GSOS Formalized in Coq

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

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

***Abstract*—Structural operational semantics provides a well-known framework to describe the semantics of programming languages, lending itself to formalization in theorem provers. The formalization of syntactic SOS rule formats, which enforce some form of well-behavedness, has so far received less attention. GSOS is a rule format that enjoys the property that the operational semantics and denotational semantics, both derived from the same set of GSOS rules, are consistent. The present paper formalizes the underlying theory in the theorem prover COQ, and proves the consistency property, also known as the *adequacy theorem*. The inspiration for our work has been drawn from the field of *bialgebraic semantics*.**

I. INTRODUCTION

Operational and denotational semantics are two well-known approaches to assigning a meaning to programming languages and process algebras. Around fifteen years ago, Turi and Plotkin [17] developed a framework that unifies both these styles. Using the language of category theory, they managed to strip away language-specific details such as concrete syntax and behavior. Given a set of operational rules, they have shown how to derive both the operational and denotational semantics from a distributive law corresponding to a set of operational rules. Their result pertains to several syntactic operational rule formats [2], [6], but the GSOS format [4] is the most prominent one of them [2], [10]. For instance, classic languages such as basic process algebra and the language WHILE [3] can be described by the GSOS format. Although implementations based on Turi and Plotkin's work have previously been developed in HASKELL by Hutton [7], Jaskelioff, Hutton and Ghani [9], and Hinze and James [6], formalized proofs in a theorem prover have not yet been provided.

The contributions of this paper are the following.

- It provides an implementation of both an operational and a denotational semantics derived from a set of GSOS rules, and a proof of their consistency (called the *adequacy theorem*), fully developed in the theorem prover COQ [16], using novel theorem proving techniques.
- A generic theory for syntactic terms, also fully developed in COQ, which is needed for the proof of the adequacy theorem.
- It lays out the foundations for further work on the formalization of bialgebraic semantics.

The advantage of a development in the constructive logic of COQ is that it enables both the execution of and formal proofs about the semantics at hand. The work in this paper is heavily inspired by Turi and Plotkin's bialgebraic semantics. The presented formalization is a shallow embedding into COQ's `Type` (i.e. the type of types), and does not possess the full generality of Turi and Plotkin's category theoretic work. However, it is still far more general than previous work on programming language semantics in theorem provers which usually concentrates on the study of a concrete language, such as [3].

An important tenet of most theorem provers, including COQ, is that every function must terminate, otherwise the underlying logic would be inconsistent. This seemingly superficial difference with HASKELL has profound implications for the development presented in this paper. In order to satisfy the syntactic checks which COQ performs on definitions to guarantee termination, the types representing the syntax and behavior of the language must be chosen carefully. As we will see, COQ's support for dependent types can be put to good use. The semantic domain potentially consists of infinite objects, as shown in the examples provided in this paper. In contrast to HASKELL, there is a clear distinction between finite and infinite worlds in COQ. The standard syntactic equality of COQ is not general enough for serious proofs about infinite objects. We have based the COQ formalization on the use of setoids, i.e. a `Type` packaged with a user-defined notion of equality and a proof of well-behavedness of the equality.

To make the content in this paper accessible to readers with limited exposure to the field of bialgebraic semantics and COQ, we explain the computational side of this work in Section II, using a simple language for the construction of streams as our running example. Furthermore, we start off with a more limited rule format to sketch the main ideas, and then treat the GSOS format. The reader is expected to have a modest amount of familiarity with category theory. All definitions and theorems in this paper have been formalized and proved in COQ. The interested reader can step through each of the proofs in the COQ-script available on the web via http://www.cs.ru.nl/~kmadlene/adequacy/.

II. A SIMPLE STREAM LANGUAGE

In this section we discuss the relation between operational and denotational semantics similar to the way they arise in the framework of Turi and Plotkin [17]. We use a simple language about streams (also used by Klin [10] in his introduction to bialgebraic semantics) as our running example, allowing us to explain the basics and provide some hints towards the COQ implementation. An example of a more involved language featuring non-determinism will be given in Section V. This section is almost exclusively limited to the computational side

$$\frac{}{AS \xrightarrow{\mathsf{a}} AS} \qquad \frac{}{BS \xrightarrow{\mathsf{b}} BS}$$

$$\frac{x \xrightarrow{l} x' \qquad y \xrightarrow{m} y'}{Alt\ x\ y \xrightarrow{l} Alt\ y'\ x'} \qquad \frac{x \xrightarrow{l} x'}{Zip\ x\ y \xrightarrow{l} Zip\ y\ x'}$$

Fig. 1. A simple language for streams.

of our work, and the code presented in this section is perfectly executable within Coq.

Consider the simple stream language defined by the operational rules in Figure 1. For now we disregard the operation *Zip*. These rules inductively define a transition relation between terms, composed of the operations AS, BS, Alt, and a label, a or b. The operations AS and BS respectively generate the streams $\mathsf{aaaa}\cdots$ and $\mathsf{bbbb}\cdots$, and the operation Alt generates the alternation between its two provided streams.

