Formalizing Bialgebraic Semantics in PVS 6.0

Sjaak Smetsers[1], Ken Madlener[1], and Marko van Eekelen[1,2]

1 **Institute for Computing and Information Sciences, Radboud University**
Postbus 9010, 6500 GL Nijmegen, The Netherlands
{S.Smetsers,K.Madlener}@cs.ru.nl

2 **School of Computer Science, Open University of the Netherlands**
Postbus 2960, 6401 DL Heerlen, The Netherlands
{M.vanEekelen}@cs.ru.nl

Abstract

Both operational and denotational semantics are prominent approaches for reasoning about properties of programs and programming languages. In the categorical framework developed by Turi and Plotkin both styles of semantics are unified using a single, syntax independent format, known as GSOS, in which the operational rules of a language are specified. From this format, the operational and denotational semantics are derived. The approach of Turi and Plotkin is based on the categorical notion of bialgebras. In this paper we specify this work in the theorem prover *PVS*, and prove the adequacy theorem of this formalization. One of our goals is to investigate whether *PVS* is adequately suited for formalizing metatheory. Indeed, our experiments show that the original categorical framework can be formalized conveniently. Additionally, we present a GSOS specification for the simple imperative programming language *While*, and execute the derived semantics for a small example program.

1998 ACM Subject Classification F.4.1 Mathematical Logic, F.3.2 Semantics of Programming Languages

Keywords and phrases operational semantics, denotational semantics, bialgebras, distributive laws, adequacy, theorem proving, *PVS*, *While*.

Digital Object Identifier 10.4230/OASIcs.WPTE.2015.47

1 Introduction

The formal definition of a real-world programming language is often a monumental undertaking. The process of verifying metatheory often exceeds human capabilities; due to its inherent complexity, mechanization time, even for the interesting core facets of the semantics of real-world programming languages, is prohibitive. The best alternative for complete verification is to employ well-established methods, such as type systems or the use of mechanized verification tools. These verification tools are usually based on typed higher-order logic. The specification languages of these tools often provide automatic code generation, which enables the execution of specifications. This feature can be used as an additional check of the developed concepts, before one starts with formally proving properties of these concepts.

In this paper, we present a formalization of both popular styles of semantic specifications: (structural) operational semantics and denotational semantics. Our main goal is to experiment with *PVS*'s latest feature, so-called *declaration parameters*, which enable the specification of polymorphic functions and data structures. The experiments are carried out with *PVS* version 6.0, released in February 2013. Previous versions of *PVS* already offered a limited form of polymorphism by means of theory level parameters. However, there are situ-

2nd International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE'15).
Editors: Yuki Chiba, Santiago Escobar, Naoki Nishida, David Sabel, and Manfred Schmidt-Schauß; pp. 47–61

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

ations where these theory parameters are inconvenient, particularly if notions are involved with a generic nature, such as the categorical concepts used in our formalization.

Our approach is based on a framework developed by Turi and Plotkin [18] unifying both styles of semantics. By exploiting the language of category theory, they managed to disassociate from language-specific details such as concrete syntax and behavior. Given a set of operational rules, they derived both operational and denotational semantics using the concept of *bialgebras* equipped with a *distributive law*. The format in which these operational rules are specified is known as the *GSOS (Grand Structural Operational Semantics) format*, introduced by [3]. This format originates from the theory of SOS, and has been given a categorical interpretation by means of bialgebras. Turi and Plotkin prove that the operational and denotational semantics generated from any set of GSOS rules are consistent for every input program. Our work is also inspired by previous comparable experiments with the *Coq* proof assistant; see [14].

The contribution of our work is threefold. First, we investigate whether *PVS* 6.0 is sufficiently expressive for formalizing metatheory. Until now, such *PVS* developments have been constrained to only a few metatheoretical experiments. This seems to be a missed opportunity, because *PVS* has shown to be very successful in proving properties of computer programs. In our attempt, we extend to a very high abstraction level, namely we use Turi and Plotkin's categorical description [18] as the point of inception. Our goal is to formally prove that their construction is sound, which is expressed in the *adequacy theorem*. The second, but not less important contribution is a *PVS* formalization providing a framework for facilitating both formal reasoning about, and experimentation with semantics of programming languages[1]. The setup is such that the user is free to choose between either denotational or operational semantics at any point. And third, we illustrate the expressiveness of our framework with an elaborate example of a standard imperative language specified in the GSOS format. Until now, concrete applications of GSOS are mostly restricted to process algebras. The reader is expected to have a modest knowledge of Category Theory. If not, we recommend [1] as an easy-going starting point.

2 Background

A brief introduction of the basic prerequisite concepts for this paper follows. Let us start with an example following the approach as, for instance, taken by [9] and [11]. We explain and illustrate the technicalities using a very simple language of streams (see also [12]). The operational rules for this stream language are given in Figure 1. These rules inductively define a transition relation that is a subset of $\mathbb{T} \times L \times \mathbb{T}$, where $\mathbb{T}$, L denote the sets of closed terms and output labels, respectively. The two basic operations AS and BS generate constant (infinite) streams of As and Bs whereas the operation Alt yields an alternation of two streams. The latter operates by repeatedly taking the head of its first argument, and calling itself recursively on the swapped tails, discarding the head of the second argument. The first step towards the formalization of such a language is to express both the *signature* and *behaviour* (i.e., the result/effect of an operation) of the operations as *functors*. In languages with support for higher-order polymorphism, like the functional language *Haskell*, one would express a functor as a type constructor class that is parameterized with the type (the object map) of the functor. The type class itself contains the corresponding (morphism)

[1] All definitions and theorems given in this paper have been formalized and proven. The files of the development can be obtained via `http://www.cs.ru.nl/~sjakie/papers/adequacy/`.

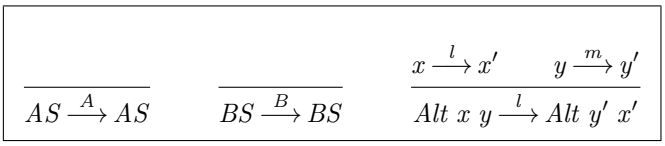


Figure 1 A simple language for streams.

map. However, *PVS* only offers first-order type variables which forces us to specify functors in a slightly more ad hoc way.

