Formal Component-Based Semantics

Ken Madlener
Institute for Computing and Information Sciences (iCIS),
Radboud University Nijmegen, The Netherlands
k.madlener@cs.ru.nl

Sjaak Smetsers
Institute for Computing and Information Sciences (iCIS),
Radboud University Nijmegen, The Netherlands
s.smetsers@cs.ru.nl

Marko van Eekelen
Institute for Computing and Information Sciences (iCIS),
Radboud University Nijmegen, The Netherlands
School of Computer Science, Open University of the Netherlands
m.vaneekelen@cs.ru.nl

One of the proposed solutions for improving the scalability of semantics of programming languages is Component-Based Semantics, introduced by Peter D. Mosses. It is expected that this framework can also be used effectively for modular meta theoretic reasoning. This paper presents a formalization of Component-Based Semantics in the theorem prover COQ. It is based on Modular SOS, a variant of SOS, and makes essential use of dependent types, while profiting from type classes. This formalization constitutes a contribution towards modular meta theoretic formalizations in theorem provers. As a small example, a modular proof of determinism of a mini-language is developed.

1 Introduction

Theorem prover formalization of programming language meta theory and semantics receives a lot of attention. Most notably, the POPLMARK Challenge [1] calls for experiments on verifications of meta theory and semantics using proof tools. One of the main issues that programming language formalizations have to cope with is the lack of reusability of existing work. Many programming languages have language constructs in common, but often have (slight) differences in their precise semantics (e.g. assignments in C versus assignments in JAVA).

Component-Based Semantics, introduced by Peter D. Mosses, aims to resolve this reusability issue by constructing language descriptions from combinations of basic abstract constructs [9]. Basic constructs are supposed to have a fixed meaning and be language-independent. As an example, the basic construct of conditional expressions should not depend on whether the expressions may have side-effects or not, terminate abruptly or even interact with other processes. One could even go as far as creating a repository of constructs that may be freely combined to build new languages. This repository is therefore necessarily open-ended, enabling users to add newly discovered basic constructs.

Modular Structural Operational Semantics (MSOS) [7], a variant of SOS, provides an adequate framework for the independent description of language components [9]. MSOS was designed to address the lack of reusability of SOS rules: *every* auxiliary entity used in a rule, such as an environment or a store, needs to be threaded through *all* rules of the language. MSOS provides a way to automatically propagate unmentioned entities between the premise(s) and conclusion of a rule, enabling the reuse of rules in different languages. SOS is very suitable for the formalization of languages and has therefore been widely adopted by the theorem prover community. MSOS has so far received less attention.

This paper proposes a formalization of Component-Based Semantics based on MSOS in the theorem

M.A. Reniers, P. Sobocinski (Eds.): Workshop on
Structural Operational Semantics 2011 (SOS 2011)
EPTCS 62, 2011, pp. 17–29, doi:10.4204/EPTCS.62.2

prover COQ [15].[1] Our main contribution is a way to constructively formalize programming language semantics: basic constructs can be developed in separate COQ files, which may be verified independently. The formalization has been tested by building a small repository of constructs. Moreover, it is possible to equip the constructs with small proofs that can be used to construct larger proofs of properties holding for a full language. For this reason, we shall use the term *component* instead of *construct* in this paper. Our formalization supports meta theoretic reasoning about a programming language, but does not support reasoning about the format of MSOS rules.

The formalization follows the original design of MSOS in its use of arrows of a category for the auxiliary entities (encapsulated in labels) appearing in the transition rules. A very elementary level of knowledge about category theory and a modest amount of familiarity with theorem proving is required to read this paper. Our formalization makes essential use of dependent types to formalize the labels in MSOS, and profits from COQ's support for type classes. Each component is represented by a parametrized so-called COQ section. To define a full language, it is sufficient to enumerate its components. The correct instantiation of the corresponding parameters can in principle be performed automatically by COQ's powerful type system.

2 Component-Based Semantics

We illustrate the description of programming languages in terms of basic abstract constructs by means of a while-loop example taken from [9]. Depending on what concrete language is being analyzed, a standard command such as while may have different interpretations. For example, if the language includes a break command that abruptly terminates the program throwing a particular exception, then the description of while should include the handler for that exception. We assume that $Cmd[\![\_]\!]$ and $Exp[\![\_]\!]$ are functions mapping concrete expressions to abstract expressions of **Cmd** and **Exp**, respectively. Below, cond-loop is a simple while-loop that takes an expression and a command, and propagates abrupt termination. The other constructs involved can be found in Table 1. The description is then:

$$\begin{aligned} Cmd[\![\mathsf{while}\ (E)\ C]\!] &= \mathsf{catch}(\mathsf{cond\text{-}loop}(Exp[\![E]\!], Cmd[\![C]\!]), \\ &\qquad\qquad \mathsf{abs}(\mathsf{eq}(\mathsf{breaking}), \mathsf{skip})) \\ Cmd[\![\mathsf{break}]\!] &= \mathsf{throw}(\mathsf{breaking}) \end{aligned}$$

A simple extension is a while-loop that handles continue commands. To describe such while-loops, all that is needed is to change the above example in such a way that $Cmd[\![C]\!]$ is encapsulated by a catch construct. Table 1 contains some possible constructs, which are used as examples throughout the rest of this paper. An example of an open-ended repository containing more constructs can be found in e.g. [9].

