Jaskelioff, Mauro Javier and Ghani, Neil and Hutton, Graham (2008) Modularity and implementation of mathematical operational semantics. In: Second Workshop on Mathematically Structured Functional Programming (MSFP 2008), 6 July 2008, Reykjavik, Iceland.

Modularity and Implementation of Mathematical Operational Semantics

Mauro Jaskelioff[1] Neil Ghani[2] Graham Hutton[3]

School of Computer Science
University of Nottingham, UK

Abstract

Structural operational semantics is a popular technique for specifying the meaning of programs by means of inductive clauses. One seeks syntactic restrictions on those clauses so that the resulting operational semantics is well-behaved. This approach is simple and concrete but it has some drawbacks. Turi pioneered a more abstract categorical treatment based upon the idea that operational semantics is essentially a distribution of syntax over behaviour. In this article we take Turi's approach in two new directions. Firstly, we show how to write operational semantics as modular components and how to combine such components to specify complete languages. Secondly, we show how the categorical nature of Turi's operational semantics makes it ideal for implementation in a functional programming language such as Haskell.

Keywords: Modularity, Category Theory, Operational Semantics, Haskell

1 Introduction

Operational semantics is one of the primary techniques for formally specifying the meaning of programs. Traditionally, one defines the operational semantics of a programming language as a relation over the syntax of programs. In structural operational semantics [19], this relation is defined by a set of inductive rules, and one seeks syntactic restrictions on these rules so that the resulting operational semantics has certain desirable properties. However, despite the relative simplicity of this syntactic approach to operational semantics, it has a number of significant drawbacks:

- Being syntactic, the restrictions on rules are often rather intricate, and it is not clear how they arise. This intricacy makes proving meta-theoretic results difficult, and makes it virtually impossible to see how these results are affected by changes to the language under consideration, or to the notion of observable behaviour

[1] Email: mjj@cs.nott.ac.uk
[2] Email: nxg@cs.nott.ac.uk
[3] Email: gmh@cs.nott.ac.uk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

of the semantics being defined. One gets the feeling that there is some deeper mathematical structure at play that is being obscured by the syntactic clutter.

- Being syntactic, operational semantics presented in this form is language-specific. As a result, it is difficult to implement a generic notion of operational semantics in a high-level programming language such as Haskell. Compare, for example, with the use of logical frameworks to implement logics and type theories; such language-independent frameworks are clearly missing for operational semantics. One would like to be able to write data types whose inhabitants are operational semantics, and programs which manipulate such data types.
- Being syntactic, it is not clear how to relate operational semantics to the more abstract and high-level denotational semantics. This is more than just a mathematical irritation: by utilising concepts from category theory, denotational semantics has given us many language-independent mathematical tools to structure programs. These include, for example, monads, initial algebra semantics and Kan extensions. The fact that these tools are language independent suggests that they somehow get at the essence of computation. The same cannot currently be said of operational semantics, with its inherently language-dependent flavour.

Overall, we are left with the feeling that we need to get at the mathematical essence of operational semantics, to allow mathematical tools to be used to structure and reason about operational semantics in a high-level manner, and to make it easier to relate it to denotational semantics. Indeed, at this point, we can begin to wonder if we really need operational and denotational semantics. A categorical semantics encompassing both approaches may actually be what we are striving for.

But the development of such an approach has already begun! Ten years ago, Daniele Turi did something rather remarkable. He abstracted from the concrete, syntactic, language-dependent approach to operational semantics and proclaimed that operational semantics was a categorical construct, namely a distributive law between syntax and behaviour. In one fell swoop [24], Turi opened the way to tackling all of the above problems. By parameterising his treatment by a functor representing syntax and a functor representing semantics, he abstracted away from the specific details of particular languages and their meaning. Moreover, it became possible to relate the operational and denotational approaches; indeed, they become two sides of the same coin, as they define the same semantic function, one by the universal property of the final coalgebra, the other by the universal property of the initial algebra. That is, the semantic function $[\![-]\!]\colon \mu\Sigma \to \nu B$ from the syntax into the behaviour, is induced by both an algebra over the final coalgebra νB and a coalgebra over the initial algebra $\mu\Sigma$. These are provably equal.

We further develop of Turi's categorical approach to operational semantics by addressing the questions of modularity and implementation. More precisely, the article makes the following contributions:

- We develop a modular operational semantics, by structuring Turi's primitive concept of behaviour. We show how to write modular semantic components and how to construct languages built from these components. We stress that a general approach to modularity in Turi's mathematical operational semantics has not been considered before (but see the related work section at the end of the article.)

- We implement our ideas in Haskell, which helps to bring Turi's categorical work to the functional programming community in a more accessible way, makes the ideas directly executable, facilitates experimentation, and allows us to benefit from Haskell's well-developed support for monadic programming. We stress that the question of how to implement Turi's mathematical operational semantics has also not been considered. This is no trivial task as there a variety of implementation issues which must be addressed so that the resulting code retains the elegance and simplicity of the category theory which inspires it.

As with any work that involves implementing mathematics, it is important to be clear about the relationship between the mathematical theory and its concrete implementation. We use categorical tools as our main technical devices, and use Haskell to illustrate and make these categorical techniques more accessible to the programming languages community. For the purposes of this article, it suffices to work in the category Set of sets and total functions, but the reader should bear in mind that the mathematical theory generalises to other categories. For example, Haskell is not based on Set but on the category CPO, which has considerable extra structure which admits partial functions. Because we do not rely on the extra structure, and because we can use Haskell to reason about its total fragment [2], Haskell provides a convenient syntax for programming in Set. Some languages require the framework to be interpreted in CPO-like categories [12], in particular for dealing with general recursion, but for all the examples we consider, the structure of Set is enough.

The article is aimed at functional programmers with a basic knowledge of category theory and semantics, but we do not assume prior knowledge of Turi's categorical approach to semantics. The Haskell code from the article is available from the authors' websites.

2 Structural Operational Semantics

Operational semantics gives meaning to terms in a language by defining a transition relation that captures execution steps in an abstract machine. Reasoning about this relation can be difficult. Therefore Plotkin proposed *structural operational semantics* (SOS), in which the transition relation is defined by structural recursion on syntax-directed rules [19]. One then uses the principle of structural induction to reason about the induced transition relation.

Example 2.1 Consider a simple process language P whose terms $p \in P$ are specified by the following grammar, which corresponds to Basic Process Algebra [1]:

$$p \quad ::= \quad !_a \mid p\,;p \mid p \sqcup p$$

The informal meaning of the operators in the language is that $!_a$ prints the character a on the screen, $p\,;q$ sequences the execution of p and q, and $p \sqcup q$ nondeterministically chooses to execute either p or q.