Syntax and Σ-algebras

We start with representing the syntax of a language as (open) terms, by introducing the following datatype:

```
TER [V : type, F : type, ar : [F → nat]] : datatype begin
  tvar (var_id : V)                          : tvar ?
  tapp (op : F, args : [below (ar (op)) → TER]) : tapp ?
end TER
```

The datatype is parametric in the type V of variables and the signature which is represented by the set F of operator symbols and the arity map ar. It has two *constructors* (*tvar* and *tapp*), and two *recognizers* (*tvar*? and *tapp*?)[2]. These recognizers are used as predicates to test whether or not a term starts with the respective constructor. The field names *var_id*, *op* and *args* are used as *accessors* to extract these components from a term. It is more convenient, however, to define operations on inductive datatypes by pattern matching. This datatype definition also illustrates *PVS*'s special typing facility: *dependent types*, i.e., types depending on values. For example, *below* (n) denotes the set of natural numbers between 0 and n.

The signature (F and ar) is encoded as a functor which we will call the *signature functor*. The following *PVS* theory[3] defines the notion of Σ-functor (consisting of a type Σ, and a map function map_Σ), and Σ-algebra (categorically, an algebra is defined as a pair consisting of an *object* X, called the *carrier* of the algebra, and a structure map $\Sigma[X] \to X$)[4]. In *PVS* syntax, a function type $T_1 \to T_2$ has the form $[T_1 \to T_2]$, and tuple types have the form $[T_1, ..., T_n]$. Associated with every n-tuple type is a set of projection functions: '1, '2, ..., i.e., by t'i one selects the i^{th} component from t.

```
ΣAlgebra [F : type, ar : [F → nat]] : theory begin
  Σ [a : type] : type      = [f : F, [below (ar (f)) → a]]
  map_Σ [a, b : type] (f : [a → b]) (sf : Σ [a]) : Σ [b] =
    (sf'1, λ (i : below (ar (sf'1))) : f (sf'2 (i)))
  Alg_Σ [a : type] : type = [Σ [a] → a]
end ΣAlgebra
```

[2] *PVS* allows question marks as constituents of identifiers.
[3] *PVS* uses parameterized theories to organize specifications, i.e, datatypes, function definitions, and properties.
[4] Here, we tacitly omit the fact that the structure map is usually equipped with two so-called *functor laws*.

In this theory we have used the new feature of *PVS* 6.0: *declaration (level) parameters*, i.e. (type) variables ranging over first-order types. These are comparable to type variables in, for instance, *Haskell* and *Coq*. Combined with the dependent typing facility, these declaration parameters enable us, for example, to specify the type part of the functor $\Sigma\,[a]$ as a dependent pair being parametric in the object type a.

To illustrate how to use these notions in combination with the simple stream language, we introduce the following auxiliary theory:

$StreamL$: **theory begin**
  $SigS$: **type** $= \{AS, BS, Alt\}$
  $arS\ (s : sigS) : nat =$ **cases** (s) **of** $AS : 0, BS : 0, Alt : 2$ **endcases**
end $StreamL$

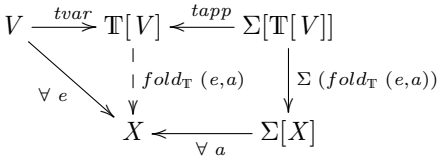
The enumeration type $SigS$ represents the set of operator symbols, whereas the function arS assigns arities to each of these symbols. We can now use $SigS$ and arS as actual parameters for $\Sigma Algebra$ theory in order to obtain the concrete signature functor.

The datatype TER introduced earlier is also a functor. To make this more explicit, we introduce $\mathbb{T}\,[v]$ as an abbreviation for $TER\,[v, F, ar]$. When *PVS* typechecks TER then it automatically generates for TER the corresponding map operation and a folding operation called *reduce*. To adhere to standard terminology and to the notation used in the present paper, we rename these generated operations to $map_{\mathbb{T}}$ and $fold_{\mathbb{T}}$, respectively. Incidentally, the latter is a standard operation for processing terms that avoids explicit recursion, see also [16] where this operation is called a *catamorphism*.

$Terms\,[F : \textbf{type}, ar : [F \to nat]]$: **theory begin importing** $TER, \Sigma Algebra\,[F, ar]$
  $\mathbb{T}\,[v : \textbf{type}] : \textbf{type} = TER\,[v, F, ar]$
  $fold_{\mathbb{T}}\,[v, x : \textbf{type}]\ (e : [v \to x], a : Alg_{\Sigma}\,[x])\ (t : \mathbb{T}\,[v]) : x = reduce\ (e, a)\ (t)$
  $map_{\mathbb{T}}\,[a, b : \textbf{type}]\ (f : [a \to b])\ (t : \mathbb{T}\,[a]) : \mathbb{T}\,[b] = map\ (f)\ (t)$
end $Terms$

The following uniqueness property is based on the categorical fact that *tapp* (which is an algebra for the functor Σ) is *initial*.

▶ **Proposition 1.** *Let* $e : V \to X$ *and* $a : Alg_{\Sigma}\,[X]$. *Then* $fold_{\mathbb{T}}\ (e, a)$ *is unique in making the following diagram commute*[5] *:*



This propery appears to be very useful as an alternative for structural induction in proofs of properties on terms. In fact, it allows for a direct translation of diagrammatic proofs into a *PVS* formalization. In our experience, these (hand-drawn) diagrammatic proofs are indispensable as the initial and most important step towards a fully formalized proof. The 'textual' *PVS* version of this proposition is:

[5] For the diagrams in this paper we adopted the categorical notation for functors by writing $\mathcal{F}$ instead of $map_{\mathcal{F}}$, for some functor $\mathcal{F}$.