An important facet of Component-Based Semantics is that the construct repositories ideally contain no redundancy. If two basic constructs with different names have the exact same semantics, then one of them should be discarded. Moreover, if a construct can be expressed purely in terms of existing basic constructs, then this construct should also be discarded. A repository therefore essentially describes a universal language that can be used to define the semantics of a concrete language in question. This universal language provides a fixed name for each basic construct, which in our formalization corresponds to the name of a COQ file.

In the rest this paper we prefer to use the term *component* instead of *construct*, to emphasize we do not only refer to syntax when we use the term component, but also to its semantics and properties

[1] The source can be obtained at `http://www.cs.ru.nl/~kmadlene/fcbs.html`.

Syntactic Categories	
Cmd	commands
Exp	expressions
Dcl	declarations
Pcd	procedure abstractions
Prm	parameter patterns, encapsulating declarations

Constructs	
Cmd ::= seq (**Cmd**, ..., **Cmd**)	normal command sequencing
Cmd ::= skip	normal termination
Cmd ::= cond-loop (**Exp**, **Cmd**)	a simple while-loop, propagating abrupt termination
Cmd ::= catch (**Cmd**, **Pcd**)	tries to handle abrupt termination of **Cmd** by procedure abstraction **Pcd**
Cmd ::= throw **Exp**	terminates abruptly with the value of the **Exp**
Pcd ::= abs (**Prm**, **Cmd**)	a parametrized procedure abstraction (with static scoping)
Prm ::= eq **Exp**	a parameter that matches only the entity computed by the **Exp**.
Exp ::= block (**Dcl**, **Exp**)	locally binds **Dcl** in the **Exp**

Table 1: A basic repository.

that it may be equipped with. For the semantics of each component to be language-independent, it is necessary that it does not depend on 1) auxiliary entities that are not mentioned by the component, 2) the transition relation of the full language, and 3) abstract syntax of the full language. In our formalization we parametrize the components on these pieces of information. However, we first review MSOS, the framework our formalization is based on.

2.1 Modular SOS

In SOS, the operational semantics of a language with effects is modeled in by a *labeled transition system (LTS)* $\langle \Gamma, A, \rightarrow \rangle$, where Γ is the set of configurations, A is the *set of actions*, and $\rightarrow \subseteq \Gamma \times A \times \Gamma$ is the *transition relation* (sometimes called *step relation*). It is possible to consider more general transition systems that include terminal states, but these are only relevant when one considers computation traces, which is outside the scope of this paper. A straightforward example of a set of configurations that we will use below is $\mathbf{Cmd} \times \rho \times \sigma$. We will call ρ and σ *auxiliary entities*, or simply *entities*.

A drawback of SOS is its lack of support for modularity. It is sometimes necessary to update existing rules by decorating the transitions with additional entities, e.g. a second store to model a separate part of memory. If we were to add an auxiliary entity to the configurations, then this entity needs to be threaded through *all* the rules that define the semantics. This prevents the rules from being reusable, and therefore plain SOS is not a suitable framework for Component-Based Semantics. One can get around this problem *informally,* by implicitly propagating the entities that are not mentioned, by using a convention such as:

$$\frac{\rho \vdash \langle c_1, \sigma \rangle \rightarrow \langle c_1', \sigma' \rangle}{\rho \vdash \langle \mathsf{seq}\,(c_1, c_2), \sigma \rangle \rightarrow \langle \mathsf{seq}\,(c_1', c_2), \sigma' \rangle} \quad \rightsquigarrow \quad \frac{c_1 \rightarrow c_1'}{\mathsf{seq}\,(c_1, c_2) \rightarrow \mathsf{seq}\,(c_1', c_2)}$$

Normal command sequencing does not manipulate any of the entities and we can therefore assume that

$$\boxed{\text{Label} := \{\ldots\}}$$

$$\mathsf{seq}\,(\mathsf{skip}, c) \to c \tag{1}$$

$$\frac{c_1 \xrightarrow{\{X\}} c_1'}{\mathsf{seq}\,(c_1, c_2) \xrightarrow{\{X\}} \mathsf{seq}\,(c_1', c_2)} \tag{2}$$

Figure 1: Normal command sequencing

$$\boxed{\text{Label} := \{\rho : \text{env}, \ldots\}}$$

$$\frac{d \xrightarrow{\{X\}} d'}{\mathsf{block}\,(d, e) \xrightarrow{\{X\}} \mathsf{block}\,(d', e)} \tag{3}$$

$$\frac{e \xrightarrow{\{\rho=\rho_0[\rho_1],X\}} e'}{\mathsf{block}\,(\rho_1, e) \xrightarrow{\{\rho=\rho_0,X\}} \mathsf{block}\,(\rho_1, e')} \tag{4}$$

$$\mathsf{block}\,(\rho_1, v) \to v \tag{5}$$

Figure 2: Local bindings

they are propagated. This informal description style enables formulation of rules independent of the auxiliary entities that may or may not be present and thereby provides reusability of the rules.