The syntax of the language is specified by its signature: a set of function symbols each equipped with a fixed arity. Such a signature is encoded as a functor which we will call the *signature functor*. If Σ denotes the signature functor, then the terms are least fixpoint of Σ, usually denoted as $T := \mu X, \Sigma X$. The corresponding types are:

$$\textbf{Inductive}\ \Sigma\ X := AS \mid BS \mid Alt\ (x\ y : X)$$
$$\textbf{Inductive}\ T := app\ (\sigma : \Sigma\ T)$$

Here X is a **Type**, and app is a constructor of T with the type $\Sigma\ T \to T$. The separation of Σ from T will be important in the rest of the paper. We provide the function map corresponding to the signature functor as well. This is an instance of the type-class $SFmap$ (i.e. setoid function map), which will be discussed in more depth in Section IV-A.

$$\begin{aligned}
&\textbf{Class}\ SFmap\ (M : \textbf{Type} \to \textbf{Type}) :=\\
&\quad sfmap_M : \forall\ X\ Y\ (f : X \to Y), M\ X \to M\ Y\\
&\textbf{Instance} : SFmap\ \Sigma :=\\
&\quad \lambda\ X\ Y\ (f : X \to Y)\ x,\\
&\qquad \textbf{match}\ x\ \textbf{with}\\
&\qquad \mid AS \quad \Rightarrow AS\\
&\qquad \mid BS \quad \Rightarrow BS\\
&\qquad \mid Alt\ x\ y \Rightarrow Alt\ (f\ x)\ (f\ y)\\
&\qquad \textbf{end}
\end{aligned}$$

A. Operational Semantics

To encode the transition relation we also have to represent the behavior of the system. This again is done with a functor, which we call a *behavior functor*. The data type L corresponds to our label set.

$$\begin{aligned}
&\textbf{Inductive}\ L := \mathsf{a} \mid \mathsf{b}\\
&\textbf{Definition}\ B\ X := L \times X\\
&\textbf{Instance} : SFmap\ B :=\\
&\quad \lambda\ A\ B\ (f : A \to B)\ a,\\
&\qquad \textbf{let}\ (l, x) := a\ \textbf{in}\ (l, f\ x)
\end{aligned}$$

We can now define a transition system as a B-*coalgebra*, i.e. a pair consisting of a state-space, in this case T, and a structure map of type $T \to B\ T$. In this paper we refer to the structure map as the coalgebra.

$$\begin{aligned}
&\textbf{Fixpoint}\ OM\ (t : T) : B\ T :=\\
&\quad sfmap_B\ app\ (\\
&\qquad \textbf{match}\ t\ \textbf{with}\\
&\qquad \mid app\ AS \quad \Rightarrow (\mathsf{a}, AS)\\
&\qquad \mid app\ BS \quad \Rightarrow (\mathsf{b}, BS)\\
&\qquad \mid app\ (Alt\ x\ y) \Rightarrow\\
&\qquad\quad \textbf{let}\ (l, x') := OM\ x\ \textbf{in}\\
&\qquad\quad \textbf{let}\ (m, y') := OM\ y\ \textbf{in}\ (l, Alt\ y'\ x')\\
&\qquad \textbf{end})
\end{aligned}$$

One may think of OM as a model of the operational semantics of the language in question (cf. the rules in Figure 1), as it specifies for each state what the next step would be. We can run a term by coiteratively unfolding OM, resulting in a stream of labels. The streams are actually the greatest fixpoint of the present behavior functor [8], usually denoted as $\nu X, BX$. This leads to the following definitions:

$$\begin{aligned}
&\textbf{CoInductive}\ Z_B := in_B\ (x : B\ Z_B)\\
&\textbf{CoFixpoint}\ unfold_B\ `(c : X \to B\ X)\ (x : X) : Z_B :=\\
&\quad in_B\ (sfmap_B\ (unfold_B\ c)\ (c\ x))\\
&\textbf{Definition}\ run := unfold_B\ OM
\end{aligned}$$

Thus, in_B is the constructor of the streams Z_B, and has type $B\ Z_B \to Z_B$. We call run the *operational semantics*, which is derived from OM, the *operational model*.

In this paper we will only consider behavior functors B that have a final coalgebra. The existence of final coalgebras is a subtle matter; in Haskell it is possible to define the greatest fixpoint of an arbitrary functor (which acts as the state-space of the final coalgebra), but in Coq the same definition is illegal due to the inability of Coq to guarantee the termination of every function in that case. The same goes for the least fixpoint operator, for the terms. The least and greatest fixpoint operators in Haskell-code would be respectively (the argument f being the functor):

$$\begin{aligned}
&\textbf{data}\ \mu\ f \quad = \mu\ (f\ (\mu\ f))\\
&\textbf{codata}\ \nu\ f = \nu\ (f\ (\nu\ f))
\end{aligned}$$

However, because there is no distinction in Haskell between finite and infinite worlds, these operators are exactly the same.[1]

The structure map of the final coalgebra is called out_B:

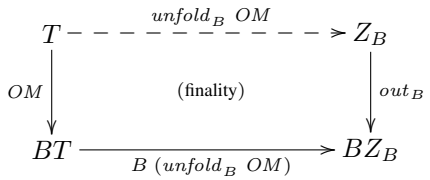
$$\begin{aligned}
&\textbf{Definition}\ out_B\ (z : Z_B) : B\ Z_B :=\\
&\quad \textbf{match}\ z\ \textbf{with}\\
&\quad \mid in_B\ x \Rightarrow x\\
&\quad \textbf{end}
\end{aligned}$$

An important property of the final coalgebra is that $unfold_B$ is the only function that makes the following diagram commute[2],

[1] In Haskell, the keywords **data** and **codata** are only used to indicate the *intended* use of the datatype, but can be interchanged.

[2] In the diagrams of this paper we will adopt the categorical notation for functors by writing F instead of $sfmap_F$, for some functor F, i.e., we explicitly indicate the instance type, and leave $sfmap_F$ itself implicit. Moreover, we omit parentheses in the notation of types.

and this function is run (= $unfold_B\ OM$).



The intuition behind the above diagram for the concrete behavior functor B for streams, used in this section, is that splitting a label off the stream generated by unfolding OM is the same as performing one step and unfolding OM on the resulting term.