$fold\_unique\ [v, x : \mathbf{type}] : \mathbf{proposition}$
$\forall\ (f : [\mathbb{T}\ [v] \to x], e : [v \to x], a : Alg_\Sigma\ [x]) :$
$f \circ tvar = e \wedge f \circ tapp = a \circ map_\Sigma\ (f) \Leftrightarrow f = fold_\mathbb{T}\ (e, a)$

Next, we show that $\mathbb{T}$ is a monad in the categorical sense, by defining the corresponding operations (so-called *natural transformations*) *unit* (embedding) and *join* (composition):

$\mathbb{T}Monad\ [F : \mathbf{type}, ar : [F \to nat]] : \mathbf{theory\ begin\ importing}\ Terms\ [F, ar]$
$unit_\mathbb{T}\ [v : \mathbf{type}]\ (vid : v) : \mathbb{T}\ [v] = tvar\ (vid)$
$join_\mathbb{T}\ [v : \mathbf{type}]\ (tt : \mathbb{T}\ [\mathbb{T}\ [v]]) : \mathbb{T}\ [v] = fold_\mathbb{T}\ (id, tapp)\ (tt)$
end $\mathbb{T}Monad$

Likewise, from category theory we borrow the notion of $\mathbb{T}$-algebra: A $\mathbb{T}$-algebra (or, more verbosely, an algebra for the $\mathbb{T}$ monad) is a 'plain' algebra a with two additional properties:

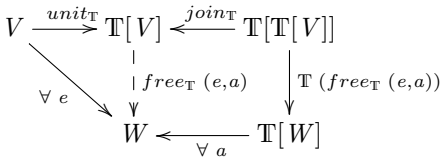
$$a \circ unit_\mathbb{T} = id \quad (1) \qquad a \circ map_\mathbb{T}\ (a) = a \circ join_\mathbb{T} \quad (2)$$

Below, these properties are encoded by the predicate $\mathbb{T}Alg?$. Additionally, we introduce a slightly modified version of $fold_\mathbb{T}$, named $free_\mathbb{T}$, taking a $\mathbb{T}$-algebra as argument instead of a Σ-algebra. In *PVS*:

$\mathbb{T}Algebra\ [F : \mathbf{type}, ar : [F \to nat]] : \mathbf{theory\ begin\ importing}\ TMonad\ [F, ar]$
$Alg_\mathbb{T}\ [a : \mathbf{type}] : \mathbf{type} = [\mathbb{T}\ [a] \to a]$
$free_\mathbb{T}\ [v, w : \mathbf{type}]\ (e : [v \to w], a : Alg_\mathbb{T}\ [w]) : [\mathbb{T}\ [v] \to w] = fold_\mathbb{T}\ (e, a \circ tapp \circ map_\Sigma\ (tvar))$
$\mathbb{T}Alg?\ [w : \mathbf{type}]\ (a : Alg_\mathbb{T}\ [w]) : bool = a \circ unit_\mathbb{T} = id \wedge a \circ map_\mathbb{T}\ (a) = a \circ join_\mathbb{T}$
end $\mathbb{T}Algebra$

We end this section with a proposition supplying $free_\mathbb{T}$ with a proof principle.

▶ **Proposition 2.** *Let* $e : V \to W$ *and* $a : Alg_\mathbb{T}\ [W]$ *such that* $\mathbb{T}Alg?\ (a)$ *holds. Then,* $free_\mathbb{T}\ (e, a)$ *is unique in making the following diagram commute:*



Behaviour and $\mathcal{B}$-coalgebras

The operational semantics of a language is given by a transition relation representing the execution steps of an abstract machine. These transition relations can be modelled in a categorial manner using coalgebras (e.g., see [10]). A coalgebra is the dual of an algebra: for a functor $\mathcal{B}$, the coalgebra consists of a carrier C, and a structure map $C \to \mathcal{B}[C]$. We express a transition relation as a $\mathcal{B}$-coalgebra, with carrier $\mathbb{T}\ [V]$, more specifically, as a function with type $\mathbb{T}\ [V] \to \mathcal{B}[\mathbb{T}[V]]$. We call this coalgebraic formalization the *operational model*.

For the stream language, the functor $\mathcal{B}$ is defined by $\mathcal{B}\ X = (L, X)$, i.e., $\mathcal{B}$ just pairs a label from L with X. Specifying the operational rules as a coalgebra is straightforward; e.g., see [14]. In Section 4 we will give a more elaborate example.

By making the behaviour functor $\mathcal{B}$ parametric in X we anticipate the fact that the terms can be executed according to the operational rules of the language yielding an infinite stream

of labels. Categorically, this stream is constructed by taking the greatest fixpoint of $\mathcal{B}$. In order to express this in *PVS*, we use *PVS*'s capability to introduce co-inductive datatypes. However, we cannot do this as a general fixpoint construction that is parametric in the behaviour functor. Incidentally, also *Coq* forbids such a construction for exactly the same technical restrictions imposed by its underlying logic, namely, such a construction would admit instantiations that have no set-theoretic semantics. Instead, we define the output stream directly as a codatatype, named $\mathbf{N}_{\mathcal{B}}$, and extract the functor $\mathcal{B}$ from this definition. In fact, no definitions are required to obtain $\mathcal{B}$: it is automatically generated from the definition of the codatatype, together with the $unfold$ operation (named *coreduce*). This unfolding operation is also know as an *anamorphism*, see [16].

$\mathbf{N}_{\mathcal{B}}\ [L : \mathbf{type}] : \mathbf{codatatype\ begin}$
  $nb\_in\ (el : L, next : \mathbf{N}_{\mathcal{B}}) : nb\_in\ ?$
$\mathbf{end}\ \mathbf{N}_{\mathcal{B}}$

The *PVS* specification of a coalgebra is as follows:

$\mathcal{B}Coalgebra\ [L : \mathbf{type}] : \mathbf{theory\ begin\ importing}\ \mathbf{N}_{\mathcal{B}}\ [L]$
  $\mathcal{B}\ [x : \mathbf{type}] : \mathbf{type} \quad = NB\_struct\ [L, x]$
  $Coalg_{\mathcal{B}}\ [x : \mathbf{type}] : \mathbf{type} = [x \to \mathcal{B}\ [x]]$
  $out_{\mathcal{B}} : Coalg_{\mathcal{B}}\ [\mathbf{N}_{\mathcal{B}}] = \lambda\ (nb : \mathbf{N}_{\mathcal{B}}) : inj\_nb\_in\ (el\ (nb), next\ (nb))$
  $unfold_{\mathcal{B}}\ [x : \mathbf{type}]\ (c : Coalg_{\mathcal{B}}\ [x])\ (z : x) : \mathbf{N}_{\mathcal{B}} = coreduce\ (c)\ (z)$
$\mathbf{end}\ \mathcal{B}Coalgebra$

