Correctness of Functionally Specified Denotational and Operational Semantics

Sjaak Smetsers[1], Ken Madlener[1], and Marko van Eekelen[1,2]
{S.Smetsers,K.Madlener,M.vanEekelen}@cs.ru.nl

[1] Institute for Computing and Information Sciences
Radboud University Nijmegen
[2] School of Computer Science
Open University of the Netherlands

Abstract. Both operational and denotational semantics are popular approaches for reasoning about properties of programs and programming languages. Each semantics has its own specific aims and applications, and studying the relation between them allows the exploitation of benefits of both styles. This paper presents a common functionally specified formalization of these styles. As specification language we use the theorem prover Pvs. This enables both the execution of the specified functions and the formal proving of properties such as adequacy. The underlying approach applies categorical work of Turi and Plotkin, proving the adequacy theorem in a uniform setting (the GSOS format). The main characteristic of the present set-up is that the proof of this theorem is syntax independent: it is not restricted to a specific programming language. This creates a framework giving the opportunity to formally prove properties about both programming languages in general and specific programs. The power of this framework lies in having the option of choosing between denotational and operational semantics at will, within just one single functional setting.
Keywords and phrases: functional specification, operational semantics, denotational semantics, bialgebras, distributive laws, adequacy, theorem proving, Pvs

1 Introduction

Formal definitions of programming languages consist of an implementation independent description of both the syntax and the semantics of programs. The description can be used not only to prove correctness of specific programs but also to develop metatheory providing a collection of general program independent properties. The latter can be a very convenient foundation for building implementations.

The formal definition of a real-world programming language can take colossal proportions. The process of verifying metatheory often exceeds human capabilities; due to its inherent complexity, development time often grows exponentially. The best alternative for complete verification is to employ well-established methods, such as type systems and operational/denotational semantics.

Functional programming languages are successfully used, not only as powerful programming tool, but also as a formalism to specify the semantics of newly developed language concepts [12, 23, 11], system descriptions [21], and abstract machines [13]. The fact that these functional specifications are executable is very

useful as a first step towards a full formalization and to get a good grip on the technicalities involved. Despite their expressiveness, functional languages cannot prevent that occasionally errors and inconsistencies arise that may not be easily discovered via execution. A promising approach in reducing these errors is to use mechanized verification tools. These verification tools are usually based on classical, typed higher-order logic. The (functional) specification languages of these tools provide automatic code generation from functional specifications, which enables the execution of such specifications. This feature is often used as an additional check of the developed concepts, before one starts with formally proving properties of these concepts.

Such functional specifications are often operational semantics. Denotational semantics are generally specified mathematically specifying the mathematical meaning of language concepts. Ideally, both denotational and operational semantics are defined and their conformance (*adequacy*) is formally proven. Generally, such adequacy proofs are done informally. Examples of that are e.g. Launchbury's natural semantics for lazy evaluation [14] and the editor arrow framework semantics by Achten et al. [1].

In this paper, we present a formalization of both popular styles of semantic specifications: (structural) operational semantics and denotational semantics. Inspired by a recent category theoretic Coq formalisation [17], this paper provides a metatheoretic framework in the Pvs theorem prover by introducing a functional, executable specification of both denotational and operational language semantics.

The approach is based on a framework developed by Turi and Plotkin [22] unifing 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 a distributive law corresponding to a set of operational rules. The format in which these operational rules are specified is known as the GSOS format; see [3].

The contribution of our work is twofold. Several years ago the PoplMark challenge was launched calling for experiments on verifications of metatheory and semantics using proof tools. Various researchers have answered this appeal, which has led to implementations of a specific type system in most of the state-of-the-art theorem provers available. However, until now no implementation has developed in Pvs; the development has been constrained to only a few metatheoretical experiments. This is a privation, because Pvs has shown to be very successful in proving properties of computer programs. In this paper, we investigate whether Pvs is adequately suited for developing metatheory. In our attempt, we extend to an even higher abstraction level, namely we use Turi and Plotkin's categorical description as the point of inception. Our main goal is to formally prove that the construction indicated by [22] is sound, resulting in the so-called *adequacy theorem*. The second contribution is a Pvs formalization providing a framework for facilitating both formal reasoning about and experimentation with semantics of programming languages allowing the user to choose between either denotational or operational semantics at any point.

Section 2 describes the relation between operational and denotational semantics which is roughly based on the categorical framework of Turi and Plotkin. For a simple language, both an operational and a denotational model is specified,

and the equivalence of these models is proven. In order to enhance expressivity, we first generalize terms in Section 3. Subsequently in Section 4, operational and denotational models are defined generically using bialgebras in order to prove the adequacy theorem in a syntax-independent manner. Finally in Section 5, to achieve a fully general formalization of the framework, we introduce the expressive GSOS format. We conclude with related and future work.

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

2 Background

We start with a brief introduction to the framework of Turi and Plotkin. To avoid getting enmeshed in category theory right from the start, we follow the approach as, for instance, taken by [8] and [9] who explain and illustrate the technicalities with definitions and examples written in HASKELL.

We use a very simple language about streams (see also [10]), of which the operational rules are given in Figure 1. These rules inductively define a transition relation on $\mathbb{T} \times L \times \mathbb{T}$, where $\mathbb{T}$, L denote the sets of closed terms, and outputted 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, by repeatedly taking the head of its first argument, and calling itself recursively on the swapped tails. The Zip function behaves similarly to Alt except that it does not discard the label of the second stream argument. At first instance we will ignore the rule for Zip. The first step in the formalization of such a language is to express both the *signature* and *behaviour* (i.e. the result of an operation) of the operations as *functors*. In HASKELL, we can model a functor as a datatype. The map that is inextricably interlaced with such a functor is defined by making the data type an instance of the Functor type class.