B. Denotational Semantics

As in [17] we consider the denotational semantics as a dual version of the operational semantics. The underlying denotational model actually operates directly on elements of the semantic domain of our stream language, i.e. the streams Z_B. For the present example this means that it prescribes how the operations of the language, which receive streams as arguments, yield new streams. The semantic functions corresponding to each of the operations are:

CoFixpoint $denAS : Z_B := in_B\ (a, denAS)$
CoFixpoint $denBS : Z_B := in_B\ (b, denBS)$
CoFixpoint $denAlt\ (x\ y : Z_B) : Z_B :=$
  match (x, y) **with**
  $\mid (in_B\ (l, x'), in_B\ (\_, y')) \Rightarrow in_B\ (l, denAlt\ y'\ x')$
  end

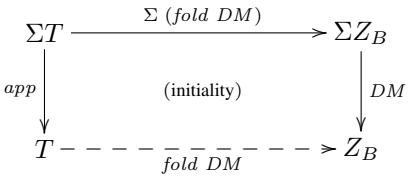
From the above functions we can define the full denotational model:

Definition $DM\ (\sigma : \Sigma\ Z_B) : Z_B :=$
  match σ **with**
  $\mid AS \quad \Rightarrow denAS$
  $\mid BS \quad \Rightarrow denBS$
  $\mid Alt\ x\ y \Rightarrow denAlt\ x\ y$
  end

In a fashion dual to the operational side, the denotational semantics evaluates a term, by folding the algebra DM over that term.

Fixpoint $fold\ `(h : \Sigma\ X \to X)\ (t : T) : X :=$
  match t **with**
  $\mid app\ \sigma \Rightarrow h\ (sfmap_\Sigma\ (fold\ h)\ \sigma)$
  end
Definition $eval : T \to Z_B := fold\ DM$

The terminology is that DM is the *denotational model*, while $eval$ is the *denotational semantics*. Likewise, there is a unique function making the following diagram commute, and this function is $eval$ (= $fold\ DM$).



III. Framework

The adequacy theorem in the case of the simple stream example says that executing run and $eval$ on the same term yields the same stream. With the definitions as they stand, a proof of it would proceed by induction on the terms of the stream language, and would therefore be rather ad hoc. A more structured development will be laid out in the present section, based on the use of a distributive law to represent operational rules. From this distributive law we derive both operational and denotational models.

As we have detailed in the previous section, it is not possible to take arbitrary fixpoints in Coq due to limitations imposed by its logic. To develop our theory independent of a particular signature or type of behavior, we will assume the existence of least/greatest fixpoints with the appropriate properties. That is, the fixpoints and their properties are parameters of the theory we develop in this paper. This is all possible in Coq as proofs and programs are in the same syntactic class, in true Curry-Howard style. The question is then how to realize these fixpoints in a fairly generic fashion. We do this for the terms in the present paper.

A. Generic Terms

A more general version of T that does not directly depend on a specific signature can not be obtained by making T parametric in the signature in Coq, as we have explained earlier. Instead, we fix T on the signature Σ, and Σ should have a certain shape. We will discuss the details of this in Section IV-B. To allow for open terms (used later on to represent meta-variables X in the operational rules) the constructor $var : X \to T\ X$ has been added.

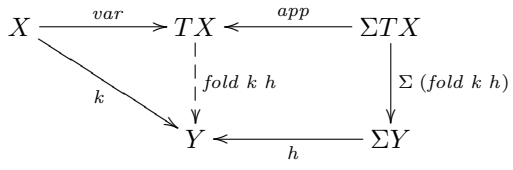
Inductive $T\ X := var\ (x : X) \mid app\ (\sigma : \Sigma\ (T\ X))$

T is also called the *free monad generated by* Σ. It is straightforward to generalize the $fold$ provided earlier to the new version of T.

Fixp. $fold\ (k : X \to Y)\ (h : \Sigma\ Y \to Y)\ (t : T\ X) : Y :=$
  match t **with**
  $\mid var\ x \Rightarrow k\ x$
  $\mid app\ \sigma \Rightarrow h\ (sfmap_\Sigma\ (fold\ k\ h)\ \sigma)$
  end

The $fold$ operation provides a recursive definition principle that avoids explicit recursion (see e.g. [12]): one only has to specify a mapping of the variables $k : X \to Y$, and an algebra $h : \Sigma\ Y \to Y$. This result is attributed to the following lemma.

Lemma 1. *Let $k : X \to Y$, and $h : \Sigma\ Y \to Y$. Then $fold\ k\ h$ is the unique function making the following diagram commute:*



The equalities $fold\ k\ h \circ var = k$ and $fold\ k\ h \circ app = h \cdot \Sigma\ (fold\ k\ h)$ depicted in the above diagram should be interpreted extensionally. Through the use of type-classes we

have overloaded the standard notion of equality in Coq to be the extensional equality, and moreover, using equality with respect to custom notions of the equality that might be defined on the types involved (setoids). See also Section IV-A.

If we choose the empty type (i.e. the type $False$) for X, then we obtain the set of closed terms (as in Section II). In that case, the left part of the diagram can be ignored, and the remaining square says precisely that $app : \Sigma\ (T\ False) \to T\ False$ is the initial algebra for functor Σ. Observe that the diagram at the end of Section II can then be obtained by taking Z_B for Y, and DM for h.