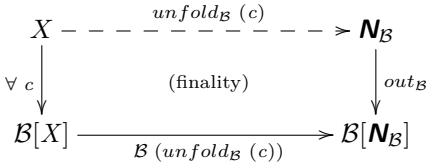
The functor $\mathcal{B}$ and operation $unfold_{\mathcal{B}}$ coincide with the generated datatype *NB_struct* and the (coinductive) function *coreduce*. As the components of *NB_struct* (e.g., see the definition of $out_{\mathcal{B}}$), are also used, we give the definition as is generated by *PVS*.

$NB\_struct\ [L, x : \mathbf{type}] : \mathbf{datatype\ begin}$
  $inj\_nb\_in\ (inj\_el : L, inj\_next : x) : inj\_nb\_in\ ?$
$\mathbf{end}\ NB\_struct$

As said before, the operational model, which represents the transition relation of the language, is specified as a coalgebra with type $Coalg_{\mathcal{B}}\ [\mathbb{T}(V)]$. The evaluation of a term (with type $\mathbb{T}\ [V]$)[6] is obtained by unfolding the operational model, using the term as 'initial seed'.

The function $out_{\mathcal{B}}$ is the (unique) final $\mathcal{B}$-colagebra on $\mathbf{N}_{\mathcal{B}}$. The following property is the dual of Propostion 1.

▶ **Proposition 3.** *Let* $c : Coalg_{\mathcal{B}}\ [X]$. *Then,* $unfold_{\mathcal{B}}\ (c)$ *is unique in making the following diagram commute:*



[6] Observe that the term might be open, i.e., it is possible to evaluate term containing variables.

The proof of this property is done by coinduction on $\mathbf{N}_{\mathcal{B}}$. The coinduction principle in *PVS* requires the construction of a proper *bisimulation relation*; e.g., see [8]. This principle is based on the fact that if two streams are bisimilar, then they are equal.

The "fusion law for anamorphisms" was introduced by [16]. We apply this law in a proof in Section 3.

$$
\begin{array}{l}
\mathit{unfold\_fusion}\ [a, b : \textbf{type}] : \textbf{lemma} \\
\quad \forall\ (f : [a \to b], c1 : \mathit{Coalg}_{\mathcal{B}}\ [a], c2 : \mathit{Coalg}_{\mathcal{B}}\ [b]) : \\
\qquad \mathit{map}_{\mathcal{B}}\ (f)\ \circ\ c1 = c2\ \circ f \Rightarrow \mathit{unfold}_{\mathcal{B}}\ (c1) = \mathit{unfold}_{\mathcal{B}}\ (c2)\ \circ f
\end{array}
$$

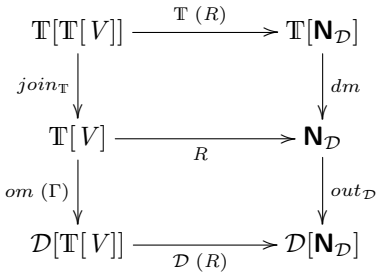
This lemma is proven by applying Proposition 3 twice: once to coalgebra *c1*, and the other to *c2*.

3 Bialgebraic semantics

In the previous section we already mentioned that the operational model (abbreviated as *om*) is specified as a coalgebra. The dual notion, the *denotational model* (*dm*), will be specified as an algebra. Before actually defining both *om* and *dm*, we introduce a general categorical concept, known as a *bialgebra*, which is used to link *om* and *dm* together.

Formally, let N, M be functors. Then, a bialgebra (for N, M) is a triple $\langle V, a, c\rangle$ such that a is a N-algebra and c is a M-coalgebra, sharing V as the common carrier. For two bialgebras, a *bialgebra homomorphism* is a mapping which is both a N-algebra homomorphism and a M-coalgebra homomorphism.

The concrete functors of our framework are $\mathbb{T}$ (for the syntax), and the so-called *free pointed functor* $\mathcal{D}$ *of* $\mathcal{B}$, i.e., $\mathcal{D}[X] = X \times \mathcal{B}[X]$ (for the behaviour). Furthermore, we consider two bialgebras with carriers $\mathbb{T}[V]$ and $\mathbf{N}_{\mathcal{D}}$ (i.e., greatest fixpoint of $\mathcal{D}$), namely, $\langle \mathbb{T}[V], \mathit{join}_{\mathbb{T}}, \mathit{om}\ (\Gamma)\rangle$ and $\langle \mathbf{N}_{\mathcal{D}}, \mathit{dm}, \mathit{out}_{\mathcal{D}}\rangle$. Here, $\Gamma : X \to \mathcal{D}[X]$ represents a *behaviour environment*. We explain later how *om* depends on Γ. The following diagram shows these two bialgebras together with a connecting homomorphism R.



Here, R is an (evaluation) function that maps each term to its execution result which is a (possibly infinite) value of type $\mathbf{N}_{\mathcal{D}}$. From this diagram we infer two different ways of defining R: (1) by considering the top square and using Proposition 2, or (2) by using the bottom square in combination with Proposition 3. This leads to the following two cases:

$$R_D\ (\Gamma) = \mathit{free}_{\mathbb{T}}\ (\mathit{unfold}_{\mathcal{D}}\ (\Gamma), \mathit{dm}) \qquad R_O\ (\Gamma) = \mathit{unfold}_{\mathcal{D}}\ (\mathit{om}\ (\Gamma))$$

From their construction it follows that both R_D (i.e., execution according to the denotational model) and R_O (i.e., execution according to the operational model) are unique. It remains to be shown that these functions are equal, which is called the *adequacy theorem*. Obviously, this depends on the way *om* and *dm* are defined. Now, the essence of the framework of

