with a lower case character.)

In CREOL, class upgrades are implemented in a distributed fashion letting all the existing objects of class `TEMP` (or a subclass) make their upgrades independently of each other [23]. An update is performed as soon as the current process in the object is suspended or completed. Each upgraded object will start to log temperature values, and will be able to respond to calls to `avgTemp`. Such calls may be generated by objects of upgraded, or new, client classes. Type safety is ensured by static checking of classes and of upgrades [42].

In PROTEUS one may add a declaration of a new variable like for `log`, change the body of a function, like adding the `log.append` statement, add a new method, e.g., the `avgTemp`, and add calls to it, at predefined program points. The upgrades will be more fine-grained than in CREOL, and to control when the updates are applied, PROTEUS requires program update points to be pre-designated by the programmer, while for CREOL the program update points are predefined by the concurrency model.

A challenge for the operational semantics is that such an upgrade as above is changing the class and interface tables, as well as variable and method bindings, in the middle of an execution. In the CREOL case upgrades are handled in the operational semantics by means of message passing. However, a complicating factor of the operational semantics is that CREOL level messages (reflecting method invocations and returns) and upgrade level messages are using the same underlying message passing mechanism.

1.2 Dynamic SOS

We are taking a *modular* approach to SOS, following the work of Mosses [30], thus building on *Modular SOS* (MSOS). This formalism uses notions of category theory, on which our work depends. The Dynamic SOS is intended as a framework for studying semantics of dynamic upgrade programming constructs, and thus existing works on dynamic upgrades should be naturally captured; we exemplify DSOS on the *dynamic software updates* of the language PROTEUS [40] and on the *dynamic class upgrades* of the concurrent object-oriented language CREOL [23, 21]. Since the many works on software updates usually look at type systems and type safety, and since their results also hold over the Dynamic SOS, here we concentrate mainly on the semantic aspect, and only briefly discuss typing aspects in Section 6.

One observation that DSOS makes 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 [40] (which is also taken up in UPGRADEJ [7] and the multi-threaded STUMP [33]). 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. The analysis techniques of [40] for safety after upgrades can be used over DSOS as well.

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 contextual *jump* to a possibly completely different static structure (i.e., 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, being externally provided.

A theoretical motivation for our developments is the close similarity of the transition systems we obtain, with the labelled 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 [4], some of which we hope can be translated to the theory developed here. In particular, the similarity with MSOS is that the states of the transition systems obtained from (D)MSOS 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. The possible connections between the terms and the structure of the labels in MSOS has been recently investigated in [9]. General results that could be investigated (starting from the work presented in [4, 9]) are:

1. generating algebraic semantics [3] from specific forms of the transition rules;
2. compositional reasoning results wrt. dynamic logic [36, 16] using specific forms of transition rules in the style of [15]; or

3. expressiveness results of the programming constructs specified within various rule formats.

A programming language that is developed within the restrictions of the rule format would get such general results for free.

1.3 Modular semantics for concurrent object-orientated languages

A second contribution of this paper is to enhance the theory of Modular SOS with a general notion of *encapsulation* that helps give semantics when a form of encapsulation of the execution is needed, such as in the setting of concurrent and distributed systems. The concurrency model that we treat here, and which is useful in an object-oriented setting, is that of the *Actor model* [17, 5] where each concurrent entity is autonomous, thought as running on one dedicated machine or processor. Therefore, the auxiliary data structures that the standard SOS employs are also localized to each actor. We capture this localization mechanism in a general manner, yet staying in the framework of MSOS, by making a construction on the category theory of MSOS, which we call the *encapsulating construction*, and show it to be in agreement with the other category notions of MSOS. This is worked out in the setting of object-oriented programming with concurrent objects of CREOL. Object-orientation has not been treated before in the MSOS style. Whereas, concurrent ML was treated in [28].

1.4 Structure of the paper

We first give a short listing of some simple notions of category theory that will be used throughout the paper and then introduce Modular SOS in Section 2. We then exemplify, in Section 3, MSOS on constructs found in the PROTEUS language, following a modular style of giving semantics to one programming construct at a time. In the end, the language and its semantics are formed by summing up the needed syntactic constructs with their respective MSOS semantic elements and rules. In Section 4.1 we introduce the encapsulating construction and use it in Section 4.2 to give modular semantics to concurrent object-oriented constructs found in the CREOL language. Both languages have dynamic upgrading constructs which are given semantics in Section 5.1 respectively Section 5.2. The main part of Section 5 is where we develop the Dynamic SOS theory, our main contribution.

2 Modular Structural Operational Semantics

The usual structure of papers on programming languages would include a section that introduces the *syntax* of the language studied, which would then be followed by a section describing the semantics. This is opposed to how DSOS and MSOS propose to develop (semantics of) programming languages. In DSOS we give semantics to a single programming construct, independently of any other constructs (as one can later see through the examples that we give). To define a programming language one puts together the syntactic constructs and the respective semantic rules. Such an approach is particularly appealing when developing a programming language assisted by a theorem prover (e.g., [35]). A main goal of the modular approach is to ensure that once the semantics has been given to one programming construct, it does not need to be changed in the future, when adding new programming constructs. This will be illustrated throughout our presentation.

Moreover, it is easy to work within different notational conventions. Translations between these notations are possible because of the common underlying theory provided by the MSOS and its category theory foundations. Nevertheless, these categorical foundations are transparent to the one giving semantics to programming languages. Standard notational conventions can be adopted for MSOS, but the methodology changes to a modular way of thinking about the semantics. The independence of notation can be seen in [31], which presents new notation conventions called IMSOS, intended to be more attractive to the designers of programming languages.

We recall briefly some standard technical notions that will be used throughout this paper. Our notation stays close to that of [30] for the MSOS related notions and to that of [34] 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 which we denote by α^s and α^t. A category is required (i) to have* identity *morphisms id_o for each object o, satisfying an identity law for each morphism with source or target in that object; and (ii) composition of any two morphisms α and β, with $\alpha^t = \beta^s$, exists (denoted $\beta \circ \alpha$, 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:*

$$F(id_o) = id_{F(o)} \quad \text{and} \quad F(\alpha\beta) = F(\alpha)F(\beta).$$

A functor $F : \mathbb{A} \to \mathbb{A}$ is called an endofunctor *applied to $\mathbb{A}$ (or on $\mathbb{A}$). Define $End(\mathbb{A})$ the* category of endofunctors on $\mathbb{A}$*, having $\mathbb{A}$ as the single object and endofunctors on $\mathbb{A}$ as morphisms.*

Modular SOS generates *arrow-labelled transition systems*, cf. [29], where the transitions are labelled with morphisms (arrows) from a category.

Definition 2.3 (ATS) *An* arrow-labelled transition system *$(\Gamma, Mor(\mathbb{A}), \to)$ is formed of a set of* states *$t_i \in \Gamma$ and transitions $\to \subseteq \Gamma \times Mor(\mathbb{A}) \times \Gamma$ labelled by morphisms $\alpha \in Mor(\mathbb{A})$ from a category $\mathbb{A}$. A* computation *in an ATS is a sequence $t_0 \xrightarrow{\alpha_0} t_1 \xrightarrow{\alpha_1} t_2 \ldots$ s.t. for any $t_i \xrightarrow{\alpha_i} t_{i+1} \xrightarrow{\alpha_{i+1}} t_{i+2}$ the two morphisms are composable in $\mathbb{A}$ as $\alpha_{i+1} \circ \alpha_i \in Mor(\mathbb{A})$.*

Notation 2.4 *Since in an ATS transitions $\xrightarrow{\alpha}$ are labelled 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. When the source and target objects of the morphism α are needed we make them explicit on the transition as $\xrightarrow{\{\alpha^s, \alpha^t\}}$.*

One goal with ATS and MSOS is to have as states *only program terms*, without the additional semantic data that an executing program may use, like stores or heaps. The additional data and the way the program manipulates it is captured by the morphisms which are labelling the transitions of the ATS. This goal is related to e.g.:

1. typing systems where the program syntax alone is under analysis;
2. Hoare logic where Hoare rules are defined for program terms only (with the pre- and post-conditions being the ones talking about the stores/heaps);
3. process algebras with process terms as the states and their observable behaviour as labels on transitions.

When giving semantics to programming languages we establish an initial multi-sorted signature defining the programming constructs of interest. This signature may be enriched upon future developments of the language with new programming constructs. The closed program terms built over this signature constitute the configurations of the arrow-labelled transition systems. Any additional structure/data (like heaps or stores) needed when giving semantics to these constructs, are objects in special categories from which we take their morphisms as transition labels.

Definition 2.5 (basic label categories) *The following three kinds of categories, called* basic label categories*, are used 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 pairs category *is a category which has one unique morphism between every two objects (i.e., in each direction).*

- ***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 read-only variables. Pairs categories correspond to additional data of a read/write type, like stores. Each store appears as one object in the category. The morphisms between two stores represent how a store may be modified by the program when executed. We take a general view where a program may change a store in radical ways, therefore, we have morphisms between every two stores. Monoid categories correspond to write-only type of data, like observable information emitted during the execution of the program, or messages sent between communicating processes.