Finally, it is straightforward to provide a function mapping for T, to turn it into a functor:

$$\textbf{Instance} : SFmap\ T :=$$
$$\lambda\ X\ Y\ (f : X \to Y), fold\ (var\ Y \circ f)\ (app\ Y)$$

B. Distributive Laws

Before we treat the GSOS, we will first consider simple distributive laws, ones that distribute a functor over another functor. Distributive laws (in the simple format) are functions $\Lambda : \Sigma \circ B \Rightarrow B \circ \Sigma$ (i.e. of type $\forall\, X, \Sigma\ (B\ X) \to B\ (\Sigma\ X)$) that happen to be natural transformations (see Section IV-A).

We will replace the operational as well as the denotational model introduced in the previous section with models that are derived from the same distributive law Λ, which corresponds to the operational rules.

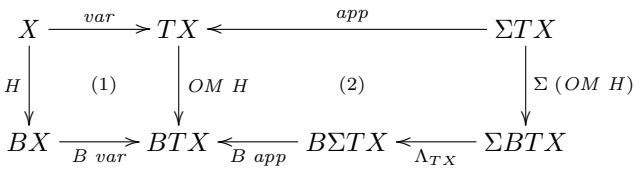
$$\begin{aligned}&\textbf{Definition}\ \Lambda : \Sigma \circ B \Rightarrow B \circ \Sigma :=\\ &\quad \lambda\ X\ \sigma,\\ &\qquad \textbf{match}\ \sigma\ \textbf{with}\\ &\qquad\ |\ AS \Rightarrow (a, AS)\\ &\qquad\ |\ BS \Rightarrow (b, BS)\\ &\qquad\ |\ Alt\ x\ y \Rightarrow\\ &\qquad\quad \textbf{let}\ (l, x') := x\ \ \textbf{in}\\ &\qquad\quad \textbf{let}\ (m, y') := y\ \textbf{in}\ (l, Alt\ y'\ x')\\ &\qquad \textbf{end}\end{aligned}$$

As a function, Λ takes an operation (an element from the signature) as argument. In the case of Alt this operation is applied to two arguments that both consist of a pairing of an action and a variable. This corresponds to the premise of a rule. The result is a pairing of an action and an operation applied to variables, corresponding to the target of the conclusion of each rule. The polymorphism in X ensures that Λ does not depend on a concrete choice of the set of variables. In summary, the type of Λ says that each operation in the language, as it is applied to behaviors on the variables, yields a behavior on an operation applied to variables.

C. Operational and Denotational Models

In the standard relational approach to operational semantics, the validity of a transition step is proved by the construction of a derivation tree. The nodes correspond to applications of the operational rules, and the leafs correspond applications of the hypotheses.

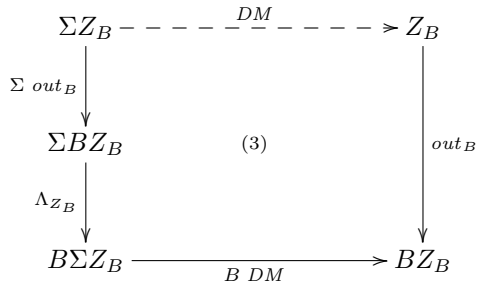
We can mimic this with the help of the definition principle for terms (the $fold$ operation) combined with the semantic model. Suppose that we have a map $H : X \to B\ X$, representing the *behavior environment*: the hypotheses about the variables in the premises. If we encounter an application of an operation, then we apply Λ, and if we encounter a variable we apply H. In a diagram,



which concretely is

$$\begin{aligned}&\textbf{Definition}\ OM\ `(H : X \to B\ X) : T\ X \to B\ (T\ X) :=\\ &\quad fold\ (sfmap_B\ (var\ X) \circ H)\\ &\qquad (sfmap_B\ (app\ X) \circ \Lambda\ (T\ X))\end{aligned}$$

The denotational model can be obtained in a dual fashion, by unfolding the semantic model. Assume the existence of a final coalgebra for the behavior B with state-space Z_B and structure map out_B.



Concretely,

$$\begin{aligned}&\textbf{Definition}\ DM : \Sigma\ Z_B \to Z_B :=\\ &\quad unfold_B\ (\Lambda\ Z_B \circ sfmap_\Sigma\ out_B)\end{aligned}$$

The denotational model operates directly on elements of the semantic domain. It tells how the operations of the language, applied to denotations, form new denotations. We remark that the hypotheses do not play a role in the denotational model, but will come into play when we construct the evaluation function $eval$. Running a term according to the operational model, and evaluating a term according to the denotational model is defined in same way as in Section II:

$$\begin{aligned}&\textbf{Definition}\ run\ (H : X \to B\ X) : T\ X \to Z_B :=\\ &\quad unfold_B\ (OM\ H)\\ &\textbf{Definition}\ eval\ (H : X \to B\ X) : T\ X \to Z_B :=\\ &\quad fold\ (unfold_B\ H)\ DM\end{aligned}$$

The distributivity property of Λ will be needed to prove the adequacy of run and $eval$.

There are many sensible rules that do not fit in the format of Λ of this section. For example, consider the operation Zip of our simple stream language, which zips two input streams together (see Figure 1). This operation differs slightly from Alt in the sense that at each transition it does not discard the head element of the second stream. If we try to encode this rule in our semantic model, for instance by adding the following alternative to Λ:

$\mid \mathit{Zip}\ x\ y \Rightarrow \mathbf{let}\ (l, x') := x\ \mathbf{in}\ (l, \mathit{Zip}\ y\ x')$

then the model does not type check anymore. The main problem is that variables used on both the left- and right-hand sides should receive a polymorphic type, which is not the case for the variable y. Also replacing y by a pattern match, as in the case for Alt will not work because then we have to reconstruct the first argument of *Zip* on the right-hand side out of the constituents. In Section V we discuss the more liberal *GSOS* rule format admitting operations like Zip.