[18] is the following: instead of defining *om* and *dm* separately, the operational rules of the language are described by using a specific syntactic format from which both *om* and *dm* are obtained generically (i.e., syntax-independently). The interrelation between *om* and *dm* is given by a so-called *distributed law* $\Lambda\ [v : \mathbf{type}] : \mathbb{T}[\mathcal{D}[v]] \to \mathcal{D}[\mathbb{T}[v]]$ from which both models can be derived.[7] Formally, a distributive law (between a monad, here $\mathbb{T}$, and an endofunctor, here $\mathcal{D}$) is a *natural transformation* for which the following two identities hold (e.g., see [4]):

$$law\_distributive : \mathbf{lemma}$$
$$\Lambda \circ unit_{\mathbb{T}} = map_{\mathcal{D}}\ (unit_{\mathbb{T}})\ \wedge\ \Lambda \circ join_{\mathbb{T}} = map_{\mathcal{D}}\ (join_{\mathbb{T}}) \circ \Lambda \circ map_{\mathbb{T}}\ (\Lambda)$$

To be a natural transformation, Λ should satisfy:

$$law\_natural : \mathbf{lemma}\ \Lambda\ \circ\ map_{\mathbb{T}}\ (map_{\mathcal{D}}\ (f)) = map_{\mathcal{D}}\ (map_{\mathbb{T}}\ (f))\ \circ\ \Lambda$$

In polymorphic functional languages, the latter property is an example of a so-called *theorem for free*; see [19]. In essence, this free theorem formalizes the fact that Λ is genuinely polymorphic.

As for *dm*, we observe that the codomain of this operation is $\mathbf{N}_{\mathcal{D}}$, calling for a definition based on $unfold_{\mathcal{D}}$. This leads to:

$$dm : Alg_{\mathbb{T}}\ [\mathbf{N}_{\mathcal{D}}] = unfold_{\mathcal{D}}\ (\Lambda\ \circ\ map_{\mathbb{T}}\ (out_{\mathcal{D}}))$$

Dually, we define *om* as a $fold_{\mathbb{T}}$ (actually, we use the special variant $free_{\mathbb{T}}$ of $fold_{\mathbb{T}}$):

$$om\ (\Gamma : [\,V \to \mathcal{D}[V]\,]) : Coalg_{\mathcal{D}}\ [\mathbb{T}(V)] = free_{\mathbb{T}}\ (map_{\mathcal{D}}\ (unit_{\mathbb{T}})\ \circ\ \Gamma, map_{\mathcal{D}}\ (join_{\mathbb{T}})\ \circ\ \Lambda)$$

For a detailed explanation, see [14].

The proof of the adequacy theorem (stating that $R_D\ (\Gamma) = R_O\ (\Gamma)$) is done by using Proposition 3 with *om* (Γ) substituted for *c*. This will immediately lead to the following commutation property:

$$\begin{array}{ccc} \mathbb{T}[V] & \xrightarrow{om\ (\Gamma)} & \mathcal{D}[\mathbb{T}[V]] \\ {\scriptstyle R_D\ (\Gamma)}\downarrow & & \downarrow{\scriptstyle \mathcal{D}\ (R_D\ (\Gamma))} \\ \mathbf{N}_{\mathcal{D}} & \xrightarrow[out_{\mathcal{D}}]{} & \mathcal{D}[\mathbf{N}_{\mathcal{D}}] \end{array}$$

For the proof of this property we apply the alternative proof principle for terms (Proposition 2) using *dm* as $\mathbb{T}Algebra$, and thus we need to verify the fact that $\mathbb{T}Alg?\ (dm)$. The key to this proof is the *unfold_fusion* lemma. Finally, nowhere in the above proofs it was required to use any (language-)specific properties of Λ, making the our approach fully syntax-independent.

4 Semantics of *While*

We have seen that our treatment of semantics is parametric in the concrete set of operational rules: the construction of the operational and denotational models did not depend

[7] The term distributed law can be explained by considering the type of Λ as a proposition, specifying that $\mathbb{T}$ distributes over $\mathcal{D}$.

$$
\begin{array}{lcl}
\mathcal{A}[\![n]\!]s & = & n \\
\mathcal{A}[\![x]\!]s & = & s(x) \\
\mathcal{A}[\![a_1 + a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1 - a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s - \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1 \times a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s \times \mathcal{A}[\![a_2]\!]s
\end{array}
$$

Figure 2 Semantics of arithmetic expressions.

$$
\frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle} \qquad \frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}
$$

$$
\langle x := a, s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s] \qquad \langle \textbf{skip}, s\rangle \Rightarrow s
$$

$$
\langle \textbf{if } c \textbf{ then } S_t \textbf{ else } S_e, s\rangle \Rightarrow \langle S_t, s\rangle \text{ if } s(c) \neq 0
$$

$$
\langle \textbf{if } c \textbf{ then } S_t \textbf{ else } S_e, s\rangle \Rightarrow \langle S_e, s\rangle \text{ if } s(c) = 0
$$

$$
\langle \textbf{while } c \textbf{ do } S, s\rangle \Rightarrow \langle \textbf{if } c \textbf{ then } (S; \textbf{while } c \textbf{ do } S) \textbf{ else skip}, s\rangle
$$

Figure 3 SOS for *While*.

on language specific properties. As such, our treatment could be classified as 'meta-meta-theoretical'.

In this section we demonstrate a paradigmatic example for many concrete (imperative) programming languages. The language we have chosen is *While*, appearing in many textbooks on semantics; e.g., see [17]. The standard structural operational semantics is given as a set of transitions, each of the form $\langle S, s\rangle \Rightarrow \gamma$, where γ is either of the form $\langle S', s'\rangle$ or simply s'.

The complete set of operational rules is given in Figure 3, in *small-step* style.

This system is based on the evaluation of arithmetic expressions which is defined separately in Figure 2, in (a sort of) *big-step* style. Although it is very well possible to specify both styles of semantics in the bialgebraic framework (for example, see [11]) we will treat arithmetic expressions differently, by expressing the semantics directly in *PVS* as a recursive function; see the *PVS* proof files. We have simplified the language by omitting boolean expressions, and restricting conditions to solely (integer) variables. The value 0 will be interpreted as *false*; any other value as *true*.

