*JFP*, **??** pages, 2022. © Cambridge University Press 2022
doi:10.1017/xxxxx

FUNCTIONAL PEARL

# *Full Abstraction for Free*

MARCO PAVIOTTI
*University of Kent,*
*(e-mail: m.paviotti@kent.ac.uk)*

NICOLAS WU
*Imperial College London,*
*(e-mail: n.wu@imperial.ac.uk)*

## Abstract

Structured recursion schemes such as folds and unfolds have been widely used for structuring both functional programs and program semantics. In this context, it has been customary to implement denotational semantics as folds over an inductive data type to ensure termination and compositionality. Separately, operational models can be given by unfolds, and naturally not all operational models coincide with a given denotational semantics in a meaningful way.

To ensure these semantics are coherent it is important to consider the property of full abstraction which relates the denotational and the operational model. In this paper, we show how to engineer a compositional semantics such that full abstraction comes for free. We do this by using *distributive laws* from which we generate both the operational and the denotational model. The distributive laws ensure the semantics are fully abstract at the type level, thus relieving the programmer from the burden of the proofs.

## 1 Introduction

For a long time structured recursion schemes have been used to ensure that functions are well defined (Meijer et al., 1991). In particular, folds have been used to ensure termination while unfolds have been used to ensure productivity of recursively defined functions. A perhaps less known fact about recursion operators is that they provide additional semantic properties when used in the context of programming languages and semantics. Hutton (1998) popularized the idea that folds can be used to define the denotational semantics, and unfolds can be used to define the operational semantics of such languages.

Given that folds and unfolds provide different semantics, a natural question is how they relate to one another. This question is answered with the idea of *full abstraction* which is the statement that syntactically different programs that behave *operationally* the same way in any context should be regarded as exactly the same entity *denotationally*.

To explain this point we define the grammar of Hutton's razor as an inductive set (left) and implement it as a data type in Haskell (right):

$$L := \mathbb{N} \mid L + L \qquad\qquad \mathbf{data}\, L = Val\, Nat \mid Add\, L\, L$$

where *Nat* is the natural numbers type. The semantics of this language can be defined by folding the structure of the syntax into the structure of a semantic domain, given by the base case $n :: Nat$ and the binary operation $Add :: Nat \to Nat$:

$$\begin{aligned}
&[\![\cdot]\!] :: L \to Nat \\
&[\![Val\, n]\!] \quad\;\; = n \\
&[\![Add\, t_1\, t_2]\!] = [\![t_1]\!] + [\![t_2]\!]
\end{aligned}$$

This kind of semantics is often called a *denotational* semantics, where the meaning of a term is given in terms of subterms and in this sense denotational semantics are modular interpretations of syntax into a given domain. As it happens, this is precisely how a generic fold operates: it takes a function that handles a base case and another for the inductive case, together known as an *algebra*, and uses these to reduce a data structure to a final value. Folds over syntax exactly correspond to the structure of denotational semantics.

However, we could have defined the semantics in many other ways that correspond to folds as as well. For example, by sending every program to $()$, i.e. $[\![\_]\!] = ()$, sending every program to itself, i.e. $[\![\cdot]\!] = id$. A perhaps more extreme example would be to interpret the syntactic constructor *Add* as the semantics multiplication.

Given the multitude of different semantics that can be given, a good question is how to provide a means of arbitrating between them. While addition of numbers into *Nat* seems reasonable, the other three are somewhat extreme and unsatisfactory.

However, the only way of arbitrating between a good semantic model and a bad one is to define precisely what programs are supposed to "do", for example, by giving an *operational* model for the language. Here, rules are used to define a reduction relation, written $t \to t'$, for programs $t$ and $t'$, where for each rule, reductions below a line are permissible given reductions above the line:

$$\frac{}{Add\,(Val\, m)\,(Val\, n) \rightsquigarrow Val\,(m+n)} \qquad \frac{t_1 \rightsquigarrow t_1'}{Add\, t_1\, t_2 \rightsquigarrow Add\, t_1'\, t_2} \qquad \frac{t_2 \rightsquigarrow t_2'}{Add\, t_1\, t_2 \rightsquigarrow Add\, t_1\, t_2'}$$

Once the operational model has been defined we can state the properties we want from a denotational model. In particular, a good denotational model has to be *sound*, which means it has to be agnostic to step reductions and, moreover, it has to be at least *computationally adequate*, that is, if two programs denote the same object in the model they should be operationally indistinguishable. Furthermore, if the model equates exactly the programs that ought to be indistinguishable the semantics is called *fully abstract*. Thus a good denotational semantics should be related in some way to the operational semantics.