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

The *signature functor* is then defined by the following data type and instance definition:

```
data Σ p = AS | BS | Alt p p
instance Functor Σ where
  fmap f AS = AS
  fmap f BS = BS
  fmap f (Alt p1 p2) = Alt (f p1) (f p2)
```

Similarly, we represent the *behaviour functor* as follows. Here L corresponds to our label set, and the functor $\mathcal{B}$ just pairs a label from L with p.

```
data L    = A | B
data B p = L :*: p
instance Functor B where
  fmap f (l :*: p) = l :*: f p
```

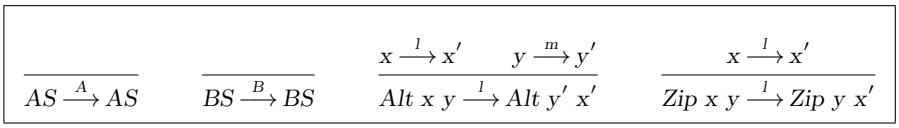


Fig. 1. A simple language for streams.

The set of terms $\mathbb{T}$ over a signature functor f is defined as:

$$\textbf{data}\ \mathbb{T}\ f = \mathsf{App}\ (f\ \mathbb{T})$$

As PVS does not support (higher-order) polymorphism this general approach cannot be followed. Instead we will mimic the same intentions with the aid of parameterized theories. We start with the set of terms, which are defined by:

```
T [F : type,                % for operations
   ar : [F -> nat]] : % for arity of operations
datatype begin
  tapp (op : F, args : [below (ar (op)) -> T]) : tapp ?
end T
```

The data type is parametric in the signature which is represented by F, ar. The following theory introduces the notion of Σ-algebra, which is essentially a pair consisting of an *object* X (called the carrier of the algebra), and a structure map $\Sigma(X) \to X$. In the PVS specification the argument object X of the functor Σ is left implicit. At the same time, we introduce a general operation for processing terms that avoids explicit recursion (and facilitates equational reasoning, as we shall see later on), here called $\mathsf{fold}_{\mathbb{T}}$; see also [19].

```
Algebras [F : type, ar : [F -> nat], X : type] : theory begin
  importing T [F, ar]
  Σ : type    = [f : F, [below (ar (f)) -> X]]
  Alg : type = [Σ -> X]
  fold_T (a : Alg) (t : T) : X = reduce (a) (t)
end Algebras
```

Observe that $\mathsf{fold}_{\mathbb{T}}$ is just a re-definition of *reduce* which is part of a theory automatically generated by PVS when $\mathbb{T}$ is typechecked.

In the representation of the behaviour functor $\mathcal{B}$ we anticipate on the fact that the terms are executed according to the operational rules of the language. This execution yields an infinite stream of labels. Categorically, this stream is obtained by taking the greatest fixpoint of $\mathcal{B}$ leading to the final coalgebra of the category of $\mathcal{B}$-coalgebras. By introducing the following (co)datatype this is straightforward in HASKELL.

```
codata N f = N (f (N f))
unfold   :: Functor b => (x -> b x) -> x -> N b
unfold g = N . fmap (unfold g) . g
```

To express this in PVS we use PVS's capability to introduce co-inductive datatypes. However, we cannot define **N** as above, since this would require some sort of higher-order polymorphism. Instead, we define **N** $\mathcal{B}$ directly as a codata type, named $\nu\mathcal{B}$, 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*).

```
νB [L : type] : codatatype begin
  nb_in (el : A, next : νB) : nb_in ?
end νB
```

A coalgebra is the dual of an algebra: for a functor $\mathcal{B}$, the coalgebra consists of an object D and a structure map $D \to \mathcal{B}(D)$. In the PVS specification, the object argument D of $\mathcal{B}$ is again left implicit.

```
Coalgebras [L, D : type] : theory begin
  importing νB [A]
  B       : type = NB_struct [L, D]
  CoAlg   : type = [D → B]
  out_νB (nb : νB) : B(νB) = inj_nb_in (el (b), next (b))
  unfold (c : CoAlg) (z : D) : νB = coreduce (c) (z)
end Coalgebras
```

The functor $\mathcal{B}$ and operation unfold coincide with the generated record *NB_struct* and the (coinductive) function *coreduce*. As the components of *NB_struct* (as can be seen in the definition of $\mathsf{out}_{\nu\mathcal{B}}$), are also used, we give its definition that is generated by PVS (L, D have been substituted for the corresponding theory parameters).

```
NB_struct : datatype begin
  inj_nb_in (inj_el : L, inj_next : D) : inj_nb_in ?
end NB_struct
```

For our concrete language we give an appropriate instance of Σ:

```
LANG : theory begin
  Symbol : type = {A, B, Alt}
  L : type = {AL, BL}
  ar (s : Symbol) : nat =
    cases s of
      A :   0,
      B :   0,
      Alt : 2
    endcases
end LANG
```

Now all the ingredients are available to represent the operational semantics of the example language in PVS.

```
OM : theory begin
  binapp (op : {s : Symbol | ar (s) = 2}, a1, a2 : T) : T =
```

```
    tapp (op, λ (i : below (2)) : if i = 0 then a1 else a2 endif)
  om (t : T) : recursive B =
    cases t of
      tapp (oper, args) :
        cases oper of
          A   : inj_bin (AL, tapp (A, args)),
          B   : inj_bin (BL, tapp (B, args)),
          Alt : let r0 = om (args (0)),
                    r1 = om (args (1))
                in inj_nb_in (inj_el (r0), binapp (Alt, inj_next (r1), inj_next (r0)))
        endcases
    endcases
  measure t BY <<
end OM
```

Observe that om has type $[\mathbb{T} \to \mathcal{B}]$ which is identical to *Coalg* (with $\mathbb{T}$ as carrier). This enables the execution of a term t by (co-iteratively) unfolding om (t):

```
run (t : T) : νB = unfold (om) (t)
```