We specify the operational rules, not directly as a distributive law, but in the so-called *GSOS-format* which is more restrictive: the rules given in this format are functions ρ of type

$$\rho\,[v : \textbf{type}] : \Sigma[\mathcal{D}[v]] \to \mathcal{B}[\mathbb{T}[v]].$$

As a first step, we introduce appropriate functors for representing the syntax and behaviour.

Syntax

In our *PVS* formalization, the syntax functor is derived from an enumeration of operator symbols, and their corresponding arity function. These can subsequently be used as actual parameters of the *Terms* theory (see Section 2) to obtain Σ. Concretely,

SigW [*V* : **type**] : **datatype begin importing** *Expr* [*V*]
  ass (*dst* : *V*, *src* : *Expr*) : *ass* ?
  skip : *skip* ?
  seq : *seq* ?

```
  ifs (con : V) :          ifs ?
  while (con : V) :        while ?
end SigW
```

The imported type $Expr\ [V]$ represents the arithmetic expressions. Aside from the function arW, the following theory contains some auxiliary definitions which facilitate the specification of the operational rules and concrete *While* programs. The representation of the syntax by $\mathbb{T}$ terms requires that the arity of each operation, as returned by arW, should correspond to the number of $\mathbb{T}$ arguments instead of the real number of arguments. This, for instance, explains why *ifs* and *while* have arities 2 and 1, respectively.

```
WhileLang [V : type] : theory begin importing SigW [V], Expr [V]
  arW (s : SigW [V]) : nat = cases (s) of
    ass (v, a) : 0, skip : 0, seq : 2, ifs (c) : 2, while (c) : 1
  endcases
  importing Terms [SigW [V], arW], EmptyFunP
  args0 [x : type] : [below (0) → T [x]] = emptyFun [below (0), T [x]]
  args1 [x : type] (a : T [x]) (i : below (1)) : T [x] = a
  args2 [x : type] (a1, a2 : T [x]) (i : below (2)) : T [x] = if i = 0 then a1 else a2 endif
  assT [x : type] (v : V, src : Expr) :   T [x] = tapp (ass (v, src), args0)
  skipT [x : type] :                      T [x] = tapp (skip, args0)
  seqT [x : type] (s1, s2 : T [x]) :      T [x] = tapp (seq, args2 (s1, s2))
  ifT [x : type] (co : V, t, e : T [x]) : T [x] = tapp (ifs (co), args2 (t, e))
  whileT [x : type] (co : V, b : T [x]) : T [x] = tapp (while (co), args1 (b))
  varT [x : type] (v : x) :               T [x] = tvar (v)
end WhileLang
```

Behaviour

As to the behaviour functor, we must remember that the semantic domain (being the greatest fixpoint $\mathbf{N}_{\mathcal{D}}$ of functor $\mathcal{D}$)[8], cannot be expressed in terms of $\mathcal{D}$ ($= X \times \mathcal{B}[X]$). Again, we will define $\mathbf{N}_{\mathcal{D}}$ as a coinductive data type, and let *PVS* generate the corresponding functor $\mathcal{D}$. The behaviour functor for *While* resembles the behaviour functor for the stream language. However, instead of using a label set, we now need a state transition function for passing the (possibly) modified store s. The store itself maps variables to integer values. Furthermore, to represent the two possibilities for γ in the state transitions, we use the *Maybe* functor which is given below. The standard map_{Mb} and $fold_{Mb}$ operations are defined in terms of the corresponding *map* and *reduce* functions generated by *PVS* itself.

```
MBF : theory begin
  Maybe [x : type] : datatype begin
    nothing :          nothing ?
    just (fromJust : x) : just ?
  end Maybe
  mapMb [a, b : type] (f : [a → b]) : [Maybe [a] → Maybe [b]] = map (f)
```

[8] Some other approaches use $\mathbf{N}_{\mathcal{B}}$ as domain. However, this is not an essential difference since one can easily show that $\mathbf{N}_{\mathcal{B}}$ and $\mathbf{N}_{\mathcal{D}}$ are isomorphic.

```
    fold_Mb [a, b : type] (nf : b, jf : [a → b]) : [Maybe [a] → b]    = reduce (nf, jf)
  end MBF
```

The data types, and basic operations for representing $\mathcal{D}$ in *PVS* are:

```
BF [ST : type] : theory begin importing MBF
  B [x : type] : type = [ST → [ST, Maybe [x]]]
  map_B [a, b : type] (f : [a → b]) (bf : B [a]) : B [b] =
    λ (s : ST) : let ber = bf (s) in (ber`1, map_Mb (f) (ber`2))
end BF

N_D [ST : type] : codatatype begin
    importing BF [ST]
    dz_in (left : N_D, right : B [N_D]) : dz_in ?
end N_D

DCoalgebra [ST : type] : theory begin importing N_D [ST]
  D [x : type]        : type = DZ_struct [ST, x]
  Coalg_D [x : type] : type = [x → D [x]]
  injD [a, b : type] (f : [a → b], g : [a → B [b]]) (x : a) : D [b] = inj_dz_in (f (x), g (x))
  outD : Coalg_D [N_D] = injD (left, right)
  map_D [a, b : type] (f : [a → b]) : [D [a] → D [b]] = injD (f ∘ inj_left, map_B (f) ∘ inj_right)
end DCoalgebra
```

For the sake of completeness, we also give the definition of *DZ_struct* as is generated from $\mathbf{N}_{\mathcal{D}}$ by *PVS*.

```
DZ_struct [ST, x : type] : datatype begin importing BF [ST]
  inj_dz_in (inj_left : x, inj_right : B [x]) : inj_dz_in ?
end DZ_struct
```

