Journal of Logical and Algebraic Methods in Programming 107 (2019) 79–107

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

# Dynamic structural operational semantics [☆]

Christian Johansen [a,∗,1], Olaf Owe [b]

[a] Institute for Technology Systems, University of Oslo, Norway
[b] Department of Informatics, University of Oslo, Norway

ARTICLE INFO

*Article history:*
Received 23 November 2017
Received in revised form 13 March 2019
Accepted 30 May 2019
Available online 5 June 2019

*Keywords:*
Modular Structural Operational semantics
Dynamic software updates
Dynamic class upgrades
Proteus
Creol
Concurrent objects

ABSTRACT

We introduce Dynamic Structural Operational Semantics (DSOS or Dynamic SOS) as a framework for describing semantics of programming languages that include dynamic software upgrades, i.e., for upgrading software code during run-time. 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.

As a side contribution we define a general encapsulating construction on Modular SOS useful in situations where a form of encapsulation of the execution is needed. We use encapsulation to give modular semantics to the concurrent object-oriented programming language Creol with active objects and asynchronous method invocations.

© 2019 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

## 1. Introduction

With renewed focus on software evolution [39,41], the interest in dynamic software upgrades is considerable [40,21,13,36,8,61,12,50,58,55,54]. However, 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

[☆] This work was partially supported by the project IoTSec – Security in IoT for Smart Grids, with number 248113/O70 part of the IKTPLUSS program funded by the Norwegian Research Council.

[∗] Corresponding author at: Dept. of Informatics, University of Oslo, P.O. Box 1080 Blindern, 0316 Oslo, Norway.
*E-mail addresses:* cristi@ifi.uio.no (C. Johansen), olaf@ifi.uio.no (O. Owe).

[1] The first author was previously known as Cristian Prisacariu.

https://doi.org/10.1016/j.jlamp.2019.05.006
2352-2208/© 2019 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

dynamic upgrading constructs independent of the programming language or the kind of system paradigm and build these into a semantic framework called Dynamic Structural Operational Semantics.[2]

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.

At runtime we distinguish between

1. the code being executed (i.e., program term);
2. *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, thread pools, local scopes;
3. *static data structures* (i.e., the data structures established at the start of the program) such as class tables, function definitions or static typing information.

The sharp distinction that we make here between static and dynamic semantic data structures can sometimes be blurred, e.g., function definitions may appear inside local blocks, thus not available at the start of the program.

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. This is in contrast to the semantics for dynamic upgrades which usually change static data structures, though some upgrade mechanisms also can reinitialise the values of program variables. The complexity of the program state depends on the features of the language, e.g., recursion is sometimes implemented using a stack-based store. Thus the operational semantics of a given code construct may need to be reformulated when the language is enriched. Modular SOS [45] 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*[3] that are manipulated by the semantics. Moreover, complex features such as abrupt termination and error propagation can be nicely handled by MSOS [45,18], 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 the language designer to a notation style. The notation can be the same as (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 the work of Mosses and New [46], 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 Proteus [61] and *dynamic class upgrades* for object-oriented languages such as the concurrent Creol language [36,34]. 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 [61,36] is given using the same style of operational semantics as for the other language constructs, often employing elaborate 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 their operational semantics. For these reasons we develop Dynamic SOS.

The two chosen languages illustrate different kinds of dynamic updates. Proteus, which is the more low-level language, allows low-level state and code updates as well as control of the possible update points in the code. Creol is a high-level language for distributed systems supporting actor-like concurrent objects communicating by asynchronous methods calls and with support for high-level synchronization mechanisms including conditional process suspension. Upgrades are done in a distributed manner; each object may upgrade itself at suspension or method completion. Thus while the update points are programmer-defined in Proteus, they are predefined in Creol. We show that DSOS can deal with both language settings in a uniform manner.

*The contributions of this paper* are:

- We define a semantic framework for programming languages where dynamic software upgrades can be given semantics in a uniform manner, thus allowing for easier comparisons between different upgrade mechanisms.
- We prove that our DSOS framework is a conservative extension of the MSOS framework (see Proposition 4.17) and that it promotes modularity (see Remark 4.18).
- We show, in Section 5, that typing aspects, commonly found in works on dynamic software upgrades, are readily doable on top of DSOS (like any other semantics).

[2] *Historical note:* A first version of this work appeared as the technical report [57].

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

- In order to test the adequacy of DSOS as a generalizing framework, we show how the semantics of two different languages with dynamic updates can be given in DSOS, i.e., we look in Section 4.1 at the language Proteus and in Section 7 at the more complex concurrent object-oriented Creol.
- We introduce in Section 6 an *encapsulation construction* on top of MSOS, which is orthogonal to and compatible with DSOS, and needed for giving semantics to distributed object-oriented languages in Section 6.1. An MSOS treatment of object-oriented programming does not seem to appear elsewhere. When defining encapsulation, we are concerned with respecting the MSOS modularity principles, as detailed in Remark 6.5.

### 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., [61, Fig. 3 & 4] from the Linux kernel, [36, Sec. 3] for complex class upgrades, or [12, 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;
 TEMP(int init){this.temp = init;} -- initialization

 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 is 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 is given below:

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

  class TEMP implements TempStat{
  int[] log;
  TEMP(int init){this.temp = init; log = empty;} -- initialization