The problem is that the existing recursion schemes proposed to give semantic interpretations, like folds, do not have any restrictions on the kind of algebras the user can provide. In other words, folds operators do not preserve nor reflect the additional information provided by the operational semantics.

In coalgebraic systems, we can associate an operational semantics to the syntax tree structure at hand by using a *coalgebra*, that is a function which takes a program as input and outputs the target program along with some information about the behavior produced. For example, the operational semantics for the language defined above can be equivalently defined as a function $opsem :: L \to [L]$[1] where values are sent to the empty set of programs, programs of the form $Add\ (Val\ n)\ (Val\ m)$ are sent to $Val\ (n+m)$ and programs of the form $Add\ t_1\ t_2$ are sent to programs of the form $Add\ t_1'\ t_2$ and $Add\ t_1\ t_2'$ whenever $t_1$ reduces to $t_1'$ and similarly for $t_2$.

Now that we have the operational information we need to ensure this is somehow respected when interpreting the syntax. As we stated earlier, the fold operator is not suitable for this job, but the *unfold* operator is. In particular, the unfold is a function that given a program produces the "trace" generated by "unfolding" or "running" the operational model on that program. In other words, the unfold operator is the unique *fully abstract* translation from syntax to the domain of traces if, and only if, the unfolding of two programs is equal whenever these programs *behave the same operationally*.

In this paper we show how to make use of *distributive laws* to ensure that the interpretation function from the syntax to the semantic domain is both a fold (is compositional) and an unfold (is fully abstract). The theory behind this approach has been known for a long time (Rutten and Turi, 1993; Turi and Plotkin, 1997; Jacobs and Rutten, 2012; Rutten, 2000), and many have used this theory for structuring functional programming (Hinze and James, 2011; Jaskelioff et al., 2011). However, none of these works focusses explicitly on the use of recursion operators for semantic properties.

In this paper, we do this by giving a fresh account on the Hutton's razor and Milner's CCS. The latter, in particular, gives raise essentially to a (correct) Domain Specific Language (DSL) for stream programming in Haskell. Our work can be seen as extending the work of Hutton (1998) which expressed they idea of denotational semantics as a fold and of operational semantics as an unfold. However, these two do not necessarily coincide and, in the case of CCS, the correspondence needs a pen and paper proof.

In our approach the parametric type structure of distributive laws is used to guide the programmer towards a fully abstract semantics for free. We found that, while CCS semantics naturally fits in the context of the distributive laws, Hutton's arithmetic language needs a slight tweak. As it is customary (Danielsson et al., 2006), we implement the theory from the literature in the total fragment of Haskell. In particular, we implement data types using recursive types and we use the appropriate recursion scheme to keep initial algebras and final coalgebras distinct, and, finally, we only use predicative polymorphism, so the Haskell programs in this paper can be interpreted in the category of sets.

We start by providing background material on folds and unfolds (Section 2) and then we use these to implement a fully abstract semantics for a simple arithmentic language for streams (Section 3) and an arithmentic language with non-determinism (Section 4). As a more involved example, we show how give a full abstract semantics of Milner's value-passing CCS (Section 5). Finally, we conclude by listing the related work (Section 6) and discussion (Section 7).

[1] Where, for simplicity, sets are modelled as lists.

## 2 Background: Folds and Unfolds

Folds and unfolds are the most basic patterns of recursion. The former are used to ensure termination of recursive definitions and the latter are used to ensure definitions of infinite data structures be productive. When folds are to structure semantics they provide a modular denotational semantics, as popularised by Hutton (1998). Similarly, unfolds provide a fully abstract interpretation by construction, but this fact is far less explored in the functional programming literature. This section provides a summary the basics of these two patterns of recursion. In Section 2.1 we introduce folds. In Section 2.2 we introduce unfolds and explain why these are fully abstract interpretations over transition systems.

### *2.1 Folds*

A fold is a recursive operator that recurses over an inductive data structure and through the use of a given combining operator reassembles the results of the recursively transformed subparts of the inductive structure.

Folds use the notion of *Functor* to describe the structure of a data type and define an inductive data type by the least fixed-point of this functor. In the code below, Σ is a variable which we will use to pass the functor as an argument to $\mu\cdot$. This in turn will compute the least fixed-point of Σ providing the constructor *In* and deconstructor $in^{\circ}$ which witness the fact that $\mu\Sigma$ is isomorphic to $\Sigma\,(\mu\Sigma)$[1].