We give an operational semantics for P by the set of structural rules in Figure 1. The rules recursively define relations $\rightarrow \subseteq P \times A \times P$ and $\rightarrow\!\!\checkmark \subseteq P \times A$, on terms P and set of characters A. We write $p \xrightarrow{a} p'$ for $(p, a, p') \in \rightarrow$ and $p \xrightarrow{a} \checkmark$ for

$$\frac{}{!_a \xrightarrow{a} \checkmark} \qquad \frac{p \xrightarrow{a} p'}{p\,;q \xrightarrow{a} p'\,;q} \qquad \frac{p \xrightarrow{a} \checkmark}{p\,;q \xrightarrow{a} q}$$

$$\frac{p \xrightarrow{a} p'}{p \sqcup q \xrightarrow{a} p'} \qquad \frac{q \xrightarrow{a} q'}{p \sqcup q \xrightarrow{a} q'} \qquad \frac{p \xrightarrow{a} \checkmark}{(p \sqcup q) \xrightarrow{a} \checkmark} \qquad \frac{q \xrightarrow{a} \checkmark}{(p \sqcup q) \xrightarrow{a} \checkmark}$$

Fig. 1: Structural operational semantics for P

$$\frac{}{n \Downarrow n} \qquad \frac{t \Downarrow n \quad u \Downarrow m}{t + u \Downarrow n + m} \qquad \frac{c \Downarrow 0 \quad t \Downarrow n}{ifz\ c\ t\ e \Downarrow n} \qquad \frac{c \Downarrow n \quad n \neq 0 \quad e \Downarrow m}{ifz\ c\ t\ e \Downarrow m}$$

Fig. 2: Structural operational semantics for Z

$$\frac{}{throw \uparrow} \qquad \frac{t \uparrow \quad u \uparrow}{(catch\ t\ u) \uparrow}$$

Fig. 3: Structural operational semantics for E

$(p, a) \in \rightarrow\checkmark$. Intuitively, the transition $p \xrightarrow{a} p'$ represents a term p which can evolve into term p' by printing the character a on the screen, whereas $p \xrightarrow{a} \checkmark$ holds for terms which can terminate successfully by printing character a.

Example 2.2 We define a simple language of arithmetic expressions, with integers, additions and a conditional expression, whose terms $z \in Z$ are specified by the following grammar:

$$z \quad ::= \quad \mathbb{Z} \mid \quad z + z \quad \mid \quad ifz\ z\ z\ z$$

The informal meaning of $ifz\ c\ t\ e$ is that if c is 0 then t is evaluated, otherwise e is evaluated. We give an operational semantics for this language in Figure 2. The rules recursively define a relation $\Downarrow \subseteq Z \times \mathbb{Z}$, where we write $t \Downarrow n$ for $(t, n) \in \Downarrow$. Intuitively, $t \Downarrow n$ means that term t can evaluate to integer n.

Note that the semantics were given in a small-step style for P and big-step style for Z. However, the mathematical approach to operational semantics that we use in this article treats these two different styles uniformly.

Example 2.3 Let us consider now a language E of exceptions:

$$e \quad ::= \quad throw \quad \mid \quad catch\ e\ e$$

The informal meaning is that $throw$ throws an exception and $catch\ t\ u$ evaluates t and, if t throws an exception, recovers from it by evaluating u. Its operational semantics is given by the rules in Figure 3 and define a predicate $\uparrow \subseteq E$, where we write $e\uparrow$ for $e \in \uparrow$. Intuitively, $e\uparrow$ means that e can throw an exception.

This language is not very useful by itself as the only possible outcome is to throw an exception. Its real utility is exhibited when one considers the language E together with some other language. For example, we will consider combining E with P and Z. However, to put the languages together we need to add extra rules explaining how $catch$ deals with the transitions defined by the other languages and how the operators in other languages deal with exceptions. In Figures 4 and 5, we show all the rules that need to be added to combine E with P and Z.

$$\frac{p \xrightarrow{a} p'}{catch\ p\ q \xrightarrow{a} catch\ p'\ q} \qquad \frac{p \xrightarrow{a} \checkmark}{(catch\ p\ q) \xrightarrow{a} \checkmark} \qquad \frac{p\uparrow \quad q \xrightarrow{a} q'}{catch\ p\ q \xrightarrow{a} q'}$$

$$\frac{p\uparrow \quad q \xrightarrow{a} \checkmark}{(catch\ p\ q) \xrightarrow{a} \checkmark} \qquad \frac{p\uparrow}{(p\,;q)\uparrow} \qquad \frac{p\uparrow}{(p \sqcup q)\uparrow} \qquad \frac{q\uparrow}{(p \sqcup q)\uparrow}$$

Fig. 4: Additional rules for combining P and E

$$\frac{t \Downarrow n}{catch\ t\ u \Downarrow n} \qquad \frac{t\uparrow \quad u \Downarrow n}{catch\ t\ u \Downarrow n} \qquad \frac{t\uparrow}{t+u\uparrow} \qquad \frac{u\uparrow}{t+u\uparrow}$$

$$\frac{c\uparrow}{(ifz\ c\ t\ e)\uparrow} \qquad \frac{c \Downarrow 0 \quad t\uparrow}{(ifz\ c\ t\ e)\uparrow} \qquad \frac{c \Downarrow z \quad z \neq 0 \quad e\uparrow}{(ifz\ c\ t\ e)\uparrow}$$

Fig. 5: Additional rules for combining Z and E

More generally, combining operational semantics is not just a matter of the tedious and error-prone task of adding extra syntactic rules, but may also involve modifying the original rules, which makes it difficult to formally relate the original and combined languages. The underlying problem is that SOS lacks a language-independent theory that would clarify what combining languages means in general, rather than for specific rules.

3 Modular Syntax

The first step towards obtaining modular operational semantics is to obtain modular syntax, in the sense that terms of a language are constructed by combining smaller languages. It is straightforward to implement the grammar for P as a recursive datatype:

```
data P = Put A | Seq P P | Alt P P
type A = Char
```

However, datatype P is monolithic. In order to obtain modular syntax we need to reveal the underlying structure, separating the operators of the language from the description of its terms. We use the standard categorical technique (for example, see [5]) of modelling terms by the free monad over a signature, and the combination of languages by the coproduct of free monads.

3.1 Terms as Free Monads

We will specify the syntax of a language by its *signature*, that is, the set of its operators and their corresponding arities. Each signature has a corresponding instance of the Functor class (see Appendix B), which we call a *signature functor*.

Example 3.1 The signature functor for P is as follows:

```
data P a = Put A | Seq a a | Alt a a
instance Functor P where
  fmap _ (Put c)   = Put c
```

$$\begin{aligned}\mathsf{fmap}\ f\ (\mathsf{Seq}\ p\ q) &= \mathsf{Seq}\ (f\ p)\ (f\ q)\\ \mathsf{fmap}\ f\ (\mathsf{Alt}\ p\ q) &= \mathsf{Alt}\ \ (f\ p)\ (f\ q)\end{aligned}$$

Terms constructed with operators from the signature functor f and with variables of type x are given by the free monad on f at x, represented by the datatype $\mathsf{Term}\ f\ x$. The flexibility of having variables of an arbitrary type will be used later on to represent the meta-variables in operational rules.

$$\textbf{data}\ \mathsf{Term}\ f\ x = \mathsf{Var}\ x \mid \mathsf{Con}\ (f\ (\mathsf{Term}\ f\ x))$$