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

  int avgTemp(){int avg=0; int i=0;
    for all x in log
      {avg = avg + x; i=i+1;}
    return avg/i; }
}}
```

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, but the constructor is modified so that the `log` variable is initialized. (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.)

The above example is presented in a syntax and style similar to Creol where 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 [36]. An upgrade is performed when 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 [64].

In Proteus one may add a declaration of a new variable, like the `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 upgrade points to be pre-designated by the programmer, while for Creol the program upgrade points are predefined by the concurrency model.

A challenge for the operational semantics is that such an upgrade as above is changing, in the middle of an execution, the class and interface tables, as well as variable and method bindings. In the Creol case, upgrades are handled in the operational semantics by sending special upgrade messages, like in the example above. 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

Dynamic SOS is intended as a framework for studying semantics of programming constructs for dynamic upgrade of software, and thus existing works on dynamic upgrades should be naturally captured; we exemplify DSOS on the *dynamic software updates* of the language Proteus [61] and on the *dynamic class upgrades* of the concurrent object-oriented language Creol [36,34]. Much of the literature on software updates focuses on type systems and type safety, and since their results also hold over Dynamic SOS, here we concentrate mainly on the semantic aspect, and only briefly discuss typing aspects in Section 5.

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

One observation that we want to emphasise with DSOS is that *upgrade points* must be identified and marked accordingly in the program code. The marking should be done with special upgrade programming constructs. Here we are influenced by the work on Proteus [61] (which is also taken up in UpgradeJ [12] and the multi-threaded Stump [51]). Opposed to a single marker as in Proteus, we propose to use multiple markers, which allow to capture also incremental upgrades, as in Creol. The purpose of identifying and marking such upgrade points is to ensure type safety after upgrades. The analysis techniques of [61] for safety after upgrades can be used over DSOS as well. Upgrade markers can be placed by a programmer or automatically by static analysis techniques, as in [61].

We are taking a *modular* approach in DSOS, following the work of Mosses [45], thus building on *Modular SOS* (MSOS). This formalism uses notions of category theory, on which our work depends. If normal program execution changes to the dynamic structures are captured by the *morphisms* in the MSOS style, the jumps will be captured in DSOS using *endofunctors*, a concept of higher abstraction, which are still seen as morphisms in an appropriate category, as we explain later on.

When seeing new frameworks, like DSOS, one may wonder about their purpose, and especially whether the same could be done with existing theories. First, we see any good attempt to unify seemingly disparate concepts as a contribution, as it allows easier comparisons and future developments of similar concepts. This is particularly so with the various dynamic software upgrade constructs out there, giving us one motivation for developing DSOS by identifying common features, and lifting these to a more abstract level of a framework. Second, one may ask whether the DSOS mechanisms can be captured by an encoding solely within the MSOS framework. The authors could not find a reasonable answer to this, and thus leave it as open question. Nevertheless, even if dynamic upgrade concepts could be encoded in MSOS, one then needs to study how natural would this encoding be, and whether it would help or not programming language designers. More specific discussions on these lines are done in the concluding Section 8.

### 1.3. Modular semantics for concurrent object-orientated languages

We 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. This is the case for the *Actor model* [29,7], used in concurrent object-oriented settings, where each concurrent entity is 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 in the MSOS style before. However, concurrent ML was treated in [42].

### 1.4. Structure of the paper

In Section 2 we first give a short listing of some simple notions of category theory that will be used throughout the paper and then recall the theory of Modular SOS. We exemplify, in Section 3, MSOS on language features found in the Proteus language, though diverging in some aspects as detailed in Subsection 3.7. We follow a modular style of giving semantics to one programming construct at a time, while the language and its semantics are formed in the end by summing up the needed syntactic constructs with their respective MSOS semantic elements and rules. In Section 4 we develop the Dynamic SOS theory, our main contribution. We exemplify the use of DSOS by giving semantics to the dynamic upgrade constructs of Proteus in Sections 4.1. We discuss in Section 5 how typing aspects from standard papers on dynamic upgrades can also be defined wrt. DSOS, and we look particularly at Proteus. In Section 6 we introduce the encapsulating construction and use it in Section 6.1 to give modular semantics to concurrent object-oriented constructs found in the Creol language. We exemplify again DSOS on the complex class upgrade feature of Creol in Section 7. We conclude and discuss possible applications and continuations of this work in Section 8.

The semantic rules that we give throughout the paper are meant to exemplify the various concepts we work with. As such, our rules could still have flaws, as they have not been validated, e.g., by implementing them in a proof assistant as advocated in [60,38,53], or by executing them, as advocated in works on Creol [34,32] where SOS-style of semantics were implemented in the rewriting logic of the Maude system [19] thus allowing for simulations and model checking. The MSOS style of semantics can also be implemented in the rewriting logic of Maude as shown in [15].

## 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 contrary 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 then one collects the syntactic constructs and the respective semantic rules. 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.

Moreover, using MSOS makes it easy to work within different standard notation conventions, but the methodology changes to a modular way of thinking about the semantics. Translations between these notations are possible because of the common methodology provided by the MSOS and its category theory foundations. Nevertheless, these categorical foundations are transparent to the one giving semantics to programming languages. The independence of notation can be seen in [46], 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 [45] for the MSOS related notions and to that of [52] for other notions of category theory.

**Definition 2.1.** 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 $\alpha$, $\beta$, possibly indexed). A morphism has a *source* object and a *target* object which we denote by $\alpha^s$ and $\alpha^t$. A category is required to have (i) *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 $\alpha$ and $\beta$, i.e., with $\alpha^t = \beta^s$, (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. if $o \xrightarrow{\alpha} o'$ then $\beta$ is $F(o) \xrightarrow{\beta} F(o')$, and moreover the following hold:

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

A functor $F : \mathbb{A} \to \mathbb{A}$ is called an *endofunctor* 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. [43], where the transitions are labelled with morphisms (arrows) from a category.

**Definition 2.3** *(ALTS).* An *arrow-labelled transition system* $(\Gamma, Mor(\mathbb{A}), \to)$ is formed by a set of *states* $t_i \in \Gamma$, including an *initial state* $t_0$, 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 ALTS 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 ALTS 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., $\alpha^s$ and $\alpha^t$. When the source and target objects of the morphism $\alpha$ are needed we make them explicit on the transition as $\xrightarrow{\{\alpha^s, \alpha^t\}}$.

One goal with ALTS and MSOS is to have as states *only program terms*, without the additional semantic data that an executing program may use, like stacks or heaps. The additional data and the way the program manipulates it, is captured by the morphisms which are labelling the transitions of the ALTS. This goal of separating the program term from the additional information needed for an analysis can intuitively be correlated with e.g.:

1. typing systems, where the program syntax alone is under analysis, and with a typing environment keeping track of additional information;

2. 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 constructs. The closed program terms built over this signature constitute the states 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 a unique morphism in each direction between every two objects.
- **monoid category:** A *monoid category* is a category that has a single object and the morphisms are elements from some predefined set *Act*.

Intuitively, discrete categories correspond to additional information that is of a read-only type, like 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 events and consider as morphisms all the 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: $|\mathbb{S}| = \mathit{IdVar} \rightharpoonup \mathit{Val}$, i.e. all partial functions from some set *IdVar* of variable identifiers to some set *Val* of values.

When several additional data are needed to define the semantics, we use *complex label categories* obtained by forming a product of basic label categories. One may use as many data components as needed to get a natural view of the semantics for each programming construct; whereas 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, while also providing 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 the trivial category[4] $\mathbf{1}$ 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

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

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:

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

**Notation 2.8.** For a composed morphism $\alpha$ 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 $\mathit{get}(\alpha, i)$, 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).* A *multi-sorted signature* $\Sigma$ is a set of function symbols, together with an *arity* mapping $\mathit{ar}()$ that assigns a natural number to each function symbol, and a family of *sorts* $S_i$. Each function symbol has a sort definition

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

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 $\Sigma$ and a set Var of sorted meta-variables is denoted $\mathsf{Terms}(\Sigma, \mathsf{Var})$ and is defined as follows (we often omit the set Var for readability):

- any meta-variable is a term;
- a function application $f(t_1, \ldots, t_{ar(f)})$ for some function symbol $f$ and set of terms $t_1, \ldots, t_{ar(f)}$, of the right sort, 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 $\alpha$ 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 $\frac{\mathrm{H}}{\mathrm{l}}$ with $H$ a set of transition literals, called the *premises*, and $l$ is a single transition literal, called the *conclusion*.

When side-conditions (e.g., equations, set memberships, definedness assertions) are needed in a rule, we write these on top of the derivation line, together with the premises, since they can easily be distinguished from transition literals. Negations of side-conditions can also be used.

**Definition 2.11** *(Generated ALTS).* The 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 term $P$, 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 notations for the labels. We write $t \xrightarrow{\{\alpha.i^s \ldots \alpha.i^t\}} t'$ to mean any morphism $\alpha$ that is a tuple of morphisms where the component indexed by $i$ is the one given on the transition, and all other components are the identity morphism, symbolized by the *three dots*. We write sources of morphisms to the left of the three dots, and targets to the right. In one transition we may refer to several components, e.g.: $t \xrightarrow{\{\alpha.i^s, \alpha.j \ldots \alpha.i^t\}} t'$. In this example the $j$ index is associated with a discrete category, and therefore we do not write the target of it on the right because it is understood as being the same. Moreover, because of the right/left convention we omit the superscripts. An even more terse notation may simply drop all references to $\alpha$ and keep only the indexes, thus the last example becomes $t \xrightarrow{\{i=o, j=h \ldots i=o'\}} t'$. Considering the objects $o, o'$ as stores, then this example transition says that the store $o$ is changed to $o'$, whereas the component $j$ may only be inspected.

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 [43, Prop. 1].

Intuitively, this result says that any transition defined using the old rule system, i.e., labelled with some $\alpha$ 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., $(\alpha, id_b)$, 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 only from the old rules then the projection from $\mathbb{A} \times \mathbb{B}$ to $\mathbb{A}$ gives an old label morphism by forgetting the identity morphism on $\mathbb{B}$. This is the case because the old transition refers only to components in $\mathbb{A}$, where the dots notation makes all other components contribute only with the identity morphism.