In [22] the denotational semantics is considered as the dual version of the operational semantics. The underlying denotational model dm essentially maps each syntactic construct to a mathematical object describing the effect of executing that construct; e.g. see [20]. In the present example this mathematical object is a function $(X \to \nu\mathcal{B}) \to \nu\mathcal{B}$, so dm : $(\Sigma\ X) \to (X \to \nu\mathcal{B}) \to \nu\mathcal{B}$, which in turn is isomorphic to dm : $(\Sigma\ \nu\mathcal{B}) \to \nu\mathcal{B}$, (e.g. see [16] for a categorical explanation of this property of parametric models). We will use the latter, algebraic format (observe that dm is a Σ-algebra with $\nu\mathcal{B}$ as carrier) in the following PVS specification:

```
DM : theory begin
  Unit : datatype begin unit : unit ? end Unit
  constCoalg (l : L) : Coalg [L, Unit] = λ (x : Unit) : inj_nb_in (l, x)
  const (l : L) : B = unfold (constCoalg (l)) (unit)
  altCoalg : Coalg [L, [νB, νB]] =
    λ (xy : [νB, νB]) : let x = xy`1, y = xy`2
      in inj_nb_in (el (x), (next (y), next (x)))
  alt (x, y : νB) : νB = unfold (altCoalg) ((x, y))
  dm (sf : Σ) : νB =
    cases sf`1 of
      A :   const (AL),
      B :   const (BL),
      Alt : alt (sf`2 (0), sf`2 (1))
    endcases
end DM
```

Evaluation is obtained by folding the algebra dm:

```
eval (t : T) : νB = fold_T (dm) (t)
```

For these specific semantic models we can prove the adequacy theorem:

run_is_eval : **theorem** $\forall\ (t : \mathbb{T}) : \mathsf{run}\ (t) = \mathsf{eval}\ (t)$

The proof proceeds by a mix of induction (on $\mathbb{T}$) and coinduction (on $\nu\mathcal{B}$). Although the proof itself is not very difficult, it is syntax-dependent and therefore rather ad hoc: choosing a different language would change the proof significantly. In the next sections we present a more structured approach.

3 Generic terms

To prepare for the more general bialgebraic treatment of semantics, we extend the representation of terms with variables:

$\mathbb{T}\ [V : \mathbf{type}, F : \mathbf{type}, ar : [F \to nat]] : \mathbf{datatype\ begin}$
  $tvar\ (var\_id : V)$: $tvar$?
  $tapp\ (op : F, args : [below\ (ar\ (op)) \to \mathbb{T}])$: $tapp$?
end $\mathbb{T}$

Those who are familiar with this subject will probably recognize $\mathbb{T}$ as the *free monad generated by the signature functor* corresponding to (F, ar). $\mathbb{T}$ itself is also a functor, and the object map for $\mathbb{T}$ is again generated automatically by PVS, together with an adjusted folding operation. To fit in with standard terminology we rename the generated operation to $\mathsf{fold}_{\mathbb{T}}$:

$\mathsf{fold}_{\mathbb{T}}\ (e : [V \to X], a : Alg)\ (t : \mathbb{T}) : X = reduce\ (e, a)\ (t)$

Here Alg is the same as defined in the previous section.

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

$fold\_unique$: **proposition** $\forall\ (e : [V \to X], a : Alg, g : [\mathbb{T} \to X]) :$
  $g \circ tapp = a \circ \mathsf{map}_{\Sigma}\ (g) \ \wedge\ g \circ tvar = e \Rightarrow g = \mathsf{fold}_{\mathbb{T}}\ (e, a)$

This uniqueness property 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.

First, we show that $\mathbb{T}$ is a monad in the categorical sense. Note that HASKELL introduces the concept of monad as a type class consisting of two overloaded operations *return* and *bind*. It is also possible to define a monad in terms of two other operations, *unit* and *join*, of which *unit* is the same as *return*. This formulation fits closely with the definition of monads in category theory. Concretely, we define *unit* and *join* by[3]:

$\mathbb{T}\mathsf{Monad}\ [V, F : \mathbf{type}, ar : [F \to nat]] : \mathbf{theory\ begin}$
  $\mathbb{T}(V) : \mathbf{type}\ = \mathbb{T}\ [V, F, ar]$
  $\mathbb{T}^2(V) : \mathbf{type} = \mathbb{T}\ [\mathbb{T}(V), F, ar]$

[3] We occasionally deviate from PVS's syntax, in particular when specifying signatures of functions, which are treated as if they were polymorphic.

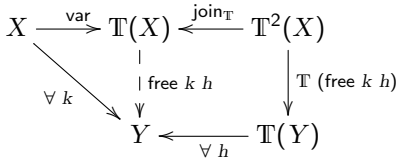
$\mathsf{unit}_{\mathbb{T}}\ (v : V) \quad : \mathbb{T}(V) = tvar\ (v)$
$\mathsf{join}_{\mathbb{T}}\ (t : \mathbb{T}^2(V)) : \mathbb{T}(V) = \mathsf{fold}_{\mathbb{T}}\ (id, tapp)\ (t)$
end $\mathbb{T}$Monad

From category theory we borrow the notion $\mathbb{T}$-algebra, which will later be used to introduce an alternative proof principle for folding. A $\mathbb{T}$-algebra (or, more verbosely, an algebra for the $\mathbb{T}$ monad) is a 'plain' algebra with two additional properties. In PVS:

$\mathbb{T}\mathsf{Alg}\ ?\ (h : [\mathbb{T}(V) \to V]) : bool =$
$h \circ \mathsf{unit}_{\mathbb{T}} = id \ \wedge\ h \circ \mathsf{map}_{\mathbb{T}}\ (h) = h \circ \mathsf{join}_{\mathbb{T}}$

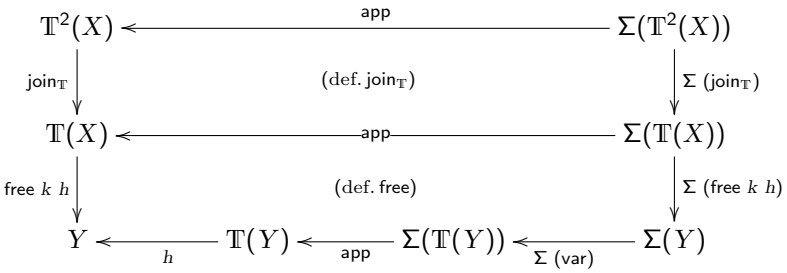
We end this section with a lemma that resembles the uniqueness property of $\mathsf{fold}_{\mathbb{T}}$.