That is, a term is either a variable or an operator from f applied to a term. It is now straightforward to make such terms into both Functors and Monads:

$$\begin{aligned}&\textbf{instance}\ \mathsf{Functor}\ f \Rightarrow \mathsf{Functor}\ (\mathsf{Term}\ f)\ \textbf{where}\\ &\quad \mathsf{fmap}\ f\ (\mathsf{Var}\ x) \ \ = \mathsf{Var}\ (f\ x)\\ &\quad \mathsf{fmap}\ f\ (\mathsf{Con}\ t) \ \ = \mathsf{Con}\ (\mathsf{fmap}\ (\mathsf{fmap}\ f)\ t)\\ &\textbf{instance}\ \mathsf{Functor}\ f \Rightarrow \mathsf{Monad}\ \ (\mathsf{Term}\ f)\ \textbf{where}\\ &\quad \mathsf{return} \qquad\quad = \mathsf{Var}\\ &\quad (\mathsf{Var}\ x) \gg\!= f \quad = f\ x\\ &\quad (\mathsf{Con}\ t) \gg\!= f \quad = \mathsf{Con}\ (\mathsf{fmap}\ (\gg\!= f)\ t)\end{aligned}$$

We will not use the fact that $\mathsf{Term}\ f$ is a monad in this article, but we mention it because it shows that terms structured in this way come equipped with a substitution operator, as given by $(\gg\!=) :: \mathsf{Term}\ f\ a \to (a \to \mathsf{Term}\ f\ b) \to \mathsf{Term}\ f\ b$ [6,15]. With this representation of terms, the natural manner in which to process terms is using a generic fold operator [16,8]:

$$\begin{aligned}&\mathsf{foldTerm} :: \mathsf{Functor}\ f \Rightarrow (a \to b) \to (f\ b \to b) \to \mathsf{Term}\ f\ a \to b\\ &\mathsf{foldTerm}\ \mathit{var}\ \_\ (\mathsf{Var}\ a) \qquad = \mathit{var}\ a\\ &\mathsf{foldTerm}\ \mathit{var}\ \mathit{con}\ (\mathsf{Con}\ \mathit{fta}) = \mathit{con}\ (\mathsf{fmap}\ (\mathsf{foldTerm}\ \mathit{var}\ \mathit{con})\ \mathit{fta})\end{aligned}$$

Intuitively, the argument of type $a \to b$ is used to process variables, and the argument of type $f\ b \to b$ (an f-algebra) is used to process operators.

Finally, the *programs* of a language are its closed terms. That is, programs are terms with variables taken from the empty datatype Zero, which comes equipped with a canonical map $\mathsf{empty} :: \mathsf{Zero} \to a$ into any other type a.

$$\textbf{type}\ \mathsf{Program}\ f = \mathsf{Term}\ f\ \mathsf{Zero}$$

Thus, we have a generic notion of syntax equipped with well-behaved substitution and a well-behaved recursion operator. Moreover, as shown in the next section, we obtain a simple and principled method for combining the syntax of languages.

3.2 Coproducts of Free Monads

We have shown that signatures define the operators of a language. In order to obtain modular syntax we will combine the signatures of small languages to obtain a signature for the complete language.

The natural way to combine two languages is to take the coproduct of the free monads modelling them. Since free constructions preserve coproducts, this is equivalent to the free monad on the coproduct of their signature functors.

```
prog      :: Program P
prog      = (put 'a' `seq` put 'b') ⊔ (put 'c')
put       :: Char → Term P a
put c     = Con (Inl (Put c))
seq, ·⊔· :: Term P a → Term P a → Term P a
seq p q   = Con (Inr (Inl (Seq p q)))
p ⊔ q     = Con (Inr (Inr (Alt p q)))
```

Fig. 6: The term $(!_a ; !_b) \sqcup !_c$, as a Program of signature P

```
data (f ⊕ g) a      = Inl (f a) | Inr (g a)
instance (Functor f, Functor g) ⇒ Functor (f ⊕ g) where
  fmap h (Inl fx)   = Inl (fmap h fx)
  fmap h (Inr gx)   = Inr (fmap h gx)
copair              :: (f a → b) → (g a → b) → (f ⊕ g) a → b
copair f _ (Inl fa) = f fa
copair _ g (Inr ga) = g ga
```

The function copair processes the coproduct of functors f and g, given that we provide two functions: one to process f and the other to process g. We can use copair to define an $f \oplus g$-algebra from an f-algebra and a g-algebra. Therefore, foldTerm can be used to process $f \oplus g$ terms.

Example 3.2 We rewrite P as the coproduct of the signatures of its operators:

```
type P      = Put ⊕ Seq ⊕ Alt
data Put a  = Put Char
data Seq a  = Seq a a
data Alt  a = Alt  a a
```

The P-term $(!_a ; !_b) \sqcup !_c$ is written in Haskell as the program *prog* in Figure 6.

Example 3.3 In a similar manner, we rewrite the operators of Z and E as separate languages. Again, the syntax of the complete languages can be recovered by the coproduct of their operators.

```
data N a    = N Int
data Add a  = Add a a
data Ifz a  = Ifz a a a

type Z = N ⊕ Add ⊕ Ifz
```

```
data Thr a = Thr
data Cat a = Cat a a

type E = Thr ⊕ Cat
```

Note that now we may define the syntax of new languages simply by choosing which constructs we would like to have. For example, we may define the syntax EP of P extended with exceptions, EZ of Z extended with exceptions, or a non-deterministic arithmetic language NDZ:

```
type EP  = Thr ⊕ Cat ⊕ Put ⊕ Seq ⊕ Alt
type EZ  = Thr ⊕ Cat ⊕ N ⊕ Add ⊕ Ifz
type NDZ = N ⊕ Add ⊕ Alt
```

Coproducts provide a structured, mathematical foundation for assembling syntax. There are, however, some practical concerns. As shown in Figure 6, we had to define auxiliary functions *put*, *seq*, and $\cdot \sqcup \cdot$ to make the definition of programs less cumbersome. These shorthands will only work for terms of signature P, and would need to be changed should the language be extended. For instance, if we were working with the language EP, then we would have to define a new auxiliary function put' as:

$$
\begin{aligned}
&put' \quad :: \mathsf{Char} \to \mathsf{Term}\ \mathsf{EP}\ a \\
&put'\ c = \mathsf{Con}\ (\mathsf{Inr}\ (\mathsf{Inr}\ (\mathsf{Inl}\ (\mathsf{Put}\ c))))
\end{aligned}
$$

Redefining these auxiliary functions every time we change our language is inherently non-modular. In the following subsection we show how to solve this problem.

3.3 Automatic Injections and Projections

Consider a function which produces terms of signature G. If $G = F_i$, this function can be easily extended to produce terms of signature $\Sigma = F_1 \oplus \ldots \oplus F_n$ by post-composing it with $\mathsf{foldTerm\ Var}\ (\mathsf{in}_i \cdot \mathsf{Con})$, where in_i is the corresponding injection into the coproduct. We would like this extension to work on terms of any signature containing G, but this is not the case here: G will not be at a fixed position i for every signature containing G. In general, we would like to be able to define functions on coproducts of datatypes for which only a limited part is known.

The solution to this problem [14,21] is to parameterise each function by injection/projection pairs corresponding to each of the summands a function is interested in. We can avoid writing this parameterisation with the following type class:

$$
\begin{aligned}
&\textbf{class}\ (\mathsf{Functor}\ sub, \mathsf{Functor}\ sup) \Rightarrow sub \hookrightarrow sup\ \textbf{where} \\
&\quad \mathsf{inj} :: sub\ a \to sup\ a \\
&\quad \mathsf{prj} :: sup\ a \to \mathsf{Maybe}\ (sub\ a)
\end{aligned}
$$

We can think of $sub \hookrightarrow sup$ as meaning "*sub* is a subtype of *sup*". The class method inj is used to inject a subtype *sub* into the supertype *sup*, and prj let us do a case analysis on a *sup* to determine if it is in fact a *sub*.