Before specifying the operational rules, we have a closer look at the GSOS-format itself. The domain of ρ (i.e., $\Sigma[\mathcal{D}[v]]$) allows us to pattern-match on the outermost symbol. The symbol is parameterized (depending on the arity) with a $\mathcal{D}$ expression which will provide access to the premises of the rule. Since only the sequence operator has rules with premises, we will elaborate on the alternative for ρ that corresponds to this operation. The first component of $\mathcal{D}$ represents the meta-variable on the left-hand side of the premise, whereas the second component represents the right-hand side. Since we do not pass the state explicitly, we have to apply the second component to the state argument of the state transition function that is yielded as a result. By inspecting the outcome of that application we can decide which of the two rules for *seq* applies, and construct the corresponding right-hand side of the conclusion. For the latter, we use the fold operation for *Maybe*. In *PVS*:

```
WhileGSOS [V : type] : theory begin
  STORE : type = [V → int]
  ρ [v : type] (sf : Σ[D[v]]) : B[T[v]] = cases sf`1 of
    ...
    seq : let a0 = sf`2 (0), a1 = sf`2 (1) in
      λ (st : STORE) : let rst = inj_right (a0) (st) in
        fold_Mb ((rst`1, just (varT (inj_left (a1)))),
          λ (s1 : x) : (rst`1, just (seqT (varT (s1), varT (inj_left (a1))))))) (rst`2),
```

```
    ...
    endcases
end WhileGSOS
```

In order to obtain a distributive law of $\mathbb{T}$ over $\mathcal{D}$, ρ needs to undergo a two-step transformation. The first step is to expand ρ's codomain using the auxiliary function *injD* defined in the theory $\mathcal{D}Coalgebra$:

$$\tau\ [v : \mathbf{type}] : [\Sigma[\mathcal{D}[v]] \to \mathcal{D}[\mathbb{T}[v]]] = injD\ (tapp \circ map_\Sigma\ (tvar \circ inj\_left), \rho)$$

Adjusting the domain is slightly more involved, and requires an appropriate use of $fold_\mathbb{T}$:

$$\Lambda\ [v : \mathbf{type}] : [\mathbb{T}[\mathcal{D}[v]] \to \mathcal{D}[\mathbb{T}[v]]] = fold_\mathbb{T}\ (map_\mathcal{D}\ (unit_\mathbb{T}), map_\mathcal{D}\ (join_\mathbb{T})\ \circ\ \tau)$$

This construction does not affect the naturality property (Lemma *law_natural*). Moreover, it guarantees that Λ is indeed a distributive law (Lemma *law_distributive*).

Running a program

There is one discrepancy between our formalization and a standard denotational semantics for *While* as, for instance given by [17]. In our case, the mathematical object describing the effect of executing each construct is the greatest fixed point $\mathbf{N}_\mathcal{D}$ of $\mathcal{D}$. In the standard case, this is a (possibly partial) state transition function, which is obtained by composing the functions that correspond to the components of this construct. Moreover, for while statements, a standard denotational semantics also requires fixed points. The result of executing a program in our framework, $\mathbf{N}_\mathcal{D}$, is not a single state transition function, but a (possibly infinite) stream of functions that still need to be interconnected. In *PVS*, however, all functions have to be total. We solve this issue by defining the following total variant[9] of a compose function which returns the constructed transition function after N execution steps.

```
composeN (N : nat, dz : N_D) (ist : STORE) : recursive ST =
  if N = 0 then ist
  else let res = right (dz) (ist) in
    cases res`2 of nothing : res`1,
                   just (x) : composeN (N - 1, x) (res`1) endcases endif measure N
```

To illustrate program execution, we use the following program that computes the 12^{th} Fibonacci number. It uses 3 variables each identified by a number. Variable 2 will hold the final result.

```
PFib10 : T = seqT (assT (0, enum (10)), seqT (assT (1, enum (1)), seqT (assT (2, enum (1)),
  (whileT (0, seqT (assT (3, eplus (evar (1), evar (2))), seqT (assT (1, evar (2)),
    seqT (assT (2, evar (3)), assT (0, emin (evar (0), enum (1))))))))))
```

We execute this program and apply the final store, obtained after 62 steps (just enough for the while loop to terminate), to variable 2, producing[10] the value 144.

---

[9] In Pvs, totality is enforced by a so-called *measure* specification which is mandatory when defining a recursive function. This measure is used to guarantee (by generating special proof obligations) that the function indeed terminates.

[10] The answer was obtained by executing our specification in the functional language *Clean*. The current, so-called, ground evaluator of *PVS* seems to have difficulties with evaluating expressions containing infinite codata structures.

$$EXEC : nat = composeN\ (62, R_O\ (emptyEnv)\ (PFib10))\ (\lambda\ (i : nat) : 0)\ (2)$$

Observe that we used R_O (based on the operational model) to execute the program. Equivalently, we could have used the denotational version R_D, obviously leading to the same result, since we proven that R_O and R_D equal.

5 PVS formalization

Our main motivation for developing this formalization was to investigate whether or not implementing abstract categorical concepts in *PVS* 6.0 is feasible. The case study we performed was based on previous, similar experiments with *Coq*.

As far as this case study is concerned, the main difference between *Coq* and *PVS* is that *Coq* is equipped with a rich type class system offering type classes as first class citizens. Therefore, in *Coq*, functors, monads, and (co)algebras can be naturally represented. *PVS* offers parameterized theories, but using these as a substitute for *Coq*'s type classes in general is definitely a setback. Moreover, like *PVS*, *Coq* suffers from the same restriction that polymorphism is only first-order. This definitely reduces the generality of the formalisation, however, we managed to separate the language-specific components from the more abstract categorial concepts such that changes in the syntax and/or behaviour of the programming language hardly affects the description in its entirety.

Additionally, for the most part these aspects of our formalization do not obstruct the proving process. There were no fundamental problems which could not be resolved due to restrictions of *PVS*'s specification language. The rich support for abstract (co)data types (including the facility for automatic generation of common theories) has shown to be adequate.

There was, however, a minor issue obstructing the proving process to some extent. When importing a parameterized theory, the user must explicitly specify which actual arguments are required. In a truly polymorphic case this matter would have been solved by the type checker (as is done in *Coq* or in *Haskell*). Unfortunately, *PVS* lacks the ability to resolve theory instantiations automatically. To some extent, this is also the case for instantiation of declaration parameters. We encountered situations in which the type checker was not capable of determining the correct instance types. However, from the discussions with the developers of *PVS* we concluded that this was not a fundamental issue but a temporary defect of the typing algorithm, that is expected to be repaired in the near future.