Example 2.6 *To build a monoid category we pick an underlying set of actions (or events) which will make a monoid of strings over this alphabet, with the empty string as the identity morphism. We can build a discrete or a pairs category by picking some underlying set of objects. One standard example of a pairs category $\mathbb{S}$ has as objects stores: $|\mathbb{S}| = IdVar \rightharpoonup Val$, 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).*

When several additional data are needed to define the semantics, we use *complex label categories* obtained by making product of basic label categories. 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. Complex labels are built using the following construction, which attaches an index to each label component. This will offer the possibility to uniquely identify each component from a complex label using the associated index. This also provides a modular way of extending the label categories.

Definition 2.7 (label transformers) *Let I_L be a countable set of indexes, $\mathbb{B}$ a basic label category, and $\mathbb{A} = \prod_{j\in J\subset I_L} \mathbb{A}_j$ a product category which is $\mathbf{1}$ the trivial category*[2] *when $J = \emptyset$. A* label transformer $\mathbf{LT}(i,\mathbb{B})$*, with $i \in I_L \setminus J$, maps $\mathbb{A}$ to the product category $\mathbb{A}\times\mathbb{B} = \mathbf{LT}(i,\mathbb{B})(\mathbb{A})$, and associates a partial operation*

$$get : Mor(\mathbb{A}\times\mathbb{B}) \times I_L \to (\cup_j Mor(\mathbb{A}_j)) \cup Mor(\mathbb{B})$$

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:

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

Notation 2.8 *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 label category, we may also refer to its source and target objects (when relevant, like in the case of discrete or pairs categories) as $\alpha.i^s$ respectively $\alpha.i^t$.*

Now we proceed to define how operational rules look like in this setting.

Definition 2.9 (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;*

[2] The trivial category has a single object and only the identity morphism for it.

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

Definition 2.10 (rules) *We call $t \xrightarrow{\alpha} t'$ a* transition literal *(or transition schema), with t,t' program terms, possibly containing meta-variables (i.e., these are program schemes). A transition schema is* closed *iff t,t' are, i.e., do not contain meta-variables. The α is a specification of a set of morphisms allowed as labels of this transition schema (see Notation 2.12). A transition* derivation rule *is of the form H/l with H a set of transition literals, called the* premises*, and l is a single transition literal, called the* conclusion.

Definition 2.11 (generated ATS) *Semantics of a program P is defined as the* generated arrow-labelled transition system *that has as states closed program terms, as initial state the program P, and as transitions all the closed transitions generated by exhaustively instantiating the derivation rules.*

Notation 2.12 (morphisms on transitions) *When writing literals we use the following notation for the labels. We write $t \xrightarrow{\{\alpha.i^s\dots\alpha.i^t\}} t'$ to mean that the morphism α is a tuple where the label 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 to the left of the three dots, and targets to the right. In one transition we may refer to several components, e.g.: $t \xrightarrow{\{\alpha.i^s,\alpha.j\dots\alpha.i^t\}} t'$. 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. An even more terse notation may simply drop all references to α and keep only the indexes, thus the last example becomes $t \xrightarrow{\{i=o,j=h\dots i=o'\}} t'$. 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. When side conditions are needed in a rule, , we write these on top of the line.*

The goal of *modularity* is to have rules defined once and for all, meaning that when a new programming construct is added and the new rules for it need to refer to new auxiliary semantic entities, i.e., to enlarge the old label category, then the old rules need not be changed. This is made precise by the essential result of [29, Prop.1].

Intuitively, this result says that any transition defined using the old rule system, i.e., labelled with some α from some category $\mathbb{A}$, is found in the new arrow-labelled transition system, over a new category $\mathbf{LT}(i,\mathbb{B})(\mathbb{A})$, 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 the old rules only 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 refers only to components in $\mathbb{A}$, where the dots notation makes all other components contribute only with the identity morphism.

Theorem 2.13 ([29, Proposition 1]) *Let $\mathbb{A}$ be a category constructed using the label transformers $\mathbf{LT}(j,\mathbb{B}_j)$ for some basic label categories $\mathbb{B}_j$ of the three kinds defined before, with $j \in J \subset Index$. Consider a set of rules R which specifies an ATS over $\mathbb{A}$, where the rules in R refer to only indexes from J. Let the category $\mathbb{A}' = \mathbf{LT}(i,\mathbb{B}_i)(\mathbb{A})$, 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 $\xrightarrow{\alpha}\xrightarrow{\beta}\dots$ specified by R over $\mathbb{A}$, a corresponding computation $\xrightarrow{\alpha'}{}'\xrightarrow{\beta'}{}'\dots$ over $\mathbb{A}'$, and vice versa.

Proof sketch: The label transformer $\mathbf{LT}(i,\mathbb{B}_i)$ 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 $\mathbb{A}\times\mathbb{B}_i$. □

3 Exemplifying MSOS for the PROTEUS language

Normally, for exemplifying how the theory of Modular SOS is applied, it would be advised to use a minimal set of programming constructs. However, we want the theory to appeal to practitioners that develop programming languages. We therefore consider various common programming constructs, without being concerned about redundancy. Moreover, the constructs that we treat in this section will sum up to the programming language PROTEUS [39]. We use though slightly different constructs, closer to those found in the CREOL language [23], which we treat in Section 4. Our style of giving semantics in this section is incremental, one construct at a time. While giving the semantics we deliberately want to be free from any specific notation convention; i.e., we want to convey the concepts of modular semantics, and not adhere to a particular established way of giving SOS semantics to programming languages. We afford to do this because of the modular framework.

Throughout the paper we work with what is sometimes called *value-added syntax*, where the values that program constructs work with are included in the language syntax as constant symbols. Denote these generally as $v \in Val$, with $n \in \mathbb{N} \subseteq Val$ and $b \in \{\mathbf{true}, \mathbf{false}\} \subseteq Val$. The $\mathsf{nil} \in Val$ is seen as a special value that statements take when finished executing. The values are considered to have sort *Expressions*, denoted usually by $e \in E$.

3.1 No labels for sequential composition

Consider a *sorted* signature $\Sigma_{3.1}$ consisting of the following programming constructs, forming a single sort *Statement*:

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

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*.

Remark 3.1 (numbering the signatures) *We use a subscript to number the different signatures that we construct. We use as number the reference of the respective subsection where the signature is defined.*

We define the following transition rules:

$$\frac{}{\mathbf{skip} \xrightarrow{\{\ldots\}} \mathsf{nil}} \qquad \frac{s_1 \xrightarrow{X} s_1'}{s_1\,;s_2 \xrightarrow{X} s_1'\,;s_2} \qquad \frac{}{\mathsf{nil}\,;s_2 \xrightarrow{\{\ldots\}} s_2}$$

where the special label variable X stands for *any morphism*, and the label $\{\ldots\}$ stands for *any identity morphism*. These rules do not specify label categories because any category can be used. This means that no additional data is needed by the respective two programming constructs. Moreover, the identify morphisms capture naturally the notion of *unobservable* transitions since they just “copy” the data represented by the objects.

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 the identity morphism on the current object in the current category of labels, whichever this may be. We also consider to have the standard arithmetic and Boolean operators which take expressions and return expressions. (See technical report [37] for a detailed example.)

3.2 Read-only label categories and a let construct

We add a set of variable identifiers as constant symbols, and denote these by $\mathbf{x} \in IdVar$. Variable identifiers have sort *Expressions*. We also include a **let** construct usually found in functional languages. Let these make a signature $\Sigma_{3.2}$, which can be added to any other signature.

$$e \ ::= \ \mathbf{x} \ \mid \mathbf{let\,var\ x} := e'\,\mathbf{in}\,e \ \mid \ldots$$

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 $\mathbf{LT}(S, \mathbb{S})$ to any category of labels, depending on the already chosen programming constructs and transition rules; in our case to the trivial category, since no specific label components were used until now. Instead of using natural numbers as indexes we use symbols. However, notational decisions are relative to the user, and our notation choices from this paper can safely be overridden.

The transition rule corresponding to the variable identifiers is:

$$\frac{\rho(\mathbf{x}) = v}{\mathbf{x} \xrightarrow{\{S=\rho\,\ldots\}} v}$$

The rule defines a transition between terms $\mathbf{x}$ and v, labelled with a morphism satisfying the condition that the label component with index S has as source an object $\rho \in |\mathbb{S}|$ that maps the variable identifier to the value v. Because the category $\mathbb{S}$ is discrete, we do not specify the target object explicitly since it is the same as the source object specified on the label, i.e., an identity morphism is used. Any other possible label components, if and when they exist, contribute with an identity morphism (symbolized by the three dots). In consequence, since all morphism are identity, this transition is unobservable. 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 only some particular identity morphisms. Note that the rules from Section 3.1 are unaffected by the fact that we have changed the label category. Neither will future rules be affected.

The semantic rules for **let** are given in a small-step style using capture-avoiding substitution.

$$\frac{e' \xrightarrow{X} e''}{\mathbf{let\,var\,x} := e'\,\mathbf{in}\,e \xrightarrow{X} \mathbf{let\,var\,x} := e''\,\mathbf{in}\,e} \qquad \frac{}{\mathbf{let\,var\,x} := v\,\mathbf{in}\,e \xrightarrow{\{\ldots\}} e[v/\mathbf{x}]}$$

3.3 Changing label categories from read-only to read/write for assignments

Having variable identifiers we may add *assignment* statements and *variable declarations* as $\Sigma_{3.3}$, which would include $\Sigma_{3.2}$.

$$d \ ::= \ \mathbf{var\,x} := e \mid \ldots \qquad s \ ::= \ \mathbf{x} := e \mid d \mid \ldots$$

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. In consequence, the new label category is made using the label transformers exactly as before, only that when adding the component with the index S we add $\mathbb{S}$ as a pairs category. All the rules from before use the identity morphisms on the objects. The new rules that we add use proper pair morphism, i.e., referring to both the source and the target stores of the morphism.