The following instances state the reflexivity of $\cdot \hookrightarrow \cdot$, and that if $f = g_i$ for some i, then f is a subtype of a coproduct $g_1 + (g_2 + (\ldots))$. Note that the sum should be associated to the right for the type-checker to be able to infer an instance, so we need to be careful when constructing coproducts.

$$
\begin{aligned}
&\textbf{instance}\ (\mathsf{Functor}\ f) \Rightarrow f \hookrightarrow f\ \textbf{where} \\
&\quad \mathsf{inj} = \mathsf{id} \\
&\quad \mathsf{prj} = \mathsf{Just} \\
&\textbf{instance}\ (\mathsf{Functor}\ f, \mathsf{Functor}\ g) \Rightarrow f \hookrightarrow f \oplus g\ \textbf{where} \\
&\quad \mathsf{inj} \qquad\ = \mathsf{Inl} \\
&\quad \mathsf{prj}\ (\mathsf{Inl}\ f) = \mathsf{Just}\ f \\
&\quad \mathsf{prj}\ \_ \qquad = \mathsf{Nothing} \\
&\textbf{instance}\ (\mathsf{Functor}\ h, f \hookrightarrow g) \Rightarrow f \hookrightarrow h \oplus g\ \textbf{where} \\
&\quad \mathsf{inj} \qquad\ = \mathsf{Inr} \cdot \mathsf{inj}
\end{aligned}
$$

```
con       :: (s ↪ t) ⇒ s (Term t x) → Term t x
con       = Con · inj
put       :: (Put ↪ s) ⇒ Char → Term s x
put c     = con (Put c)
seq       :: (Seq ↪ s) ⇒ Term s x → Term s x → Term s x
seq p q   = con (Seq p q)
· ⊔ ·     :: (Alt ↪ s) ⇒ Term s x → Term s x → Term s x
p ⊔ q     = con (Alt p q)
n         :: (N ↪ s) ⇒ Int → Term s x
n m       = con (N m)
add       :: (Add ↪ s) ⇒ Term s x → Term s x → Term s x
add p q   = con (Add p q)
ifz       :: (Ifz ↪ s) ⇒ Term s x → (Term s x, Term s x) → Term s x
ifz c (t, e) = con (Ifz c t e)
thr       :: (Thr ↪ s) ⇒ Term s x
thr       = con Thr
cat       :: (Cat ↪ s) ⇒ Term s x → Term s x → Term s x
cat p q   = con (Cat p q)
```

Fig. 7: Modular constructors for the operators of P, Z, and E

```
prj (Inr a) = prj a
prj _       = Nothing
```

For convenience, we define an auxiliary function $f \oslash g$ for processing supertypes, which applies a function g if we are in the case of a chosen subtype and a function f otherwise.

```
(⊘)       :: (sub ↪ sup) ⇒ (sup x → a) → (sub x → a) → sup x → a
(f ⊘ g) x = case prj x of
              Nothing → f x
              Just y  → g y
```

Finally, we will rewrite the auxiliary functions in Figure 6 so that they work with any signature which satisfies certain requirements expressed as type constraints. In Figure 7 we show the modular constructors for the operators in P, Z, and E.

With the use of coproducts and the functorial representation of signatures, we achieved our goal of obtaining and implementing a modular syntax.

4 Transition Relations as Coalgebras

As shown in Section 2, operational semantics are given by a transition relation which represents execution steps in an abstract machine. Transition relations can be modeled in a generic, categorical way by coalgebras [10]. Given an endofunctor B, a B-coalgebra is an object X and a structure map $X \to BX$. The carrier of the coalgebra X can be seen as the states of an abstract machine while the endofunctor B represents the observable behaviour of the machine.

Every relation $R \subseteq X \times Y$ can be written as a function $X \to \mathcal{P}Y$ mapping every element in X to its set of related elements in Y. The simplest technique for interpreting the powerset functor in Haskell is to use the list functor. Thus, we interpret relations $R \subseteq X \times Y$ as Haskell functions $\mathsf{X} \to [\mathsf{Y}]$.

Example 4.1 The SOS rules for the language P defined two transition relations $\to \subseteq P \times A \times P$ and $\to\checkmark \subseteq P \times A$. Given our decision to interpret relations as Haskell functions, it is natural to write these as coalgebras with carrier $\mathsf{Program\ P}$, and structure functions $\mathsf{Program\ P} \to [(\mathsf{A}, \mathsf{Program\ P})]$ and $\mathsf{Program\ P} \to [\mathsf{A}]$ respectively. In order to make the coalgebraic structure explicit, let us define functorial composition, the functor $\mathsf{Pr}\ a$, which pairs a character from A with a, and the constant A functor:

```
data (h ∘ g) x = Comp{ deComp :: (h (g x)) }
instance (Functor h, Functor g) ⇒ Functor (h ∘ g) where
  fmap f (Comp c) = Comp (fmap (fmap f) c)
data Pr a  = Pr A a
data KA a = KA A
```

Using these definitions, we can express the transition relation $\to$ by a $([\,] \circ \mathsf{Pr})$-coalgebra on $\mathsf{Term\ P}$ and the transition relation $\to\checkmark$ by a $([\,] \circ \mathsf{KA})$-coalgebra on $\mathsf{Term\ P}$. Furthermore, we can pack *both* transition relations into a $([\,] \circ (\mathsf{Pr} \oplus \mathsf{KA}))$-coalgebra on $\mathsf{Term\ P}$ which captures all the observable behaviour of language P.

Example 4.2 Consider the language Z of Section 2. A simple inductive argument shows that the $\Downarrow$ relation is a function. Hence, we can describe the induced transition relation by a KI-coalgebra, where KI is the constant Int functor.

```
data KI a = KI Int
```

Example 4.3 The transition relation $\uparrow$ can be represented by a KE-coalgebra, where KE is the constant unit functor.

```
data KE a = KE
```

As shown in these last two examples, when the transition relation is a function, we can remove the powerset (or list). In this manner, the determinism of the underlying transition system is made explicit, avoiding the need for a separate proof. Being able to describe precisely what is observable by choosing the appropriate behaviour functor is an important advantage of the coalgebraic approach.

4.1 Execution of transition systems

In order to execute a transition system specified by a coalgebra, we *unfold* the coalgebra [10,9] to construct a tree of observations. The appropriate notion of tree is given by the greatest fixpoint [4] of the behaviour functor of the coalgebra.

[4] In Haskell there is no distinction between least and greatest fixpoints of recursive datatypes. To distinguish between them, we write least fixpoints as **data** and greatest fixpoints as **codata**.

$$\begin{aligned}&\textbf{codata}\ \mathsf{Nu}\ f = \mathsf{Nu}\ (f\ (\mathsf{Nu}\ f))\\&\mathsf{unfold} \quad :: \mathsf{Functor}\ b \Rightarrow (x \to b\ x) \to x \to \mathsf{Nu}\ b\\&\mathsf{unfold}\ g = \mathsf{Nu} \cdot \mathsf{fmap}\ (\mathsf{unfold}\ g) \cdot g\end{aligned}$$

In conclusion, coalgebras provide an abstract model of transition systems, where the type of the transition system and its corresponding notion of equality are determined by a functor. However, as discussed in the next section, this is not sufficient to model structural operational semantics.