Finally, the goal of our experiment was not to compare *PVS* with *Coq*. In terms of development times, for our specific example these were about the same. Of course, we have to take into account that we first did our experiment with *Coq* and all theoretical difficulties were already solved when we started the exercise with *PVS*. We are convinced that the time it takes to formalize a relatively complex system, such as the bialgebraic framework, is mainly determined by the experience of the user. This development process is barely retarded by (the peculiarities of) the specific proof assistant itself.

6 Related work

This work was inspired by our earlier work on modularity, the formalization of Modular Structural Operational Semantics (MSOS) [15]. The present paper can be seen as a contribution to field of bialgebraic semantics, starting with Turi and Plotkin's research [18], and resulting in a uniform categorical treatment of semantics. They abstracted from concrete syntactical and semantical details by characterizing these language-dependent issues

by a distributive law between syntax and behaviour. By means of a categorical construct, both an operational and a denotational model were obtained, and moreover the adequacy of these models could be proven. Klin [12] gives an introduction to the basics of bialgebras for operational semantics that was used in the present formalization. He also sketches the state-of-the-art in this field of research.

The distributive law actually describes a syntactic format for specifying operational rules. This abstract so-called GSOS format has been applied to several areas of computer science. For example, in his thesis [2] Bartels gives concrete syntactic rule formats for abstract GSOS rules in several concrete cases. Variable binding, which is a fundamental issue in, for example, λ-calculus or name passing π-calculi, is addressed in [6]. The authors show that name binding fits in the abstract GSOS format. This was refined further in [7].

In [5] a framework is introduced, called MTC, for defining and reasoning about extensible inductive datatypes which is implemented as a Coq library. It enables modular mechanized metatheory by allowing language features to be defined as reusable components. Similar to our work, MTC's modular reasoning is based on universal property of folds [16], offering an alternative to structural induction.

A significant contribution to the work on interpreters for programming languages, is that of the application of monads in order to structure semantics. Liang et al. [13] introduced monad transformers to compose multiple monads and build modular interpreters. Jaskelioff et al. use [11] as a starting point, and provide monad-based modular implementation of mathematical operational semantics in *Haskell*. The authors also give some concrete examples of small programming languages specified in GSOS-format. Our example in Section 2 is inspired by this work. Although, [11] strictly follows the approach of Turi and Plotkin, there is no formal evidence that their construction is correct, i.e., there are no 'pen and paper' or machine-checked proofs given. The latter issue is addressed by recent work of [9] who introduce modular proof techniques for equational reasoning about monads.

7 Conclusions

We presented a formalization in *PVS* version 6.0 of Turi and Plotkin's work based on category theory. Our main goal was to investigate whether this could be done in a convenient way. Except for some minor flaws in *PVS*'s type checker discussed in Section 5, we did not encountered any fundamental issues that seriously hindered the proving process. Our experiment also resulted in a *PVS* framework which can be used for formal reasoning about programming languages in general, in addition to reasoning about specific programs. Moreover, it offers the user the possibility to choose between either denotational or operational semantics at any point in his application.

Our future plans comprise of experimenting with our framework in formal reasoning with case studies in specific examples of denotational and operational semantics, and to extend the framework with an axiomatic semantics.

References

1 Michael Barr and Charles Wells. *Category theory for computing science*, volume 49. Prentice Hall New York, 1990.

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

3 Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation can't be traced. *J. ACM*, 42(1):232–268, January 1995.

4 Marcello M Bonsangue, Helle Hvid Hansen, Alexander Kurz, and Jurriaan Rot. Presenting distributive laws. In *Algebra and Coalgebra in Computer Science*, LNCS, pages 95–109. Springer, 2013.

5 Ben Delaware, Bruno C.d.S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, New York, NY, USA, 2013. ACM.

6 Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, LICS '99, pages 193–202, Washington, DC, USA, 1999. IEEE Computer Society.

7 Marcelo Fiore and Sam Staton. A congruence rule format for name-passing process calculi from mathematical structural operational semantics. In *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science*, LICS '06, pages 49–58, Washington, DC, USA, 2006. IEEE Computer Society.

8 Ulrich Hensel and Bart Jacobs. Coalgebraic theories of sequences in pvs. *J. Log. Comput.*, 9(4):463–500, 1999.

9 Ralf Hinze and Daniel W.H. James. Proving the unique fixed-point principle correct: an adventure with category theory. *SIGPLAN Not.*, 46(9):359–371, September 2011.

10 Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:62–222, 1997.

11 Mauro Jaskelioff, Neil Ghani, and Graham Hutton. Modularity and implementation of mathematical operational semantics. *Electron. Notes Theor. Comput. Sci.*, 229(5):75–95, March 2011.

12 Bartek Klin. Bialgebras for structural operational semantics: An introduction. *Theoretical Computer Science*, 412(38):5043–5069, 2011. CMCS Tenth Anniversary Meeting.

13 Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 333–343, New York, NY, USA, 1995. ACM.

14 Ken Madlener and Sjaak Smetsers. GSOS formalized in Coq. In *The 7th International Symposium on Theoretical Aspects of Software Engineering (TASE2013)*, pages 199–206, 2013. Birmingham, UK, 2013. IEEE.

15 Ken Madlener, Sjaak Smetsers, and Marko C. J. D. van Eekelen. Formal component-based semantics. In Michel A. Reniers and Pawel Sobocinski, editors, *SOS*, volume 62 of *EPTCS*, pages 17–29, 2011.

16 Erik Meijer, Maarten M. Fokkinga, and Ross 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, London, UK, UK, 1991. Springer-Verlag.

17 Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: a formal introduction*. John Wiley & Sons, Inc., New York, NY, USA, 1992.

18 Daniele Turi and Gordon Plotkin. Towards a mathematical operational semantics. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science*, LICS '97, pages 280–291, Washington, DC, USA, 1997. IEEE Computer Society.

19 Philip Wadler. Theorems for free! In *Functional Programming Languages And Computer Architecture*, pages 347–359. ACM Press, 1989.