$\textbf{newtype}\ \mu\Sigma = \mathit{In}\ \{\mathit{in}^{\circ} :: \Sigma\ (\mu\Sigma)\ \}$

A fold is a recursive function from the type $\mu\Sigma$ to any type $a$ endowed with a function $alg :: \Sigma\, a \to a$. A fold takes as input the function $alg$ and recursively transforms an inductively defined data structure into an element of type $a$ using $alg$ to recombine the results from the recursive calls

$$
\begin{aligned}
&(\!|\cdot|\!) :: \mathit{Functor}\ \Sigma \Rightarrow (\Sigma\, a \to a) \to \mu\Sigma \to a\\
&(\!|\,alg\,|\!) = alg \cdot \mathit{fmap}\,(\!|\,alg\,|\!) \cdot in^{\circ}
\end{aligned}
$$

For example, recall the language we introduced earlier in Section 1. The signature functor Σ of this language can be defined as:

$\textbf{data}\ \mathit{ValAddF}\ x = \mathit{Val}\ \mathit{Nat} \mid \mathit{Add}\ x\ x\ \textbf{deriving}\ \mathit{Functor}$

where $\mathit{Val} :: \mathit{Nat} \to \Sigma\, x$ and $\mathit{Add} :: x \to x \to \Sigma\, x$ are the constructors of the language. The language described by Σ is denoted by $\mu\Sigma$. To simplify the presentation and avoid cluttering the code with the *In* operator we can define *smart constructors* as well

$$
\begin{aligned}
&\mathit{add} :: \mu\,\mathit{ValAddF} \to \mu\,\mathit{ValAddF} \to \mu\,\mathit{ValAddF} \qquad && \mathit{val} :: \mathbb{N} \to \mu\,\mathit{ValAddF}\\
&\mathit{add}\ x\ y = \mathit{In}\ (\mathit{Add}\ x\ y) && \mathit{val} = \mathit{In} \cdot \mathit{Val}
\end{aligned}
$$

The interpretation function can now be defined as a fold over a Σ-algebra. For example, if we wanted to interpret this language into the domain of natural numbers, we could use the following Σ-algebra on *Nat*.

[1] Here we assume inductive and coinductive data types will be distinguished by their keywords $\mu\cdot$ and $\nu\cdot$.

$desem :: ValAddF\ Nat \to Nat$
$desem\ (Val\ n) = n$
$desem\ (Add\ n\ m) = n + m$

Then the fold over *desem*, namely $(\!|\, desem\,|\!)$, is by definition equal to the interpretation function $[\![\cdot]\!]$ we have shown in Section 1.

## *2.2 Unfolds*

While folds *destruct* data, their dual, the unfolds, *construct* data. In particular, given a seed value and an *observation*, the unfold corecursively "runs" the observations starting from the initial seed.

For example, for a set of states $X$ and an alphabet set $L$ we can define a transition system on $X$ as a function $c : X \to L \times X$ implementing the *transition* map. In particular, for a state $x_1 \in X$, $c(x_1)$ returns a pair $(l, x_2) \in L \times X$ where $l \in L$ is the observable action and $x_2 \in X$ is the next state.

Now consider the transition system below composed by three states $x_1, x_2$ and $x_3$ with the transitions labelled by the set of labels $L = \{a, b\}$ $c_X$ defined such that $x_1 \mapsto (a, x_2)$, $x_2 \mapsto (b, x_3)$ and $x_3 \mapsto (c, x_3)$.

$$\begin{tikzcd} x_1 \arrow[r, "a"] & x_2 \arrow[r, "b"] & x_3 \arrow[loop right, "b"] \end{tikzcd}$$

This can be implemented by defining the *Functor* of behaviors, say *BHV* representing the shape of the information observed in each transition and an observation map $obs ::$ $States \to BHV\ States$ taking a starting state to the next state together with the additional information about the observable behaviors produced in the process

**data** $BHV\ k = Nat :< k$
**data** $States = X_1 \mid X_2 \mid X_3$

$opsem :: States \to BHV\ States$
$opsem\ X_1 = 1 :< X_2$
$opsem\ X_2 = 2 :< X_3$
$opsem\ X_3 = 2 :< X_3$

We can set some notation to make it explicit that a this transition map is in fact the operational semantics of our transition system

$$x \xrightarrow{l}_X x' \overset{\triangle}{\iff} obs(x) = (l, x')$$

In the context of transition systems, the unfold function executes the operational semantics from a starting state while collecting the observable behaviors that appear in each transition. Thus, we need define the type for the collection of behaviors first.