Lemma 1. *Let $k : X \to Y$ and $h : \mathbb{T}(Y) \to Y$ such that h is a $\mathbb{T}$-algebra. Set* $\mathsf{free}\ k\ h := \mathsf{fold}\ k\ (h \circ \mathsf{app} \circ \mathsf{map}_{\mathbb{T}}\ (\mathsf{var}))$. *Then* $\mathsf{free}\ k\ h$ *is unique in making the following diagram commute*[4] *:*



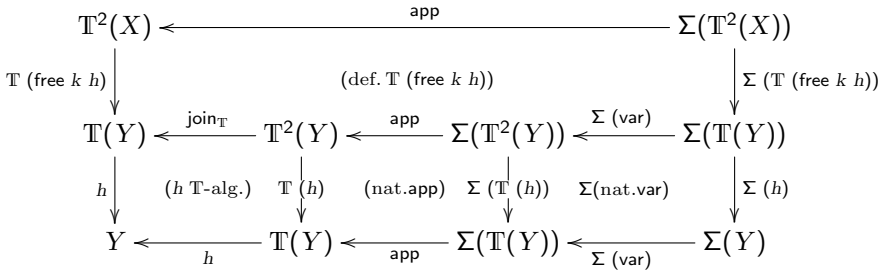
We only show that the diagram commutes, and not that $\mathsf{free}\ k\ h$ is unique. Before giving a proof in PVS itself, we develop a diagrammatic version for the application case (indicated by the right part of the above diagram) on paper. The idea is to construct a separate diagram for each side, and subsequently use Proposition *fold_unique* to show that both compositions can be written as the same application of $\mathsf{fold}_{\mathbb{T}}$. The uniqueness property then yields the desired result.

For the path $\mathsf{free}\ k\ h \circ \mathsf{join}_{\mathbb{T}}$ we have:



The (correctness of the) subdiagrams used in this diagram follow(s) directly from the definition of $\mathsf{join}_{\mathbb{T}}$ and free. For the path $h \circ \mathsf{map}_{\mathbb{T}}\ (\mathsf{free}\ k\ h)$ on the right-hand side we have:

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



Proving the subdiagrams used above is just a matter of simple casuistics. Observe that in both diagrams the paths at the bottom are identical. These diagrams can be translated directly into Pvs. The main advantage of using diagrams is that they are constructed in a type driven manner. The operations on the arrows follow almost directly from the object types. In a pure textually conducted proof, the type information is usually not taken into account, and hence equational reasoning is solely based on the operations themselves.

4 Bialgebras

The essence of [22]'s framework is the following: instead of defining the operational and denotational models separately, the operational rules of the language are described by a specific syntactic format from which both semantical models can be obtained generically (i.e. syntax-independently).

As we have seen in Section 2, the operational model is a $\mathcal{B}$-coalgebra whereas the denotational model is a Σ-algebra. Such a combination is a special case of a general categorical concept known as a *bialgebra*. Formally, a bialgebra (for Σ, $\mathcal{B}$) is a triple $\langle \mathsf{V}, \mathsf{a}, \mathsf{c} \rangle$ such that a is a Σ-algebra and c is a $\mathcal{B}$-coalgebra. For two bialgebras, a *bialgebra homomorphism* is a mapping that is both a Σ-algebra homomorphism and a $\mathcal{B}$-coalgebra homomorphism. We are interested in Λ-bialgebras: bialgebras equipped with a so-called *distributed law* Λ. In this section, such a law corresponds to functions of type $\Sigma(\mathcal{B}(V)) \to \mathcal{B}(\Sigma(V))$. We will show how the operational and denotational model can be derived for any arbitrary instance of Λ.

The law that represents the operational rules of our simple stream language is specified by:

$$
\begin{array}{l}
\Lambda\ (sf : \Sigma(\mathcal{B}(V))) : \mathcal{B}(\Sigma(V)) = \\
\quad \textbf{cases}\ sf\text{`}1\ \textbf{of} \\
\qquad B: \quad inj\_nb\_in\ (BL, (B, \lambda\ (i : below\ (0)) : inj\_next\ (sf\text{`}2\ (i)))), \\
\qquad A: \quad inj\_nb\_in\ (AL, (A, \lambda\ (i : below\ (0)) : inj\_next\ (sf\text{`}2\ (i)))), \\
\qquad Alt : \textbf{let}\ a0 = sf\text{`}2\ (0), a1 = sf\text{`}2\ (1) \\
\qquad\quad \textbf{in}\ inj\_nb\_in\ (inj\_el\ (a0), (Alt, \lambda\ (i : below\ (2)) : \\
\qquad\qquad \textbf{if}\ i = 0\ \textbf{then}\ inj\_next\ (a1)\ \textbf{else}\ inj\_next\ (a0)\ \textbf{endif})) \\
\quad \textbf{endcases}
\end{array}
$$

We now clarify how om and dm are obtained from this law. As to om, suppose we have a function $\Gamma : V \to \mathcal{B}(V)$ that maps each variable to its behaviour. For a given

term, the idea is to use Γ in the variable case, and to apply Λ in the application case. A first attempt to express this equationally could be:

$$\mathsf{om} \circ \mathit{tvar} = \Gamma\ [1] \ \wedge\ \mathsf{om} \circ \mathit{tapp} = \Lambda \circ \mathsf{map}_\Sigma\ (\mathsf{om})\ [2]$$