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

The requirement above the line in the second rule can be ensured by the typing system, and thus can be removed. This is even desired when we want the rules to be in a standard rule format [4].

3.4 Functions

Consider *function identifiers* as constants denoted by $\mathbf{f} \in \mathit{IdFun}$, and *function definitions* and *function applications*, in the signature $\Sigma_{3.4}$ below. This may be added to any signature that includes variable identifiers, like $\Sigma_{3.3}$.

$$d \ ::= \ \mathbf{fun}\,\mathbf{f}(\mathbf{x})\,\{s\} \mid \ldots \qquad\qquad s \ ::= \ \mathbf{f}\,e \mid \ldots$$

Function declarations are stored in a new label component which is a pairs category[3] containing objects which associate function identifiers to lambda terms. 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 $\mathbf{LT}(F,\mathbb{F}) \circ \mathbf{LT}(S,\mathbb{S})$. Since variable identifiers are needed, the stores component is added as well.

Another semantics, like that of [40], may want to consider these two as a single store-like data structure. In this paper we prefer to use disjoint structures when possible. At an implementation stage one could merge these two kinds of stores into one, and take care of differentiating the variable identifiers from the function identifiers correctly.

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. This is typical for reduction semantics, as in [40]; however we could also use evaluation contexts, e.g., as done in [30]. We exemplify the use of evaluation contexts in Section 4.2.4 for the semantics of threads.

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

3.5 Records

We add to $\Sigma_{3.4}$ 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 two language constructs for *record definition* and *record projection*, thus making $\Sigma_{3.5}$:

$$d \ ::= \ \mathbf{record}\ \mathbf{r}\,\{\mathbf{l_i} = e_i\} \mid \ldots \qquad\qquad e \ ::= \ \mathbf{r}.\mathbf{l} \mid \ldots$$

Record definitions 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 $\{\mathbf{l_i} = e_i\}$, with i ranging here over the list of record elements). Extend the previous labels category with: $\mathbf{LT}(R,\mathbb{R})$. The transition rules for the two new programming constructs are:

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

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

The rules above give a "lazy" semantics for records, where the evaluation of the expressions is postponed until the record label is referenced. This is similar to inlining constructs, as e.g. in the Promela [18, ch.3]. Moreover, these rules implement a small-step semantics. Big-step or eager semantics could also be given.

The choice of syntax for the records is biased by our goal to reach PROTEUS. Nevertheless, using the theory we presented, one may give semantics to more complex records as those in e.g. [19, Chap.9].

[3] Normally, the program at runtime just inspects the function definitions, therefore we could consider using a read-only, discrete, category label component. However, we are using above function definitions as programming constructs. Their semantics is exactly to change the stored definitions of functions.

3.6 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 $\Sigma_{3.3} \subset \Sigma_{3.6}$.

$$s \ ::= \ \mathbf{if}\, e\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2 \mid \ldots$$

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

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

3.7 Comparison with Proteus

By now we have reached the language Proteus of [40] (omitting reference constructs). We add the upgrade construct in Section 5.1. We have used single variable identifiers above, but this can be easily generalized to lists. Moreover, since we investigate only semantic aspects in this paper (i.e., no typing systems), we assume only syntactically correct programs, including static typing. Discussions about typing over MSOS and DSOS are relegated to Section 6.

The transition rules that we gave for Proteus used a label category formed of three components: $\mathbb{S}$, $\mathbb{F}$, and $\mathbb{R}$. In [40] 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 the resulting semantic object obtained for Proteus in [40, Fig.12].

Proposition 3.2 (conformance with Proteus semantics) *Considering reductions $\Rightarrow$ to be either a compilation or an evaluation step from [40, Fig.12], and the transitions $\xrightarrow{\alpha}$ obtained with the MSOS rules for* Proteus*, we have that*

$$\Omega, H, e \Rightarrow \Omega, H', e' \ \textit{ iff } \ e \xrightarrow{\alpha} e' \textit{ with}$$

$$\alpha^s = (\rho_s, \rho_f, \rho_r), \alpha^t = (\rho'_s, \rho'_f, \rho'_r), H = \rho_s \cup \rho_f \cup \rho_r, H' = \rho'_s \cup \rho'_f \cup \rho'_r.$$

When we add types in Section 6 then the typing environment Ω may change and will be captured by the types label $\mathbb{TY}$ on the morphisms: $\Omega = \rho_{ty}, \Omega' = \rho'_{ty}$.

Proof: The proof of this proposition essentially uses the relation between standard labelled transition systems and the arrow-labelled transition systems of the MSOS [30, Prop.3&4]. Here we are looking at the particular rules of Proteus. It is not difficult to see that the changes (and inspections) to the heap that are made in the original rules of [40, Fig.12] are matched by the ones mentioned on the arrows of the MSOS rules given above. □

4 Modular SOS for the concurrent object-oriented language Creol

We will show how to give semantics in a modular style to concurrent object-oriented constructs as used by the language Creol. For this we define a new "encapsulating mechanism" in next section.

4.1 Modular encapsulation for Concurrent Object-Orientation

We focus here on the concurrency notion from the Actor model [5] which has proved well suited for the object-oriented languages. In this setting concurrent objects communicate through asynchronous method calls and have their own execution unit (like a virtual CPU), thus having standard programming constructs be run *inside* the 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.

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. We want that definitions of new semantic rules would 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 (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 diagram on the right commutes.*

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

Definition 4.2 (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). Other categories may be used if one needs to capture relations between the objects, like ownership. The intuition is that each functor attaches one data object to each programming object identifier, thus capturing one snapshot of the private data of all objects. We have access to these encapsulated data by taking the appropriate identifier, i.e., $F(o)$ is the data encapsulated in the programming object identified by o.

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 label categories.

Proposition 4.3 *When the encapsulating construction is applied to $\mathbb{A}$ which is built only with discrete or pairs 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 diagram on the right commutes.

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

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 $id_{F(o)}$. The the diagram becomes the one to the right, which clearly commutes for any natural transformation.

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

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., $\eta = \{(F(o),F'(o)) \mid o \in |\mathbb{O}|\}$. $\square$

The fact that the morphisms of $\mathbb{E}$ are identified by the objects on which they act allows us to work in the notations with the pair of objects instead of the morphism. In the encapsulating construction we may also refer to the morphisms, as 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. In consequence, we are free to use the get operation to refer to a particular morphism component of a $\eta(o)$, i.e., we may write $\eta(o).i$ or any other preferred notation like $o.i$ or $o \mapsto i$ or $\langle o \mid i\rangle$ or $o : i$.

These intuitions hold also when monoid categories are part of the labels. In this case there are multiple natural transformations between two functors.

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 spirit of MSOS, 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 Subsection 4.2.6 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 a different set of identifiers. This has application in languages with object groups, like ABS [22], where objects execute inside a group.

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 rules that 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. This aspect becomes apparent when treating *threads* in Subsection 4.2.4. Henceforth we denote the encapsulated (or local or internal) category by $\mathbb{I}$ when its components are irrelevant.

4.2 Modular SOS for concurrent object-orientation

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 Subsection 4.2.1.

4.2.1 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.

$$O \ ::= \ \langle \mathbf{o} \mid s \rangle$$

This signature $\Sigma_{4.2.1}$ 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 labelled from a category constructed using the encapsulating construction applied to some appropriate $\mathbb{I}$: $\mathbb{E} = \mathbf{Enc}(\mathbb{O},\mathbb{I})$, where $|\mathbb{O}| = IdObj$. Since any of the constructs before can be run inside the object construction, therefore we encapsulate the category that we built before. Thus, the label category that we use in the rules for the object construction below would be

$$\mathbf{Enc}(\mathbb{O},\mathbf{LT}(F,\mathbb{F})(\mathbf{LT}(S,\mathbb{S})(\mathbf{LT}(R,\mathbb{R})(\mathbf{1})))).$$

We give one transition rule that *encapsulates* any transition at the level of the statements inside the objects.

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

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, whereas 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 $X \in Mor(\mathbb{I})$.

4.2.2 Systems of objects

Objects may run in parallel, thus forming systems of distributed objects. 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 rules:

$$\frac{obj_1 \xrightarrow{X} obj_1'}{obj_1 \parallel obj_2 \xrightarrow{X} obj_1' \parallel obj_2} \text{(INT-1)} \qquad \frac{obj_2 \xrightarrow{X} obj_2'}{obj_1 \parallel obj_2 \xrightarrow{X} obj_1 \parallel obj_2'} \text{(INT-2)}$$

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:

$$\frac{\langle \mathbf{o_1} \mid s_1 \rangle \xrightarrow{\mathbf{o_1}:X} \langle \mathbf{o_1} \mid s_1' \rangle \qquad obj_2 \xrightarrow{\eta} obj_2'}{\langle \mathbf{o_1} \mid s_1 \rangle \parallel obj_2 \xrightarrow{\eta[\mathbf{o_1}:X]} \langle \mathbf{o_1} \mid s_1' \rangle \parallel obj_2'} \text{(NON-INT)}$$

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.