**Theorem 2.13** *([43, Prop. 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 ALTS 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 $\mathbb{A}'$. For each computation $\xrightarrow{\alpha}\xrightarrow{\beta} \ldots$ specified by $R$ over $\mathbb{A}$, we have a corresponding computation $\xrightarrow{\alpha'}\xrightarrow{\beta'} \ldots$ over $\mathbb{A}'$, and vice versa.*

**Proof Sketch.** This result is a consequence of [43, Prop. 1] and is explicitly stated in the corresponding technical report [44, Cor. 1]. 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 reverse 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 forward 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 language Proteus

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. The constructs that we treat in this section are following those in the programming language Proteus [61, Fig. 2], though slightly different (see Remark 3.1 for more concrete comparisons), being closer to the Creol language [36], which we treat in Section 6. Thus we focus on imperative constructs with static typing and static variable binding. To simplify the presentation we assume that program variables have distinct names (which could be achieved by adding the declaration level to variables during static analysis). Our style of giving semantics in this section is incremental, one construct at a time. We deliberately try to not adhere to any specific notational convention or particular established way of giving operational semantics to programming languages; as advocated by the MSOS framework and exemplified recently in the works of [46,18].

This section can be skipped by a reader knowledgeable of MSOS, though, if some later notation seems unclear one can come back to this section for clarifications.

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 $e \in Expressions$.

### 3.1. No label categories for sequential composition

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

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

where **skip** is a constant, standing for the program that does nothing, and $\_\,;\,\_$ is a binary function symbol, standing for *sequential composition*. We assume the special *value* nil to be left and right identity for sequential composition.

Each of our signatures is numbered with the reference of the respective subsection where it is defined. These signatures can be combined (i.e., modularity), or one can include another when some construct depends on another.

We define the following transition rules:

$$\frac{}{\mathbf{skip} \xrightarrow{\{\ldots\}} \mathsf{nil}} \ (\text{R.3.1.1}) \qquad \frac{s_1 \xrightarrow{X} s_1'}{s_1 \,;\, s_2 \xrightarrow{X} s_1' \,;\, s_2} \ (\text{R.3.1.2})$$

The special label variable $X$ stands for *any morphism*, and the label $\{\ldots\}$ stands for *any identity morphism*. The two 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.

Rule R.3.1.2 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 $s_1 \,;\, s_2$. Rule R.3.1.1 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, as in e.g. [45, Table 6].

### 3.2. Read-only label categories and a let construct

We add variable identifiers as constant symbols, denoted by $\mathbf{x} \in IdVar$ and having sort *Expressions*. We also include a **let** construct usually found in functional languages, forming the signature $\Sigma_{3.2}$:

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

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 $|\mathbb{S}| = IdVar \rightharpoonup Val$ 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 (here the letter $S$ refers to the component $\mathbb{S}$).

The transition rule corresponding to the variable identifiers is the axiom:

$$\frac{\rho(\mathbf{x}) = v}{\mathbf{x} \xrightarrow{\{S=\rho\ \ldots\}} v} \; (\text{R.3.2.1})$$

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. Any other possible label components, if and when they exist, contribute with an identity morphism (symbolized by the three dots). In consequence, since all morphisms 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 textual substitution $[v/x]$ as in [61] or [42, Sec. 4.1], assuming that all variable names are distinct (i.e., an application of Barendregt's variable convention [11, p. 26]).

$$\frac{e' \xrightarrow{X} e''}{\mathbf{let\,var}\ \mathbf{x} := e'\ \mathbf{in}\ e \xrightarrow{X} \mathbf{let\,var}\ \mathbf{x} := e''\ \mathbf{in}\ e} \; (\text{R.3.2.2}) \qquad \frac{}{\mathbf{let\,var}\ \mathbf{x} := v\ \mathbf{in}\ e \xrightarrow{\{\ldots\}} e[v/\mathbf{x}]} \; (\text{R.3.2.3})$$

### 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 $\Sigma_{3.2}$:

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

Both assignments and declarations (which are a subsort of *Statement*) 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 morphisms, i.e., referring to both the source and the target stores of the morphism.

$$\frac{e \xrightarrow{X} e'}{\mathbf{var}\ \mathbf{x} := e \xrightarrow{X} \mathbf{var}\ \mathbf{x} := e'} \; (\text{R.3.3.1}) \qquad \frac{\mathbf{x} \notin \rho}{\mathbf{var}\ \mathbf{x} := v \xrightarrow{\{S=\rho\ \ldots\ S=\rho[\mathbf{x}\mapsto v]\}} \mathsf{nil}} \; (\text{R.3.3.2})$$

The condition appearing above the line in R.3.3.2 can be ensured by the typing system, and thus could be removed. This is even desired when we want the rules to be in a standard rule format [5,49]. However, rule formats for DSOS are deferred to future work, discussed in Section 8.1, where one would need to look at more recent works on formats for data [48,24] and for MSOS [17]. The rules for assignment are similar.

$$\frac{e \xrightarrow{X} e'}{\mathbf{x} := e \xrightarrow{X} \mathbf{x} := e'} \; (\text{R.3.3.3}) \qquad \frac{\mathbf{x} \in \rho}{\mathbf{x} := v \xrightarrow{\{S=\rho\ \ldots\ S=\rho[\mathbf{x}\mapsto v]\}} \mathsf{nil}} \; (\text{R.3.3.4})$$

Note that we do not treat aliasing nor pointers, which would require more semantic details and working with both an *environment* and a *store*. Such details can be found treated in the MSOS style in [45].

### 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. Since it needs variable identifiers, then either $\Sigma_{3.4}$ includes $\Sigma_{3.2}$, or it should be combined with a signature that includes variable identifiers, like $\Sigma_{3.3}$.

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

Function declarations are stored in a new label component which is a pairs category[5] 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.

[5] 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 function definitions as programming constructs.

Another semantics, like that of [61], may want to consider these two as a single store-like 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.

The transition rules below are as in Proteus, using a functional languages style and again the notation $s[v/\mathbf{x}]$ for substitution of all occurrences of the variable in the statement body of the function.

$$\frac{e \xrightarrow{X} e'}{\mathbf{f}\, e \xrightarrow{X} \mathbf{f}\, e'}\ (\text{R.3.4.1}) \qquad \frac{\rho_f(\mathbf{f}) = \lambda(\mathbf{x}).s}{\mathbf{f}\, v \xrightarrow{\{F=\rho_f\ \dots\}} s[v/\mathbf{x}]}\ (\text{R.3.4.2}) \qquad \frac{}{\mathbf{fun}\, \mathbf{f}(\mathbf{x})\ \{s\} \xrightarrow{\{F=\rho_f\ \dots\ F=\rho_f[\mathbf{f}\mapsto\lambda(\mathbf{x}).s]\}} \mathsf{nil}}\ (\text{R.3.4.3})$$

### 3.5. Records

We define 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 $\Sigma_{3.5}$:

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

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 any previous labels category with: $\mathbf{LT}(R, \mathbb{R})$. The transition rules for the two new programming constructs are:

$$\frac{\mathbf{r} \notin \rho_r}{\mathbf{record}\ \mathbf{r}\,\{\mathbf{l_i} = e_i\} \xrightarrow{\{R=\rho_r\ \dots\ R=\rho_r[\mathbf{r}\mapsto\{\mathbf{l_i}=e_i\}]\}} \mathsf{nil}}\ (\text{R.3.5.1}) \qquad \frac{\rho_r(\mathbf{r}) = \{\mathbf{l_i} = e_i\},\ \exists i : \mathbf{l_i} = \mathbf{l},\ e_i = e}{\mathbf{r.l} \xrightarrow{\{R=\rho_r\ \dots\}} e}\ (\text{R.3.5.2})$$

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 C/C++ or Promela [30, Chap. 3]. Moreover, these rules specify small-step semantics. Big-step or (more common) eager semantics could also be given in the modular framework, as illustrated for the **if** statement in Section 3.6.

The choice of syntax for the records is biased by our goal to stay close to Proteus. Nevertheless, MSOS can be used to also give semantics to more complex records such as those in [31, Chap. 9].

### 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 called $\Sigma_{3.6}$.

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

The semantics does not rely on any particular form of the label categories. Moreover, this is big-step semantics.

$$\frac{e \xrightarrow{X} \mathbf{true}}{\mathbf{if}\, e\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2 \xrightarrow{X} s_1}\ (\text{R.3.6.1}) \qquad \frac{e \xrightarrow{X} \mathbf{false}}{\mathbf{if}\, e\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2 \xrightarrow{X} s_2}\ (\text{R.3.6.2})$$

### 3.7. Comparison with Proteus

We have omitted reference constructs that Proteus has, but these could be added following [42, Sec. 4.2]. We add the upgrade construct in Section 4.1. 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 5.

The transition rules that we gave above used a label category formed of three components: $\mathbb{S}$, $\mathbb{F}$, and $\mathbb{R}$. In [61, Sec. 4.3] the semantics of Proteus keeps all this information in one single structure called *heap*. When the information from our three labels is put together it forms the same semantic object, as one can check against [61, Fig. 11]. Our choice was made with the intention to obtain 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.

**Remark 3.1** *(Conformance with* Proteus *semantics).* Considering reductions $\Rightarrow$ to be either a compilation or an evaluation step from [61, Fig. 12], and the transitions $\xrightarrow{\alpha}$ obtained with the MSOS rules that we gave above, we have that

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

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

When we add types in Section 5 then the typing environment $\Omega$ may change and will be captured by the types label $\mathbb{TY}$ on the morphisms: $\Omega = \rho_{ty}$, $\Omega' = \rho'_{ty}$.

To prove this statement we need to use the relation between standard labelled transition systems and the arrow-labelled transition systems of the MSOS [45, Prop. 3&4]. Further we are looking at the semantic 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 [61, Fig. 12] are matched by the ones mentioned on the arrows of the MSOS rules given above.

We first correlate the functional syntax used by Proteus with our more imperative definitions from Sections 3.1–3.6. The constructs for sequential composition from Sec. 3.1 are encoded in the functional style of Proteus using multiple applications of the **let** construct. Our syntax for the **let** construct (Sec. 3.2) as well as for variable definition (Sec. 3.3) and function definition (Sec. 3.4) are the same as in Proteus, albeit looking more imperative than functional (e.g.: instead of the Proteus notation $\mathbf{z} \mapsto \lambda(x).e$ for function definition we use $\mathbf{fun}\,\mathbf{f}(\mathbf{x})\,\{s\}$ with $\mathbf{f}$ as the $\mathbf{z}$ and $s$ as the $e$). For records we chose in Sec. 3.5 to name them **record r** and to use this name in projections **r**.**l**, whereas Proteus uses just expressions when doing projections, which for us is the body of the record $\{\mathbf{l_i} = e_i\}$. The **if** statement from Proteus uses as test the comparison of two expressions, whereas in Sec. 3.6 we use only one expression and let the rules decide that the **if** is executed only when this expression evaluates to a Boolean.

We also correlate the transition rules of Proteus from [61, Fig. 12] with our rules from Sections 3.1–3.6. Proteus uses evaluation contexts [61, Fig. 11] and one rule (cong) for context reductions in [61, Fig. 12]. We achieve the same effect by adding for each construct explicit rules that evaluate expressions until their final value form. This is not new, e.g., [42] does this in the MSOS style for a functional language and explains well in [42, Sec. 5.1] the correlations with other related styles of semantics including evaluation context reduction. For the **let** construct the treatment in [61, Fig. 11] uses one evaluation context **let** $z = E$ **in** $e$ which is meant to ensure that first the expression $E$ is evaluated to a value, after which the corresponding rule (let) from [61, Fig. 12] is applicable. In our case, rule R.3.2.2 corresponds to the evaluation context, whereas R.3.2.3 is the same as in Proteus. For our variable declarations and assignments in Sec. 3.3 rules R.3.3.1 and R.3.3.3 correspond to evaluating the expression to a final value, whereas R.3.3.2 corresponds to the last compilation rule of [61, Fig. 12] where the heap is updated (in our case the label $S$ is involved). For functions our rule R.3.4.3 corresponds exactly to the compilation rule for functions from [61, Fig. 12] (the remaining compilation rule from [61, Fig. 12] is not applicable to us because we do not have typing information). Rule R.3.4.1 corresponds to the evaluation contexts for function applications from [61, Fig. 11], whereas R.3.4.2 corresponds to the rule (call) from [61, Fig. 12]. In Sec. 3.5 we chose to give a lazy semantics to records, where we store and return the expressions corresponding to some record entry, whereas Proteus gives an eager semantics where they store and return the corresponding values. For this Proteus keeps in the heap the records with values, whereas we keep in the label component $\mathbb{R}$ the records with their original expressions. Moreover, Proteus uses evaluation contexts for records to produce their corresponding values, whereas we do not. However, it is straightforward to give eager rules in the MSOS style; we only need to add rules for evaluating expressions until their final values (corresponding to the contexts of Proteus) and then rules similar to the current ones but which work on values instead of expressions. Our two rules from Sec. 3.6 first evaluate the test expression, and if it evaluates to a Boolean, one or the other of the branches is taken as the continuing statement. This matches the two transition rules (if-t) and (if-f) of Proteus from [61, Fig. 12] which work only on values, and also explicit the evaluation contexts from [61, Fig. 11] together with the evaluation context reduction rule (if-t) from [61, Fig. 12] for this statement. We provided big-step style rules only for exemplification purposes, whereas small-step style rules would be similar to what we did for the other previous constructs. The other rules from [61, Fig. 12] are not applicable, especially rule 2 is for coercions, which we do not consider, rules 5-6 are for references and are similar to [42, Sec. 4.2], whereas rules 10 and 12 are for updates, which we consider further down.