MSOS is a variant of SOS that has special support for the propagation of unmentioned entities. The key distinction is that it separates phrases of the language from entities by moving the entities into a label on the transition. That is, transitions are of the form $\gamma \xrightarrow{\alpha} \gamma'$, such that γ and γ' merely consist of abstract syntax (which may include computed values), and α is a label containing the auxiliary entities. Before we discuss the associated transition systems, let us consider some examples of rules specified in MSOS. Figures 1 and 2 provide examples of normal command sequencing and local bindings. The abstract syntax is standard, and the meta-variables c, d, e, ρ and v stand for commands, declarations, expressions, environments and values, respectively.

The meta-variable X plays an important rôle in the rules. It binds the unmentioned entities, allowing us to propagate them between the premise(s) and conclusion of each rule, without specifically describing what these entities are. Different occurrences of X in the same rule stand for the same entities. Note that the rules assume neither the presence or absence of particular auxiliary entities: the only entities that are mentioned are the ones used by the transitions in the rule in question. The Label box specifies what entities the label should at least include. Entities in labels can be matched in rules using notation such as '$\{\rho = \rho_0[\rho_1], X\}$', where $\rho_0[\rho_1]$ stands for updating ρ_0 by ρ_1. Rules without labels on them are *unobservable*, meaning that they implicitly assume that the entities remain unchanged during the transition (e.g. in rule (1)). As an aside, we remark that skip too is a component: it has an empty label and an empty set of rules.

Mosses [7] recognized that the arrows of a category provide an adequate mathematical structure for labels. That is, two consecutive steps are only allowed to be made when their labels are composable, i.e., $\gamma \xrightarrow{p \longrightarrow q} \gamma' \xrightarrow{r \longrightarrow s} \gamma''$ is only allowed if if $q = r$. Hence, the associated transition systems are a triple $\langle \Gamma, A, \to \rangle$ similar to LTSes, with the difference that Γ strictly consists of abstract syntax, and the additional requirement that A are the arrows of a *label category* $\mathbb{A}$. The label category is a product of elementary categories that correspond to the entities, which we will discuss in Section 4. The values of the auxiliary entities are the objects of $\mathbb{A}$. As an example, a simple step with rule (1) looks as follows, if the label contains an environment and a store:

$$\mathsf{seq}\,(\mathsf{skip}, c) \xrightarrow{\langle \rho,\sigma \rangle \longrightarrow \langle \rho,\sigma \rangle} c \tag{6}$$

Identity arrows are used to express unobservability, used in e.g. rule (1).

3 Formalization

In Component-Based MSOS, the source configuration γ of a transition $\gamma \xrightarrow{\alpha} \gamma'$ plays a special rôle. Namely, it determines to which component the rule permitting that particular transition belongs. The formalization defines for each component a so-called local transition relation, which describes the rules for source configurations that belong to that particular component. Provided with the grammar of the full language, we construct the transition relation of the full language by combining the local transition relations. Components may optionally provide proof of a property that it satisfies, which can likewise be combined to build the proof of that property about the full language (if all components satisfy that property). This will be demonstrated in Section 5.

We make use of Coq's support for type classes [13] to automatically "fill in the details", i.e. combining the components and filling in the parameters to construct the full language. Type classes, however, are not strictly necessary for the formalization. It is possible in our formalization to construct several full languages from the same repository, but it is not possible to create an extension of an existing full language without completely specifying the extended language's grammar.

3.1 Types for transition relations

The transition relations of labeled transition systems (see Section 2.1) can be assigned the following type:

$\mathsf{Step}\ \Gamma\ \mathsf{A}\colon \Gamma \to \mathsf{A} \to \Gamma \to \mathsf{Prop}$

In other words, they are predicates which takes arguments γ, α and γ' and return an element of Prop (the built-in sort of propositional types in Coq). Just like the labeled transition systems associated with SOS specifications, there is no apparent distinction between syntax and the auxiliary entities.

Following the principles of MSOS, we update the type of Step to feature arrows of a category as labels on the transitions. Step now becomes parametric in the full label category $\mathbb{A}$ of the full language (which has a collection O of objects), resulting in the following type:

$\mathsf{Step}\ \Gamma\ \mathsf{O}\ (\mathbb{A}\colon \mathsf{Category}\ \mathsf{O})\colon \Gamma \to \mathsf{Arrows}\ \mathbb{A} \to \Gamma \to \mathsf{Prop}$

We have to remark that to avoid confusion, we are not following the exact syntax used in our formalization at this point. Moreover, we omit the definition of Category in this paper, but we elaborate on Arrows in Section 4.