IV. Coq Formalization

In this section we discuss some details of the Coq formalization, and continue the development of the theory for terms.

A. Equational Reasoning with Setoids

Infinite objects, as in most theorem provers, live in a world separate from finite objects, and do not adhere to Coq's standard notion of equality. One often works instead with *bisimulation*, a weaker notion of equality on infinite objects. Coq does not support user-defined extensions of its standard notion of equality (i.e. quotient types) as it would endanger the decidability of type checking. To overcome this issue, it is common practice to work with *setoids*, **Type**s packaged with a user-defined notion of equality and a proof of well-behavedness of the equality. The commuting diagrams in Section II use bisimulation as the underlying notion of equality in the Coq formalization. Finally, *setoid morphisms* are functions whose domain and codomain are setoids and respect those equalities.

The recent addition of type-classes to Coq [14] enables the use of canonical names for standard mathematical notation. These type-classes are first-class as they are powered by proof search and implicit arguments. Declaring instances of the $Equiv$, $Setoid$ and $Setoid\_Morphism$ type-classes enables fluent rewriting modulo setoid equality in proofs. In our development we have tacitly overloaded the canonical name "=" with setoid equality.

First, we introduce setoid counterparts for the standard categorical notions of functor and natural transformation. The *setoid functor* is taken from the MathClasses library [15]. It consists of an object map M and two classes: a class $SFmap$, which is the function map, and a class $SFunctor$ carrying proofs of the setoid functor axioms.[3]

$\mathbf{Class}\ \mathit{SFunctor}\ (M : \mathbf{Type} \to \mathbf{Type})$
$\quad `\{\forall\ `\{\mathit{Equiv}\ X\}, \mathit{Equiv}\ (M\ X)\}\ `\{\mathit{SFmap}\ M\} := \{...\}$

For reasons of space, we omit the full definition of $SFunctor$. The second argument lifts a notion of equality on X to a notion of equality on $M\ X$. Furthermore, it carries two sanity properties stating that the object map makes a setoid on X into a setoid on $M\ X$, and that the function map is a setoid morphism in its function argument (allowing us to rewrite equivalent functions with one another), and the following two familiar properties about the function map:

$$\mathit{sfmap}_M\ \mathit{id} = \mathit{id}$$
$$\mathit{sfmap}_M\ (f \circ g) = \mathit{sfmap}_M\ f \circ \mathit{sfmap}_M\ g$$

Given two object maps M and N, one can define a family of functions:

$\mathbf{Notation}\ M \Rightarrow N := \forall\ X, M\ X \to N\ X$

The family of functions $\eta : M \Rightarrow N$ is a *setoid natural transformation* if η_X is a setoid morphism whenever X is a setoid, and if it satisfies a commutation law:

$$\eta_Y \circ \mathit{sfmap}_M\ f = \mathit{sfmap}_N\ f \circ \eta_X \tag{1}$$

Again for reasons of brevity, we no not include the corresponding type-class definition $SNatural$.

B. Dependent Types for Generic Terms

Attempting to encode terms as the least fixpoint μ of a signature functor as is done in the Haskell code in Section II results in an error in Coq, as such definitions violate Coq's syntactic check for positivity, which guarantees termination of structurally recursive functions.

We bypass this issue by exploiting the fact that signatures of binding-free languages have a fairly simple structure. That is, a term on such a signature is essentially a rose tree, in which each parent node has an arbitrary number of child nodes, dictated by the arity of the operation corresponding to the parent node. A leaf of the tree is either a parent node with zero children nodes, or a variable, if we consider open terms.

The signature is nothing more than an assignment of an arity to each of the language's operations.

$\mathbf{Variable}\ \mathit{Operation} : \mathbf{Type}$
$\mathbf{Variable}\ \mathit{ar} : \mathit{Operation} \to \mathit{nat}$
$\mathbf{Definition}\ \Sigma\ X := \{\, x : \mathit{Operation}\ \&\ \mathit{vector}\ X\ (\mathit{ar}\ x)\}$

A parent node in the tree is described by a dependent pair, consisting of the operation x and a vector of length the arity of x ($vector$ is essentially a richly typed version of $list$). One can think of the notation "$\{\_\,\&\,\_\}$" as a type-theoretic variant of set comprehension. Dependent pairs can be crafted using the notation "$(\_\,\&\,\_)$", and $projT1$, $projT2$ are the corresponding projections.

The function map for the signature functor is:

$\mathbf{Instance} : \mathit{SFmap}\ \Sigma :=$
$\quad \lambda\ X\ Y\ (f : X \to Y)\ (\sigma : \Sigma\ X),$
$\quad\quad \mathbf{match}\ \sigma\ \mathbf{with}$
$\quad\quad \mid (s\ \&\ v) \Rightarrow (s\ \&\ \mathit{map}\ f\ v)$
$\quad\quad \mathbf{end}$

The induction principle for T pushes the use of dependent types even further. It is the basis for the proofs of the Lemmas in Section IV-C.

$\mathbf{Definition}\ T\_\mathit{induction}\ `(P : T\ X \to \mathbf{Type}) :$
$\quad (\forall\, x : X, P\ (\mathit{var}\ x)) \to$

[3] Unlike Haskell, Coq admits variable names starting with an uppercase letter. Furthermore, the backtick causes Coq to automatically generalize missing variables.

$$(\forall\, x : \Sigma\, \{\, t : T\ X\ \&\ P\ t\,\}, P\ (app\ (sfmap_\Sigma\ projT1\ x))) \\ \to \forall\, t, P\ t$$

$T\_induction$ is essentially a dependently-typed version of *fold*: we can re-obtain *fold* by setting $P := \lambda\ \_, Y$:

Definition $fold\ '(k : X \to Y)\ (h : \Sigma\ Y \to Y) : T\ X \to Y := T\_induction\ (\lambda\ \_, Y)\ k\ (h \circ (sfmap_\Sigma\ projT2))$

C. Theory about Terms

Now that we have the full definitions in place, we can continue our formalized treatment of the theory about terms.