### *2.2.1 Coinductive Data Types*

Since running the operational semantics may result in non-termination we are interested in (possibly) infinite data types. In our example above where $BHV$ was defined as the labels paired with the target state, we were interested in the infinite collections of labels. Hence, once the shape of the behaviours $B$ is known, we want to have the *greatest* fixed-point of $B$, written $\nu B$, and implemented as follows[1]

$\mathbf{newtype}\ \nu b = Out^{\circ}\ \{out :: b\ (\nu b)\}$

Now, for each starting element of type $x$ the unfold maps this element to the unique element in $\nu b$ that satisfies the following equation

$[\![\cdot]\!] :: Functor\ b \Rightarrow (x \to b\ x) \to x \to \nu b$
$[\![coalg]\!] = Out^{\circ} \cdot fmap\ [\![coalg]\!] \cdot coalg$

Let us exemplify this by using the functor $BHV$ of behaviors we defined above. Its greatest fixed-point is the data type of streams (infinite lists) of numbers

$\mathbf{type}\ Stream = \nu BHV$

Now, starting from a state, say $X_2$, the unfold over the transition map *opsem* is obtained by iterating *opsem* infinitely many times while collecting observable behaviours which yields the *infinite trace* "22222222222222 . . . " for the state $X_2$.

### *2.2.2 Full Abstraction for unfolds*

One can easily see that despite $X_2$ and $X_3$ are different states, their trace is equal. That is $[\![opsem]\!]\,X_2 = [\![opsem]\!]\,X_3$. This means that that two states produce the same trace if and only if they are mapped into the same streams. To see this, consider the state $X_2$ again and notice that *opsem* takes $X_2$ to $X_3$ with label 2 then $[\![opsem]\!]\,X_2$ must map $X_2$ to an infinite stream such that the first label is 2 and the rest of the stream is where $X_3$ is mapped to, hence the unfold preserves the behaviors when mapping into $\nu b$, so if two states are producing the same trace (operationally) so must be their unfolding. The other direction is provided by the fact that the unfold maps every state to its *unique* trace. In other words, there is no other trace that can represent one state so it the traces of two states are equal then their states behave the same operationally. For a more formal explanation of this phenomenon, we defer the interested reader to (Hinze, 2008, Appendix, Theorem 1).

### *2.2.3 Other behavioral functors*

Depending on the functor $B$ we choose we can model different kind of semantics.

Another example is big-step operational semantics which is modelled by the constant functor $Big\ x = V$ where $V$ is the set of values for a language. Its associated $V$-coalgebra $\cdot \Downarrow \cdot :: \mu\Sigma \to V$ sends programs in to the values in $V$ they reduce to. We can use classic notation for big-step semantics as follows

$$t \Downarrow v \overset{\triangle}{\Longleftrightarrow} \cdot \Downarrow \cdot (t) = v$$

[1] Notice that in Haskell, inductive and coinductive data types coincide. Here we assume $\mu\cdot$ and $\nu\cdot$ are different.

So far we have considered semantics for deterministic languages. However, when the language is non-deterministic we need a way to map a program into multiple programs. To this end we can use the finite powerset functor $\mathscr{P}_{\text{fin}}(-)$ which can be implemented in functional programming using the type of lists. In this case, the operational semantics is a function of type $x \to [x]$ or even $x \to [(L,x)]$ where each transition is annotated with a label in the set $L$.

As in the previous cases, we can set some notation as below stating that $x_1$ reduces to $x_2$ if there exists $x_2$ in the list of states $x_1$ maps to.

$$x_1 \to_X x_2 \overset{\triangle}{\iff} x_2 \in opsem(x_1)$$

Now the we introduced both folds and unfolds and how to use unfolds on transition systems we need to put these two pieces together and this is the topic of the next section.

## 3 Simple Recursion Schemes for Program Semantics

In the previous section we have shown how to give a compositional denotational semantics for a small programming language by using folds (Section 2.1), how coalgebras can be used to describe operational semantics and, furthermore, how unfolds are fully abstract interpretations of transition systems (Section 2.2).

Our aim is now to get compositionality and full abstraction with the same interpretation function. We do this by using a distributive law between syntax and semantics. In this section we first exemplify distributive laws via the arithmetic language (Section 3.1) and then we generalise the framework to a generic signature and behavioral functor (Section 3.2).

### *3.1 Distributive Laws*

To understand how this is useful to achieve our goal consider the signature and the behavioral functor we defined in Section 2, in particular, *ValAddF* and *BHV*.

Say that we want to define an operational semantics for the language described by $\mu\,ValAddF$ and, in particular, we would like to have a semantics that behaves as per the following inference rules:

$$\frac{}{val\,n \overset{n}{\rightsquigarrow} val\,n} \qquad\qquad \frac{t1 \overset{n}{\rightsquigarrow} t1' \qquad t_2 \overset{m}{\rightsquigarrow} t_2'}{add\,t_1\,t_2 \overset{n+m}{\rightsquigarrow} add\,t_1'\,t_2'}$$

Mathematically, $\overset{n}{\to}$ is a inductively defined family of relations which happens to be a function as well. In particular, it is defined inductively over the structure of the language mapping a value $val\,n$ to $(n, val\,n)$ and a program $add\,t_1\,t_2$ to $(n+m, add\,t_1'\,t_2')$ if the two subprograms $t_1$ and $t_2$ map to $(n, t_1')$ and $(m, t_2')$ respectively.