Component-Based MSOS requires both a modular way to specify the step relation and a modular way to specify the abstract syntax. The component seq of Figure 1 implicitly specifies its own signature, namely the production rule **Cmd** ::= seq (**Cmd**, **Cmd**), and specifies two new rules. It also assumes that a syntactical category **Cmd** exists, and to be able to define rule (2), it assumes that a transition relation on **Cmd** exists. We therefore parametrize the component (i.e. its local transition relation and lemmas) with Γ, representing the syntactic category, the full transition relation S on Γ, and the component's construct C (where P is a type that stands for its parameters, see the next section). Since the components always define the semantics for precisely one construct of the language, we restrict the input configuration to the phrases built by that construct. We call the transition relation of a component a *local step*, to emphasize the difference with a transition relation defined on a full syntactic category.

$\mathsf{LocalStep}\ \Gamma\ \mathsf{O}\ (\mathbb{A}\colon \mathsf{Category}\ \mathsf{O})\ (\mathsf{S}\colon \mathsf{Step}\ \Gamma\ \mathsf{O}\ \mathbb{A})\ \mathsf{P}\ (\mathsf{C}\colon \mathsf{Construct}\ \mathsf{P}\ \Gamma)\colon$
$\quad \mathsf{restr}\ \mathsf{C} \to \mathsf{Arrows}\ \mathbb{A} \to \Gamma \to \mathsf{Prop}$

To define the full language, it is sufficient to enumerate the components it is built of. This results in a transition relation of type Step for each syntactic category, which we call a *global* step relation. This is described later on in this section.

3.2 Grammar

As a running example, we define a language that consists of just the components skip and seq (see Figure 1). Although it is a fairly simple example, it allows us to explain the formalization without having to get ahead too much on labels, which are treated in Section 4. The grammar of our skip-seq language is straightforwardly encoded by the following inductive type:

```
Inductive Cmd := skip | seq (c1 c2: Cmd).
```

Recall from Section 2 that each component is parametrized on its abstract construct. The arguments are passed on as an injection-projection pair which we will call Construct. Injection corresponds to applying a constructor and projection corresponds to pattern matching. Construct consists of two properties saying that i and p are (partial) inverses of each other. This is needed to prove properties about the component.

```
Class Inject P Γ := inject: P → Γ.
Class Project P Γ := project: Γ → option P.
Class Construct P Γ {i: Inject P Γ} {p: Project P Γ} := {
 H_i: ∀ x: P, p (i x) = Some x;
 H_p: ∀ γ: Γ, match project γ with
           | None ⇒ True
           | Some x ⇒ i x = γ end }.
```

For constructs that take several arguments, such as **Cmd** ::= seq (**Cmd**, **Cmd**), the arguments are tupled. The Class keyword declares the definitions to be type classes. The convenience of type classes is that class fields (such as inject or project) may be used without explicitly mentioning which instance of that class should be used. The curly brackets around i and p indicate that these arguments are implicit. In this case, these implicit arguments become class constraints, i.e., order to build an instance of Construct, instances of Inject and Project need to be present. For our example language, the corresponding instances are:

```
Instance: Inject unit Cmd := λ _, skip.
Instance: Inject (Cmd∗Cmd) Cmd := λ p, let (c1, c2) := p in seq c1 c2.

Instance: Project unit Cmd :=
  λ γ, match γ with
           | skip ⇒ Some tt
           | _ ⇒ None end.
Instance: Project (Cmd∗Cmd) Cmd :=
  λ γ, match γ with
           | seq c1 c2 ⇒ Some (c1, c2)
           | _ ⇒ None end.

Instance: Construct unit Cmd.
Instance: Construct (Cmd∗Cmd) Cmd.
```

The type class mechanism can be seen at work here: we do not have to specify the arguments i and p, for they can be resolved from the signatures. In fact, the manual declaration of these type class instances is straightforward and can be omitted by an augmentation of COQ's type class resolution algorithm, but we skip the details here. The reader may have noted that when the full language has two constructs with the same signature, the type class instance resolution algorithm may fill in the wrong Construct instance. This is solved in the formalization by adding an argument (i.e. a string) to Construct, enabling us to uniquely identify each instance.

Returning to the LocalStep type, the Construct argument is actually a class constraint (i.e. it is an implicit argument) in the formalization. In fact, the category and the Step relation are also class constraints. Some components require the presence of other components. For instance, the component seq "imports" the (very basic) component skip. To this end, the Skip construct becomes an additional constraint of seq. This does not interfere with modularity: all other details about the full language remain opaque.