5 Mathematical Operational Semantics

Coalgebras provide an abstract model of transition systems. Unfortunately, they do not support a proper theory of SOS. In particular, the carrier of a coalgebra is unstructured, and hence a purely coalgebraic approach will not be able to take advantage of the fact that the carrier of the coalgebra is the set of terms, and hence, has an algebra structure. Therefore, in order to develop a mathematical operational semantics, what we need is a structure which contains both coalgebraic and algebraic features. Turi constructed such a structure in his categorical framework for SOS by focusing on the operational rules rather than on the transition relation.

In this section we present our implementation of Turi's framework. To begin with, let us consider a typical operational rule and analyse its structure:

$$\frac{p \xrightarrow{a} p'}{p\,;q \xrightarrow{a} p'\,;q} \qquad\qquad \frac{premisses}{source \to target}$$

In general, a rule consists of some premisses and a conclusion. The source of the conclusion consists of an operator of the language (the ; operator, in the example above) applied to some metavariables (p and q) which stand for arbitrary terms. Premisses are transitions from these metavariables. Finally, the target of the conclusion is a term with metavariables taken from the source of the conclusion and from the premisses (q and p', respectively).

In the previous two sections we showed how to abstract syntax by a signature functor and observable behaviour by a behaviour functor. Using these concepts we can abstract the structure of operational rules.

5.1 The Type of Operational Rules

Given a language with syntax determined by a signature functor s and behaviour functor b, its structural operational semantics is given by rules of the form:

$$\begin{aligned}\textbf{type}\ \mathsf{OR}\ s\ b = \forall\ x\ y \cdot (x \to y) \to \quad &\text{-- Term environment}\\(x \to b\ y) \to \quad &\text{-- Behaviour environment}\\s\ x \to \quad &\text{-- Source of the conclusion}\\b\ (\mathsf{Term}\ s\ y) \quad &\text{-- Target of the conclusion}\end{aligned}$$

The type above says that operational rules are defined by a function which given two environments and the source of the conclusion of a rule, returns the transition

in the conclusion of the rule. The x in the type declaration above corresponds to variables in the source of a transition (*source variables*) and y to variables in the target of a transition (*target variables*). The environments are:

Term environment: specifies which target variable corresponds to each source variable. It enables us to use source variables to construct the term in the target of the conclusion.

Behaviour environment: specifies the transition corresponding to each source variable. It plays the role of the premisses in an operational rule.

Note that source and target variables are polymorphic to ensure that the defined semantics do not depend on their actual nature. The distinction between source variables and target variables guarantees that the induced semantics depends only on the behaviour of its subterms, and not on the actual subterms.

Example 5.1 In Figure 8 we give operational semantics[5] to the constructs of P given in Example 3.1 with a behaviour functor $([\,] \circ (\mathsf{KA} \oplus \mathsf{Pr})$.

Function *orP* implements the operational rules of P given in Figure 1 by pattern-matching on the operator in the source of the conclusion. In the case of $\mathsf{Put}\ c$, the only possible transition is to print c and terminate. In the case of $\mathsf{Seq}\ p\ q$, we analyse the type of each possible transition of p to see which transition to perform (for the definition of the monad instance for lists, see appendix B.3). If p may print a character c and terminate, then $\mathsf{Seq}\ p\ q$ prints c and continues execution with term q. If p may print c and continue execution with term p' then $\mathsf{Seq}\ p\ q$ prints c and continues execution with term $\mathsf{Seq}\ p'\ q$. In the case of $\mathsf{Alt}\ p\ q$, the possible transitions are the union of the possible transitions from p and from q.

It is important to note that values of type $\mathsf{OR}\ s\ b$ are isomorphic to Turi's abstract operational rules (see Appendix A). Consequently, not only are they a structured, language-independent formulation of SOS, but also they are guaranteed to induce a transition relation with bisimulation as a congruence and to generate an adequate denotational model. We prefer OR rules rather than Turi's abstract operational rules since they lead to a natural implementation in a functional language.

5.2 Obtaining a Transition Relation

Every operational rule $\mathsf{OR}\ s\ b$ induces a lifting *opMonad* of the syntax monad $\mathsf{Term}\ s$ to the category of b-coalgebras. The function *opMonad* (the operational monad [22]) takes a b-coalgebra on x and returns a b-coalgebra on $\mathsf{Term}\ s\ x$. Intuitively, *opMonad* shows that given an operational rule and the semantics of variables x in the terms, we can give semantics to terms with variables from x.

$$
\begin{aligned}
&\mathit{opMonad} && :: (\mathsf{Functor}\ s, \mathsf{Functor}\ b) \Rightarrow \\
&&& \quad \mathsf{OR}\ s\ b \to (x \to b\ x) \to \mathsf{Term}\ s\ x \to b\ (\mathsf{Term}\ s\ x) \\
&\mathit{opMonad}\ \mathit{op}\ k &&= \mathsf{snd} \cdot \mathsf{foldTerm}\ \langle \mathsf{Var}, \mathsf{fmap}\ \mathsf{Var} \cdot k \rangle \\
&&& \qquad\qquad \langle \mathsf{Con} \cdot \mathsf{fmap}\ \mathsf{fst}, \mathsf{fmap}\ \mathsf{join} \cdot \mathit{op}\ \mathsf{fst}\ \mathsf{snd} \rangle \\
&&& \quad \textbf{where}\ \langle f, g \rangle\ a = (f\ a, g\ a)
\end{aligned}
$$

[5] For clarity, in Figure 8 we have omitted the constructors for functorial composition.

```
orP                    :: OR P ([] ∘ (KA ⊕ Pr))
orP _ _  (Put c)       = [Inl (KA c)]
orP te be (Seq p q)    = be' p >>= copair
                           (λ(KA c)    → [Inr (Pr c (te' q))])
                           (λ(Pr c p') → [Inr (Pr c (Con (Seq p' (te' q))))])
                           where te' = Var · te
                                 be' = map (fmap Var) · be
orP te be (Alt p q)    = map (fmap Var) (be p ++ be q)
```

Fig. 8: Operational semantics for P

In order to execute a Program we unfold the coalgebra obtained from *opMonad*:

```
run     :: (Functor s, Functor b) ⇒ OR s b → Program s → Nu b
run op = unfold (opMonad op empty)
```

It is a simple exercise for the reader to use *opMonad* to obtain a denotational model $s\ (\mathsf{Nu}\ b) \to (\mathsf{Nu}\ b)$ corresponding to a given OR.

This concludes our functional implementation of Turi's mathematical operational semantics. In the next section we tackle the question of how to modularly combine operational rules.

6 Modular Operational Semantics

Operational rules $\mathsf{OR}\ s\ b$ are defined for a signature functor s and behaviour functor b. In Section 3 we showed how to obtain modular syntax by abstracting from a specific signature functor, and instead considering a signature functor equipped with type constraints as in Figure 7. Our goal now is to obtain a similar abstraction technique for behaviours.

Many behaviours that appear in practice, including all the behaviours in our examples, are of the form $m \circ b$, where m is a monad and b is a coproduct of functors. If we only consider operational semantics with behaviours of this form we can abstract from a concrete $m \circ b$ and obtain modular behaviours:

- Rather than considering a concrete monad, we consider a monad m which supports certain required operations. This is exactly the technique proposed by Liang et al. [14] and implemented in the Haskell library.
- The same technique used to abstract from concrete syntax using the $\hookrightarrow$ relation can be used to abstract coproducts of behaviours b.