It is easy to see that the above specification corresponds to an observation function over $\mu\,ValAddF$, i.e. a function of type $\mu\,ValAddF \to BHV\,(\mu\,ValAddF)$ sending a program to another program together with an observation as in the following recursive program:

```
opsemSimple :: μValAddF → BHV (μValAddF)
opsemSimple (In (Val n))       = n :< In (Val n)
```

$$\begin{aligned}
\mathit{opsemSimple}\ (\mathit{In}\ (\mathit{Add}\ t_1\ t_2)) = \mathbf{let}\ & (n :< t_1') = \mathit{opsemSimple}\ t_1 \\
& (m :< t_2') = \mathit{opsemSimple}\ t_2 \\
\mathbf{in}\ & \\
& (n+m) :< \mathit{add}\ t_1'\ t_2'
\end{aligned}$$

Since this is a recursive definition where the recursive call is only used on a strictly smaller input we can equivalently write *opsem* as a fold over an *ValAddF*-algebra on $\mathit{BHV}\ (\mu\mathit{ValAddF})$, i.e. a map $\mathit{ValAddF}\ (\mathit{BHV}\ (\mu\mathit{ValAddF})) \to \mathit{BHV}\ (\mu\mathit{ValAddF})$ as in the program below:

$$\begin{aligned}
&\mathit{opsemAlg} :: \mathit{ValAddF}\ (\mathit{BHV}\ (\mu\mathit{ValAddF})) \to \mathit{BHV}\ (\mu\mathit{ValAddF}) \\
&\mathit{opsemAlg}\ (\mathit{Val}\ n) = n :< \mathit{In}\ (\mathit{Val}\ n) \\
&\mathit{opsemAlg}\ (\mathit{Add}\ (n :< t_1')\ (m :< t_2')) = (n+m) :< \mathit{In}\ (\mathit{Add}\ t_1'\ t_2')
\end{aligned}$$

Next we notice that this algebra can be further decomposed by noticing that *In* is applied to the target program of type $\mathit{ValAddF}\ (\mu\mathit{ValAddF})$. In fact we can remove *In* resulting in an alternative definition of *opsemAlg* which we call *opsemDistr*

$$\begin{aligned}
&\mathit{opsemDistr} :: \mathit{ValAddF}\ (\mathit{BHV}\ (\mu\mathit{ValAddF})) \to \mathit{BHV}\ (\mathit{ValAddF}\ (\mu\mathit{ValAddF})) \\
&\mathit{opsemDistr}\ (\mathit{Val}\ n) = n :< \mathit{Val}\ n \\
&\mathit{opsemDistr}\ (\mathit{Add}\ (n :< t_1')\ (m :< t_2')) = (n+m) :< \mathit{Add}\ t_1'\ t_2'
\end{aligned}$$

which post-composed with *fmap In* will give *opsemAlg*. It is an easy simplification now to parametrise *opsemDistr* by substituting $\mu\mathit{ValAddF}$ with a universally quantified type variable, thus obtaining the following parametric function:

$$\mathit{opsemDistr} :: \forall x.\ \mathit{ValAddF}\ (\mathit{BHV}\ x) \to \mathit{BHV}\ (\mathit{ValAddF}\ x)$$

Notice how this does not change the implementation of *opsemDistr* since when we defined *opsemDistr* we did no rely on the structure of $\mu\mathit{ValAddF}$.

The above study case dualises nicely, of course. In fact, the denotational semantics is an *ValAddF*-algebra over the type $\nu B$ defined by $\mathit{dsem} = \mathit{opsemDistr} \cdot \mathit{fmap}\ \mathit{out}$.

### 3.2 The general case

In the example above, *opsemDistr* is a distributive law between the signature functor *ValAddF* and the behavioural functor *BHV* .