3.3 Semantics

A straightforward way to encode transition relations in a theorem prover is by means of an inductive predicate [2]. Making the definition inductive guarantees that the only valid transitions are the ones that can be built by its constructors, which correspond to the rules. The encoding of rules is straightforward using nested implications, where universal quantifications are added for variables that occur in the rules. As an example, we give the transition relation for seq:

```
Inductive ls: restr Seq → Arrows 𝔸 → Cmd → Prop :=
| seq_1: ∀ c1 c2 c1' ar, step c1 ar c1' → ls (Seq · (c1, c2)) ar (i (c1', c2))
| seq_2: ∀ c2 ar, unobs ar → ls (Seq · (skip tt, c2)) ar c2.
```

The premise unobs ar expresses unobservability of the label, i.e., it has to stay unchanged. We have suppressed the class constraints here for readability. That is, ls requires suitable instances of Category, Step, Construct and Label (the latter is presented in Section 4). The type restr C is used to restrict phrases of the full language to ones built by constructor C. By means of an inductive type with a single constructor, we can ensure that the only way to build an instance of type restr C is by providing an object of P:

```
Inductive restr `(C : Construct P Γ) := restr_cons (γ: Γ).
Notation "C · γ" := (restr_cons C γ) (at level 50, left associativity).
```

The backtick performs implicit generalization: necessary variables to the argument C are automatically declared as implicit arguments of restr. Writing e.g. Seq · (c_1, c_2) is similar to applying the "real" constructor (e.g. seq c_1 c_2), but not exactly the same. One can obtain c_1, c_2 by straightforward pattern matching on restr_cons. In contrast, it is only possible obtain c_1, c_2 from seq c_1 c_2 by using the elimination principle of Cmd, which is not available inside the component.

The inductive predicate ls is made into a type class instance to enable resolution:

```
Instance LS_Seq: LocalStep O := ls.
```

The semantics of the full language is essentially defined by a case distinction on the constructors of the datatypes. The full step relation is defined as an inductive predicate s that combines the existing local step relations of the used components into one global step relation. This is done by means of an inductive predicate that has a single constructor. The constructor assumes a localstep of any of the local transition relations of the syntactic category in question (passing along s itself), and returns an object of s (as above, in ls). The reader interested in the details is referred to the source code. This construction satisfies equations such as:

$$\mathsf{localize\ Skip\ S\_Cmd} = \mathsf{LS\_Skip}$$
$$\mathsf{localize\ Seq\ S\_Cmd} = \mathsf{LS\_Seq}$$

The operator localize maps the given Step instance (in this case S_Cmd) to the canonical LocalStep w.r.t. the provided construct. These equations are necessary to prove properties about the components. For example, consider the component seq, which imports the component skip. To be able to prove properties about seq, the local step relation of skip (which is empty) needs to be accessible. This is done by passing on the first equation as an argument. The equality is overloaded with the obvious meaning that the Step instances agree on all inputs (i.e. ar, γ and γ'). In conjunction with COQ's built-in support for setoid rewriting (rewriting modulo an equivalence relation), this enables us to perform short proofs for meta theory (used in Section 5).

4 Labels

Auxiliary entities such as environments and stores in SOS are encapsulated in a label on the transitions in MSOS. In Section 2 we have explained that the labels on the transitions have the structure of arrows of a category: the labels of consecutive transitions should be composable. A subtle difference between MSOS and SOS is that the chosen label category may restrict the transition relation specified by the rules, whereas in SOS it is solely the rules that determine this relation. This can be seen by assuming the label category to be a discrete category, i.e., the category with just identity arrows.

Mosses [7] has shown that a suitable category is the product $\mathbb{A} \mathrel{\hat{=}} \prod_{i \in I} \mathbb{A}_i$ of elementary categories representing the auxiliary entities. The usual types of entities used in SOS rules are environments, stores and labels, which correspond to read-only, read-write or write-only permissions, respectively. In MSOS, each entity (with index i) has a corresponding set of objects S_i that, together with the permissions, determines its corresponding category $\mathbb{A}_i$:

- read-only: $\mathbb{A}_i$ is the discrete category with S_i as its objects;
- read-write: $\mathbb{A}_i$ is the pre-order category with S_i as its objects, and S_i^2 as its morphisms;
- write-only: $\mathbb{A}_i$ is the category with a single object $*$, and the free monoid on S_i as its morphisms.

A distinguishing feature of MSOS is its inherent support for write-only entities. For example, a transition in a system with a single write-only entity can be pictured as $\gamma \xrightarrow{* \longrightarrow *} \gamma'$. If it appears as the conclusion of a rule, then the premises of that rule can not possibly depend on the value of that entity, because it is simply $*$. For this reason, we have adopted the use of arrows as labels in our formalization. An alternative is to consider a relation on a product of entities as the label category. This is a special case that does not provide true support for write-only entities.

Recall that the components are parametrized by a label category $\mathbb{A}$ on a collection of objects O. To build the product category, O is instantiated with the entity map $i \mapsto \mathbb{A}_i$. Inside the component, the label category is entirely opaque. In other words, it is impossible to learn anything from $\mathbb{A}$ except that it is a product category. The Label box in the component specification expresses what entities the full label should *at least* include. For example, Figure 2 requires that the full label includes an environment entity.