4.2.3 Methods inside objects

Methods are like functions only that they have a **return** statement which is treated specially.[4] We thus add method definition and invocation as $\Sigma_{4.2.3}$:

$$d \ ::= \ \mathbf{mtd}\ \mathbf{m}(\mathbf{x})\{s\} \mid \ldots$$

$$s \ ::= \ \mathbf{return}\, e \mid \mathbf{m}(e) \mid \ldots$$

The transition rules for methods are simple and use another pairs label 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:

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

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

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

We do not complicate the presentation more because our aim is only to exemplify how the CREOL language can be given a MSOS style of semantics, using the encapsulation construction.

4.2.4 Threads

We take the model of threads studied in [1, 2] and consider the following programming constructs of sort statement in a signature $\Sigma_{4.2.4}$ which normally would include also other constructs for statements from before:

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

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 $\mathbf{LT}(T,\mathbb{T})$. The label category used to give the transition rules for statements becomes now:

$$\mathbf{LT}(T,\mathbb{T})(\mathbf{LT}(R,\mathbb{R})(\mathbf{LT}(F,\mathbb{F})(\mathbf{LT}(S,\mathbb{S})(\mathbf{1})))).$$

We need more algebraic structure for the thread pools, 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 **yield**, which needs the whole program term that follows it, we give semantics to threads using *evaluation contexts*. The MSOS is perfectly suited for describing semantics using evaluation contexts. One may define rules for a programming construct both using evaluation contexts and without; and then pick the preferred rules. An essential result is to show that both sets of rules generate the same arrow-labelled transition system.

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 giving alternative rules using evaluation contexts, we prefer to give the following rule, and remove the two rules for sequential composition from Subsection 3.1. 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].

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

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

4.2.5 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.

Class identifiers are introduced from a set *IdClass*, usually written as **C**. Class definitions include *method definitions* and *attribute definitions*:

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

$$d \ ::= \ \mathbf{class\,C}\,\{At;M\} \mid \dots \qquad s \ ::= \ \mathbf{x} := \mathbf{new\,C} \mid \mathbf{m}(e) \ \mid \dots$$

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., $\rho_c : \mathit{IdClass} \rightharpoonup (\mathit{IdMethods} \rightharpoonup \mathit{MtdDef})$. Objects $\rho_a \in |\mathbb{A}|$ are mappings $\mathit{IdClass} \rightharpoonup At$. The encapsulation is a global component of its own, to which the label transformer associates index E. The transition rule for class definitions is:

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

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.[5] 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.[6] The rule for object creation is:

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

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. However, for our purposes, 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.

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

[5] 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 Subsection 4.2.3.

[6] This kind of objects have so simple structure that it may look awkward to have such a category defined on them; but it is perfectly fine.

4.2.6 Asynchronous method calls as in CREOL

We take the model of asynchronous method calls from [21] and consider two programming constructs for calling a method and reading the result of the completion of a call:

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

where $\mathbf{t} \in \mathit{IdFut}$ are special identifiers used for retrieving the result of the method call. This mechanism has been studied as "futures" in programming languages [24, 14, 12]. Denote this signature $\Sigma_{4.2.6}$, which can be added to any previous signature.

The asynchronous method calls, as discussed in [21], work with asynchronous message passing, as in the Actor 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 $\mathbf{LT}(M,\mathbb{M})$ is applied at least to an encapsulating category. Similarly to the thread pools, define set operations $\oplus$ and $\ominus$ to add and remove messages from any set $\mathscr{M}\mathscr{S} \in 2^{\mathit{MsgTerm}}$. For our exemplification purposes the messages are of the form: $\mathit{invoke}(\mathbf{o},n,\mathbf{m}(v))$ and $\mathit{comp}(n,v)$, 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 **return**: $\mathbf{mtd}\ \mathbf{m}(\mathit{caller},\mathit{label},\mathbf{x})\,\{s\,;\mathbf{return}\,e\}$.

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 $\mathbf{LT}(L,\mathbb{L})$. The category $\mathbb{L}$ is a pairs category with objects $|\mathbb{L}|$ being mappings $\mathit{IdFut} \rightharpoonup \mathbf{Nat}$.

$$\frac{\mathit{fresh}(n,\rho) \qquad \rho_m(\mathbf{o}') = \mathscr{M}\mathscr{S}}{\langle \mathbf{o} \mid \mathbf{t}!\mathbf{o}'.\mathbf{m}(v)\,\rangle \xrightarrow{\mathbf{o}:L=\rho,M=\rho_m \dots M=\rho_m[\mathbf{o}'\mapsto \mathscr{M}\mathscr{S}\oplus \mathit{invoke}(\mathbf{o},n,\mathbf{m}(v))],\mathbf{o}:L=\rho[\mathbf{t}\mapsto n]} \langle \mathbf{o} \mid \mathsf{nil}\rangle}$$

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

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

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

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. The freshness of n in ρ, that is required in the first rule, can be obtained in various ways, which only complicate rules, and we decide to leave these details out of this presentation.

Remark 4.4 *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.*

There are several variations on giving semantics to asynchronous method calls; the above is just our choice. Other choices can be to have a global store where the values that are returned by the call are kept and retrieved by the caller (not using the completion message as we do above). Other choices do not necessarily block on a read, as we do in the last rule above, but put the waiting process in the thread pool.

5 Dynamic SOS

To give intuitions for Dynamic SOS consider the program term as acting on a data structure during its execution, like a store or a heap. Classical operational semantics describes how each programming construct changes these data structures (or uses the information stored in them). The dynamic upgrades can be seen as coming from outside the program, being controlled by an external user through the upgrade information. A program needs to be analysed in the presence of such dynamic upgrades.

DSOS considers that there is a separate data structure containing information about upgrades. This upgrade data structure is changed by the external user at any point in the execution of the program, and the program may only inspected it. The program can decide at which points in the execution it is safe to do an upgrade. The upgrade operation takes information from the upgrade data and changes the data structures that the program maintains. Therefore, this may change the behaviour of the program.

Since the upgrade points are decided by the program, upgrade programming constructs can be added to the language. A programmer can use these, or a tool can detect program points, and insert such upgrade constructs in the code as necessary. The semantics of an upgrade construct essentially describes how the upgrade changes the data structures of the program.

These ideas are simple and capture only how the semantics of upgrades should be thought and defined. Complications may appear in the definition of the actual update functions of the data structures, as well as in the analysis technique of the programming language for detecting the upgrade points. These also interact with the typing system. Much of the related works on dynamic upgrading constructs [26, 13, 8, 23, 6, 40, 7] focus on these aspects, which are usually developed on top of the semantics. We discuss typing aspects in Section 6.

Dynamic SOS builds on the modular approach from the previous sections by incorporating the following aspects.

1. The arrow-labelled transition system is enriched by adding new kinds of transitions labelled not with morphisms, but with endofunctors.
2. In consequence, the syntax for writing transition rules is enriched to use endofunctors.
3. The label transformer is enriched accordingly, and also the label categories that we use. Defining the endofunctors, though, is something we are already familiar with, as we shortly see.
4. The program syntax is assumed to have programming constructs denoting upgrade points of various kinds, the semantics of which are given with the endofunctors.

Definition 5.1 (upgrade transition systems) *An* upgrade transition system *(UTS) is a classic transition system* $(\Gamma, L, \to, T)$ *as in Definition 2.3 where the set of labels is* $L = Mor(\mathbb{A}) \cup Mor(End(\mathbb{A}))$. *We call the transitions labelled by endofunctors,* jumps*, and distinguish them by labeling with capital letters* $E \in Mor(End(\mathbb{A}))$. *The other transitions are called* steps.

A computation *in UTS is a sequence of transitions starting with a step, i.e.,*

$$\pi ::= \xrightarrow{\alpha} \mid \pi; \xrightarrow{\alpha} \mid \pi; \xrightarrow{E}$$

where the sequence operation must match the end state of the previous transition with the start state of the next, and is restricted as:

$$\pi; \xrightarrow{\alpha} \quad \textit{iff} \quad \pi^t = \alpha^s$$

with the $(\cdot)^t$ *defined for computations* π *inductively as*

$$(\xrightarrow{\alpha})^t = \alpha^t \qquad (\pi;\xrightarrow{\alpha})^t = (\xrightarrow{\alpha})^t \qquad (\pi;\xrightarrow{E})^t = E(\pi^t).$$

Corollary 5.2 *When there are no jumps, a computation in UTS is defined exactly as for ATSes.*

Requiring a computation to start with a step transition captures our intuition that dynamic upgrades may happen only during the execution of program, but not before it starts.

Definition 5.3 (upgrade label transformers) *Consider a second indexing set* I_U *disjoint from* I_L*. The* upgrade label transformer *is defined the same as the label transformer from Definition 2.7, but using the upgrade indexes* $j \in I_U$*. The* $\mathbf{ULT}(j,\mathbb{U})$ *maps a category* $\mathbb{A}$ *to a product category* $\mathbb{A}\times\mathbb{U}$*, where* $\mathbb{U}$ *may only be a discrete category.*

The $\mathbb{U}$ categories are called the *upgrade components* of the labels. These are discrete since the program is not supposed to change the upgrade information. 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 upgrade component. In essence, the upgrade label transformer is a special case of the label transformer, i.e., uses a disjoint set of indexes I_U and only discrete categories $\mathbb{U}$.