However, the types of om in (1) and (2) are incompatible. We must also remember that om should yield a result of type $\mathbb{T}(V) \to \mathcal{B}(\mathbb{T}(V))$. Therefore, we adjust both the results of Γ and Λ in the following way:

$$\mathsf{om} \circ \mathit{tvar} = \mathsf{map}_\mathcal{B}\ (\mathit{tvar}) \circ \Gamma \ \wedge\ \mathsf{om} \circ \mathit{tapp} = \mathsf{map}_\mathcal{B}\ (\mathit{tapp}) \circ \Lambda \circ \mathsf{map}_\Sigma\ (\mathsf{om})$$

Now we can apply Proposition *fold_unique*, and define om as

$$\begin{array}{l}\mathsf{om}\ (\Gamma : [\,V \to \mathcal{B}(V)]\,)\ (t : \mathbb{T}(V)) : \mathcal{B}(\mathbb{T}(V)) = \\ \quad \mathsf{fold}_\mathbb{T}\ (\mathsf{map}_\mathcal{B}\ (\mathit{tvar}) \circ \Gamma, \mathsf{map}_\mathcal{B}\ (\mathsf{app}) \circ \Lambda)\ (t)\end{array}$$

The denotational model is obtained by taking the dual construction of om: $\mathsf{fold}_\mathbb{T}$ becomes unfold, *tapp* is replaced by $\mathsf{out}_{\nu\mathcal{B}}$, and composition is reversed. This yields to

$$\mathsf{dm}\ (\mathit{sb} : \Sigma(\nu\mathcal{B})) : \nu\mathcal{B} = \mathsf{unfold}\ (\Lambda \circ \mathsf{map}_\mathbb{T}\ (\mathsf{out}_{\nu\mathcal{B}}))\ (\mathit{sb})$$

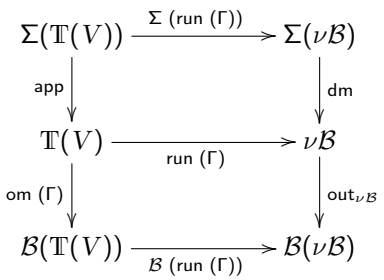
Running a term according to the operational model, and evaluating a term according to the denotational model is done in the same way as in Section 2,

$$\begin{array}{l}\mathsf{run}\ (\Gamma : [\,V \to \mathcal{B}(V)]\,)\ (t : \mathbb{T}(V)) : \nu\mathcal{B}\ = \mathsf{unfold}\ (\mathsf{om}\ (\Gamma))\ (t) \\ \mathsf{eval}\ (\Gamma : [\,V \to \mathcal{B}(V)]\,)\ (t : \mathbb{T}(V)) : \nu\mathcal{B} = \mathsf{fold}_\mathbb{T}\ (\mathsf{unfold}\ (\Gamma), \mathsf{dm})\ (t)\end{array}$$

To show adequacy, it is sufficient to acknowledge that Λ is a natural transformation, i.e. the concrete definition of Λ for a specific language does not affect the structure of the proof. Unfortunately, there are not many sensible rules that fit in this simple format. The *Zip* rule, for example, cannot be expressed without violating typability. E.g., the following attempt will not typecheck anymore.

$$\begin{array}{l}\Lambda\ (\mathit{sf} : \Sigma(\mathcal{B}(V))) : \mathcal{B}(\Sigma(V)) = \\ \quad \textbf{cases}\ \mathit{sf}\text{`}1\ \textbf{of} \\ \quad \ldots \\ \quad\quad \mathit{Zip} : \textbf{let}\ \mathit{a0} = \mathit{sf}\text{`}2\ (0), \mathit{a1} = \mathit{sf}\text{`}2\ (1) \\ \quad\quad \textbf{in}\ \mathit{inj\_nb\_in}\ (\mathit{inj\_el}\ (\mathit{a0}), (\mathit{Zip}, \lambda\ (i : \mathit{below}\ (2)) : \\ \quad\quad\quad \textbf{if}\ i = 0\ \textbf{then}\ \mathit{a1}\ \textbf{else}\ \mathit{inj\_next}\ (\mathit{a0})\ \textbf{endif})) \\ \quad \textbf{endcases}\end{array}$$

We end this section with a diagram showing the concrete bialgebras used so far, namely $\langle \mathbb{T}(V), \mathsf{app}, \mathsf{om}\ (\Gamma)\rangle$ and $\langle \nu\mathcal{B}, \mathsf{dm}, \mathsf{out}_{\nu\mathcal{B}}\rangle$, together with their connecting homomorphism run (Γ) (which is provably equal to eval (Γ)).



5 GSOS

For a general approach to formalized semantics, the plain syntactic format as prescribed by the distributive law appeared to be too restrictive. Instead of using natural transformations from $\Sigma(\mathcal{B}(V))$ to $\mathcal{B}(\Sigma(V))$ (distributing a functor over a functor), we will consider laws that distribute a monad over a functor. More specifically, our laws are functions with signature $\mathbb{T}(\mathcal{D}(V)) \to \mathcal{D}(\mathbb{T}(V))$, where $\mathcal{D}(X) = X \times \mathcal{B}(X)$, also known as the *cofree copointed functor of* $\mathcal{B}$. This signature, however, does not directly match the syntactical format in which the operational rules are described. This, so-called GSOS-format is slightly more restrictive: the rules ρ given in this format are functions with type $\Sigma(\mathcal{D}(V)) \to \mathcal{B}(\mathbb{T}(V))$. We require that each ρ is a natural transformation, which means that is should satisfy:

$$
\begin{array}{l}
\mathit{rho\_natural} : \textbf{lemma} \; \forall \, (f : [X \to Y]) : \\
\quad \rho \circ \mathsf{map}_{\Sigma} \, (\mathsf{map}_{\mathcal{D}} \, (f)) = \mathsf{map}_{\mathcal{B}} \, (\mathsf{map}_{\mathbb{T}} \, (f)) \circ \rho
\end{array}
$$

Before giving a PVS specification of ρ for the complete example language, we have to remember that the semantic domain (being the greatest fixpoint $\nu\mathcal{D}$ of functor $\mathcal{D}$)[5], cannot be expressed in terms of $\mathcal{D}$. Again, we will define $\nu\mathcal{D}$ as a coinductive data type, and let PVS generate the corresponding functor $\mathcal{D}$[6].