Putting together modular behaviours with modular syntax yields the following definition of *modular operational rules*:

```
type MOR s t m b = ∀ x y · (x → y) →
                           (x → m (b y)) →
                           s x → m (b (Term t y))
```

Modular operational rules MOR differ from concrete operational rules OR in two ways:

(i) they distinguish between the signature s of the semantic component being defined, and the signature of the complete language t,

(ii) they consider behaviours to be the functorial composition of a monad m and a functor b.

Given a MOR, we can *ossify* it and obtain a concrete OR by fixing the signature of the complete language to be the signature of the language being defined, and providing a behaviour which satisfies the behaviour requirements of the given MOR.

$$\begin{aligned}&\mathit{ossify} && :: \mathsf{MOR}\ s\ s\ m\ b \to \mathsf{OR}\ s\ (m \circ b)\\ &\mathit{ossify}\ \mathit{mor}\ \mathit{te}\ \mathit{be} &&= \mathsf{Comp} \cdot \mathit{mor}\ \mathit{te}\ (\mathit{deComp} \cdot \mathit{be})\end{aligned}$$

6.1 Combining Modular Operational Rules

Combining modular operational rules is a simple matter of taking their copair. The requirements on the behaviour of the combined rules is the combination of the requirements on behaviour of each component.

$$\begin{aligned}&(\Cup) && :: \mathsf{MOR}\ s\ t\ m\ b \to \mathsf{MOR}\ s'\ t\ m\ b \to \mathsf{MOR}\ (s \oplus s')\ t\ m\ b\\ &(\mathit{op1} \Cup \mathit{op2})\ \mathit{te}\ \mathit{be} &&= \mathsf{copair}\ (\mathit{op1}\ \mathit{te}\ \mathit{be})\ (\mathit{op2}\ \mathit{te}\ \mathit{be})\end{aligned}$$

This is the fundamental tool for combining modular operational rules. The constraint that the monad m and behaviour b should be the same for the input rules of $\Cup$ appears to be a severe restriction that undermines our original goal. However, as it will be shown next, we can sidestep this restriction by defining modular operational rules over an abstract monad and behaviour.

6.2 Defining Modular Operational Rules

The following example shows how to write the modular components for the operators of P:

$$\begin{aligned}&\mathit{morPut} && :: (\mathsf{Put} \hookrightarrow s, \mathsf{KA} \hookrightarrow b, \mathsf{Monad}\ m) \Rightarrow \mathsf{MOR}\ \mathsf{Put}\ s\ m\ b\\ &\mathit{morPut}\ \_\ \_\ (\mathsf{Put}\ c) &&= \mathsf{return}\ (\mathsf{inj}\ (\mathsf{KA}\ c))\end{aligned}$$

$$\begin{aligned}&\mathit{morSeq} && :: (\mathsf{Seq} \hookrightarrow s, \mathsf{KA} \hookrightarrow b, \mathsf{Pr} \hookrightarrow b, \mathsf{Monad}\ m)\\ &&&\quad \Rightarrow \mathsf{MOR}\ \mathsf{Seq}\ s\ m\ b\\ &\mathit{morSeq}\ \mathit{te}\ \mathit{be}\ (\mathsf{Seq}\ p\ q) &&= \mathit{be}'\ p \ggg\!= \mathsf{return} \cdot \mathsf{fmap}\ (\text{`}\mathit{seq}\text{`}\mathit{te}'\ q) \circledcirc \lambda(\mathsf{KA}\ c) \to\\ &&&\quad \mathsf{return}\ (\mathsf{inj}\ (\mathsf{Pr}\ c\ (\mathit{te}'\ q)))\\ &&&\quad \textbf{where}\ \mathit{te}' = \mathsf{Var} \cdot \mathit{te}\\ &&&\qquad\quad \mathit{be}' = \mathsf{fmap}\ (\mathsf{fmap}\ \mathsf{Var}) \cdot \mathit{be}\end{aligned}$$

$$\begin{aligned}&\mathit{morAlt} && :: (\mathsf{Alt} \hookrightarrow s, \mathsf{Functor}\ b, \mathsf{MonadPlus}\ m)\\ &&&\quad \Rightarrow \mathsf{MOR}\ \mathsf{Alt}\ s\ m\ b\\ &\mathit{morAlt}\ \mathit{te}\ \mathit{be}\ (\mathsf{Alt}\ p\ q) &&= \mathsf{fmap}\ (\mathsf{fmap}\ \mathsf{Var})\ (\mathit{be}\ p\ \text{`}\mathsf{mplus}\text{`}\ \mathit{be}\ q)\end{aligned}$$

The fundamental idea is that each component should have the least possible requirements on syntax and behaviour, as given by the type constraints in the type

```
morN                    :: (N ↪ s, KI ↪ b, Monad m) ⇒ MOR N s m b
morN _ _ (N n)          = return (inj (KI n))

---

morAdd                  :: (Add ↪ s, KI ↪ b, Monad m) ⇒ MOR Add s m b
morAdd te be (Add p q)  = be' p >>= return · fmap (`add` te' q) ⊜ λ(KI n) →
                          be' q >>= return · fmap (te' p `add`) ⊜ λ(KI m) →
                          return (inj (KI (n + m)))
                          where te' = Var · te
                                be' = fmap (fmap Var) · be

---

morIfz                  :: (Ifz ↪ s, KI ↪ b, Monad m) ⇒ MOR Ifz s m b
morIfz te be (Ifz c t e) = be' c >>= return · fmap (`ifz` (te' t, te' e))
                                ⊜ λ(KI n) → if n ≡ 0 then be' t else be' e
                           where te' = Var · te
                                 be' = fmap (fmap Var) · be
```

Fig. 9: Modular operational rules for the operators of Z

signatures above. For *morPut*, the only requirements are that Put is in the syntax of the complete language, and that KA is in the behaviour. The semantic component for *morSeq* is a bit more subtle. After obtaining the behaviour of the first argument with *be* it needs to check whether execution of the first argument has finished, as indicated by a behaviour KA. This check is implemented by a case analysis with the $\circledcirc$ operator. In the case of a behaviour KA, it will move on to the second argument with a Pr transition, as it is done in the non-modular semantics of P. The most interesting part is the handling of the case where the behaviour is not KA, but some possibly unknown behaviour [6]. Here, a step is made by propagating the unknown behaviour and continuing execution with the term obtained by adding the (Seq $[-]$ q) context to the resulting term of the transition. The semantic component for Alt is almost the same as the corresponding case in the non-modular semantics of P, except that now we do not explicitly require a behaviour $[]$, but ask for the monad in the behaviour to support the mplus operation (described in appendix B.4).

Example 6.1 Modular operational semantics for the modular components of Z and E are given in Figures 9 and 10, respectively. In *morAdd*, the monad in the behaviour forces the choice of an order of evaluation of the arguments of Add. In the next subsection, we will show how to modularly obtain an addition operator with non-deterministic order of evaluation.

6.3 Putting it all together

After writing modular operational rules for all the fragments, it is time to reap the fruits of our hard work. The following examples show how straightforward it is to obtain semantics for new languages by combining modular components.