Because the upgrade components are discrete categories, when referring to an upgrade component of a morphism label we in fact refer to the current upgrade object. Modularity is not disturbed, and new data categories may be added with the label transformer in the same way, without any interference with the upgrade components.

The semantics of dynamic software upgrades is given in terms of endofunctors on the product category. These endofunctors are obtained from combining *basic endofunctors*, which are defined in terms of only some of the data and the upgrade 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 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 upgrade these at runtime. Therefore, the upgrade 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 5.4 *Properties for label categories and their products.*

1. *In discrete or pairs categories morphisms are uniquely defined by the objects.*
2. *Let* $\mathbb{A}$ *and* $\mathbb{B}$ *be both either pairs or discrete categories. In the category returned by the label transformer* $\mathbf{LT}(i,\mathbb{B})(\mathbb{A})$ *the morphisms are uniquely defined by the objects.*
3. *Let* $\mathbb{A}$ *and* $\mathbb{B}$ *be both either pairs or discrete categories and* $\mathbb{C}$ *a monoid category. In the category returned by the label transformer* $\mathbf{LT}(j,\mathbb{C})(\mathbb{A})$*, as well as in* $\mathbf{LT}(i,\mathbb{B})(\mathbf{LT}(j,\mathbb{C})(\mathbb{A}))$*, each morphism is uniquely determined by the objects up to the morphism components coming from the monoid category; i.e., when the monoid components are projected away.*

Proof: Verifying the three properties is an easy exercise in category theory. □

Proposition 5.5 *Let* $\mathbb{A}$ *be a discrete or a pairs category, and* $F:\mathbb{A}\to\mathbb{A}$ *an endofunctor on* $\mathbb{A}$*.* F *is completely defined by its application to the objects of* $\mathbb{A}$*.*

Proof: Consider that for F we know how it is applied to the objects in $|\mathbb{A}|$. Consider one morphism $o \xrightarrow{\alpha} o'$, which is uniquely defined by the two objects o, o' (which may also be the same object, in a discrete category). The functor associated to this morphism is 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. □

Endofunctors are meant to describe how upgrade information from the objects of the $\mathbb{U}$ components changes the objects from the data components. The result above suggests that for pairs or discrete categories (or products of such), defining such endofunctors resorts to only defining their application on the objects of the category (i.e., a total function). These objects are normally tuples involving both upgrade and data objects.

Essentially, each upgrade functor specifies how the upgrade information from the upgrade component acts on the data information from the data component, changing the latter and also the information in the upgrade component (e.g., when doing incremental upgrades using only part of the upgrade information, which disappears after the upgrade operation). These will become more clear in the examples below, when we see the specific structure of the objects of the data and upgrade component categories.

Nevertheless, abstracting away from this structure, an upgrade endofunctor defines a correspondence between the data before and after some upgrade, for any upgrade information. Therefore we may have many upgrade endofunctors, and in defining the semantics of the upgrade constructs we specify which of these endofunctors to be used.

We will work in this paper with simple endofunctors, as in Proposition 5.5, for which the application on the objects is enough. However, DSOS should handle more complex categories, where the action of the endofunctors on the morphisms may also be relevant, e.g., monoid categories.

It remains to see how to combine such endofunctors from acting locally, on basic label categories, to a single endofunctor on the whole label category. We essentially make pairs of endofunctors over the product of categories.

Proposition 5.6 (endofunctors as morphisms) *Consider two categories $\mathbb{A}$ and $\mathbb{B}$ with $\mathbb{E}nd(\mathbb{A})$ and $\mathbb{E}nd(\mathbb{B})$ as in Definition 2.2. 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.*

1. *Any morphism $(E_{\mathbb{A}}, E_{\mathbb{B}})$ in the product $\mathbb{E}nd(\mathbb{A}) \times \mathbb{E}nd(\mathbb{B})$ is an endofunctor on $\mathbb{A} \times \mathbb{B}$ which takes any object $(a,b) \in |\mathbb{A} \times \mathbb{B}|$ to an object $(E_{\mathbb{A}}(a), E_{\mathbb{B}}(b))$ and any morphism (α,β) to $(E_{\mathbb{A}}(\alpha), E_{\mathbb{B}}(\beta))$.*
2. *If the categories $\mathbb{A}$ and $\mathbb{B}$ have the property of Proposition 5.4(2), like discrete or pairs categories, and their products, then the pairs of endofunctors are also completely defined by their application on the objects.*

Proof : The proof uses basic notions of category theory, and becomes even easier in the light of the proof of Proposition 5.5. □

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

However, Proposition 5.5 talks about products of only pairs categories or products of only discrete categories. Whereas, the product of a discrete with a pairs category is different since there may be tuples of objects with no morphism between them. This is an issue when putting together an upgrade component, which is always discrete, and a pairs data component. To be in line with our intuition that an upgrade operation should be arbitrarily definable and dependent on both the upgrade and the data objects, we will define endofunctors on *discretized categories*.

Definition 5.7 A discretized category $\mathbb{A}^d$ *is obtained from a category $\mathbb{A}$ by removing all non-identity morphisms.*

The only requirement that we ask of the endofunctors is that once an *information-less object* is reached, then no more change of data objects can be performed. This is a termination condition where inaction from the functor is required. Intuitively, an upgrade should not change the data of the program if there is no upgrade information.

Definition 5.8 *For any upgrade category $\mathbb{U}$ we identify at least one (or more) objects as being* information-less object*, and denote such objects with a "bottom" symbol at subscript, e.g., $o_\perp, u_\perp$.*

The categories that we encountered in our examples all have information-less objects, e.g.:

- when the underlying objects are *sets* then $o_\perp$ is the $\emptyset$;

- when the underlying objects are *partial functions* then $o_\perp$ is the minimal partial function completely undefined;
- for a category with a single object, like the *monoid 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 information-less object.

All examples above have the set of objects *equipped with a partial order*, in which case the information-less objects are the minimal objects in the partial order.

Definition 5.9 (no sudden jumps) *An endofunctor E on $\mathbb{D}^d \times \mathbb{U}$ is said to* have no sudden jumps *iff* $\forall u_\perp \in |\mathbb{U}| : E((d,u_\perp)) = (d,u_\perp)$.

All endofunctors that we give as examples in this paper can be easily checked to have no sudden jumps, i.e., are inactive on information-less objects.

Notation 5.10 *For some indexing set $I \subset I_L$ (or $I \subset I_U$) we denote by $\mathbb{D}_I$ (respectively $\mathbb{U}_I$) the product category $\times_{i\in I}\mathbb{D}_i$ obtained using the (upgrade) label transformer using the indexes from I attached to the respective category component.*

Definition 5.11 (extending endofunctors) *For the discretized version of a product category $\mathbb{D}_I \times \mathbb{U}_K$ obtained using* **LT** *and* **ULT***, define an* upgrade endofunctor *E as a total function over the objects of some part of this category, i.e., over $|\mathbb{D}_{I'} \times \mathbb{U}_{K'}|$, with $\emptyset \neq I' \subseteq I$ and $\emptyset \neq K' \subseteq K$. This endofunctor must have no sudden jumps.* Extend *E to the whole product category by pairing it with the identity endofunctor on the remaining component categories, as in Proposition 5.6.*

Proposition 5.12 (composing upgrade endofunctors) *For two endofunctors defined on disjoint sets of indexes, their extensions can be composed in any order, resulting in the same endofunctor on the union of the indexing sets.*

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 5.11 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 the tuple objects is also a 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 the product of the two endofunctors E' and E'' and only then 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. $\square$

Proposition 5.12 ensures modularity of the Dynamic SOS as follows. One defines a basic endofunctor for some dynamic upgrade construct, and this is never changed upon addition of other dynamic upgrade constructs and their upgrade categories and related endofunctors. Moreover, the method of *extending* the basic endofunctors with the identity functor on the rest of the indexes, from Def. 5.11, ensures modularity when new data or upgrade components are added by the label transformers.

When designing a programming language the label transformers may be applied on an already used index, resulting in changing the respective category component, e.g.:

- we may change a read-only component into a read/write component.
- we may decide to have more upgrade functors on one particular component, i.e., to define a new way of updating, maybe needed by a new programming constructs.
- we may leave one functor unspecified, as the identity functor, and at a later point add a proper functor for the specific component.

The encapsulation construction from Section 4.1 can be applied to endofunctors as well. This is expected, because if we encapsulate the categories on which the endofunctors act, then the endofunctors would become undefined. While by encapsulating them the endofunctors would be preserved. Once encapsulated, we may refer to the endofunctors using the object identifiers, the same as we were referring to the localized data components.

Each endofunctor is matched (using a transition rule) by a dynamic upgrade construct in the programming language, for which it captures the desired upgrade mechanism; this is exemplified in the next section.

Much of the work in [40] is concerned with analyzing PROTEUS program terms to automatically insert upgrade constructs at the appropriate points in the program where the upgrade would not cause type errors. The same analyses can be done also when the language is given a DSOS semantics.