Lemma 2. *Σ and T are setoid functors.*

The proof of Lemma 2 uses the full dependently-typed induction principle for T. The full principle has also been used to prove the properties in this section by induction on the structure of T.

In the remainder of this section we show that T is a monad in the categorical sense. To this end, we need to show that it has a unit, var in this case, and a multiplication, namely:

Definition $join\ X : T\ (T\ X) \to T\ X := fold\ id\ (app\ X)$

These satisfy the two standard coherence conditions of monads.

Lemma 3. *The terms form a monad, i.e. the following identities hold:*

$$join_X \circ sfmap_T\ join_X = join_X \circ join_{TX}$$
$$join_X \circ sfmap_T\ var_X = join_X \circ var_{TX} = id$$

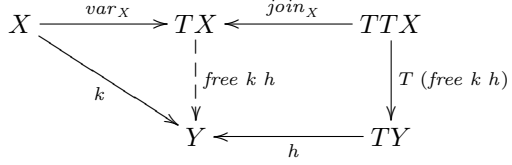
It is a well-known fact from category theory that the category of Σ-algebras is isomorphic to the category of *algebras for the term monad*. These are "plain" algebras h for the functor T, with two additional properties:

$$h \circ var_Y = id$$
$$h \circ sfmap_T\ h = h \circ join_Y$$

A T-algebra homomorphism is a homomorphism of the underlying algebra.

We conclude this section by providing an alternative proof principle for open terms, that differs from Lemma 1.

Lemma 4. *Let $k : X \to Y$ and $h : T\ Y \to Y$ such that h is an algebra for the term monad. Set free $k\ h := fold\ k\ (h \circ app_Y \circ sfmap_\Sigma\ (var_Y))$. Then free $k\ h$ is unique in making the following diagram commute:*



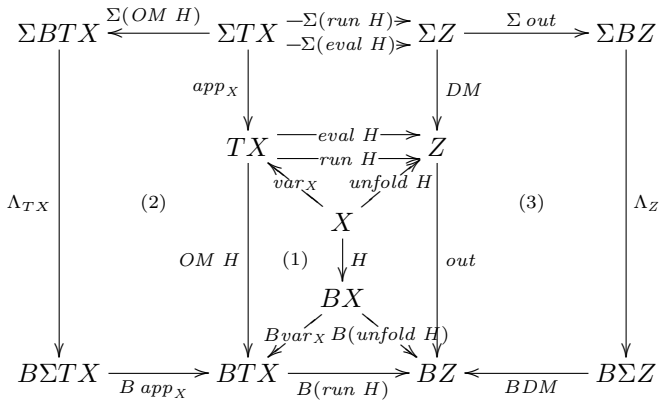
We have now set up a theory for syntax. Similarly, we could develop a theory for behavior. We do not pursue this goal for two reasons. First, since our presentation is essentially a deep embedding of SOS rules, most of it hinges on a structured encoding of the terms. Secondly, one may expect that more variation in the behavior is desired to model phenomena such as time or probability (see [2]; see also [10] for an overview). Moreover, finality proofs can be tricky to carry out in Coq due to guardedness restrictions that it puts on corecursive definitions.

V. Proving the Adequacy Theorem

Our approach to prove the adequacy theorem for the GSOS rules is to first carry it out for the simple rule format. The proof for GSOS follows the same proof skeleton, but requires some additional intermediate verifications.

A. Adequacy Theorem for Rules in Simple Format

Recall how the operational and denotational models are obtained from a single semantic model. We combine the diagrams of the operational and denotational models into the following diagram.

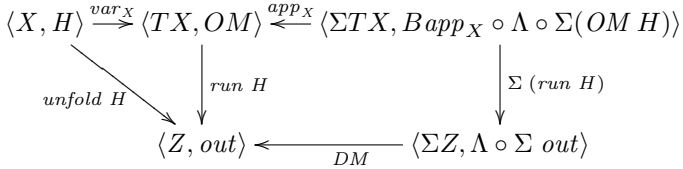


The following theorem holds for *open* terms, which is a mild generalization of what has been presented in the literature [17], [10], [2].

Theorem 5 (Adequacy)**.**

$$\forall\, H\ t, run\ H\ t = eval\ H\ t.$$

Proof: Consider the following diagram in the category of B-coalgebras. That is, the objects are pairs (consisting of the object and the structure map) and the arrows are coalgebra homomorphisms.



Except for the arrow $\Sigma\ (run\ H)$, it is trivial to see that each of the arrows are in fact coalgebra homomorphisms, as it can be directly read off the complete diagram above. To show this for $\Sigma\ (run\ H)$, one uses naturality of Λ and the fact that run is a coalgebra homomorphism. Commutativity of the diagram follows from the finality of out. The theorem then follows from applying the forgetful functor to the above diagram and the definition principle used to define $eval\ H$. ■

$$\frac{}{!a \xrightarrow{\mathsf{a}} \mathit{Done}} \qquad \frac{x \not\rightarrow \qquad y \xrightarrow{l} y'}{x \bullet y \xrightarrow{l} y'} \qquad \frac{x \xrightarrow{l} x'}{x \bullet y \xrightarrow{l} x' \bullet y}$$

$$\frac{x \xrightarrow{l} x'}{x \sqcup y \xrightarrow{l} x'} \qquad \frac{y \xrightarrow{l} y'}{x \sqcup y \xrightarrow{l} y'} \qquad \frac{}{\mathit{Done} \not\rightarrow}$$

Fig. 2. Basic process algebra.