In general, given a signature and a behavior functor $\Sigma$ and $b$ respectively, a *distributive law* is a parametric function distributing $\Sigma$ over $b$ as follows:

$$\lambda :: (\mathit{Functor}\ \Sigma, \mathit{Functor}\ b) \Rightarrow \forall x.\ \Sigma\ (b\ x) \to b\ (\Sigma\ x)$$

At this point, given a distributive law $\lambda$ between $\Sigma$ and $b$, the following facts hold

- the operational model is a $b$-coalgebra of type $\mu\Sigma \to b\ (\mu\Sigma)$ given by a fold over the $\Sigma$-algebra $(\mathit{fmap}\ \mathit{In} \cdot \lambda) :: \Sigma\ (b\ (\mu\Sigma)) \to b\ (\mu\Sigma)$

  $\mathit{opsem}\ \lambda = (\!|\ \mathit{fmap}\ \mathit{In} \cdot \lambda\ |\!)$

- dually, the denotational model is a $\Sigma$-algebra of type $\Sigma\ (\nu b) \to \nu b$ given by the unfold over the $b$-coalgebra $(\lambda \cdot \mathit{fmap}\ \mathit{out}) :: \Sigma\ (\nu b) \to b\ (\Sigma\ (\nu b))$

$$desem\ \lambda = [\![ \lambda \cdot fmap\ out ]\!]$$

- the fold over the denotational model is compositional
- unfold over the operational model is fully abstract

Furthermore, the fold over the denotational model corresponds to the unfold over the operational model. This map is called the *universal semantics* (Turi and Plotkin (1997)).

$$sem :: \forall \Sigma\, b.\ (Functor\ \Sigma, Functor\ b) \Rightarrow (\forall x.\ \Sigma\ (b\ x) \to b\ (\Sigma\ x)) \to \mu\Sigma \to \nu b$$
$$sem\ \lambda\ t = (\!|\ desem\ \lambda\ |\!)\ t = [\![ opsem\ \lambda ]\!]\ t$$

As shorthand, we use semantic brackets for this definition: $[\![t]\!]\lambda = sem\ \lambda\ t$.

As a last remark of this section, it is worthwhile to notice the correspondence between distributive laws and certain *rule formats*. The kind of distributive laws we have seen in section, in particular, correspond to Structural Operational Semantics (SOS) (Plotkin (2004)). For example, for the behavioral functor *BHV* and a generic signature functor the distributive law yields an operational semantics of the following shape:

$$\frac{x_1 \overset{l_1}{\rightsquigarrow} x_1' \quad \dots \quad x_n \overset{l_n}{\rightsquigarrow} x_n'}{\sigma(x_1, \cdots, x_n) \overset{l}{\rightsquigarrow} \sigma'(y_1, \cdots, y_m)} \tag{3.1}$$

for a set of meta-variables $\{y_1 \dots, y_m\} \subseteq \{x_1', \dots, x_n'\}$, constructors from the signature $\sigma, \sigma' \in \Sigma$ and labels $\{l_1, \dots, l_n, l\} \in L$. Notice that the distributive law does not force the labels in the conclusions to belong to those in the premises, but there are however restrictions on the variables in the target belonging to the targets in the premises. We will see in the next sections that these restriction limit the expressivity of the languages we can model.

## 4 Full Abstraction for a Simple Non-Deterministic Arithmetic Language

The language we introduced in Section 3 is a basic language for talking about streams. We would like now to implement a fully abstract interpretation for the language we have introduced in Section 1, which is the language of arithmetic expressions originally introduced by Hutton (1998), also known as Hutton's razor.

In Section 4.1 we introduce Hutton's razor while in Section 4.2 we show how to give a fully abstract semantics and in Section 4.3 we abstract to copointed functors to define the behavior of these kind of languages. The full general case here is deferred to Section 5.

### *4.1 Hutton's razor*

This language has the same syntax we have seen in Section 3 so we can just reuse the signature functor *ValAddF*. The operational semantics of Hutton's razor is defined using the following inductive relation $(\to) \subseteq \mu\Sigma \times \mu\Sigma$:

$$\frac{}{\mathtt{add}(\mathtt{val}(n), \mathtt{val}(m)) \rightsquigarrow \mathtt{val}(n+m)} \qquad \frac{t_1 \rightsquigarrow t_1'}{\mathtt{add}(t_1, t_2) \rightsquigarrow \mathtt{add}(t_1', t_2)}$$

$$\frac{t_2 \rightsquigarrow t_2'}{\mathtt{add}(t_1, t_2) \rightsquigarrow \mathtt{add}(t_1, t_2')}$$