A similar, but rather coarse analysis of upgrade points is done for the concurrent object-oriented language CREOL of [23], where acceptable upgrade points are taken to be those execution points of an object where it is “idle” (called *quiescent* states in [20], where the processor has been released and no pending process has been activated yet). A more fine-grained analysis in the style of [40] could be carried out, but it would be necessarily more complex because of the concurrency and object-oriented aspects, and also because of the special asynchronous method calls and late bindings. Such an analyses for the CREOL language is beyond the scope of this paper, and we leave it for future work.

5.1 Exemplifying DSOS for PROTEUS

For this section knowledge of PROTEUS [40] is not needed since our discussions will use only standard programming languages terminology. Nevertheless, we constantly refer to PROTEUS and the work in [40] for completeness and guidance for the familiar reader.

The transition rules that we gave for PROTEUS constructs [40, Fig.2] in Section 3 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. In [40, Sec.4.3] 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 [40, Fig.11]. Our choice was made with the intention to allow for a more clear separation of concerns, where we can see from the transition rules which programming construct works with what part of the program state, and in what way it interacts with the other parts. One can easily correlate our rules with the ones in [40, 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. Updating types is discussed in Section 6.

In [40, Fig.11] the update information comes in the form of a partial mapping from top-level identifiers to values (we omit the types for now). 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 $\mathbb{R}$. Therefore, the corresponding update categories are: $\mathbb{U}_{\mathbb{S}}$, $\mathbb{U}_{\mathbb{F}}$, and $\mathbb{U}_{\mathbb{R}}$, discrete categories containing the same objects as respectively $\mathbb{S}$, $\mathbb{F}$, and $\mathbb{R}$.

We observed that the objects of the update categories are, in general, the same as the objects in the corresponding data categories. However, 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 that the update data comes as a mapping from type names to pairs of type and type transformer.

PROTEUS uses a single update construct, which marks points in the program where updates can take place. We separate these update constructs into three kinds, each dealing with variables, functions, or records. Thus, our update signature Σ_{upd} contains:

$$s \;::=\; \mathbf{upgrade}^{v}\Delta \mid \mathbf{upgrade}^{f}\Delta \mid \mathbf{upgrade}^{r}\Delta \mid \ldots$$

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

Having defined the update categories, 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 and update objects, e.g., from $|\mathbb{S}| \times |\mathbb{U}_{\mathbb{S}}|$. Define an update transition rule as:

$$\frac{}{\mathbf{upgrade}^{v}\Delta \xrightarrow{E^{v}_{\Delta}} \mathsf{nil}}$$

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 [40, Fig.13] but restricted to consider only those variable identifiers specified in Δ and remove them from the update objects. Thus, both the data object and the update object may be changed by an endofunctor. 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 by removing all the mappings for the variable identifiers appearing in Δ; and changes ρ by replacing all mappings from variable identifiers appearing in Δ with the corresponding ones from ρ_u:

$$E^{v}_{\Delta}(\rho,\rho_u) = \begin{cases} (\rho[\mathbf{x} \mapsto \rho_u(\mathbf{x}) \mid \mathbf{x} \in \Delta \cap \rho_u],\, \rho_u \backslash \Delta) & \text{if } dom(\rho_u) \cap \Delta \neq \emptyset \\ (\rho,\rho_u) & \text{otherwise} \end{cases}$$

For the typed case we would need a more complex safety check which can be taken from [40, Fig.24] where it is called $\mathsf{updateOK}(-)$ and which also checks that the update information is well typed, not only that all needed identifiers are part of the update, as we did here. In fact one could do any kind of sanity checks of the update information against the data. However, at the level of the functor definition one does not have access to the program term, as is done in [40, Fig.16]. Any such information must either be put in the data part (e.g., as done when having threads), or be dealt with statically, as is done in [40, Sec.5] to obtain the definition of $\mathsf{updateOK}(-)$.

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 why it was immediate to take the definition from [40, 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 updates in tight correlation with the normal programming constructs.

The above definition was simple and natural, but more complicated definitions can be devised, especially when the update objects do not have the same structure as the data objects, as is the case for CREOL in Section 5.2.

Note also that for PROTEUS the semantics that we give here in the DSOS style does not depart from the semantics given in [40], which is made precise in Proposition 5.13 using notation from [40].

Proposition 5.13 *For any update information ρ_u, which in* PROTEUS *[40] is denoted upd, that updates only variable identifiers, we have that*

$$\Omega, H, \mathbf{update}^{\Delta} \xrightarrow{upd} \Omega, H', 0 \;\; \textit{iff} \;\; \mathbf{upgrade}^{v}\bar{\Delta} \xrightarrow{E^{v}_{\bar{\Delta}}} \mathsf{nil} \; \textit{with}$$

$\bar{\Delta}$ containing all those identifiers not in Δ, $H = \rho_s \cup \rho_f \cup \rho_r, H' = \rho'_s \cup \rho_f \cup \rho_r$, where $E^{v}_{\bar{\Delta}}(\rho_s, \rho_u) = (\rho'_s, \rho'_u)$.

This proposition can also be given for full updates of PROTEUS.

Proof: The transition $\xrightarrow{upd}$ is defined in [40, Fig.12] conditioned on the $\mathsf{updateOK}(-)$. This condition is now part of the definition of the endofunctor $E^{v}_{\bar{\Delta}}$. Only variable bindings are changed in the heap H which is reflected in the $\mathbf{upgrade}^{v}\bar{\Delta}$ construct. Put several such constructs in sequence to update other entities too. The choice we made to have incremental updates can be changed to match the choice in PROTEUS exactly; in which case the $\rho'_u = \emptyset$. □

5.2 Exemplifying Dynamic SOS for CREOL

First we identify the data components that are subject to the dynamic upgrade. For CREOL our example upgrades classes that have only methods and attributes. Thus, the data components subject to the upgrade are $\mathbb{C}$ and $\mathbb{A}$ holding the methods respectively attributes for each class. In [23] extra complexity appears in the form of *dependencies between upgrades*. In consequence, classes have associated *upgrade numbers*, that are only inspected by the objects during method calls, and changed only by the upgrade constructs. A discrete category $\mathbb{UN}$, with objects $|\mathbb{UN}| = \mathit{IdClass} \rightharpoonup \mathbf{Nat}$, mappings from class identifiers to natural numbers, is added as a global component $\mathbf{LT}(UN, \mathbb{UN})$. Denote the product of all these data categories as $\mathbb{D} = \mathbb{C} \times \mathbb{A} \times \mathbb{UN}$.

Next we identify the upgrade information, looking at [23], as three components: two holding the actual new code for methods and attributes, and another holding the dependencies, i.e.,

- a discrete category $\mathbb{UC}$ with the same objects as $\mathbb{C}$, $|\mathbb{UC}| = \mathit{IdClass} \rightharpoonup (\mathit{IdMethods} \rightharpoonup \mathit{MtdDef})$, holding information about which class names need to be upgraded and what is the new information to be used;
- another discrete category $\mathbb{UA}$ has objects $\mathit{IdClass} \rightharpoonup A$;
- and another $\mathbb{UD}$ having objects $|\mathbb{UD}| = \mathit{IdClass} \rightharpoonup (\mathit{IdClass} \rightharpoonup \mathbf{Nat})$ holding 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 observed that often the objects of the upgrade 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 upgrade data comes as a mapping from type names to pairs of type and type transformer. We are not concerned with types though. The example for CREOL also shows that the upgrade component $\mathbb{UD}$ does not have a correspondent among the data components.

Finally, we decide on the upgrade constructs and associate appropriate endofunctors. For CREOL there are more details to consider than we had for PROTEUS. In [23] there is no actual upgrade construct, but only upgrade messages floating in the distributed system and holding the upgrade information. We achieve the same results using the upgrade constructs:

$$S \ ::= \ \mathbf{upgrade}^c\Delta$$

where Δ is a set of class identifiers.

Essentially the technique of [23] corresponds, in PROTEUS terminology, to a single upgrade construct which appears at every "ideal" point in the program and which treats one class at a time. The ingenious analysis of the program code of PROTEUS can establish at each program point which identifiers can be upgraded without breaking the type safety. This preliminary analysis labels each program point with a set of *capabilities*. In our situation we can apply the same analysis and use upgrade constructs which are labelled with the set of identifiers that can be safely upgraded at that point. One could use the same information to have *incremental upgrades*, where at each point the upgrade is made only for those identifiers which are safe, when possible (dependencies between the names in the upgrade information may not allow for such splitting of the upgrade).

The corresponding upgrade transition rule is:

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

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 [23]. We need some notation first.

Definition 5.14 (dependencies check) *We define a binary relation* $\subseteq$ *on partial mappings* $\rho,\rho' \in IdClass \rightharpoonup \mathbb{N}$ *as:*

$$\rho \subseteq \rho' \ \ \textit{iff} \ \ \forall \mathbf{C} \in IdClass : \mathbf{C} \in \rho \Rightarrow \mathbf{C} \in \rho' \ \wedge \ \rho(\mathbf{C}) \leq \rho'(\mathbf{C}).$$

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.