[6] We say possibly unknown behaviour because in this case the behaviour might be the known behaviour Pr, or an unknown behaviour.

$$
\begin{array}{l}
morThr \quad :: (\mathsf{Thr} \hookrightarrow s, \mathsf{KE} \hookrightarrow b, \mathsf{Monad}\ m) \Rightarrow \mathsf{MOR}\ \mathsf{Thr}\ s\ m\ b \\
morThr\ \_\ \_\ \mathsf{Thr} \quad = \mathsf{return}\ (\mathsf{inj}\ \mathsf{KE}) \\
\hline
morCat \quad :: (\mathsf{Cat} \hookrightarrow s, \mathsf{KE} \hookrightarrow b, \mathsf{Monad}\ m) \Rightarrow \mathsf{MOR}\ \mathsf{Cat}\ s\ m\ b \\
morCat\ te\ be\ (\mathsf{Cat}\ p\ q) = be'\ p \gg\!= \mathsf{return} \cdot \mathsf{fmap}\ (`cat`\ te'\ q) \ominus \lambda \mathsf{KE} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad be'\ q \\
\qquad\qquad\qquad\qquad \mathbf{where}\ te' = \mathsf{return} \cdot te \\
\qquad\qquad\qquad\qquad\qquad\quad be' = \mathsf{fmap}\ (\mathsf{fmap}\ \mathsf{return}) \cdot be
\end{array}
$$

Fig. 10: Modular operational rules for the operators of E

Example 6.2 We construct $modP$ a modular version of the language P which combines the modular operational rules of its operators. The requirements on syntax and behaviour of $modP$ are the combination of the requirements of its components. To obtain a concrete operational semantics for P we fix the syntax to be exactly P and we instantiate the monad m to be the list monad $[\,]$, which satisfies the requirement of being a $\mathsf{MonadPlus}$:

$$
\begin{array}{l}
morP :: (\mathsf{Put} \hookrightarrow s, \mathsf{Seq} \hookrightarrow s, \mathsf{Alt} \hookrightarrow s, \mathsf{KA} \hookrightarrow b, \mathsf{Pr} \hookrightarrow b, \mathsf{MonadPlus}\ m) \Rightarrow \\
\qquad \mathsf{MOR}\ \mathsf{P}\ s\ m\ b \\
morP = morPut \Cup morSeq \Cup morAlt \\
orP' \ :: \mathsf{OR}\ \mathsf{P}\ ([\,] \circ (\mathsf{Pr} \oplus \mathsf{KA})) \\
orP' \ = ossify\ morP
\end{array}
$$

Example 6.3 A modular version of language Z is defined as the combination of the semantics of N, Add, and Ifz. To obtain a concrete version we fix the syntax and semantics with *ossify*. In this case the monad in the behaviour has no requirements, so we can instantiate it to the identity monad (described in Appendix B.2):

$$
\begin{array}{l}
morZ :: (\mathsf{N} \hookrightarrow s, \mathsf{Ifz} \hookrightarrow s, \mathsf{Add} \hookrightarrow s, \mathsf{Monad}\ m, \mathsf{KI} \hookrightarrow b) \Rightarrow \mathsf{MOR}\ \mathsf{Z}\ s\ m\ b \\
morZ = morN \Cup morAdd \Cup morIfz \\
z :: \mathsf{OR}\ \mathsf{Z}\ (\mathsf{Id} \circ \mathsf{KI}) \\
z = ossify\ morZ
\end{array}
$$

Example 6.4 Adding exceptions to P is just a matter of adding the semantics of throw and catch:

$$
\begin{array}{l}
ep :: \mathsf{OR}\ \mathsf{EP}\ ([\,] \circ (\mathsf{KE} \oplus \mathsf{Pr} \oplus \mathsf{KA})) \\
ep = ossify\ (morThr \Cup morCat \Cup morP)
\end{array}
$$

Example 6.5 Adding exceptions to Z is once again, just a matter of adding the semantics of throw and catch:

$$
\begin{array}{l}
ez :: \mathsf{OR}\ \mathsf{EZ}\ (\mathsf{Id} \circ (\mathsf{KE} \oplus \mathsf{KI})) \\
ez = ossify\ (morThr \Cup morCat \Cup morZ)
\end{array}
$$

Example 6.6 A version of Z which can also print characters is the following:

$$
\begin{array}{l}
zp :: \mathsf{OR}\ (\mathsf{Put} \oplus \mathsf{Seq} \oplus \mathsf{Z})\ (\mathsf{Id} \circ (\mathsf{KA} \oplus \mathsf{Pr} \oplus \mathsf{KI})) \\
zp = ossify\ (morPut \Cup morSeq \Cup morZ)
\end{array}
$$

Our modular semantics of addition has a fixed evaluation order from left to right. In order to define a version of addition with a non-deterministic order of evaluation we extend the semantics with $\sqcup$ and define non-deterministic addition *addND* as syntactic sugar for the term $(a + b) \sqcup (b + a)$.

$$
\begin{aligned}
&zP :: \mathsf{OR}\ (\mathsf{Put} \oplus \mathsf{Seq} \oplus \mathsf{Alt} \oplus \mathsf{Z})\ ([\,] \circ (\mathsf{KA} \oplus \mathsf{Pr} \oplus \mathsf{KI})) \\
&zP = \mathit{ossify}\ (\mathit{morPut} \Cup \mathit{morSeq} \Cup \mathit{morAlt} \Cup \mathit{morZ}) \\
&\mathit{addND}\ a\ b = (a\ `\mathit{add}`\ b) \sqcup (b\ `\mathit{add}`\ a)
\end{aligned}
$$

In order to obtain a combined semantics we need to provide a monad which supports the operations required by the modular components. One way to obtain such a monad is to use monad transformers together with liftings of operations as in [14]. Another way would be to use the coproduct of monads [5,7] or to use Lawvere theories [20]. Note that the requirements do not specify any order on the layering of effects, so there could be many different monads that satisfy these requirements, each yielding different combined semantics.

7 Related Work

A practical approach to modular operational semantics for certain specific effects has recently been put forward by Mosses [17], but it is based on the syntactic rather than semantic approach to SOS. Turi showed with a few examples how operational rules which are parametric in their behaviour could be instantiated to different settings [23] but did not attempt to systematize this technique. Lenisa et al. [13] defined an operation that combines two operational rules on the same behaviour $\mathsf{OR}\ s\ b$ and $\mathsf{OR}\ s'\ b$ into an operational rule $\mathsf{OR}\ (s \oplus s')\ b$, but did not consider the problem of semantics with different behaviour. The advantage of defining the combination operation for MOR rather than OR is that elements of MOR are flexible enough to allow the separate definition of operators which depend on other operators. This flexibility is especially advantageous if each operator has different requirements on the behaviour functor, as each operator will be defined with less requirements, yielding a more general semantics. Kick [11] presented the dual of the syntax combination operation for ORs, that is, an operation which takes two operational rules $\mathsf{OR}\ s\ b$ and $\mathsf{OR}\ s\ b'$, and returns a $\mathsf{OR}\ s\ (b \otimes b')$ (where $\otimes$ is functorial product). This operation does not seem to be powerful enough to support the combinations we are trying to obtain. However, it would be interesting to see how this operation, in the particular case of the behaviour being of the form $\mathcal{P}B$, could be used to obtain results similar to ours by exploiting the isomorphism $\mathcal{P}(A + B) \cong \mathcal{P}(A) \times \mathcal{P}(B)$.