The semantics of this language is clearly non-deterministic as the operation `add` can perform two different reductions when the inner term is not `val`. Thus, to model it, we use an observation function for the functor $\mathscr{P}_{\text{fin}}(-)$ which we implement using finite lists of programs. These lists should be compared modulo duplication and swapping of elements. With this in mind the implementation of the inference rules above is the following:

$$\begin{array}{ll}
\mathit{smallstep} :: \mu\mathit{ValAddF} \to [\mu\mathit{ValAddF}] & \\
\mathit{smallstep}\ (\mathit{In}\ (\mathit{Val}\ n)) & = [\,] \\
\mathit{smallstep}\ (\mathit{In}\ (\mathit{Add}\ (\mathit{In}\ (\mathit{Val}\ n))\ (\mathit{In}\ (\mathit{Val}\ m)))) & = [(\mathit{val}\ (n+m))] \\
\mathit{smallstep}\ (\mathit{In}\ (\mathit{Add}\ t_1\ t_2)) & = [\mathit{add}\ t_1'\ t_2 \mid t_1' \leftarrow \mathit{smallstep}\ t_1] \\
 & \quad +\!\!+\ [\mathit{add}\ t_1\ t_2' \mid t_2' \leftarrow \mathit{smallstep}\ t_2]
\end{array}$$

Notice how the base case for *val* is empty as there is no reduction of the *val* case.

There are essentially two problems with this implementation. The first is that the unfold over this observation function yields the finitely branching *empty* trees. It is an easy computation to perform by unfolding the definition of, for example, $[\![\mathit{smallstep}]\!]\ (\mathit{add}\ (\mathit{val}\ n)\ (\mathit{val}\ m))$. Intuitively, this because after one step of computation we end up with the value $[\mathit{val}\ (n+m)]$ by definition of the second case and if we perform another step of computation by applying $[\![\mathit{smallstep}]\!]$ on the programs of the list, by the first case these (in this case we have just one program) will reduce to the empty list.

The second problem is that, in order to reap the benefits of Turi and Plotkin's approach we would need a parametric distributive law. This is rather problematic because, in the implementation of $\lambda$ below, it is not clear what should go in the holes "_":

$$\begin{array}{l}
\lambda :: \mathit{ValAddF}\ ([x]) \to [(\mathit{ValAddF}\ x)] \\
\lambda\ (\mathit{Add}\ (n :< \mathit{xs})\ (m :< \mathit{ys})) = ([\mathit{Add}\ x\ \_ \mid x \leftarrow \mathit{xs}]\ +\!\!+\ [\mathit{Add}\ \_\ y \mid y \leftarrow \mathit{ys}])
\end{array}$$

In fact, in the case of $\mathit{Add}\ (n :< \mathit{xs})\ (m :< \mathit{ys})$ the two subprograms have already generated their list of outputs of type $x$ and since $x$ is universally quantified we do not know which element of $x$ actually generated this reduction which is what we need to complete the holes.

### 4.2 Full abstraction for the razor

The first problem can be addressed by tweaking the definition of the behavioral functor. The idea is to make explicit when the semantics is proceeding and when it stops. In particular, we would like to map *val n* to the number *n* so that the reduction semantics for values is not empty.

Thus, the first tweak is to use the behavioral functor $\mathbb{N} + \mathscr{P}_{\text{fin}}(-)$ where the left injection into the sum is the when the semantics stops with a number and the right injection is when the semantics is reducing (non-deterministically) to a program. This can be implemented as follows:

$$\textbf{data}\ \mathit{StopAndGo}\ k = \underline{\mathit{Nat}} \mid \mathit{Step}\ [k]\ \textbf{deriving}\ \mathit{Functor}$$

Notice here that the underscore is the constructor of the data type.

The second tweak is more is more profound as we need to modify the type of the distributive law. In particular, we would like to retain a copy of the input before it gets destructured. To do this, we pair the type *StopAndGo x* with an additional *x* which is supposed to retain the original copy of the input. Notice that the resulting type $(x, \mathit{StopAndGo}\, x)$ is also a *Functor* on $x$.

$\rho :: \forall x.\ \mathit{ValAddF}\ (x, \mathit{StopAndGo}\, x) \to \mathit{StopAndGo}\ (\mathit{ValAddF}\ x)$

This new structure we have just inserted ensures that when the $\rho$-rule needs to repeat the input as is, it just needs to use the left projection.

However, it has to be noted that this is not a distributive law anymore since $(x, \mathit{StopAndGo}\, x)$ and $\mathit{StopAndGo}\, x$ are not the same. To avoid confusion, from here on we call this type of parametric function a $\rho$-rule (following original work by Turi and Plotkin). Notice also that this $\rho$-rule not being a distributive law has some theoretical consequences that we are going to need to address. For the time being let us look at how we can quickly implement the operational semantics we are after. So consider first the following implementation of a $\rho$-rule