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

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

The upgrade message used in [23] is the special case where Δ contains one class identifier and the three upgrade objects also contain this single class identifier. The apparent complication in the definition of the endofunctor comes from the complicated upgrade 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 abuse the notation and use set operations between Δ and mappings ρ, referring to the domain of the map. The notation $\rho[\ldots]$ denotes the update of the partial map.

Compared to PROTEUS, challenging in the dynamic upgrading mechanism of CREOL is the fact that the concurrent objects must be upgraded also (i.e., their local attributes), where inheritance needs particular attention, i.e., when a super-class is upgraded in a class hierarchy and objects of a sub-class must be aware of this upgrade.

Objects are the active unit of computation in a distributed object-oriented setting. In CREOL though also the classes are active, and the messages. 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 [23] 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. This is captured by the semantics of the objects and is not part of the upgrade mechanism. Essentially, in the presence of dynamic class upgrades the objects must be more aware of the data on which they work, as assumptions that were made when upgrades where not considered, like the fact that the data produced by the system is changed only by the system, do not hold any more.

6 Typing aspects over DSOS for PROTEUS

This section is meant to substantiate our claims that the typing analysis that makes the main results of [40] can also be carried over to PROTEUS with a DSOS semantics. Therefore, this section contains details pertaining to PROTEUS which for space reasons could not be included. Nevertheless, the general arguments that we make should be understandable without these details, which an interested reader can use when closely comparing with PROTEUS.

We need to add *type identifiers* $\mathbf{t} \in \mathit{IdType}$ and *type definitions* $\mathbf{type}\ \mathbf{t} = \tau$, with τ being basic types, record, functions, or reference types, as in [40, Fig.2]. We work with a new label category $\mathbb{TY}$, which has type environments $|\mathbb{TY}| = \mathit{IdType} \rightharpoonup \tau$ as objects, mapping type names to type definitions. This pairs category is attached to the existing labels using $\mathbf{LT}(Ty, \mathbb{TY})$. A transition rule updates the type environment consuming a type definition, similar to what we did with variable definitions in Section 3.3. Up to now we followed the modularity principle and none of the previous rules need to be changed. However, when we add type information in the syntax for variable and function definitions we need to change the respective rules too; this is inevitable as the program terms change. For the label categories there are two options: one more economical, chosen in PROTEUS, where the object of the label categories would map identifiers to tuples of type and value; and a second more modular option, to add new label categories mapping the respective identifiers to their types alone. These categories are treated by the respective changed rules; e.g., the label $\mathbf{LT}(Ft, \mathbb{FT})$, which has objects $|\mathbb{FT}| = \mathit{IdFun} \rightharpoonup \tau$, is used in the changed rule from Subsection 3.4:

$$\frac{}{\mathbf{fun}\,\mathbf{f}(\mathbf{x} : \tau_1)\,\{s : \tau_2\} \xrightarrow{\{F=\rho_f, Ft=\rho_t \ldots F=\rho_f[\mathbf{f} \mapsto \lambda(\mathbf{X}).s], Ft=\rho_{ft}[\mathbf{f} \mapsto (\tau_1 \to \tau_2)]\}} \mathsf{nil}}$$

The compilation procedure from [40, Sec.4.2], which inserts type coercions, is analogously done over DSOS as it makes no use of the semantics definition, but only of the programming language syntax and typing. In this way the program code can be annotated with $\mathbf{con_t}$ and $\mathbf{abs_t}$ at those points where the type name $\mathbf{t}$ is know to be further used concretely respectively abstractly. The update operation from [40, Fig.13] changes (besides the data) also the remaining program code, using type transformers, to make any abstract use of a type into the correct new type. We can avoid this update of the remaining program code by adding two new rewrite rules and one label component to deal with statements of the form $\mathbf{abs_t}e$. The label component $\mathbf{LT}(Ab, \mathbb{AB})$ has objects $|\mathbb{AB}| = \mathit{IdType} \rightharpoonup \mathbf{c}$, that map a type name to a type transformer function. The upgrade functor in DSOS just changes this label component, not touching the continuing program code, and the runtime makes sure to use the correct type by applying the type transformer as:

$$\frac{\rho_{ab}(\mathbf{t}) = \mathbf{c}}{\mathbf{abs_t}e \xrightarrow{Ab=\rho_{ab} \ldots} \mathbf{c}(e)} \qquad \frac{\mathbf{t} \notin \rho_{ab}}{\mathbf{abs_t}e \xrightarrow{Ab=\rho_{ab} \ldots} e}$$

When adding types, the safety check is performed by $\mathsf{updateOK}(-)$ and makes sure that the update information is well typed so that the continuing program will be type safe under the upgraded data. Essentially $\mathsf{updateOK}(-)$ checks that the new type definitions are safe and that the associated type transformers are well typed in the updated type information. It also checks that any new values or function definitions are well typed w.r.t. the updated information.

To avoid cluttering more the notation, consider upgrading only type definitions and function declarations, i.e., involve only the pairs categories $\mathbb{F}$, $\mathbb{FT}$, $\mathbb{TY}$, and the discrete category $\mathbb{AB}$. We would define an endofunctor E^t_Δ on

$\mathbb{TY} \times \mathbb{AB} \times \mathbb{UTY}$ for updating type definitions, or E^f_Δ on $\mathbb{F} \times \mathbb{FT} \times \mathbb{UF} \times \mathbb{UFT}$ for updating function declarations. The upgrade label categories $\mathbb{UF}$ and $\mathbb{UFT}$ contain the same objects as the respective categories, whereas $\mathbb{UTY}$ maps type identifiers to pairs of a type and a type transformer, as in PROTEUS.

Consider only $E^t_\Delta(\rho_{ty}, \rho_{ab}, \rho_{uty}) =$

$$\begin{cases} \begin{pmatrix} \rho_{ty}[\mathbf{t} \mapsto \sigma \mid \forall \mathbf{t} \in \Delta \cap \rho_{uty} \wedge \rho_{uty}(\mathbf{t}) = (\sigma, \mathbf{c})], \\ \rho_{ab}[\mathbf{t} \mapsto \mathbf{c} \mid \forall \mathbf{t} \in \Delta \cap \rho_{uty} \wedge \rho_{uty}(\mathbf{t}) = (\sigma, \mathbf{c})], \\ \rho_{uty} \setminus \Delta, \end{pmatrix} & \text{if } \mathsf{updateOK}(-) \\ \\ (\rho_{ty}, \rho_{ab}, \rho_{uty}) & \text{otherwise.} \end{cases}$$

In the first line we now use the check $\mathsf{updateOK}(-)$ as:

$$\begin{pmatrix} \vdash \rho_{ty}[\rho_{uty}] \ \wedge \ \ dom(\rho_{uty}) \in \Delta \ \wedge \\ (\forall \mathbf{t} \in dom(\rho_{uty}) : \rho_{uty}(\mathbf{t}) = (\sigma, \mathbf{c}) \Rightarrow \rho_{ty}[\rho_{uty}] \cdots \vdash \mathbf{c} : \rho_{ty}(\mathbf{t}) \to \sigma) \ \wedge \\ (\forall \mathbf{f} \in dom(\rho_{uf}) : \rho_{ty}[\rho_{uty}] \cdots \vdash \rho_{uf}(\mathbf{f}) : \rho_{uft}(\mathbf{f})) \end{pmatrix}$$

We have been superficial in the above definition and omitted some details like capabilities and other typing information. To be complete one would use the exact type-and-effect system of [40, Sec.5], i.e., from Fig.18-22, and extract the above definition of $\mathsf{updateOK}(-)$ from Fig.23-24. When looking at the definition in [40, Fig.24] one can correlate the first line above with lines 3-4 (where the *bindOK* is omitted), the second line with a simplified view of Fig.24(b), and the third line with the rest of Fig.24 that checks the new values.[7] In particular, the $types(H)$ that Fig.24 extracts from the heap, in our case come from the labels like $\mathbb{FT}$, which we omitted through "...". Useful could be to automate this proof in a proof assistant, on the lines of [35], which would contain all the meticulous details that have already been done in [40].

Considering the same typing system of [40, Sec.5], proving type soundness w.r.t. the DSOS semantics is not more than redoing the lengthy details from the appendix of [40]. The statement in Proposition 6.1 reflects the DSOS style, but can easily be matched by the respective statement in [40, Th.A.22].

Proposition 6.1 (type soundness) *For a program term P and an object o from the label category used in the semantics we have that if for some type environment Ω,*

$$\Omega \vdash P : \sigma, \Omega' \ \textit{ and } \ \Omega \vdash o$$

then either P is a value, or there exists a transition $P \xrightarrow{\alpha} P'$, with $o = \alpha^s$, $o' = \alpha^t$, for which $\Omega' \vdash o'$ and $\Omega' \vdash P' : \sigma, \Omega''$, where Ω', Ω'' are the effects of the typing judgments containing generated typing information. Particularly interesting is the above statement with an empty type environment and the object containing only empty maps.

Proof sketch: The check $\Omega \vdash o$ corresponds to the check that the heap is well typed in PROTEUS. The program P and the object o together make up the configuration that is used in PROTEUS. When the code is not a value, it can reduce to a new program of the same type and a changed heap which is still well typed. For upgrades this ensures well typedness of the changed heap. □

7 Conclusion and Further Work

We have built on the modular SOS of [30, 29] 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

[7] Note that the third line of $\mathsf{updateOK}(-)$ would be needed for E^f_Δ but not for E^t_Δ.

category theory of MSOS. The encapsulation can be used also in other situations 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 considered the dynamic class upgrades of CREOL, as well as the more classical upgrades of PROTEUS.