This is reflected in our formalization by providing two functors P_M and P_U to each component, that project full labels to their mentioned entities and unmentioned entities, respectively:

$$\mathbb{A} \xrightarrow{\mathsf{P}_M} \prod_{i\in M} \mathbb{A}_i, \qquad \mathbb{A} \xrightarrow{\mathsf{P}_U} \mathbb{U}.$$

The idea is that the product of mentioned entities is transparent to the component, whereas $\mathbb{U}$ is opaque. We use the functor P_U to express unobservability, needed e.g. in rule (1). Additionally, the component requires that $(\mathsf{P}_M, \mathsf{P}_U)$ is an isomorphism, which is crucial to enable modular proof. Let us consider determinism as an illustration of this.

Property 1 *Assume configurations $\gamma\ \gamma'\ \gamma'' \colon \Gamma$ and labels $ar' \colon x \longrightarrow y$, $ar'' \colon x \longrightarrow z$. The step relation on Γ is* deterministic *when both $\gamma \xrightarrow{\mathsf{ar}'} \gamma'$ and $\gamma \xrightarrow{\mathsf{ar}''} \gamma''$ imply that $\gamma' = \gamma''$ and $ar' = ar''$.*

The requirement that the arrows are equivalent ensures not only that the post configurations are equal, but also the outputs through the write-only components are equal. To prove that the component seq is deterministic, one proceeds by straightforward case analysis on the structure of the input configuration. In the case that it is $\mathsf{seq}(\mathsf{skip}, c)$, we have two arrows ar', ar'' such that $\mathsf{P}_U\ \mathsf{ar}' = \mathsf{P}_U\ \mathsf{ar}'' = id$, and $\mathsf{P}_M\ \mathsf{ar}' = \mathsf{P}_M\ \mathsf{ar}'' = ()$ (the empty tuple). In other components that do have mentioned entities, these projections of P_M have to be equivalent. Using the isomorphism we can then conclude that $\mathsf{ar}' = \mathsf{ar}''$.

4.1 Formalization of labels

The category theory we have used in our formalization is provided by the MATH-CLASSES library by van der Weegen and Spitters [14]. Their library makes extensive use of a technique called “unbundling”, which boils down to separating the components of mathematical structures into separate type classes. An example of this are categories. In Section 3.1, we have treated Category as a record structure containing Arrows as a field for presentation purposes. However, in the actual formalization, Arrows is a separate type class:

```
Class Arrows (O: Type): Type :=  Arrow: O → O → Type.
Infix "⟶" :=  Arrow (at level 90, right associativity).
```

To build a Category, among other components, an equivalence relation on the corresponding Arrows instance is necessary, to enable the comparison of arrows. We use this relation in our formalization to define the predicate unobs for unobservability. The following instances are used for the entity categories:

```
Instance arrows_ro: Arrows O :=  λ x y, x = y.
Instance arrows_rw: Arrows O :=  λ x y, unit.
Instance arrows_wo: Arrows unit :=  λ x y, list O.
```

We now define the type class Label, which is used to provide the projection functors. Label assumes the presence of the following objects:

```
I M: Type
O: I → Type
A: ∀ i: I, Arrows (O i)
O_M: M → Type
A_M: ∀ i: M, Arrows (O_M i)
```

In other words, for both index sets I and M it is required that a collection of arrows exists.

```
Class Label := {
 cover_O: ∀ i: M, O_M i = O (to_I i);
 cover_A: ∀ i: M, A_M i = ⟪ λ T, Arrows T | eq_sym (cover_O i) ⟫ A (to_I i) }.
```

The cover_O property says that for every index of the mentioned entities, the objects have to correspond to the objects of the full category. Likewise, the arrows of the mentioned entities have to correspond. A cast operation [6] on the objects (indicated by ⟪_|_⟫) is needed to be able to express the latter, but we omit the details in this paper. Given an instance of Label, we can derive the functors P_M and P_U together with the fact that they are isomorphic. Each component has a Label type class constraint which leaves O and A parametric, but specifies O_M and A_M.

To illustrate how a rule is interpreted with help of the Label construction, we consider rule (4) of Figure 2. Let us first write it using informal notation. Assume that $\mathsf{ar}\colon \mathsf{x} \longrightarrow \mathsf{y}$, $\mathsf{ar}'\colon \mathsf{x}' \longrightarrow \mathsf{y}'$ and $\mathsf{proj}_\rho = \pi_\rho \circ \mathsf{P}_M$ is the projection of the component with index ρ.

$$\frac{\mathsf{proj}_\rho\ \mathsf{x}' = \rho_0[\rho_1] \qquad \mathsf{proj}_\rho\ \mathsf{x} = \rho_0 \qquad \mathsf{P}_U\ \mathsf{ar} = \mathsf{P}_U\ \mathsf{ar}' \qquad e \xrightarrow{\mathsf{ar}'} e'}{\mathsf{block}\,(\rho_1, e) \xrightarrow{\mathsf{ar}} \mathsf{block}\,(\rho_1, e')}$$

In COQ-syntax, this rule is:

```
rule4:
  ∀ (ρ0 ρ1: Env) (e e′: Exp) ‘(ar′: x′ ⟶ y′),
    projρ x′ = update ρ0 ρ1 → projρ x = ρ0 →
    fmap_PU ar = fmap_PU ar′ → step ar′ e e′ →
(* ---------------------------------------------- *)
    ls (Block · (ρ1, e)) ar (i (ρ1, e′))
```

Note that the use of equality in the above code is highly overloaded, which is made possible by the use of type classes. Like the MATH-CLASSES library, we represent the functors by means of a function that maps the objects, which have the actual names P_M and P_U, and functions that map the arrows, which have the fmap_ prefix.