## 4. Dynamic structural operational semantics

To intuitively understand Dynamic SOS, consider how the program term is acting on a data structure (produced and changed) during its execution, like a store or a heap, or a more complex structure encoding a distributed run-time environment. Classical operational semantics describes how each programming construct changes these data structures (or uses the information stored in them). The dynamic upgrades use upgrade data that is seen as coming from outside the program, being controlled by an external entity. It is irrelevant for the upgrade programming construct where or how the upgrade data appears. What is important though is how the upgrade construct uses the upgrade data and when during the execution of the program. This is described through the semantics of the upgrade constructs and is ensured type-safe through static analysis (like for any other programming constructs). Implementing a way to insert upgrade data can be done in various ways, independent of the semantics of the upgrade constructs; e.g., in Creol a pool of messages is maintained for communications between the concurrent objects (i.e., part of the way a program executes), and this is also used for class upgrades by inserting into the pool a special upgrade message which is not matched by any programming constructs, but only by the upgrade mechanism.

DSOS considers that there is a separate data structure containing information about upgrades. This upgrade data structure is changed by the external entity at any point in the execution of the program, and the program may only inspect it. The upgrade operation takes information from the upgrade data and changes the data structures that the program maintains. This may change the future behaviour of the program. Upgrade constructs are added to the programming language and used to mark safe upgrade points, either by a programmer or by a tool that inserts such upgrade constructs in the code as necessary. The semantics of an upgrade construct describes only how performing the upgrade would change the data structures of the program.

These ideas capture only how the semantics of upgrades should be thought of 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 [40,21,13,36,8,61,12] focus on these aspects, which are usually developed on top of the semantics. We discuss typing aspects in Section 5.

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 (called jumps).
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. However, it turns out that for our examples, defining the endofunctors is as simple as defining functions; as shown below.
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 4.1** *(Upgrade transition systems).* An *upgrade transition system* (UTS) is a triple $(\Gamma, L, \rightarrow)$ with

- $\Gamma$ the set of program terms (also called states), including an initial state $t_0$;
- $L = Mor(\mathbb{A}) \cup Mor(End(\underline{\mathbb{A}}))$ the set of labels, with $End(\underline{\mathbb{A}})$ the category of endofunctors over $\underline{\mathbb{A}}$, where $\underline{\mathbb{A}}$ is any category having the same objects as $\mathbb{A}$ (i.e., $|\underline{\mathbb{A}}| = |\mathbb{A}|$), which we call a *variation of* $\mathbb{A}$, and
- $\rightarrow\ \subseteq \Gamma \times L \times \Gamma$ the labelled transitions relation.

We call the transitions labelled by endofunctors, *jumps*, and label them with capital letters $E \in Mor(End(\underline{\mathbb{A}}))$. The other transitions are called *steps*. A *computation* in UTS is a possibly infinite sequence of transitions starting in the initial state, i.e., $\pi = t_0 \xrightarrow{l_0} t_1 \xrightarrow{l_1} t_2 \ldots$, with the following restrictions:

1. $l_0 \in Mor(\mathbb{A})$, i.e., computations must start with a step;
2. for any finite prefix $\pi_0 \xrightarrow{l_i} t_{i+1} \xrightarrow{l_{i+1}} t_{i+2} \ldots$ then
   (a) when $l_i, l_{i+1} \in Mor(\mathbb{A})$ then $l_i^t = l_{i+1}^s$, i.e., two adjacent morphisms are composable in $\mathbb{A}$;
   (b) when $l_i \in Mor(End(\underline{\mathbb{A}}))$ and $l_{i+1} \in Mor(\mathbb{A})$ then $l_i(\pi_0^t) = l_{i+1}^s$, with $(\cdot)^t$ defined for finite non-empty computations as: $(\pi' \xrightarrow{\alpha})^t = \alpha^t$ and $(\pi' \xrightarrow{E})^t = E(\pi'^t)$.

**Corollary 4.2.** *When there are no jumps, a computation in UTS is defined exactly as for ALTSes.*

Requiring a computation to start with a step transition captures our intuition that dynamic upgrades may happen only during the execution of the program, but not before it starts. Note that the Definition 4.1.2b makes use of the endofunctors only wrt. their applications on the objects of the category, whereas their application on morphisms is not important for the whole Definition 4.1. This is why we only require that $|\underline{\mathbb{A}}| = |\mathbb{A}|$, allowing $\underline{\mathbb{A}}$ to be any variation of $\mathbb{A}$ with more or less morphisms. More relevant discussions follow Definition 4.8.

**Definition 4.3** *(Upgrade rules).* Rules for defining jump transitions are similar as in Definition 2.10 only that transition literals can have both morphisms and endofunctors labelling the arrow.

As we see further, in this paper the endofunctors are rather simple and accordingly the transition rules that we see in the examples for dynamic upgrade constructs, like in Sections 4.1 or 7, are only axiom rules, where for some program term we define a specific endofunctor which is then the one specified on the arrow of the conclusion transition literal.

**Definition 4.4** *(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 **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 because the program does not change the upgrade information, i.e., any morphisms on the transitions must include only identity morphisms for the upgrade components. 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.

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. The MSOS properties are preserved since new data categories may be added with the label transformer as before (see also Proposition 4.17 and Remark 4.18).

**Corollary 4.5.** *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 $\mathbb{U}$.*

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.

**Proposition 4.6.** *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. □

For discrete or pairs categories the endofunctors have a special property, they are completely defined by their application to the objects of the category.

**Proposition 4.7.** *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 only 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). 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. □

However, Proposition 4.7 talks about products of only pairs categories or products of only discrete categories, which have the property of Proposition 4.6.1. Whereas, the product of a discrete with a pairs category does not satisfy Proposition 4.6.1 since there may be tuples of objects with no morphism between them. This situation appears when putting together an upgrade component (which is always discrete) and a pairs data component. To capture our intuition that an upgrade operation should be arbitrarily definable (i.e., as in Proposition 4.7) and dependent on both the upgrade and the data objects, we define endofunctors on *discretized categories* as below.

**Definition 4.8.** The *discrete version of a category* $\mathbb{A}$ is the discrete category $\mathbb{A}^d$ that has objects $|\mathbb{A}^d| = |\mathbb{A}|$.

Endofunctors are meant to describe how upgrade information from the objects of the $\mathbb{U}$ components change the objects from the data components, thus defining a correspondence between the data before and after some upgrade, for any upgrade information. Proposition 4.7 suggests that for pairs or discrete categories, defining such endofunctors resorts to only defining their application on the objects of the category.

In general, endofunctors must also relate the morphisms, which restricts their definition. When a pairs category is coupled with a discrete upgrade category then the endofunctor definition on the objects must be made in such a way that morphisms from the pairs category are somewhat preserved. This would, for example, not allow to freely change the upgrade object, e.g., since $(p_1, u) \xrightarrow{(\alpha, id_u)} (p_2, u)$, we cannot define $E(p_1, u) = (p_3, u_1)$ and $E(p_2, u) = (p_4, u_2)$ because there is no morphism between these last two. This example is intuitive when doing incremental upgrades using only part of the upgrade information that disappears after the upgrade operation. It is interesting to study what kinds of practically useful upgrades can be defined if endofunctors can be defined on whatever label categories.

**Remark 4.9.** We will work in this paper with endofunctors restricted as in Proposition 4.7, for which it is enough to know their application on the objects. However, there are situations where the application of the endofunctors on the morphisms may be relevant, e.g., when using monoid categories for handling errors as in [45, Sec. 3.7]. In this case we can still define the endofunctors only by their application on the objects, and choose to define their application on the morphisms by simply matching the monoid part (cf. also Proposition 4.6(3)).

**Notation 4.10.** 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 applied multiple times, each time to a different index $i \in I$ and the respective category component $\mathbb{D}_i$ (i.e., $\circ_{i\in I}\mathbf{LT}(i, \mathbb{D}_i)$ when $I \subset I_L$).

**Definition 4.11** *(basic endofunctors).* For a product category $\mathbb{D}_I \times \mathbb{U}_K$ obtained using **LT** and **ULT**, consider the discrete version of this to be $\mathbb{D}_I^d \times \mathbb{U}_K$, and define a *basic upgrade endofunctor* $E^b$ as a total function over the objects of this category.

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

**Proposition 4.12** *(Endofunctors as morphisms). Consider two categories $\mathbb{A}$ and $\mathbb{B}$ with $End(\mathbb{A})$ and $End(\mathbb{B})$ as in Definition 2.2. Define the product of two such categories $End(\mathbb{A}) \times End(\mathbb{B})$ 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 $End(\mathbb{A}) \times End(\mathbb{B})$ 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 $(\alpha, \beta)$ to $(E_{\mathbb{A}}(\alpha), E_{\mathbb{B}}(\beta))$.*
2. *If the categories $\mathbb{A}$ and $\mathbb{B}$ are (products of) discrete or (products of) pairs categories, then the pairs of endofunctors are 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 4.7. □

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

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 4.13.** 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* (like for thread pools) then $o_\perp$ is the $\emptyset$;
- when the underlying objects are *partial functions* (like for heaps) 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 tupling 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 4.14** *(No sudden jumps).* An endofunctor $E$ on $\mathbb{D} \times \mathbb{U}$ is said to *have no sudden jumps* iff $\forall u_\perp \in |\mathbb{U}| : E((d, u_\perp)) = (d, u_\perp)$. Both $\mathbb{D}$ and $\mathbb{U}$ can be arbitrary product categories.

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.

**Definition 4.15** *(Extending endofunctors).* For a product category $\mathbb{D}_I \times \mathbb{U}_K$ obtained using **LT** and **ULT**, define a basic upgrade endofunctor $E^b$ as in Definition 4.11 over some part of this category, i.e., over $\mathbb{D}_{I'}^d \times \mathbb{U}_{K'}$, with $\emptyset \neq I' \subseteq I$ and $\emptyset \neq K' \subseteq K$. This basic endofunctor must have no sudden jumps. *Extend* $E^b$ to the whole product category by pairing it with the identity endofunctor on the remaining component categories, as in Proposition 4.12.

Note that extending with identity endofunctors can be done over arbitrary kinds of label categories, i.e., the restriction to discretized category is needed only for defining the basic endofunctors. Note that any basic endofunctor is defined over the discrete variation of $\mathbb{D}_I \times \mathbb{U}_K$, thus respecting the requirement from Definition 4.1. Moreover, any extension of a basic endofunctor is also over a variation (not necessarily discrete) of the larger product category, i.e., the objects are the same, thus still respecting the requirement from Definition 4.1.