$$
\begin{array}{l}
\nu\mathcal{D} \, [A : \textbf{type}] : \textbf{codatatype begin} \\
\quad \mathit{dz\_in} \, (\mathit{left} : \nu\mathcal{D}, \mathit{right} : [A, \nu\mathcal{D}]) : \mathit{dz\_in} \, ? \\
\textbf{end} \; \nu\mathcal{D} \\
\mathit{Coalgebras} \, [A, X : \textbf{type}] : \textbf{theory begin} \\
\quad \textbf{importing} \; \nu\mathcal{D} \, [A] \\
\quad \mathcal{D} \quad : \textbf{type} = \mathit{DZ\_struct} \, [A, X] \\
\quad \mathcal{B} \quad : \textbf{type} = [A, X] \\
\quad \mathit{CoAlg} : \textbf{type} = [X \to \mathcal{D}] \\
\quad \mathsf{out}_{\nu\mathcal{D}} \, (nd : \nu\mathcal{D}) : \mathcal{D}(\nu\mathcal{D}) = \mathit{inj\_dz\_in} \, (\mathit{left} \, (nd), \mathit{right} \, (dn)) \\
\quad \mathsf{unfold} \, (c : \mathit{CoAlg}) \, (z : X) : \nu\mathcal{D} = \mathit{coreduce} \, (c) \, (z) \\
\textbf{end} \; \mathit{Coalgebras} \\
\mathit{DMap} \, [A, X, Y : \textbf{type}] : \textbf{theory begin} \\
\quad \textbf{importing} \; \mathit{Coalgebras} \, [A, X] \\
\quad \textbf{importing} \; \mathit{Coalgebras} \, [A, Y] \\
\quad \mathsf{map}_{\mathcal{B}} \, (f : [X \to Y]) \, (\mathit{fb} : \mathcal{B} \, [A, X]) : \mathcal{B} \, [A, Y] = \\
\qquad (\mathit{fb}\text{`}1, f \, (\mathit{fb}\text{`}2)) \\
\quad \mathsf{map}_{\mathcal{D}} \, (f : [X \to Y]) \, (\mathit{fd} : \mathcal{D} \, [A, X]) : \mathcal{D} \, [A, Y] = \\
\qquad \mathit{inj\_dz\_in} \, (f \, (\mathit{inj\_left} \, (\mathit{fd})), \mathsf{map}_{\mathcal{B}} \, (f) \, (\mathit{inj\_right} \, (\mathit{fd}))) \\
\textbf{end} \; \mathit{DMap}
\end{array}
$$

The following property, introduced by [19] as the "fusion law for anamorphisms", holds:

[5] In this paper we deviate slightly from other approaches which use $\nu\mathcal{B}$ as domain. This is not an essential difference since one can easily show that $\nu\mathcal{B}$ and $\nu\mathcal{D}$ are isomorphic.

[6] We had to 'inline' the definition of the behaviour functor $\mathcal{B}$ in $\nu\mathcal{D}$ in order to get $\nu\mathcal{D}$ accepted by PVS. This explains the tuple type $[A, \nu\mathcal{D}]$ in the right component of $\nu\mathcal{D}$.

```
importing Coalgebras [A, X]
importing Coalgebras [A, Y]
unfold_fusion : lemma
  ∀ (f : [X → Y], c : CoAlg [A, X], d : CoAlg [A, Y]) :
    map_D (f) ∘ c = d ∘ f ⇒ unfold (c) = unfold (d) ∘ f
```

We define ρ as follows:

```
ρ (sf : Σ(D(V))) : B(T(V)) =
  cases sf‘1 of
    A   : (AL, tapp (A, λ (i : below (0)) : tvar (inj_right (sf‘2 (i))‘2))),
    B   : (BL, tapp (B, λ (i : below (0)) : tvar (inj_right (sf‘2 (i))‘2))),
    Alt : let a0 = sf‘2 (0), a1 = sf‘2 (1)
          in (inj_right (a0)‘1, tapp (Alt, λ (i : below (2)) :
              if i = 0 then tvar (inj_right (a1)‘2) else tvar (inj_right (a0)‘2) endif)),
    Zip : let a0 = sf‘2 (0), a1 = sf‘2 (1)
          in (inj_right (a0)‘1, tapp (Zip, λ (i : below (2)) :
              if i = 0 then tvar (inj_left (a1)) else tvar (inj_right (a0)‘2) endif))
  endcases
```

Proving that ρ is indeed a natural transformation is easy, one can simply use PVS's *grind* strategy.

As with the plain format, naturality of ρ appears to be the only property needed to prove the adequacy theorem. In PVS we can exploit this fact by using an axiomatic speciation of ρ. More specifically, ρ itself is redefined as an *uninterpreted function* (by omitting the body) whereas ρ's naturality is expressed as an axiom. This will prevent unindented use of ρ's actual implementation while constructing proofs.

The symmetry of the codomain and domain of Λ appears to be important in the adequacy proof. In order to obtain a distributive law of $\mathbb{T}$ over $\mathcal{D}$, ρ needs to undergo a two-step transformation. Expanding ρ's codomain is the first step, using an auxiliary function $to\mathcal{D}$:

```
toD (f : [X → T(V)], g : [X → B(T(V))]) (x : X) : D(T(V))
   = inj_dz_in (f (x), g (x))
τ : [Σ(D(V)) → D(T(V))] = toD (tapp ∘ map_Σ (tvar ∘ inj_left), ρ)
```

Adjusting the codomain is slightly more involved, and requires an appropriate use of $\mathsf{fold}_{\mathbb{T}}$:

```
Λ : [T(D(V)) → D(T(V))] = fold_T (map_D (tvar), map_D (join_T) ∘ τ)
```

This construction does not affect the naturality property, as stated by the following lemma:

```
law_natural : lemma ∀ (f : [X → Y]) :
  Λ ∘ map_T (map_D (f)) = map_T (map_T (f)) ∘ Λ
```