B. The GSOS Format

Consider the process algebra language of Figure 2. The parallel composition operation "$\sqcup$" and the possibility of a state not having any outgoing transitions clearly make this a non-deterministic language. We use the *fin* type family to define the behavior functor; *fin* n consists of the first n natural numbers.

Definition $B\ X := \{\, n : \mathit{nat}\ \&\ \mathit{fin}\ n \to A \times X \,\}$
CoInductive $\mathit{cotree} := \mathit{node} : B\ \mathit{cotree} \to \mathit{cotree}$

The object of the corresponding final coalgebra consists of the cotrees.

Lemma 6. *cotree is the object of the final coalgebra for the functor B.*

As an aside, it may seem more natural to set $B\ X := \mathit{list}\ (A \times X)$, but one runs into problems trying to provide a guarded corecursive definition of *unfold* for that choice of B.

Note that the sequencing operation "$\bullet$" does not fit in the simple rule format for the same reasons as *Zip* in Section III-C. Moreover, the second rule in Figure 2 has a transition to a variable in its conclusion, which is neither supported in the simple format.

The operational rules of basic process algebra can however be encoded in the GSOS format. Let B be arbitrary and Σ be constrained to the conditions set out in Section IV-B, and set $D\ X := X \times B\ X$. The *abstract GSOS format* are natural transformations of the following type:[4]

$$\rho : \Sigma \circ D \Rightarrow B \circ T$$

The difference with the simple format is that the arguments of each operation are now pairings consisting of both the variable and the behavior on that variable, and each rule yields a term.

For reference we include the encoding of the rules of Figure 2 in abstract GSOS:

```
Definition ρ : Σ ∘ D ⇒ B ∘ T :=
  λ X σ,
    match σ with
    | ! a      ⇒ (1 & (λ _, (a, app Done)))
    | x ⊔ y ⇒ (_ & merge ([id, var] ∘ projT2 (π₂ x))
                        ([id, var] ∘ projT2 (π₂ y)))
    | (_, (0 & _)) • (_, b)
            ⇒ sfmap_B var b
    | (_, b) • (y, _)
            ⇒ sfmap_B (λ x', app (var x' • var y)) b
    | Done  ⇒ (0 & case0 _)
    end
```

Here *merge* and *case0* have the following types:

```
Definition merge '(f : fin n → X) '(g : fin m → X) :
  fin (n + m) → X
Definition case0 (f : fin 0 → Type) (i : fin 0), f i
```

C. From GSOS to Distributive Laws

Recall that the symmetry of the (co)domains was vital to the proof of the adequacy theorem. The rules ρ need to undergo a two-step transformation to obtain a distributive law of T over D. First expand ρ's codomain:

```
Definition τ : Σ ∘ D ⇒ D ∘ T :=
  λ X, ⟨ app X ∘ sfmap_Σ (var X ∘ π₁), ρ ⟩.
```

To obtain Λ we apply *fold*.

```
Definition Λ : T ∘ D ⇒ D ∘ T :=
  λ X, fold (sfmap_D (var X))
            (sfmap_D (join X) ∘ τ (T X))
```

A general proof (included in the Coq development) shows that the definition principle for terms yields natural transformations. From this fact, together with the assumption that ρ is a natural transformation, it is straightforward to show that Λ is natural as well. The obtained distributive law, which distributes the term monad over the copointed functor D, enjoys more structure than the plain distributive laws, as we will prove in Proposition 7.

Proposition 7. *The following two identities hold:*

$$\Lambda_X \circ \mathit{var}_{DX} = \mathit{sfmap}_D\ \mathit{var}_X$$
$$\Lambda_X \circ \mathit{join}_{DX} = \mathit{sfmap}_D\ \mathit{join}_X \circ \Lambda_{TX} \circ \mathit{sfmap}_T\ \Lambda_X$$

The first identity says that the law should behave trivially on variables. The second identity characterizes compositionality of the semantics. Proposition 7 is a key ingredient in the proof of Lemma 8 and Lemma 9. Lenisa, Power and Watanabe [11] prove that the GSOS format corresponds precisely to a distributive law of a monad over a copointed functor, but for the adequacy theorem it is sufficient to show that a GSOS rule implies such a law.

D. Adequacy Theorem for the GSOS Format

The proof of the adequacy theorem for GSOS is analogous to the situation in Theorem 5, but B should be replaced by D, Σ by T, and *app* by *join* and entails more proof obligations due to the richer structure on Λ. We adopt the definitions of the operational/denotational models and semantics to this new situation. Recall that we used the definition principle of terms to obtain an operational model from the plain distributive laws. We repeat this construction for distributive laws that stem from GSOS rules, with the difference that we use the alternative definition principle.

[4] Literature on process algebra often considers just the case where B is the finite power set and calls that the GSOS format.

Def. OM_{GSOS} '$(H : X \to D\ X) : T\ X \to D\ (T\ X) :=$
$\quad free\ (sfmap_D\ (var\ X) \circ H)$
$\qquad (sfmap_D\ (join\ X) \circ \Lambda\ (T\ X))$

As in the proof, we need to verify that each of the relevant arrows are coalgebra homomorphisms, but for the functor D instead of B. For the arrow corresponding to $join$ this follows from the following fact:

Lemma 8. *$sfmap_D\ join_X \circ \Lambda_{TX}$, used in OM_{GSOS}, is an algebra for the term monad.*

The D-coalgebras are isomorphic to the B-coalgebras, and it is straightforward to verify that if out is a final B-coalgebra, then $\langle id, out \rangle$ is a final D-coalgebra (with the same state-space). Hence, the denotational model for GSOS rules and run_{GSOS} can be obtained by finality, analogous to Section III-B. We obtain $eval_{GSOS}$ by making use of the alternative definition principle for terms.