**Proposition 4.16** *(Composing upgrade endofunctors). For two basic 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 $\mathbb{D}_I \times \mathbb{U}_J$ built with the label transformer over the index sets $I \cup J$. Without loss of generality we explain the proof for the simpler category $\mathbb{D} \times \mathbb{D}' \times \mathbb{U} \times \mathbb{U}' \times \mathbb{K}$. Consider two endofunctors $E$, $E'$ built over $\mathbb{D} \times \mathbb{U}$ respectively $\mathbb{D}' \times \mathbb{U}'$; the disjointness is important. The category $\mathbb{K}$ can be any upgrade or data categories.

Extend each endofunctor from above to the whole category as in Definition 4.15 by pairing it with the identity endofunctor on the remaining category; e.g., for $E$ denote its extension as $\tilde{E}$ to be the product $E \times ID_{\mathbb{D}' \times \mathbb{U}' \times \mathbb{K}}$. The similar extension for $E'$ is $\tilde{E}' = E' \times ID_{\mathbb{D} \times \mathbb{U} \times \mathbb{K}}$. Since the identity endofunctors can be seen as products of smaller identity endofunctors, we can rewrite the above endofunctors to: $\tilde{E} = E \times ID_{\mathbb{D}' \times \mathbb{U}'} \times ID_{\mathbb{K}}$ and $\tilde{E}' = E' \times ID_{\mathbb{D} \times \mathbb{U}} \times ID_{\mathbb{K}}$. We have 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 $ID_{\mathbb{D} \times \mathbb{U}} \times E' \times ID_{\mathbb{K}}$.

We need to show that

$$\tilde{E}' \circ \tilde{E} = \tilde{E} \circ \tilde{E}' = E \times E' \times ID_{\mathbb{K}}.$$

Pick now two objects from the big category: $(d_1, u_1, d'_1, u'_1, d^k_1)$ and $(d_2, u_2, d'_2, u'_2, d^k_2)$. The morphism between the tuple objects is also a tuple of respective morphisms $(\alpha_d, \alpha_u, \alpha'_d, \alpha'_u, \beta)$. Apply now the endofunctor $\tilde{E}$ to obtain tuples of objects $(E(d_1, u_1), d'_1, u'_1, d^k_1)$ and $(E(d_2, u_2), d'_2, u'_2, d^k_2)$, and morphism $(E(\alpha_d, \alpha_u), \alpha'_d, \alpha'_u, \beta)$. To this apply the second endofunctor to obtain objects $(E(d_1, u_1), E'(d'_1, u'_1), d^k_1)$ and $(E(d_2, u_2), E'(d'_2, u'_2), d^k_2)$, and morphism $(E(\alpha_d, \alpha_u), E'(\alpha'_d, \alpha'_u), \beta)$.

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

From the above it is easy to see how one could first make the product of the two endofunctors $E \times E'$ and afterwards extend this to the whole category, as $E \times E' \times ID_{\mathbb{K}}$, and the result of the application of this product results in the same objects and morphisms as the compositions above. □

**Proposition 4.17.** *The Dynamic SOS is a conservative extension of MSOS.*

**Proof.** The semantic objects of DSOS are the UTS which by Corollary 4.2 are including ALTS as a sub-class.

The labels in DSOS are defined both using MSOS label transformers as well as the upgrade label transformers of Definition 4.4 but using a disjoint set of indexes (see Corollary 4.5).

The operational rules are defined either as in MSOS or using endofunctors as labels (see Section 4.1). □

**Remark 4.18** *(Modularity of DSOS).* Proposition 4.16 ensures modularity of 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 Definition 4.15, 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 new ways of updating, maybe needed by 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.

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 Section 4.1.

Much of the work in [61] 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 [36], where acceptable upgrade points are taken to be those execution points of an object where it is “idle” (i.e., where the processor has been released and no pending process has been activated yet). A more fine-grained analysis in the style of [61] 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.

### 4.1. Exemplifying DSOS for Proteus

For this section knowledge of Proteus [61] is not needed since our discussions will use only standard programming languages terminology, including the notions presented in Section 3. Nevertheless, we constantly refer to Proteus and the work in [61] for completeness and guidance for the reader.

Four kinds of update information are present in Proteus. In this exemplification we treat only the two not related to types, i.e., the update and the addition of new bindings to the heap. Updating or adding new types is discussed in Section 5. In [61, 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, we can see the identifiers in the heap 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. 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 $\mathbb{R}$.

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, respectively records. Thus, our update signature $\Sigma_{upd}$ contains:

$$s \ ::= \ \textbf{upgrade}^{v}\,\Delta \mid \textbf{upgrade}^{f}\,\Delta \mid \textbf{upgrade}^{r}\,\Delta \mid \ldots$$

where $\Delta$ 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:

$$\frac{}{\textbf{upgrade}^{v}\,\Delta \xrightarrow{E^{v}_{\Delta}} \mathsf{nil}}\ (\text{R.4.1.1})$$

with $E^{v}_{\Delta} \in Mor(End(\mathbb{S} \times \mathbb{U}_{\mathbb{S}}))$ an endofunctor on the product category $\mathbb{S} \times \mathbb{U}_{\mathbb{S}}$, defined below the same as in [61, Fig. 13] but restricted to consider only those variable identifiers specified in $\Delta$ 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 $\rho$ of $|\mathbb{S}|$ and one update object $\rho_u$ of $|\mathbb{U}_{\mathbb{S}}|$ the endofunctor $E^{v}_{\Delta}$ changes $\rho_u$ by removing all the mappings for the variable identifiers appearing in $\Delta$; and changes $\rho$ by replacing all mappings from variable identifiers appearing in $\Delta$ with the corresponding ones from $\rho_u$:

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

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 [61, 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 7.

Our goal in this section was to exemplify the use of DSOS to give semantics to the Proteus updates without departing from the semantics given in [61]. We make this claim more precise in the following, using notation from [61], but with only a sketch of a proof, since a full proof would require too much background from [61].

**Remark 4.19.** For any update information $\rho_u$, which in Proteus [61] is denoted *upd*, and updates only variable identifiers, we have that

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

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

The transition $\xrightarrow{upd}$ is defined in [61, Fig. 12] conditioned by the updateOK(−) safety check. In our case this condition would be part of the definition of the endofunctor $E^{v}_{\bar{\Delta}}$, and above our untyped example reduced this check to only a membership check. For the typed case we would need a more complex safety check which can be taken from [61, Fig. 24] 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 [61, 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 [61, Sec. 5] to obtain the definition of updateOK(−).

The remark is only about variable bindings being changed in the heap $H$, which is reflected on the right side in the use of the $\mathbf{upgrade}^{v}\bar{\Delta}$ construct. The remark can also be given for the general updates of Proteus by putting several of our constructs in sequence to update other entities too. Our choice to have incremental updates can be changed to match the choice in Proteus exactly; in which case the $\rho'_u = \emptyset$.

## 5. Typing aspects over DSOS for Proteus

This section is meant to substantiate our claims that the typing analyses that make the main results of [61] can also be carried over to a DSOS semantics of Proteus. However, for space reasons we only aim to make general arguments that should be understandable without too many Proteus typing details, and an interested reader can then use our arguments when closely comparing with [61].

We need to add *type identifiers* $\mathbf{t} \in IdType$ and *type definitions* $\mathbf{type\ t} = \tau$, with $\tau$ being basic types, records, functions, or reference types, as in [61, Fig. 2]. We work with a new label category $\mathbb{TY}$, which has type environments as objects, $|\mathbb{TY}| = IdType \rightharpoonup \tau$, mapping type names to type definitions. This pairs category is attached to the existing labels using $\mathbf{LT}(Ty, \mathbb{TY})$. A transition rule would update 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 add new rules too; this is inevitable as the program terms are different. 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 new rules; e.g., the label $\mathbf{LT}(Ft, \mathbb{FT})$, which has objects $|\mathbb{FT}| = IdFun \rightharpoonup \tau$, is used in the new rule below which is similar to rule R.3.4.3:

$$\frac{}{\mathbf{fun\,f}(\mathbf{x} : \tau_1)\,\{s : \tau_2\} \xrightarrow{\{F=\rho_f\,, Ft=\rho_t\ \dots\ F=\rho_f[\mathbf{f}\mapsto\lambda(\mathbf{X}).s], Ft=\rho_{ft}[\mathbf{f}\mapsto(\tau_1\rightarrow\tau_2)]\}} \mathsf{nil}} \quad (\text{R.5.0.1})$$

The compilation procedure from [61, Sec. 4.2], which inserts type coercions, is analogously done over DSOS as it makes no use of the semantics definitions, 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 known to be further used concretely, respectively abstractly. The update operation from [61, 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 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 $|\mathbb{AB}| = IdType \rightharpoonup \mathbf{c}$, that map a type name to a type transformer function. The upgrade endofunctor 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:

$$\frac{\rho_{ab}(\mathbf{t}) = \mathbf{c}}{\mathbf{abs_t}e \xrightarrow{Ab=\rho_{ab}\ \dots} \mathbf{c}(e)} \ (\text{R.5.0.2}) \qquad \frac{\mathbf{t} \notin \rho_{ab}}{\mathbf{abs_t}e \xrightarrow{Ab=\rho_{ab}\ \dots} e} \ (\text{R.5.0.3})$$

When adding types, the check performed by updateOK(−) ensures that the update information is well typed so that the continuing program will be type safe under the upgraded data. Essentially 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 $E^{t}_{\Delta}$ on $\mathbb{TY} \times \mathbb{AB} \times \mathbb{UTY}$ for updating type definitions, and $E^{f}_{\Delta}$ 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 data categories, whereas $\mathbb{UTY}$ maps type identifiers to pairs of a type and a type transformer, as in Proteus.

For $\Delta$ a set of type identifiers, consider $E^t_\Delta(\rho_{ty}, \rho_{ab}, \rho_{uty}) =$

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

In the first alternative we now use the check updateOK$(-)$ defined 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 [61, Sec. 5], i.e., from Fig. 18–22, and extract a complete definition of updateOK$(-)$ from Fig. 23–24 by expanding our above definition. When looking at the definition in [61, Fig. 24] one can correlate from above

1. the first line with lines 3-4, where the $bindOK$ is omitted and $\rho_{ty}[[\rho_{uty}]]$ denotes the updating of the types of $\rho_{ty}$ using the information from $\rho_{uty}$ as in the third operation from [61, Fig. 13], which should be well-typed,
2. the second line with a simplified view of Fig. 24(b), and
3. the third line with the rest of Fig. 24 that checks the new values.[6]

In particular, the $types(H)$ that Fig. 24 extracts from the heap, in our case come from the labels, e.g., from $\mathbb{FT}$, which we omitted through "…". It would be useful to automate this proof in a proof assistant, on the lines of [53], which would contain all the meticulous details that have already been done in [61].

Considering the same typing system of [61, Sec. 5], proving type soundness w.r.t. the DSOS semantics is not more than redoing the lengthy details from the appendix of [61]. The statement below reflects the DSOS style, but can easily be matched by the respective statement in [61, Thm. A.22]. This is a specific result for the language of [61], meant here for exemplification, and not an essential part of the DSOS framework. Therefore, we only outline how the proof should be carried out, following the standard method for such type soundness proofs [63,7,1].

**Remark 5.1** *(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 $\Omega$,

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

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 $\Omega'$, $\Omega''$ are the effects of the typing judgements containing generated typing information.

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.

## 6. Encapsulating MSOS for concurrent object-oriented languages

We show how to give semantics in a modular style to concurrent object-oriented constructs as used by the language Creol. For this we first define a new *encapsulating mechanism* for concurrent object-orientation. This construction extends MSOS in a conservative manner, upholding the modularity principles as explained in Remark 6.5. The construct is not specific to object-orientation, but can be applied to other programming settings where execution is encapsulated in some way, e.g., where one works with isolating execution environments like in ambient calculus [14] or distributed settings [28,27].

The Creol language adopts the concurrency notion from the Actor model [7] which has proved well suited for object-oriented languages, e.g. in the Active Objects paradigm [37]. In this setting concurrent objects interact through asynchronous method calls and have their own execution unit (like a virtual CPU), thus having standard programming constructs running *inside* the object. This notion of encapsulation of the execution must be captured in the category theory of the labels and the