The upgrade information is externally provided and is not available to the program. This is why the upgrade components cannot be modified nor inspected by the program constructs, unlike the self produced data. The program can only decide upgrade points and what is allowed to be upgraded safely at a point. This is done using the **upgrade** constructs which can be automatically inserted in the code using techniques as in [40]. An upgrade allows the program data to be modified in accordance with the available upgrade information. We discard from the upgrade object only the used upgrade information, hence we use an incremental upgrade method. However, this is not fixed and depends on the decision when defining the upgrade endofunctors.

We have concentrated on the semantic framework, and less on the typing aspects. The cited papers that investigate forms of dynamic upgrade do thorough investigations into typing issues. These investigations can be done over a Dynamic SOS. We have exemplified DSOS for the PROTEUS language from [40] and discussed the typing aspects. DSOS could be done also for UPGRADEJ [7] or STUMP [33] since these also adopt the idea of upgrade points. We have also applied DSOS to the CREOL language [23, 21], where the combination of distributed objects with concurrency and asynchronous method calls with futures, interfaces and inheritance, dynamic binding and behaviour types, make the example non-trivial.

7.1 Possible continuations

Two questions remain. Can the DSOS mechanisms be captured solely within the MSOS; and how natural would this be? Can any dynamic upgrade construct, as captured by the endofunctor in the DSOS, be implemented using the programming language constructs alone? This last question is specific to the programming language and the upgrade mechanism; therefore, it cannot have a general answer at the level of DSOS. For specific situations this seems plausible as long as discrete categories are not used by the program. For the first question we see a negative answer because the endofunctors capture general functions on the objects which cannot readily be captured with the pairs and discrete categories. However, if we use only pairs categories then an encoding seems possible, though how natural it would be is not clear since the morphisms have the computational interpretation of capturing the way data is being manipulated by the program, whereas the endofunctors encode actions outside the view of the program but which act on the data that the program works with.

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. This was the goal of the PlanComps[8] project which achieved quite significant results [11, 41].

We can mention a few requirements of such a database. One is a ready integration of the (D)MSOS rules with a proof assistant like Coq, where the work in [35] 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. Nice advancements have been done by people from the PlanComps project, e.g., [9, 10] as well as relating with the recent K framework [32, 38]. Such a database needs to be maintainable by the community, as with a wiki.

Another interesting problem is upgrading running code at a more basic level than what CREOL or PROTEUS do where the upgrade happens for methods inside classes and the new execution can be seen only if the currently running code decides to call the upgraded methods; or where types are upgraded and the new code is seen if it is

[8] `http://plancomps.csle.cs.rhul.ac.uk/` or `http://www.plancomps.org`

accessed. We mean trivial examples like a reactive while loop (i.e., which waits for input from a user to proceed with a round of computation and response) where no methods are called, but where non-trivial computation and checks are done. A bug in such a code (maybe on a branch that is very rarely taken) may be caused by a wrong operation (like plus instead of minus). One wants to correct this running code, and no method or type upgrading would do it. We also do not accept arguments like: put the executing body of the while in a function which is called at each iteration, then upgrade the function when it is finished.

Upgrading such running code could be possible if we view the code as data, having one component of label category keeping track of the current executing code. One could use a program counter variable updated by all execution operations. Kept the program counter together with the actual execution code term t, An upgrade operation of the executing code works with an upgrade component that also contains a new code term t_u and an associated new program counter. The execution of the upgrade operation would then replace the execution term with the new one, and the continuing code would be the one given by the new term t_u and the associated program counter. The upgrade data for the program may have more complex structure, and the upgrade composition may be more involved than just complete replacing. For example, the new program counter may be depending on the old execution term and the current program counter also; so it may be a function of these. This may well be a map between the possible program counters in the old term t and new program counters in t_u.

Acknowledgments: We are grateful to Martin Churchill and Martin Steffen (and several anonymous reviewers) for helping us to improve the paper.

References

[1] M. Abadi and G. D. Plotkin. A model of cooperative threads. In Z. Shao and B. C. Pierce, editors, *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 29–40. ACM, 2009. (see also the LMCS journal version 2010-6(4)).

[2] M. Abadi and G. D. Plotkin. A model of cooperative threads. *Logical Methods in Computer Science*, 6(4), 2010.

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

[4] L. Aceto, W. J. Fokkink, and C. Verhoef. Structural Operational Semantics. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*, chapter 3. Elsevier, 2001.

[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 D. Thomas, editor, *20th European Conference ON Object-Oriented Programming (ECOOP)*, volume 4067 of *Lecture Notes in Computer Science*, pages 452–476. Springer, 2006.

[7] G. M. Bierman, M. J. Parkinson, and J. Noble. UpgradeJ: Incremental Typechecking for Class Upgrades. In J. Vitek, editor, *22nd European Conference on Object-Oriented Programming (ECOOP)*, volume 5142 of *Lecture Notes in Computer Science*, 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 R. Crocker and G. L. S. Jr., editors, *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 403–417. ACM, 2003.

[9] M. Churchill and P. D. Mosses. Modular bisimulation theory for computations and values. In F. Pfenning, editor, *16th International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, volume 7794 of *Lecture Notes in Computer Science*, pages 97–112. Springer, 2013.

[10] M. Churchill, P. D. Mosses, and M. R. Mousavi. Modular semantics for transition system specifications with negative premises. In P. R. D'Argenio and H. C. Melgratti, editors, *24th International Conference on Concurrency Theory (CONCUR)*, volume 8052 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2013.

[11] M. Churchill, P. D. Mosses, N. Sculthorpe, and P. Torrini. Reusable components of semantic specifications XII. *Trans. Aspect-Oriented Software Development*, 12:132–179, 2015.

[12] F. S. de Boer, D. Clarke, and E. B. Johnsen. A Complete Guide to the Future. In R. de Nicola, editor, *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer-Verlag, 2007.

[13] 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.

[14] C. Flanagan and M. Felleisen. The semantics of future and an application. *J. Funct. Program.*, 9(1):1–31, 1999.

[15] 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.

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

[17] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In N. J. Nilsson, editor, *3rd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 235–245. William Kaufmann, 1973.

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

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

[20] E. B. Johnsen, M. Kyas, and I. C. Yu. Dynamic Classes: Modular Asynchronous Evolution of Distributed Concurrent Objects. In A. Cavalcanti and D. Dams, editors, *2nd World Congress ON Formal Methods (FM)*, volume 5850 of *Lecture Notes in Computer Science*, pages 596–611. Springer, 2009.

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

[22] E. B. Johnsen, O. Owe, D. Clarke, and J. Bjork. A formal model of service-oriented dynamic object groups. *Science of Computer Programming*, 115–116:3–22, 2016.

[23] E. B. Johnsen, O. Owe, and I. Simplot-Ryl. A Dynamic Class Construct for Asynchronous Concurrent Objects. In M. Steffen and G. Zavattaro, editors, *7th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'05)*, volume 3535 of *LNCS*, pages 15–30. Springer, 2005.

[24] R. H. H. Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

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

[26] S. Malabarba, R. Pandey, J. Gragg, E. T. Barr, and J. F. Barnes. Runtime support for type-safe dynamic java classes. In E. Bertino, editor, *14th European Conference on Object-Oriented Programming (ECOOP)*, volume 1850 of *Lecture Notes in Computer Science*, pages 337–361. Springer, 2000.

[27] T. Mens and S. Demeyer. *Software Evolution*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[28] P. D. Mosses. A modular SOS for ML concurrency primitives. Technical Report RS-99-57, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999.

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

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

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

[32] P. D. Mosses and F. Vesely. Funkons: Component-based semantics in K. In S. Escobar, editor, *10th International Workshop on Rewriting Logic and Its Applications (WRLA)*, volume 8663 of *Lecture Notes in Computer Science*, pages 213–229. Springer, 2014.

[33] I. Neamtiu and M. W. Hicks. Safe and timely updates to multi-threaded programs. In M. Hind and A. Diwan, editors, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 13–24. ACM, 2009.

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

[35] 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.

[36] V. R. Pratt. Semantical Considerations On Floyd-Hoare Logic. In *IEEE Symposium On Foundations of Computer Science (FOCS'76)*, pages 109–121, 1976.

[37] C. Prisacariu and O. Owe. Dynamic Structural Operational Semantics (long version). Technical Report 426, Department of Informatics, University of Oslo, December 2012.

[38] G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[39] G. Stoyle, M. W. Hicks, G. M. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: safe and predictable dynamic software updating. In J. Palsberg and M. Abadi, editors, *32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, pages 183–194. ACM, 2005.

[40] 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.

[41] L. T. van Binsbergen, N. Sculthorpe, and P. D. Mosses. Tool support for component-based semantics. In L. Fuentes, D. S. Batory, and K. Czarnecki, editors, *15th International Conference on Modularity*, pages 8–11. ACM, 2016.

[42] I. C. Yu, E. B. Johnsen, and O. Owe. Type-Safe Runtime Class Upgrades in Creol. In *FMOODS*, volume 4037 of *LNCS*, pages 202–217. Springer, 2006.