The proof of this lemma proceeds with structural induction on $\mathbb{T}(\mathcal{D}(V))$, using naturality of ρ, and a well-known fact on $\mathsf{join}_{\mathbb{T}}$, namely $\mathsf{join}_{\mathbb{T}} \circ \mathsf{map}_{\mathbb{T}}\,(\mathsf{map}_{\mathbb{T}}\,(f)) =$

$\mathsf{map}_{\mathbb{T}}\ (f)\ \circ\ \mathsf{join}_{\mathbb{T}}$. From Λ we derive both om and dm, in a similar manner to the plain format, as such we write om as a $\mathsf{fold}_{\mathbb{T}}$, and dually, dm as an unfold. As to om, recall that the algebra corresponding to the second argument of $\mathsf{fold}_{\mathbb{T}}$ will have type $\Sigma(\mathcal{D}(\mathbb{T}(\mathsf{V}))) \to \mathcal{D}(\mathbb{T}(V))$. The following diagram shows how such an algebra is obtained from Λ. Again both domain and codomain must be adjusted. To enhance legibility, we have omitted the brackets in the types as well as the type variable V:

$$\Sigma\ \mathcal{D}\ \mathbb{T} \xrightarrow{\Sigma\ (tvar)} \Sigma\ \mathbb{T}\ \mathcal{D}\ \mathbb{T} \xrightarrow{tapp} \mathbb{T}\ \mathcal{D}\ \mathbb{T} \xrightarrow{\Lambda} \mathcal{D}\ \mathbb{T}\ \mathbb{T} \xrightarrow{\mathcal{D}\ (\mathsf{join}_{\mathbb{T}})} \mathcal{D}\ \mathbb{T}$$

For the first argument of $\mathsf{fold}_{\mathbb{T}}$ is suffices to adjust Γ's domain only. This can be done with a simple map leading to

$$\begin{array}{l}\mathsf{om}\ (\Gamma : [V \to \mathcal{D}(V)]) : [\mathbb{T}(\mathsf{V}) \to \mathcal{D}(\mathbb{T}(V))] = \\ \quad \mathsf{fold}_{\mathbb{T}}\ (\mathsf{map}_{\mathcal{D}}\ (tvar)\ \circ\ \Gamma, \mathsf{map}_{\mathcal{D}}\ (\mathsf{join}_{\mathbb{T}})\ \circ\ \Lambda\ \circ\ tapp\ \circ\ \mathsf{map}_{\Sigma}\ (tvar))\end{array}$$

Obtaining dm from Λ is less difficult. The coalgebra given as argument to unfold has type $\mathbb{T}(\nu\mathcal{D}) \to \mathcal{D}(\mathbb{T}(\nu\mathcal{D}))$, which almost coincides with the type of Λ, after $\nu\mathcal{D}$ is substituted for V. Only a small adjustment of the codomain to expand $\nu\mathcal{D}$ to $\mathcal{D}(\nu\mathcal{D})$ (in $\mathbb{T}(\nu\mathcal{D})$) is necessary. This is done below.

$$\mathsf{dm} : [\mathbb{T}(\nu\mathcal{D}) \to \nu\mathcal{D}] = \mathsf{unfold}\ (\Lambda\ \circ\ \mathsf{map}_{\mathbb{T}}\ (\mathsf{out}_{\nu\mathcal{D}}))$$

The functions run and eval remain the same, bringing us to the main result of this paper: the adequacy theorem.

$$\mathit{run\_is\_eval} : \textbf{theorem}\ \forall\ (\Gamma : [V \to \mathcal{D}(V)]) : \mathsf{run}\ (\Gamma) = \mathsf{eval}\ (\Gamma)$$

The proof of this theorem is done by coinduction on the semantic domain $\nu\mathcal{D}$. The coinduction principle in Pvs requires the construction of a proper *bisimulation relation*; e.g. see [7]. This principle is based on the fact that if two streams are bisimilar, then they are equal. The definition of such a bisimulation is in our case reasonably straightforward. However, proving that it indeed fulfills the bisimulation criteria is more complicated. The crux of the proof is based on the following commutation property:

$$\begin{array}{ccc} \mathbb{T}(V) & \xrightarrow{\mathsf{om}} & \mathcal{D}(\mathbb{T}(V)) \\ {\scriptstyle \mathsf{eval}\ (\Gamma)}\downarrow & & \downarrow{\scriptstyle \mathcal{D}\ (\mathsf{eval}\ (\Gamma))} \\ \nu\mathcal{D} & \xrightarrow[\mathsf{out}_{\nu\mathcal{D}}]{} & \mathcal{D}(\nu\mathcal{D}) \end{array}$$

For the proof of this property we use of the alternative proof principle for terms (Lemma 1) using dm as a $\mathbb{T}$Monad, and thus we need to verify the following fact:

$$\mathit{dm\_is\_T\_algebra} : \textbf{lemma}\ \mathbb{T}\mathsf{Alg}\ ?\ (\mathsf{dm})$$

This property consists of two parts. The proof of the first part (bij coinduction on $\nu\mathcal{D}$) is straightforward. The proof of the second part (stating that $\mathsf{dm}\ \circ\ \mathsf{map}_{\mathbb{T}}\ (\mathsf{dm}) = \mathsf{dm}\ \circ\ \mathsf{join}_{\mathbb{T}}$) is more subtle. The key to this proof is the *unfold_fusion* lemma.