[6] Note that the third line of updateOK$(-)$ would be needed for $E^f_\Delta$ but not for $E^t_\Delta$.

rule definitions. The term “encapsulate” has a 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 used to give semantics to that code. These data components become now local to the specific object. Some specific new constructs, like for class declarations or object creation, may also need global data structures (not encapsulated).

We want to keep the modularity in defining semantics for such object-oriented constructs, i.e., 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 see how each object may perform a local transition, while the system of objects performs a concurrent transition combining these and changing several of the local data in the objects.

**Definition 6.1** *(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 $\beta$ of $Mor(\mathbb{B})$ with $\beta^s = F(o)$ and $\beta^t = G(o)$ s.t. for any morphism $\alpha$ 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 6.2** *(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.

In this paper we consider the objects of $\mathbb{O}$ to be identifiers for programming objects. A pairs category may be used if one needs to capture specific object-oriented relations like ownership. The intuition is that each $F \in |\mathbf{Enc}(\mathbb{O}, \mathbb{A})|$ attaches to each programming object identifier $o \in |\mathbb{O}|$ one data object from $|\mathbb{A}|$, thus capturing one snapshot of the working data of all programming objects in the system, thus, using $F(o)$ we have access to the data encapsulated in the programming object identified by $o$.

A natural transformation $\eta \in Mor(\mathbf{Enc}(\mathbb{O}, \mathbb{A}))$ between two such snapshots $F$, $F'$ can be thought as capturing one way of transforming one snapshot into the other. These intuitions hold also when monoid categories are part of the labels. In this case there are multiple natural transformations between two functors.

**Notation 6.3.** We can either write $\eta$ 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., $\eta = \{(F(o), F'(o)) \mid o \in |\mathbb{O}|\}$. As such we may refer to the data morphisms from the encapsulated label category $\mathbb{A}$, since these are indexed by the programming object identifiers, i.e., $\eta(o)$ and call these “local” morphisms associated to $o$. In consequence, we are free to use the get operation to refer to a particular component of the encapsulated label category morphisms $\eta(o)$, 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 $o : i$.

One property of the encapsulation construction is that the resulting category is similar to the encapsulated category in the following sense.

**Proposition 6.4.** *When the encapsulating construction is applied to a label category $\mathbb{A}$ where the morphisms are uniquely defined by the objects (i.e., as in Proposition 4.6), then the morphisms of $\mathbb{E} = \mathbf{Enc}(\mathbb{O}, \mathbb{A})$ are uniquely defined by the objects (i.e., by the functors).*

**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 $\eta$ 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 $o'$, then Diagram 1 commutes.
In our case this diagram becomes simpler because in $\mathbb{O}$ the only morphisms are the identities, which means that $\alpha$ 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 $id_{F(o)}$. In consequence Diagram 1 becomes Diagram 2, which clearly commutes for any $\eta$.
The natural transformation $\eta$ assigns the morphism $\eta(o)$ between $F(o)$ and $F'(o)$ in $\mathbb{A}$, which is unique by the assumption that in $\mathbb{A}$ morphisms are uniquely determined by the objects on which they act, i.e., $\eta(o) = (F(o), F'(o))$. The same for any $o' \in |\mathbb{O}|$ the $\eta(o')$ is unique. In consequence, the $\eta$ is uniquely defined by the two functors on which it is applied.

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

*Diagram 1.*

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

*Diagram 2.* □

**Remark 6.5** *(Modularity of encapsulation).* The category built by the encapsulating construction can be used with the label transformer to attach more global data structures. Therefore, the encapsulating construction is modular, in the sense that new global programming constructs and rules may be added without changing the rules for encapsulation. The reference mechanism provided by the label transformer is used as normal. We see this in Subsection 6.1.6 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 [35], 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 6.1.4. Henceforth we denote the encapsulated (or local or internal) category by $\mathbb{I}$ when its components are irrelevant.

The encapsulation construction from Section 6 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.

The way of applying the encapsulating construction will use transitions labelled both with morphisms from $\mathbb{I}$ as well as from $\mathbf{Enc}(\mathbb{O},\mathbb{I})$ (or an extension of this when new global label components are needed as for class definitions in Section 6.1.5), which will not fit the MSOS type of transition systems. As such we define a slight variation of ALTS where the label morphisms can come from one of several categories.

**Definition 6.6** *(Encapsulated ALTS).* For a set of categories $\{\mathbb{A}_j\}$ define an *encapsulating arrow-labelled transition system* $(\Gamma, \bigcup_j Mor(\mathbb{A}_j), \to)$ formed by a set of *states* $t \in \Gamma$, including an *initial state* $t_0$, and transitions $\xrightarrow{\alpha}$ labelled by morphisms $\alpha$ from one of the categories $\mathbb{A}_j$. A *computation* in an encapsulated ALTS 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 both coming from the same category $\mathbb{A}_j$ and are composable in $\mathbb{A}_j$ as $\alpha_{i+1} \circ \alpha_i \in Mor(\mathbb{A}_j)$.

### 6.1. 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, possibly 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 ENC rule R.6.1.1.

**Notation 6.7.** We reuse and extend the notation from Section 3 to specify (partly) the morphisms on arrows of encapsulated ALTS in the rules below. In particular, when specifying encapsulating morphisms from $\mathbf{Enc}(\mathbb{O},\mathbb{I})$ we use the notation $\mathbf{o} : X$ to partly specify the natural transformation, saying that the specific programming object $\mathbf{o}$ has in the encapsulated category the morphism $X$ (whichever that is). Similarly, the use of $\ldots$ around this (i.e., at the level of the category $\mathbf{Enc}(\mathbb{O},\mathbb{I})$) means that the rest of the morphisms from the natural transformation (i.e., for the other programming object identifiers) would be identity morphisms. We let $X$ stand for an arbitrary morphism also in the $\mathbf{Enc}(\mathbb{O},\mathbb{I})$ when this is clear from the context.

#### 6.1.1. 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 $\Sigma_{6.1.1}$ should include some signature defining statements; any of the constructs before can run inside the object construction, but the exact set of constructs is not relevant 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 objects, we encapsulate the category that we built before into:

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

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

$$(\textsc{enc})\ \frac{s \xrightarrow{X} s'}{\langle\, \mathbf{o} \mid s \,\rangle \xrightarrow{\{\mathbf{o}:X\ldots\}} \langle\, \mathbf{o} \mid s' \,\rangle}\ (\text{R.6.1.1})$$

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 $\{\mathbf{o} : X \ldots\}$ specifies only part of the natural transformation, whereas the rest may be any identity morphism. This specifies 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 $X \in Mor(\mathbb{I})$.

### *6.1.2. Systems of objects*

Objects may run in parallel, thus forming systems of distributed objects. For this we add a parallel construct $\|$ of sort *Objects*:

$$O \;\; ::= \;\; \langle\, \mathbf{o_1} \mid s_1 \,\rangle \parallel \langle\, \mathbf{o_2} \mid s_2 \,\rangle \;(\mathbf{o_1} \neq \mathbf{o_2}) \mid \ldots$$

Note that rule R.6.1.16 for object generation ensures that all parallel objects are different by making use of a globally unique (i.e., fresh) identity for each new object, thus explaining our notation above. Denote generically $obj \in O$ an object term as above (possibly indexed or primed). We choose an *interleaving* semantics for our parallel operator, hence the rules:

$$(\textsc{int-1})\ \frac{obj_1 \xrightarrow{X} obj_1'}{obj_1 \parallel obj_2 \xrightarrow{X} obj_1' \parallel obj_2}\ (\text{R.6.1.2}) \qquad (\textsc{int-2})\ \frac{obj_2 \xrightarrow{X} obj_2'}{obj_1 \parallel obj_2 \xrightarrow{X} obj_1 \parallel obj_2'}\ (\text{R.6.1.3})$$

Note that the $X$ in the rules R.6.1.2 and R.6.1.3 stands for any morphism in the encapsulating category.

For non-interleaving concurrency we need to specify more precisely the label components:

$$(\textsc{non-int})\ \frac{\langle\, \mathbf{o_1} \mid s_1 \,\rangle \xrightarrow{\mathbf{o_1}:X} \langle\, \mathbf{o_1} \mid s_1' \,\rangle \qquad obj_2 \xrightarrow{\eta} obj_2'}{\langle\, \mathbf{o_1} \mid s_1 \,\rangle \parallel obj_2 \xrightarrow{\eta[\mathbf{o_1}:X]} \langle\, \mathbf{o_1} \mid s_1' \,\rangle \parallel obj_2'}\ (\text{R.6.1.4})$$

The label of the conclusion specifies the morphism which is the natural transformation $\eta$ 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.

### *6.1.3. Methods inside objects*

We define methods like functions only that they have a **return** statement which is treated specially. We thus add method definition and invocation as $\Sigma_{6.1.3}$:

$$d \;\; ::= \;\; \textbf{mtd}\ \mathbf{m}(\mathbf{x})\,\{s\} \mid \ldots \qquad\qquad s \;\; ::= \;\; \textbf{return}\, e \mid y := \mathbf{m}(e) \mid \ldots$$

For simplicity we limit the discussion to methods with one input. The transition rules for methods use another pairs label category $\mathbb{MD}$, identified by the index *MD*, for storing method definitions (the same as was done for function definitions). This is added by the label transformer to the local labels category $\mathbb{I}$ that is encapsulated. In order to define the semantics of the (local) method call statement $y := \mathbf{m}(e)$ in isolation, we add to $\mathbb{I}$ a new pairs label category $\mathbb{RS}$, identified by the index *RS*, which has *stacks of variable identifiers* as objects, used to control the passing of the return value to the actual output variable of the call, i.e., the top of the stack. Since **return** is an example of abrupt termination, the above rules are inspired by the MSOS style of semantics done for error propagation [45, Sec. 3.7] and exception handling [18, Sec. 3.3] only that our label category needs more complicated objects, i.e., stacks. Standard stack operations are used, i.e.: $top(RS) \in IdVar$, $pop(RS) \in IdVar$, which also updates the stack by removing the top element, and $push(RS, y)$ that adds a variable identifier to the top of the stack, returning an updated stack. The rules then ensure that calls are handled in a stack-based manner, assuming that every method body has a return statement. We use end only in the semantics to delimit the end of a method body.

$$\frac{}{\textbf{mtd}\ \mathbf{m}(\mathbf{x})\,\{s\} \xrightarrow{\{MD=\rho_m\ \ldots\ MD=\rho_m[\mathbf{m}\mapsto\lambda(\mathbf{x}).(s)]\}} \mathsf{nil}}\ (\text{R.6.1.5})$$

$$\frac{e \xrightarrow{X} e'}{y := \mathbf{m}(e) \xrightarrow{X} y := \mathbf{m}(e')}\ (\text{R.6.1.6}) \qquad \frac{\rho_m(\mathbf{m}) = \lambda(\mathbf{x}).(s)}{y := \mathbf{m}(v) \xrightarrow{\{MD=\rho_m, RS=\rho_{rs} \dots RS=push(\rho_{rs}, y)\}} (s)[v/\mathbf{x}];\ \mathsf{end}}\ (\text{R.6.1.7})$$

$$\frac{e \xrightarrow{X} e'}{\mathbf{return}\, e \xrightarrow{X} \mathbf{return}\, e'}\ (\text{R.6.1.8}) \qquad \frac{top(\rho_{rs}) = y \qquad \mathsf{end} \notin s}{\mathbf{return}\, v;\ s;\ \mathsf{end} \xrightarrow{\{RS=\rho_{rs} \dots RS=pop(\rho_{rs})\}} y := v}\ (\text{R.6.1.9})$$

An alternative could have been to use the let construct to bind the return value in the statements following the call, however, that would require identification of these statements in the rules. 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, i.e., all the above rules would be encapsulated.

### 6.1.4. Threads

We take the model of threads studied in [2,3] and consider the following programming constructs of sort statement in a signature $\Sigma_{6.1.4}$ which normally would include also other constructs for statements from before:

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

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 $\rho_t$ 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 $\rho_t$ but does not contain $s$.

The authors of [2] give semantics using *evaluation contexts*, which is useful since **yield** needs the whole program term that follows it. However, evaluation contexts break one important aspect of MSOS, i.e., the semantics for each programming construct should not depend on any other constructs. But the evaluation constructs that we define below are very simple, involving only the sequential construct; as opposed to a semantics fully based on evaluation contexts, as done for Proteus, where more constructs are involved. Therefore, we prefer to use here evaluation contexts as a structuring aid to capture “the program term that follows”. Alternatively, we could have used similar methods as in Section 6.1.3 since **yield** has an abrupt termination style, like **return**.

Evaluation contexts are statements with a *hole* [ ]:

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