Def. $run_{GSOS}\ (H : X \to D\ X) : T\ X \to D\ (T\ X) :=$
$\quad unfold_D\ (OM_{GSOS}\ H)$
Definition $DM_{GSOS} : T\ Z_D \to Z_D :=$
$\quad unfold_D\ (\Lambda\ Z_D \circ sfmap_T\ out_D)$
Def. $eval_{GSOS}\ (H : X \to D\ X) : T\ X \to D\ (T\ X) :=$
$\quad free\ (unfold_D\ H)\ DM_{GSOS}$

Recalling Lemma 4, to ensure the uniqueness of $eval_{GSOS}$, we need to verify the following fact:

Lemma 9. *DM_{GSOS} is an algebra for the term monad.*

We can now conclude the following:

Theorem 10 (Adequacy for GSOS rules)**.**

$$\forall\ H\ t, run_{GSOS}\ H\ t = eval_{GSOS}\ H\ t.$$

VI. Related Work

The work in this paper is part of a line of research called *bialgebraic semantics*, initiated by the work of Turi and Plotkin [17]. Bialgebras appear in the present paper in the form of diagrams (2) and (3). Hinze and James [6] give a pen and paper proof of the adequacy theorem based on Haskell definitions for several rule formats, using proof techniques similar to ours. The most powerful rules distribute a monad over a comonad and also appear in [17], [2]. Although these laws provide the most abstract perspective of well-behaved rules, they have not yet been applied in concrete studies of rule formats [10].

An implementation of Turi and Plotkin's work has been developed by Hutton [7] in Haskell and extended for modularity by Jaskelioff, Ghani and Hutton [9]. Both papers define the terms and and the final coalgebra as the greatest fixpoint of a functor. Direct translations to Coq are not possible; in this paper we have presented an alternative approach based on dependent types.

Niqui [13] extends the class of productive specifications definable in Coq by developing the λ-coiteration scheme in Coq, based on Bartels' work [2]. In the further work section of his paper he mentions that adding monadic, pointed or cofree structure on the bialgebraic nature of λ-coiteration can help to build even more powerful schemes.

Aceto et al. [1] have developed a tool, called the PREG Axiomatizer, to prove the bisimilarity of two ground terms written in a language specified in GSOS extended with predicates. It derives a sound set of axioms from the GSOS rules, and uses that to prove the bisimulation.

VII. Conclusions

We have shown how operational and denotational semantics can be obtained from operational rules in the GSOS format in the theorem prover Coq. Moreover, we have formally proved the theorem that says that these forms of semantics are consistent. Our formalization facilitates both formal reasoning about and the execution of programming language semantics.

Directions of further work would be to add support for variable binding, which requires the use of a different base category [5] (the present formalization is based on **Type**), and further generalization to support different rule formats, as in [2], [6].

References

[1] L. Aceto, G. Caltais, E.-I. Goriac, and A. Ingolfsdottir. PREG Axiomatizer – A ground bisimilarity checker for GSOS with predicates. In *Algebra and Coalgebra in Computer Science*, pages 378–385. Springer, 2011.

[2] F. Bartels. *On generalised coinduction and probabilistic specification formats*. PhD thesis, CWI, Amsterdam, April 2004.

[3] Y. Bertot, G. Huet, J.-J. Lévy, and G. Plotkin, editors. *Theorem proving support in programming language semantics*, chapter 15, pages 337–361. Cambridge University Press, 2009.

[4] B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced. *J. ACM*, 42(1):232–268, 1995.

[5] M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proc. 14th LICS Conf.*, pages 193–202. IEEE, 1999.

[6] R. Hinze and D. W. James. Proving the unique fixed-point principle correct: an adventure with category theory. In *Proc. 16th ACM SIGPLAN ICFP*, pages 359–371. ACM, 2011.

[7] G. Hutton. Fold and unfold for program semantics. In *Proc. 3rd ACM SIGPLAN ICFP*, pages 280–288. ACM, 1998.

[8] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:62–222, 1997.

[9] M. Jaskelioff, N. Ghani, and G. Hutton. Modularity and implementation of mathematical operational semantics. *Electr. Notes Theor. Comput. Sci.*, 229(5):75–95, 2011.

[10] B. Klin. Bialgebras for structural operational semantics: An introduction. *Theor. Comput. Sci.*, 412(38):5043–5069, 2011.

[11] M. Lenisa, J. Power, and H. Watanabe. Category theory for operational semantics. *Theor. Comput. Sci.*, 327(1-2):135–154, 2004.

[12] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 124–144. Springer-Verlag New York, Inc., 1991.

[13] M. Niqui. Coalgebraic reasoning in Coq: Bisimulation and the λ-Coiteration scheme. In *TYPES*, volume 5497 of *LNCS*, pages 272–288. Springer, 2009.

[14] M. Sozeau and N. Oury. First-class type classes. In *TPHOLs*, volume 5170 of *LNCS*, pages 278–293. Springer, 2008.

[15] B. Spitters and E. van der Weegen. Type classes for mathematics in type theory. *MSCS*, 21:1–31, 2011.

[16] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.3*, 2010. http://coq.inria.fr.

[17] D. Turi and G. Plotkin. Towards a mathematical operational semantics. In *Proc. 12th LICS Conf.*, pages 280–291. IEEE, 1997.