5 Example of Modular Proof

Once the full language is declared, it is possible to combine proofs of the components to prove that a particular property holds for the full language. Like the local step relations, properties are parametrized by a global step relation S. We say that a property holds for a step relation if it holds for all the possible configurations, but we are a bit more general and allow the user to express that a property holds for a particular configuration.

Not all properties can be proved by induction, and likewise not all properties have a modular proof. We consider a class of admissible, well-behaved properties P such that P S (I γ) does not depend on anything but the localized version of S w.r.t. C (here I γ injects γ into Γ):

```
Definition admissible Γ O (P: Step Γ O 𝔸 → Γ → Prop) :=
  ∀ ‘(C: Construct A Γ) (S: Step Γ O 𝔸) (γ: restr C),
    P (globalize (localize C S)) (I γ) → P S (I γ).
```

The operator globalize is the inverse of localize: it takes a local step relation ls and makes it global, behaving like ls on phrases constructed by C and not permitting any steps to be made that start from other configurations. The idea of admissible properties is that they warrant that proof by induction is possible.

Lemma 1 *Determinism is admissible.*

We will demonstrate how this lemma is used to show that our skip-seq language is deterministic by illustrating the seq case (skip is similar). Inside the COQ section of seq, we have proved the following lemma that says that the component is deterministic.

Lemma det_Seq (c_1 c_2: Cmd): det_global S_Cmd c_1 → det_local LS_Seq (Seq · (c_1, c_2)).

Note that it assumes that the global step relation is deterministic on c_1, which is essentially the induction hypothesis.

Recall the equivalence relations on Step, LocalStep of Section 3. Both det_global and globalize respect this relation. Using COQ's built-in support for rewriting modulo equivalence relations (called setoid rewriting), it can be shown that:

$$\begin{array}{lll}
 & \mathsf{det\_global\ S\_Cmd\ (seq\ c_1\ c_2)} & \text{(fold I)} \\
= & \mathsf{det\_global\ S\_Cmd\ (I\ (Seq\cdot(c_1,\ c_2)))} & \text{(Lemma 1)} \\
= & \mathsf{det\_global\ (globalize\ (localize\ Seq\ S\_Cmd))\ (I\ (Seq\cdot(c_1,\ c_2)))} & \text{(rewrite eq\_Seq)} \\
= & \mathsf{det\_global\ (globalize\ LS\_Seq)\ (I\ (Seq\cdot(c_1,\ c_2)))} & \text{(fold det\_local)} \\
= & \mathsf{det\_local\ LS\_Seq\ (Seq\cdot(c_1,\ c_2))} &
\end{array}$$

Now, the latter holds because this is a property proved in the component seq. The proof for seq can therefore be completed by applying det_Seq, using the equation localize Skip S_Cmd = LS_Skip and the induction hypothesis.

Other components follow the same prescription. In future work, we want to automate the weaving of local proofs by generalizing the above, and exploiting automated proof search with the help of the type class mechanism in COQ. Experiments have already demonstrated that this is feasible, but fragile.

6 Related Work

A specification language for MSOS, called the MSOS Definition Formalism (MSDF), has been developed by Mosses and Chalub (see [3]). It combines BNF notation with textual representation of MSOS transitions. A large number of basic components have already been identified and specified in MSDF. A tool that translates ML and (a part of) JAVA into this repository have been developed by Chalub and Braga [3], which can be executed in the MAUDE tool. MSDF provides its own specification language for datatypes, which can be constructed from primitives such as sequences, lists, maps, etc. In contrast, our formalization directly uses types defined in COQ.

Implicit-MSOS is an improvement of MSOS that reduces the amount of clutter in the rules even further by implicitly propagating unmentioned entities [10]. The interpretation of Implicit-MSOS is given in terms of MSOS, and we expect that it can be built on top of our formalization by clever use of type classes.

Delaware et al. [5] have very recently investigated the possibility of modular metatheory in COQ. Their focus is on extending a programming language with new features, taking Featherweight JAVA as an example. In their paper, they demonstrate how to develop a modular proof of type-safety of a number of concrete extensions of Featherweight JAVA. The considered extensions do not have effects, i.e., there are no entities.

The formalization of the operational semantics of OCAML$_{\text{light}}$ in HOL by Scott Owens makes use of labels to encode mutations to the store in them [11]. These mutations are correlated to a reduction in the program. The labels explicitly carry mutations and therefore simplify the notation, but do not enable a high degree of reusability of the rules.