Placing a program term $s$ in the hole 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. In the semantic rules we also place nil inside the context hole, but nil is used only in the rules, involving also the two identity equations to make this semantics construct disappear from the term when another statement exists. Strictly speaking, during the semantics we work with equivalence classes of terms, in fact with the representative of the class being the smallest term, i.e., not containing nil; and similarly for the evaluation contexts.

Instead of giving alternative rules using evaluation contexts, we prefer to give the following rule R.6.1.10, and do not use the two rules for sequential composition from Subsection 3.1. A second rule is required when object terms are present. The $X$ label of R.6.1.11 comes from an encapsulated $\mathbb{I}$, whereas the one on the right comes from a global label.

$$\frac{s \neq \mathsf{nil} \qquad s \xrightarrow{X} s'}{Ev[s] \xrightarrow{X} Ev[s']}\ (\text{R.6.1.10}) \qquad \frac{s \neq \mathsf{nil} \qquad \langle\, \mathbf{o} \mid s \,\rangle \xrightarrow{X} \langle\, \mathbf{o} \mid s' \,\rangle}{\langle\, \mathbf{o} \mid Ev[s] \,\rangle \xrightarrow{X} \langle\, \mathbf{o} \mid Ev[s'] \,\rangle}\ (\text{R.6.1.11})$$

Now we can give the rules for the new programming constructs, which may be compared to the ones given in [2, Fig. 4].

$$\frac{}{\mathbf{async}\,(s) \xrightarrow{\{T=\rho_t \dots T=\rho_t \oplus s\}} \mathsf{nil}}\ (\text{R.6.1.12}) \qquad \frac{}{Ev[\mathbf{yield}] \xrightarrow{\{T=\rho_t \dots T=\rho_t \oplus Ev[\mathsf{nil}]\}} \mathsf{nil}}\ (\text{R.6.1.13})$$

$$\frac{s \in \rho_t}{\mathsf{nil} \xrightarrow{\{T=\rho_t \dots T=\rho_t \ominus s\}} s}\ (\text{R.6.1.14})$$

### 6.1.5. 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 are not essential here; these can be easily added as in [33].

*Class identifiers* are introduced from a set *IdClass*, usually written as **C**. Class definitions include *method definitions* and an *intialization* with initialized *attribute definitions* and initial statements;

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

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

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

$$\frac{\rho_a' = \rho_a[\mathbf{C} \mapsto At] \qquad \rho_c' = \rho_c[\mathbf{C} \mapsto \{\mathbf{m} \mapsto \lambda(\mathbf{x}).(s) \mid \mathbf{m} \in M\}]}{\mathbf{class}\,\mathbf{C}\,\{At\,;M\} \xrightarrow{\{A=\rho_a,C=\rho_c\ \ldots\ A=\rho_a',C=\rho_c'\}} \mathsf{nil}} \ (\text{R.6.1.15})$$

Each object is an instance of a class. In consequence we associate to each object the name of the class it belongs to, and from where method definitions can be retrieved.[7] Therefore, to the internal category $\mathbb{I}$ we add one more category $\mathbb{CN}$, with index $CN$, and objects $|\mathbb{CN}| = \mathit{IdClass}$ just class identifiers.[8] The rule for object creation uses both the global label components $\mathbb{A}$ and $\mathbb{C}$ as well as the encapsulated component where only two objects of the natural transformation are relevant, and mentioned on the arrow:

$$\frac{\mathit{fresh}(\mathbf{o}') \qquad \forall i \neq CN \quad \mathbf{o}' : i = d_\perp^i \qquad \rho_a(\mathbf{C}) = At}{\langle\, \mathbf{o} \mid Ev[\,\mathbf{x} := \mathbf{new}\,\mathbf{C}\,]\,\rangle \xrightarrow{\{A=\rho_a,C=\rho_c,\mathbf{o}:S=\rho\ \ldots\ \mathbf{o}':CN=\mathbf{C},\mathbf{o}:S=\rho[\mathbf{x}\mapsto\mathbf{o}']\}} \langle\, \mathbf{o} \mid Ev[\mathsf{nil}]\,\rangle \parallel \langle\, \mathbf{o}' \mid At\,\rangle} \ (\text{R.6.1.16})$$

There are different ways of ensuring freshness of the object identifiers and different ways of initialising generated objects, for instance by means of constructors. Our notion allows initialised attribute declarations as well as initial statements, for instance a call to a local method (used to start desired active behaviour in the case of Creol). Due to the assumption of distinct variables names, we do not need to separate attributes and local variables.

In Creol the semantic rule for object creation also initialises the structures of the new object to some default value. In order for this to be modular we need to know what is the "default" value for future label categories too. Therefore, we ask that any basic label category comes with a special object, *denoted* $d_\perp$ possibly indexed by the same index of the label category, as in the rule. This is similar to how in Definition 4.13 we defined information-less objects, and examples are similar, e.g.: for sets use $\emptyset$; for partial functions (like for heaps) use the minimal partial function completely undefined.

The transition rule for method application must include the object because it needs the global class definitions where the method definitions are found.

$$\frac{\mathbf{C} \in \rho_c \qquad \mathbf{m} \in \rho_c(\mathbf{C}) \qquad \rho_c(\mathbf{C})(\mathbf{m}) = \lambda(\mathbf{x}).(s)}{\langle\, \mathbf{o} \mid y := \mathbf{m}(v)\,\rangle \xrightarrow{\{\mathbf{o}:CN=\mathbf{C},C=\rho_c\ \ldots\}} \langle\, \mathbf{o} \mid s[v/\mathbf{x}]\ ?(y)\,\rangle} \ (\text{R.6.1.17})$$

### 6.1.6. Asynchronous method calls

We take the model of asynchronous method calls from [34] and consider two programming constructs for calling a method and reading the result of the completion of a call:

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

[7] 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 7; otherwise we could do without, and use the method definitions local to objects as in Subsection 6.1.3. 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.

[8] The objects of this category have such a simple structure that it may look awkward to have a category defined on them, but it is perfectly fine for MSOS and encouraged for separation of concerns (not optimisation).

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 [25,22,20]. We here limit the discussion to *local futures* as opposed to shared futures. Denote this signature $\Sigma_{6.1.6}$.

The asynchronous method calls, as discussed in [34], work with asynchronous message passing, as in the Actor model [7]. In order to keep track of the messages in the system we add a global data component as a pairs category $\mathbb{M}$ with objects $|\mathbb{M}| = \mathit{IdObj} \to 2^{\mathit{MsgTerm}}$ being mappings from object identifiers to message sets (i.e., each object has a pool of messages that can be manipulated by any of the distributed objects of the system). 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 $\mathcal{MS} \in 2^{\mathit{MsgTerm}}$. For our exemplification purposes the messages are of the form: $\mathit{invoke}(\mathbf{o}, n, \mathbf{m}(v))$ and $\mathit{compl}(n, v)$, where $\mathbf{o}$ is an object identifier, $n \in \mathbb{N}$ is a natural number (representing the *future* of the call), 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 *future*, and the statements may end with a return statement: $\mathbf{mtd}\ \mathbf{m}(\mathit{caller}, \mathit{future}, \mathbf{x})\,\{s\,;\,\mathbf{return}\, e\}$.

The special future identifiers $\mathbf{t}$ can be seen as variables that 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 $\mathit{IdFut} \rightharpoonup \mathbf{Nat}$.

$$\frac{\mathit{fresh}(n, \rho) \qquad \rho_m(\mathbf{o}') = \mathcal{MS} \qquad o' \neq o}{\langle\, \mathbf{o} \mid \mathbf{t}!\mathbf{o}'.\mathbf{m}(v)\,\rangle \xrightarrow{\mathbf{o}:L=\rho, M=\rho_m \,\ldots\, M=\rho_m[\mathbf{o}' \mapsto \mathcal{MS} \oplus \mathit{invoke}(\mathbf{o}, n, \mathbf{m}(v))], \mathbf{o}:L=\rho[\mathbf{t} \mapsto n]} \langle\, \mathbf{o} \mid \mathsf{nil}\,\rangle} \ (\mathrm{R.6.1.18})$$

$$\frac{\rho_m(\mathbf{o}) = \mathcal{MS} \qquad \mathit{invoke}(\mathbf{o}', n, \mathbf{m}(v)) \in \mathcal{MS}}{\langle\, \mathbf{o} \mid s\,\rangle \xrightarrow{M=\rho_m \,\ldots\, M=\rho_m[\mathbf{o} \mapsto \mathcal{MS} \ominus \mathit{invoke}(\mathbf{o}', n, \mathbf{m}(v))]} \langle\, \mathbf{o} \mid \mathbf{async}\,(\mathbf{m}(\mathbf{o}', n, v))\,;\, s\,\rangle} \ (\mathrm{R.6.1.19})$$

$$\frac{\rho(\mathit{caller}) = \mathbf{o}' \qquad \rho(\mathit{future}) = n \qquad \rho_m(\mathbf{o}') = \mathcal{MS}}{\langle\, \mathbf{o} \mid \mathit{Ev}[\mathbf{return}\, v]\,\rangle \xrightarrow{\mathbf{o}:S=\rho, M=\rho_m \,\ldots\, M=\rho_m[\mathbf{o}' \mapsto \mathcal{MS} \oplus \mathit{compl}(n, v)]} \langle\, \mathbf{o} \mid \mathsf{nil}\,\rangle} \ (\mathrm{R.6.1.20})$$

$$\frac{\rho(\mathbf{t}) = n \qquad \rho_m(\mathbf{o}) = \mathcal{MS} \qquad \mathit{compl}(n, v) \in \mathcal{MS}}{\langle\, \mathbf{o} \mid \mathbf{t}?(\mathbf{x})\,\rangle \xrightarrow{\mathbf{o}:L=\rho, M=\rho_m \,\ldots\, M=\rho_m[\mathbf{o} \mapsto \mathcal{MS} \ominus \mathit{compl}(n, v)]} \langle\, \mathbf{o} \mid \mathbf{x} := v\,\rangle} \ (\mathrm{R.6.1.21})$$

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 $\rho$, 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 6.8.** Rules R.6.1.19 and R.6.1.21 are dependent on additional program constructions (**async**(...) and assignment, respectively), 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 R.6.1.19 would involve the $\mathbb{T}$ local category and R.6.1.21 would involve $\mathbb{S}$. In this way dependency on program constructs is removed, but still the rules depend on the two local label components, which are shared with other constructs.

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.

## 7. 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 [36] extra complexity appears in the form of *dependencies between upgrades*. In consequence, in [36] 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 $\mathbf{LT}(\mathit{UN}, \mathbb{UN})$. 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 [36], 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 $\mathit{IdClass} \rightharpoonup \mathit{At}$;
- 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. 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 [36] there is no actual upgrade construct, but only upgrade messages floating in the distributed system and holding the upgrade information. Essentially the technique of [36] 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[9]:

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

where $\Delta$ is a set of class identifiers. The corresponding upgrade transition rule is:

$$\frac{}{\langle\, \mathbf{o} \mid \mathbf{upgrade}^{c}\,\Delta \,\rangle \xrightarrow{E^{c}_{\Delta}} \langle\, \mathbf{o} \mid \mathsf{nil} \,\rangle}\ \text{(R.7.0.22)}$$

with $E^{c}_{\Delta} \in \mathit{Mor}(\mathit{End}(\mathbb{D} \times \mathbb{U}_{\mathbb{D}}))$ an endofunctor on the product category from above, which is defined following the work in [36]. We need some notation first.

**Definition 7.1** *(Dependencies check).* We define a binary relation $\subseteq$ on partial mappings $\rho, \rho' \in \mathit{IdClass} \rightharpoonup \mathbb{N}$ as:

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

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

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

The upgrade message used in [36] is the special case where $\Delta$ 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 $\Delta$ and mappings $\rho$, referring to the domain of the map.

Compared to Proteus, challenging in the dynamic upgrading mechanism of Creol is the fact that the concurrent objects must be upgraded too (i.e., their local attributes), where inheritance would need particular attention, i.e., when a super-class is upgraded in a class hierarchy, then objects of a sub-class must be aware of this upgrade. Objects are the active unit of computation in a distributed object-oriented setting, and they use messages for communication. In Creol with upgrades also

[9] 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 classes are active since they may be changed at runtime. 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.

## 8. Conclusion and further work

Based on the work of [45,43] we have built a Dynamic SOS framework intended to be used for defining the semantics of dynamic software upgrades. At the same time we have given modular SOS definitions for concurrent object-oriented programming constructs, where we defined an *encapsulating construction* on the underlying category theory of MSOS. The encapsulation can be used also in other situations where a notion of localization of the program execution is needed.

We have considered two examples of languages with dynamic software upgrades. For the language Proteus we also discussed typing aspects. DSOS could similarly be used for upgrade constructs from UpgradeJ [12] or Stump [51] since these too adopt the idea of upgrade points of Proteus. We have also exemplified DSOS on the class upgrade construct of Creol [36,34], where the combination of distributed objects with concurrency and asynchronous method calls make the example non-trivial. Erlang [10,9] is a language close to Creol which supports the actor model using a functional programming style. The concurrency of Erlang could be handled in a manner similar to Creol. We are aware of works on hot-code-replacement in Erlang where upgrades are imitated by message passing, but we are not aware of language support for dynamic upgrades in Erlang. A dynamic upgrade mechanism for Erlang could in principle be made in the style of Creol upgrades, but this discussion is outside the current paper.

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 [61]. 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.

For the question whether DSOS could be encoded solely in the MSOS, mentioned in the introduction, 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. At the same time a discrete category can always be replaced by a pairs category without changes to the rules, in which case an encoding seems even more plausible. Thus, this open question seams like a natural immediate continuation of this work.