8 Conclusion

We have developed a modular approach to operational semantics which allows us to define the semantics of a language as a combination of the semantics of its individual components. Our approach is based on writing the operational semantics on partially known syntax and behaviour, and on the representation of an operational semantics as a polymorphic function that distributes syntax over behaviour.

This high-level modular approach leads to a simple and natural implementation in Haskell, which serves to make our work more accessible and also to allow readers to experiment further with our constructions.

As with Turi's original work, this paper is fundamentally first-order. Therefore, in terms of future work, our primary aim is to consider modular operational semantics for languages with more advanced features, such as binding and recursion. Incorporating binding operations into Turi's framework is a difficult task, see [4,3] for example, but we have some preliminary ideas in this direction. In addition, we would like to investigate the extent to which our ideas are applicable to other models of program execution, such as abstract and virtual machines.

Our eventual aim is to be able to write modular operational semantics in Haskell in as clean and simple way as modular interpreters [14].

Acknowledgement

We would like to thank Sam Staton, Peter Mosses and Wouter Swierstra for their valuable feedback, and the FP lab in Nottingham and anonymous referees for their useful comments.

References

[1] Jan A. Bergstra and Jan Willem Klop. Algebra of communicating processes with abstraction. *Theor. Comput. Sci.*, 37:77–121, 1985.

[2] Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. Fast and loose reasoning is morally correct. In *POPL*, pages 206–217, 2006.

[3] Marcelo Fiore and Sam Staton. Comparing operational models of name-passing process calculi. *Electr. Notes Theor. Comput. Sci.*, 106:91–104, 2004.

[4] Marcelo Fiore and Daniele Turi. Semantics of name and value passing. In *Proc. 16th LICS Conf.*, pages 93–104. IEEE, Computer Society Press, 2001.

[5] Neil Ghani and Christoph Lüth. Composing monads using coproducts. *Intl. Conference on Functional Programming 2002*, 37(9):133–144, 2002.

[6] Neil Ghani and Cristoph Lüth. Monads and modular term rewriting. In *Proceedings of CTCS'97*, number 1290 in Lecture Notes in Computer Science, pages 69–86. Springer-Verlag, 1997.

[7] Neil Ghani and Tarmo Uustalu. Coproducts of ideal monads. *Journal of Theoretical Informatics and Applications*, (38):321–342, 2004.

[8] Tatsuya Hagino. *A categorical programming language*. PhD thesis, University of Edinburgh, 1987.

[9] Graham Hutton. Fold and Unfold for Program Semantics. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming*, Baltimore, Maryland, September 1998.

[10] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, 1997.

[11] Marco Kick and A. John Power. Modularity of behaviours for mathematical operational semantics. *Electr. Notes Theor. Comput. Sci.*, 106:185–200, 2004.

[12] Bartek Klin. Adding recursive constructs to bialgebraic semantics. *Journal of Logic and Algebraic Programming*, 60-61, 2004.

[13] M. Lenisa, J. Power, and H. Watanabe. Distributivity for endofunctors, pointed and co-pointed endofunctors, monads and comonads. In Horst Reichel, editor, *Proceedings 3rd Workshop on Coalgebraic Methods in Computer Science, CMCS'00, Berlin, Germany, 25–26 March 2000*, volume 33. Elsevier, Amsterdam, 2000.

[14] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In ACM, editor, *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco, California, January 22–25, 1995*, pages 333–343, New York, USA, 1995. ACM Press.

[15] Saunders Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. Springer-Verlag, 1971. Second edition, 1998.

[16] Erik Meijer, Maarten 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, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[17] Peter D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:195–228, 2004. Special issue on SOS.

[18] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003.

[19] Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[20] Gordon D. Plotkin and John Power. Computational effects and operations: An overview. *Electr. Notes Theor. Comput. Sci.*, 73:149–163, 2004.

[21] Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(3):1–14, 2008.

[22] Daniele Turi. *Functorial Operational Semantics and its Denotational Dual*. PhD thesis, Free University, Amsterdam, June 1996.

[23] Daniele Turi. Categorical modelling of structural operational rules: Case studies. In *Category Theory and Computer Science*, pages 127–146, 1997.

[24] Daniele Turi and Gordon D. Plotkin. Towards a mathematical operational semantics. In *Proc. 12th LICS Conf.*, pages 280–291. IEEE, Computer Society Press, 1997.

A ORs are Turi's Operational Rules

Operational rules OR*s* are isomorphic to Turi's abstract operational rules AOR. Categorically, this can be seen by the universal property of right Kan extensions and their end formula [15], which is valid in any parametric model. More concretely, the isomorphism is given by the following functions:

$$
\begin{aligned}
&\textbf{type}\ \mathsf{AOR}\ s\ b &&= \forall\ a \cdot s\ (a, b\ a) \to b\ (\mathsf{Term}\ s\ a)\\
&\mathit{fromOR} &&:: \mathsf{OR}\ s\ b \to \mathsf{AOR}\ s\ b\\
&\mathit{fromOR}\ \mathit{os} &&= \mathit{os}\ \mathsf{fst}\ \mathsf{snd}\\
&\mathit{toOR} &&:: (\mathsf{Functor}\ s) \Rightarrow \mathsf{AOR}\ s\ b \to \mathsf{OR}\ s\ b\\
&\mathit{toOR}\ \mathit{aor}\ \mathit{te}\ \mathit{be} &&= \mathit{aor} \cdot \mathsf{fmap}\ \langle \mathit{te}, \mathit{be} \rangle\\
&\qquad\qquad\qquad &&\textbf{where}\ \langle f, g \rangle\ a = (f\ a, g\ a)
\end{aligned}
$$

B Additional definitions

B.1 Functors and Monads

A datatype is shown to have a functorial or a monadic structure by an instance of the following classes, plus the proof that certain coherence conditions hold [18].

$$
\begin{aligned}
&\textbf{class}\ \mathsf{Functor}\ f\ \textbf{where}\\
&\quad \mathsf{fmap}\ :: (a \to b) \to f\ a \to f\ b\\
&\textbf{class}\ \mathsf{Monad}\ m\ \textbf{where}\\
&\quad \mathsf{return} :: a \to m\ a\\
&\quad (\gg\!=)\ :: m\ a \to (a \to m\ b) \to m\ b
\end{aligned}
$$

The multiplication of a monad for all Monad instances is:

```
join :: Monad m ⇒ m (m a) → m a
join = (≫=id)
```

B.2 The Identity Monad

The identity monad is given by the following instances:

```
data Id a = Id a
instance Functor Id where
  fmap f (Id a) = Id (f a)
instance Monad Id where
  return      = Id
  (Id a) ≫= f = f a
```

B.3 The List Monad

The list monad is given by the following instances:

```
instance Functor [] where
  fmap f []     = []
  fmap f (x:xs) = f x : fmap f xs
instance Monad [] where
  return x      = [x]
  [] ≫= f       = []
  (x:xs) ≫= f   = f x ++ (xs ≫= f)
```

B.4 The class of MonadPlus *monads*

Monads that support choice and failure, such as the list monad via ++ and [], are instances of the class MonadPlus:

```
class (Monad m) ⇒ MonadPlus m where
  mzero :: m a
  mplus :: m a → m a → m a
```