6 PVS formalization

One of our motivations for developing this formalization was to investigate whether or not implementing abstract categorical concepts in PVS is feasible, and furthermore to reason about these concepts. 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 fully polymorphic, and equipped with a rich type class system offering type classes as first class citizens. In COQ, functors, monads, and (co)algebras can be naturally represented. PVS merely offers a very rudimentary form of polymorphism via parameterized theories. Using these parameterized theories as a substitute for COQ's type classes is definitely a setback. Nevertheless, for the most part this aspect of our formalization does not hamper the proving process. There were no fundamental problems which cannot be solved 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). Moreover, the type classes used in specifications or in proofs (the original description of [9] contains numerous occurrences of these) are resolved automatically during type checking. In fact, the user never has to bother about which overloaded instance is actually used, which makes programming/proving much easier. Unfortunately, PVS lacks the ability for resolving theory instantiations automatically. As a consequence, the user has to do this by hand. However, even after all overloading has been removed manually, one can still get stuck in seemingly unsolvable cases, especially when constructing a proof. In the latter case, each formulae is usually displayed in a legible manner, by omitting the (sometimes lengthy) type information about symbols. However, this often obstructs equational reasoning; one cannot simply replace one formulae by another equivalent formulae just by copying the text as it appears in the proof sequent. The type checker is often not capable of determining the correct instance type, although this should be evident from the context. The prover command `show-expanded-sequent` reveals relevant context information about symbols appearing in the proof sequent, but the use of this information often leads to code explosion even for relatively simple expressions, making proofs lengthy and very hard to read.

7 Related work

This work was inspired by our earlier work on modularity, the formalization Modular Structural Operational Semantics (MSOS) [18]. The present paper can be seen as a contribution to the so-called field of bialgebraic semantics, starting with Turi and Plotkin's research [22], 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 was obtained, and moreover the adequacy of these models

could be proven. In [10] an introduction is given to the basics of bialgebras for operational semantics that was used in the present formalization. The author of [10] 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 [5]. The authors show that name binding fits in the abstract GSOS format. This was refined further in [6].

In [4] 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 [19] offering an alternative to structural induction.

Another interesting line of related work on interpreters, is that of the the application of monads in order to structure semantics. Liang et al. [15] introduced monad transformers to compose multiple monads and build modular interpreters. Jaskelioff et al. use [9] 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 Haskell example in Section 2 is inspired by this work. Although, [9] strictly follows the approach of Turi and Plotkin, there is no formal evidence that their construction is correct. The latter issue is addressed by recent work of [8] who introduce modular proof techniques for equational reasoning about monads.

8 Conclusions

We presented a formalization in Pvs of Turi and Plotkin's work based on category theory. This 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.

References

1. P. Achten, M. van Eekelen, M. de Mol, and R. Plasmeijer. Editorarrow: An arrow-based model for editor-based programming. *Journal of Functional Programming*, 23:185–224, 2 2013.
2. F. Bartels. *On generalised coinduction and probabilistic specification formats.* PhD thesis, CWI, Amsterdam, April 2004.
3. B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced. *J. ACM*, 42(1):232–268, Jan. 1995.
4. B. Delaware, B. C. Oliveira, and T. 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.

5. M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, LICS '99, pages 193–, Washington, DC, USA, 1999. IEEE Computer Society.
6. M. Fiore and S. 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.
7. U. Hensel and B. Jacobs. Coalgebraic theories of sequences in pvs. *J. Log. Comput.*, 9(4):463–500, 1999.
8. R. Hinze and D. W. James. Proving the unique fixed-point principle correct: an adventure with category theory. *SIGPLAN Not.*, 46(9):359–371, Sept. 2011.
9. M. Jaskelioff, N. Ghani, and G. Hutton. Modularity and implementation of mathematical operational semantics. *Electron. Notes Theor. Comput. Sci.*, 229(5):75–95, Mar. 2011.
10. B. Klin. Bialgebras for structural operational semantics: An introduction. *Theoretical Computer Science*, 412(38):5043–5069, 2011. CMCS Tenth Anniversary Meeting.
11. P. W. M. Koopman, R. Plasmeijer, and P. Achten. An executable and testable semantics for itasks. In S.-B. Scholz and O. Chitil, editors, *IFL*, volume 5836 of *Lecture Notes in Computer Science*, pages 212–232. Springer, 2008.
12. P. W. M. Koopman, R. Plasmeijer, and P. Achten. An effective methodology for defining consistent semantics of complex systems. In Z. Horváth, R. Plasmeijer, and V. Zsók, editors, *CEFP*, volume 6299 of *Lecture Notes in Computer Science*, pages 224–267. Springer, 2009.
13. P. W. M. Koopman, M. C. J. D. van Eekelen, and M. J. Plasmeijer. Operational machine specification in a functional programming language. *Softw., Pract. Exper.*, 25(5):463–499, 1995.
14. J. Launchbury. A natural semantics for lazy evaluation. In M. S. V. Deusen and B. Lang, editors, *POPL*, pages 144–154. ACM Press, 1993.
15. S. Liang, P. Hudak, and M. 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.
16. S. Mac Lane. *Categories for the Working Mathematician.* Number 5 in Graduate Texts in Mathematics. Springer-Verlag, 1971.
17. K. Madlener and S. Smetsers. Gsos formalized in coq. In *The 7th International Symposium on Theoretical Aspects of Software Engineering (TASE2013)*, 2013. Birmingham, UK, 2013. IEEE. To appear.
18. K. Madlener, S. Smetsers, and M. C. J. D. van Eekelen. Formal component-based semantics. In M. A. Reniers and P. Sobocinski, editors, *SOS*, volume 62 of *EPTCS*, pages 17–29, 2011.
19. E. Meijer, M. 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, London, UK, UK, 1991. Springer-Verlag.
20. H. R. Nielson and F. Nielson. *Semantics with applications: a formal introduction.* John Wiley & Sons, Inc., New York, NY, USA, 1992.
21. R. Plasmeijer, P. Achten, and P. W. M. Koopman. itasks: executable specifications of interactive work flow systems for the web. In R. Hinze and N. Ramsey, editors, *ICFP*, pages 141–152. ACM, 2007.
22. D. Turi and G. Plotkin. Towards a mathematical operational semantics. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science*, LICS '97, pages 280–, Washington, DC, USA, 1997. IEEE Computer Society.
23. V. Zsók, P. W. M. Koopman, and R. Plasmeijer. Generic executable semantics for d-clean. *Electr. Notes Theor. Comput. Sci.*, 279(3):85–95, 2011.