A programming language designer might also ask whether any dynamic upgrade construct that can be captured by the endofunctors in the DSOS, can be implemented using the programming language constructs alone. This 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.

### 8.1. Possible continuations

A theoretical motivation for giving semantics to dynamic upgrades using DSOS 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 [5], some of which we hope can be translated to the theory developed here. In particular, the states of the transition systems obtained from DSOS 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 [16] and endeavours into rule formats with data, like we would need in DSOS, are being investigated [49,48]. General results that could be investigated (starting from the work presented in [5,16,48]) are:

1. generating algebraic semantics [4,6,24] from specific forms of the transition rules;
2. compositional reasoning results wrt. dynamic logic [56,26] using specific forms of transition rules in the style of [23]; 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.

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[10] project which achieved quite significant results [18,62]. For DSOS we would probably need to extend their results to include the dynamic upgrade semantic concepts of DSOS and also the encapsulation concept. Then all the FunCons of PlanComps would be reusable, and on top would define similar concepts for dynamic upgrade constructs.

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 [53] 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., [16,17] as well as relating with the recent K framework [47,59]. 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 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 the label category keeping track of the current executing code. One could use a program counter variable updated by all execution operations. 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 $t_u$. However, this is an open question left for future work.

## Declaration of Competing Interest

There are no conflicting interests that the authors can think of.

## Acknowledgements

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

## References

[1] Martin Abadi, Luca Cardelli, A Theory of Objects, Springer Science & Business Media, 2012.

[2] Martín Abadi, Gordon D. Plotkin, A model of cooperative threads, in: Zhong Shao, Benjamin C. Pierce (Eds.), 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, ACM, 2009, pp. 29–40.

[3] Martín Abadi, Gordon D. Plotkin, A model of cooperative threads, Log. Methods Comput. Sci. 6 (4) (2010).

[4] Luca Aceto, Bard Bloom, Frits W. Vaandrager, Turning SOS rules into equations, Inf. Comput. 111 (1) (1994) 1–52.

[5] Luca Aceto, Wan J. Fokkink, Chris Verhoef, Structural operational semantics, in: Jan A. Bergstra, Alban Ponse, Scott A. Smolka (Eds.), Handbook of Process Algebra, Elsevier, 2001, chapter 3.

[6] Luca Aceto, Anna Ingólfsdóttir, Mohammad Reza Mousavi, Michael A. Reniers, Algebraic properties for free!, Bull. Eur. Assoc. Theor. Comput. Sci. 99 (2009) 81–103.

[7] Gul Agha, Ian A. Mason, Scott F. Smith, Carolyn L. Talcott, A foundation for actor computation, J. Funct. Program. 7 (1) (1997) 1–72.

[8] Sameer Ajmani, Barbara Liskov, Liuba Shrira, Modular software upgrades for distributed systems, in: Thomas Dave (Ed.), 20th European Conference on Object-Oriented Programming, ECOOP, in: Lecture Notes in Computer Science, vol. 4067, Springer, 2006, pp. 452–476.

[9] Joe Armstrong, Programming Erlang: Software for a Concurrent World: Pragmatic Bookshelf, 2nd edition, 2013.

[10] Joe Armstrong, Robert Virding, Claes Wikström, Mike Williams, Concurrent Programming in ERLANG, 2nd edition, Prentice Hall, 1996.

[11] Henk Barendregt, The Lambda Calculus: Its Syntax and Semantics, Studies in Logic and the Foundations of Mathematics, vol. 103, North-Holland, 1981.

[12] Gavin M. Bierman, Matthew J. Parkinson, James Noble UpgradeJ, Incremental typechecking for class upgrades, in: Jan Vitek (Ed.), 22nd European Conference on Object-Oriented Programming, ECOOP, in: Lecture Notes in Computer Science, vol. 5142, Springer, 2008, pp. 235–259.

[13] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, Steven Richman, Lazy modular upgrades in persistent object stores, in: Ron Crocker, Guy L. Steele Jr. (Eds.), ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA, ACM, 2003, pp. 403–417.

[14] Luca Cardelli, Andrew D. Gordon, Mobile ambients, Theor. Comput. Sci. 240 (1) (2000) 177–213.

[10] https://plancomps.csle.cs.rhul.ac.uk/.

[15] Fabricio Chalub, Christiano Braga, Maude MSOS tool, in: Proceedings of the 6th International Workshop on Rewriting Logic and Its Applications, WRLA 2006, Electron. Notes Theor. Comput. Sci. 176 (4) (2007) 133–146.

[16] Martin Churchill, Peter D. Mosses, Modular bisimulation theory for computations and values, in: Frank Pfenning (Ed.), 16th International Conference on Foundations of Software Science and Computation Structures, FOSSACS, in: Lecture Notes in Computer Science, vol. 7794, Springer, 2013, pp. 97–112.

[17] Martin Churchill, Peter D. Mosses, Mohammad Reza Mousavi, Modular semantics for transition system specifications with negative premises, in: Pedro R. D'Argenio, Hernán C. Melgratti (Eds.), 24th International Conference on Concurrency Theory, CONCUR, in: Lecture Notes in Computer Science, vol. 8052, Springer, 2013, pp. 46–60.

[18] Martin Churchill, Peter D. Mosses, Neil Sculthorpe, Paolo Torrini, Reusable components of semantic specifications XII, Trans. Aspect-Oriented Softw. Dev. 12 (2015) 132–179.

[19] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, José F. Quesada, Maude: specification and programming in rewriting logic, Theor. Comput. Sci. 285 (2) (2002) 187–243.

[20] Frank S. de Boer, Dave Clarke, Einar Broch Johnsen, A complete guide to the future, in: Rocco de Nicola (Ed.), Proc. 16th European Symposium on Programming, ESOP'07, in: Lecture Notes in Computer Science, vol. 4421, Springer-Verlag, 2007, pp. 316–330.

[21] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, Paola Giannini, More dynamic object re-classification: Fickle$_{II}$, ACM Trans. Program. Lang. Syst. 24 (2) (2002) 153–191.

[22] Cormac Flanagan, Matthias Felleisen, The semantics of future and an application, J. Funct. Program. 9 (1) (1999) 1–31.

[23] Wan Fokkink, Rob J. van Glabbeek, Paulien de Wind, Compositionality of Hennessy-Milner logic by structural operational semantics, Theor. Comput. Sci. 354 (3) (2006) 421–440.

[24] Daniel Gebler, Eugen-Ioan Goriac, Mohammad Reza Mousavi, Algebraic meta-theory of processes with data, in: Johannes Borgström, Bas Luttik (Eds.), Proc. 20th Int. Workshop on Expressiveness in Concurrency and 10th Workshop on Structural Operational Semantics, EXPRESS/SOS, in: Electronic Proceedings in Theoretical Computer Science (EPTCS), vol. 120, 2013, pp. 63–77.

[25] Robert H. Halstead Jr., Multilisp: a language for concurrent symbolic computation, ACM Trans. Program. Lang. Syst. 7 (4) (1985) 501–538.

[26] David Harel, Dexter Kozen, Jerzy Tiuryn, Dynamic Logic, MIT Press, 2000.

[27] Matthew Hennessy, A Distributed Pi-Calculus, Cambridge University Press, 2007.

[28] Matthew Hennessy, James Riely, Resource access control in systems of mobile agents, Inf. Comput. 173 (1) (2002) 82–120.

[29] Carl Hewitt, Peter Bishop, Richard Steiger, A universal modular ACTOR formalism for artificial intelligence, in: Nils J. Nilsson (Ed.), 3rd International Joint Conference on Artificial Intelligence, IJCAI, William Kaufmann, 1973, pp. 235–245.

[30] Gerard J. Holzmann, The Spin Model Checker, Addison-Wesley, 2003.

[31] Hans Hüttel, Transitions and Trees: An Introduction to Structural Operational Semantics, Cambridge Univ. Press, 2010.

[32] Einar Broch Johnsen, Olaf Owe, An asynchronous communication model for distributed concurrent objects, in: 2nd International Conference on Software Engineering and Formal Methods, SEFM, IEEE Computer Society Press, 2004, pp. 188–197.

[33] Einar Broch Johnsen, Olaf Owe, Inheritance in the presence of asynchronous method calls, in: 38th Hawaii International Conference on System Sciences, HICSS-38, IEEE Computer Society, 2005.

[34] Einar Broch Johnsen, Olaf Owe, An asynchronous communication model for distributed concurrent objects, Softw. Syst. Model. 6 (1) (2007) 39–58.

[35] Einar Broch Johnsen, Olaf Owe, Dave Clarke, Joakim Bjork, A formal model of service-oriented dynamic object groups, Sci. Comput. Program. 115–116 (2016) 3–22.

[36] Einar Broch Johnsen, Olaf Owe, Isabelle Simplot-Ryl, A dynamic class construct for asynchronous concurrent objects, in: Martin Steffen, Gianluigi Zavattaro (Eds.), 7th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems, FMOODS'05, in: Lecture Notes in Computer Science, vol. 3535, Springer, 2005, pp. 15–30.

[37] Farzane Karami, Olaf Owe, Toktam Ramezanifarkhani, An evaluation of interaction paradigms for active objects, J. Log. Algebraic Methods Program. 103 (2019) 154–183.

[38] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, Robert Bruce Findler, Run your research: on the effectiveness of lightweight mechanization, in: 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '12, ACM, 2012, pp. 285–296.

[39] Meir M. Lehman. Programs, Life cycles, and laws of software evolution, Proc. IEEE 68 (9) (1980) 1060–1076.

[40] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl T. Barr, J. Fritz Barnes, Runtime support for type-safe dynamic Java classes, in: Elisa Bertino (Ed.), 14th European Conference on Object-Oriented Programming, ECOOP, in: Lecture Notes in Computer Science, vol. 1850, Springer, 2000, pp. 337–361.

[41] Tom Mens, Serge Demeyer, Software Evolution, 1 edition, Springer, 2008.

[42] Peter D. Mosses, A Modular SOS for ML Concurrency Primitives, Technical Report RS-99-57, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999.

[43] Peter D. Mosses, Foundations of modular SOS, in: Miroslaw Kutylowski, Leszek Pacholski, Tomasz Wierzbicki (Eds.), Mathematical Foundations of Computer Science, MFCS'99, in: Lecture Notes in Computer Science, vol. 1672, Springer, 1999, pp. 70–80.

[44] Peter D. Mosses, Foundations of Modular SOS, Technical Report BRICS RS-99-54, Basic Research in Computer Science, December 1999.

[45] Peter D. Mosses, Modular structural operational semantics, J. Log. Algebraic Program. 60–61 (2004) 195–228.

[46] Peter D. Mosses, Mark J. New, Implicit propagation in structural operational semantics, Electron. Notes Theor. Comput. Sci. 229 (4) (2009) 49–66.

[47] Peter D. Mosses, Ferdinand Vesely, Funkons: component-based semantics in K, in: Santiago Escobar (Ed.), 10th International Workshop on Rewriting Logic and Its Applications, WRLA, in: Lecture Notes in Computer Science, vol. 8663, Springer, 2014, pp. 213–229.

[48] Mohammad Reza Mousavi, Michel A. Reniers, Jan Friso Groote, Notions of bisimulation and congruence formats for SOS with data, Inf. Comput. 200 (1) (2005) 107–147.

[49] Mohammad Reza Mousavi, Michel A. Reniers, Jan Friso Groote, SOS formats and meta-theory: 20 years after, Theor. Comput. Sci. 373 (3) (2007) 238–272.

[50] Iulian Neamtiu, Tudor Dumitraş, Cloud software upgrades: challenges and opportunities, in: International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems, 2011, pp. 1–10.

[51] Iulian Neamtiu, Michael W. Hicks, Safe and timely updates to multi-threaded programs, in: Michael Hind, Amer Diwan (Eds.), Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, ACM, 2009, pp. 13–24.

[52] Benjamin C. Pierce, Basic Category Theory for Computer Scientists, MIT Press, 1991.

[53] Benjamin C. Pierce, Chris Casinghino, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Brent Yorgey, Software foundations, e-book, http://www.cis.upenn.edu/~bcpierce/sf/, 2012.

[54] Luís Pina, Michael Hicks, Tedsuto: a general framework for testing dynamic software updates, in: IEEE International Conference on Software Testing, Verification and Validation, ICST, 2016, pp. 278–287.

[55] Luís Pina, Luís Veiga, Michael Hicks, Rubah: DSU for Java on a stock JVM, in: ACM SIGPLAN Notices, vol. 49, ACM, 2014, pp. 103–119.

[56] Vaughan R. Pratt, Semantical considerations on floyd-hoare logic, in: IEEE Symposium on Foundations of Computer Science, FOCS'76, 1976, pp. 109–121.

[57] Cristian Prisacariu, Olaf Owe, Dynamic Structural Operational Semantics (preliminary version). Technical Report 426, Department of Informatics, University of Oslo, December 2012. Online at http://folk.uio.no/cristi/papers/TR426.pdf.

[58] Mario Pukall, Christian Kästner, Walter Cazzola, Sebastian Götz, Alexander Grebhahn, Reimar Schröter, Gunter Saake, Javadaptor: flexible runtime updates of Java applications, Softw. Pract. Exp. 43 (2) (2013) 153–185.

[59] Grigore Roşu, Traian Florin Şerbănuţă, An overview of the K semantic framework, J. Log. Algebraic Program. 79 (6) (2010) 397–434.
[60] Peter Sewell, Francesco Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, Rok StrniŠa, Ott: effective tool support for the working semanticist, J. Funct. Program. 20 (1) (2010) 71–122.
[61] Gareth Stoyle, Michael W. Hicks, Gavin M. Bierman, Peter Sewell, Iulian Neamtiu, Mutatis Mutandis: safe and predictable dynamic software updating, ACM Trans. Program. Lang. Syst. 29 (4) (2007).
[62] L. Thomas van Binsbergen, Neil Sculthorpe, Peter D. Mosses, Tool support for component-based semantics, in: Lidia Fuentes, Don S. Batory, Krzysztof Czarnecki (Eds.), 15th International Conference on Modularity, ACM, 2016, pp. 8–11.
[63] Andrew K. Wright, Matthias Felleisen, A syntactic approach to type soundness, Inf. Comput. 115 (1) (1994) 38–94.
[64] Ingrid Chieh Yu, Einar Broch Johnsen, Olaf Owe, Type-safe runtime class upgrades in Creol, in: IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems, FMOODS, in: LNCS, vol. 4037, Springer, 2006, pp. 202–217.