In a theorem prover (and functional languages), abstract syntax and transition relations are typically encoded as inductive types, of which the constructors correspond to the grammar production rules and the constructors correspond to the rules of the step relation. The inductive definition ensures that those constructors are the only way to build instances of those types. This corresponds to the notions “initial algebra” and “least relation”, sometimes used in this context (e.g. [10]). To facilitate Component-Based Semantics, we have to be able to build these inductive types from “partial versions” that define just the rules and production rules of the component in question. To our best knowledge, there is no theorem prover (or functional language) that supports (multiple) inheritance of inductive types natively.

7 Conclusions and Future Work

In this paper we have presented a formalization of Component-Based Semantics in the theorem prover COQ. The formalization makes essential use of dependent types, and profits from COQ’s support for type classes. Our formalization is based on the ideas of MSOS, and makes use of the idea of labels as arrows in categories, as proposed by Mosses [7]. Splitting the label category into a transparent part for the mentioned entities and an opaque part for the unmentioned entities enables modular proof. We have demonstrated this by crafting a proof of determinism of a mini-language from smaller local proofs provided by the components used.

In future work we plan apply this work with the aim of scalable verification of specific programs. Another direction of research is to investigate whether the full generality of labels as arrows (which our formalization provides) can be exploited for entities of types other than read-only, read-write and write-only. We expect that by choosing a suitable category, it is possible to enforce information flow policies, which has applications to security. Our work also enables formal investigation of the appropriate definitions of bisimulation in MSOS, which as of now has an experimental status [7].

Acknowledgments. The authors wish to thank Peter D. Mosses for introducing them to the notion of Component-Based Semantics, and Bas Spitters for introducing them to type classes in COQ. The authors would also like to thank Peter D. Mosses, Julien Schmaltz and the anonymous reviewers for their comments on an earlier version of this paper.

References

[1] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich & Steve Zdancewic (2005): *Mechanized Metatheory for the Masses: The POPLMARK Challenge*. In: *TPHOLs*, *LNCS* 3603, Springer, pp. 50–65, doi:10.1007/11541868_4.

[2] Yves Bertot, Gérard Huet, Jean-Jacques Lévy & Gordon Plotkin, editors (2009): *Theorem proving support in programming language semantics*, chapter 15, pp. 337–361. Cambridge University Press. Available at http://hal.inria.fr/inria-00160309/.

[3] Fabricio Chalub & Christiano Braga (2007): *Maude MSOS Tool*. *ENTCS* 176(4), pp. 133–146, doi:10.1016/j.entcs.2007.06.012.

[4] Adam Chlipala (2011): *Certified Programming with Dependent Types*. Available at `http://adam.chlipala.net/cpdt/`. To appear.

[5] Benjamin Delaware, William R. Cook & Don Batory (2011): *Modular Mechanized Metatheory*. In preparation.

[6] Chung-Kil Hur (2010): *Heq: A Coq Library for Heterogeneous Equality*. Available at `http://www.pps.jussieu.fr/~gil/Heq/`. Informal presentation at the 2nd Coq Workshop.

[7] Peter D. Mosses (2004): *Modular Structural Operational Semantics*. *J. of Logic and Algebraic Programming* 60-61, pp. 195–228, doi:`10.1016/j.jlap.2004.03.008`.

[8] Peter D. Mosses (2005): *A Constructive Approach to Language Definition*. *J. of Universal Computer Science* 11(7), pp. 1117–1134, doi:`10.3217/jucs-011-07-1117`.

[9] Peter D. Mosses (2009): *Component-Based Semantics*. In: *Proc. of SAVCBS'09*, ACM Press, pp. 3–10, doi:`10.1145/1596486.1596489`.

[10] Peter D. Mosses & Mark J. New (2009): *Implicit Propagation in Structural Operational Semantics*. *ENTCS* 229(4), pp. 49–66, doi:`10.1016/j.entcs.2009.07.073`.

[11] Scott Owens (2008): *A Sound Semantics for OCaml light*. In: *Proc. of ESOP'08*, *LNCS* 4960, Springer, pp. 1–15, doi:`10.1007/978-3-540-78739-6_1`.

[12] Gordon D. Plotkin (2004): *A Structural Approach to Operational Semantics*. *J. of Logic and Algebraic Programming* 60-61, pp. 17–139.

[13] Matthieu Sozeau & Nicolas Oury (2008): *First-Class Type Classes*. In: *TPHOLs*, *LNCS* 5170, Springer, pp. 278–293, doi:`10.1007/978-3-540-71067-7_23`.

[14] Bas Spitters & Eelis van der Weegen (2011): *Type Classes for Mathematics in Type Theory*. *MSCS* 21, pp. 1–31, doi:`10.1017/S0960129511000119`.

[15] The Coq Development Team (2010): *The Coq Proof Assistant Reference Manual – Version V8.3*. Available at `http://coq.inria.fr`.