$$\begin{array}{ll}
\rho\ (\mathit{Val}\, n) & = \underline{n} \\
\rho\ (\mathit{Add}\ (\_, \underline{n})\qquad (\_, \underline{m})) & = \underline{n+m} \\
\rho\ (\mathit{Add}\ (x_1, (\mathit{Step}\, xs_1))\ (x_2, (\mathit{Step}\, xs_2))) & = \mathit{Step}\ ([\mathit{Add}\, x_1\, x_2' \mid x_2' \leftarrow xs_2] +\!\!+ \\
 & \quad [\mathit{Add}\, x_1'\, x_2 \mid x_1' \leftarrow xs_1]) \\
\rho\ (\mathit{Add}\ (x_1, \_)\ (\_, (\mathit{Step}\, xs_2))) & = \mathit{Step}\ [\mathit{Add}\, x_1\, x_2' \mid x_2' \leftarrow xs_2] \\
\rho\ (\mathit{Add}\ (\_, (\mathit{Step}\, xs_1))\ (x_2, \_)) & = \mathit{Step}\ [\mathit{Add}\, x_1'\, x_2 \mid x_1' \leftarrow xs_1]
\end{array}$$

Note how this additional argument plays the role of copy of the input, for example in the third line of this definition where $x_1$ is the original program that can reduced to a list of programs $xs_1$ and in this case, since we are not interested in this information we may retain the original program. The relational definition corresponding to this $\rho$-rule is the following:

$$\frac{}{\mathit{val}\, n \rightsquigarrow \underline{n}} \qquad \frac{t_1 \rightsquigarrow \underline{n} \qquad t_2 \rightsquigarrow \underline{m}}{\mathit{add}\, t_1\, t_2 \rightsquigarrow \underline{n+m}} \qquad \frac{t_1 \rightsquigarrow t_1'}{\mathit{add}\, t_1\, t_2 \rightsquigarrow \mathit{add}\, t_1'\, t_2} \qquad \frac{t_2 \rightsquigarrow t_2'}{\mathit{add}\, t_1\, t_2 \rightsquigarrow \mathit{add}\, t_1\, t_2'}$$

It is probably worth noticing how we started with a set of rules, the arithmetic expression language by Hutton, and using distributive laws forced us to consider a slight different version of the operational semantics. This kind of format corresponds to the copointed structural operational semantics (Copointed SOS) due to the use of the copointed functor in the distributive law.

### *4.3 On CoPointed Functors*

The type $(x, b\, x)$ is a *Functor* on $x$ and, in particular, is the *free copointed functor* for $b$. The terminology is borrowed from its dual, the *free pointed functor Either* $x\ (b\, x)$ since $b$ is augmented with an extra point.

**type** $b_\times\ x = (x, b\, x)$

The type $\cdot_\times$ is clearly a functor with the obvious *pmap* function:

$\mathit{pmap} :: \mathit{Functor}\, b \Rightarrow (x \to y) \to b_\times\, x \to b_\times\, y$
$\mathit{pmap} f\ (x, y) = (f\, x, \mathit{fmap} f\, y)$

There is also an obvious counit map which is simply returning the copy of the input:

$\varepsilon :: b_\times\ a \to a$
$\varepsilon\ (x, \_) = x$

Given an observation function for the functor $b$, say $g :: a \to b\ a$, an *observation function for the copointed functor* $b_\times$ is a function of type $id \wedge g :: a \to b_\times\ a$ which pairs up $g$ with the identity on $a$. In other words, observations for copointed functors and their respective functors are in one-to-one correspondence[1]. This is implemented via $\lfloor\cdot\rfloor_\times$ and $\lceil\cdot\rceil_\times$

$\lfloor\cdot\rfloor_\times :: (Functor\ g) \Rightarrow (a \to g_\times\ a) \to a \to g\ a$
$\lfloor ccoalg\rfloor_\times\ x = \mathbf{let}\ (\_, c') = ccoalg\ x\ \mathbf{in}\ c'$
$\lceil\cdot\rceil_\times :: (Functor\ g) \Rightarrow (a \to g\ a) \to a \to g_\times\ a$
$\lceil g\rceil_\times = id \wedge g$

where $\wedge$ is

$(\wedge) :: (x \to a) \to (x \to b) \to (x \to (a, b))$
$(f \wedge g)\ x = (f\ x, g\ x)$

Now we are looking at repeating the development that we did in Section 3. To do this, we will first need to turn the $\rho$-rule into a distributive law. However, to be able to model more complex languages we will need what have been called GSOS (for Guarded Structural Operational Semantics) which subsume the copointed $\rho$-rules. Therefore, we will introduce GSOS before giving the general method for constructing the semantics.

## 5 Full Abstraction for CCS

As a bigger case study we implement a fully abstract semantics for Milner's *Calculus of Communicating Systems* (CCS) (Milner, 1980). We first recap CCS (Section 5.1). We then explain our choice of the behavior functor (Section 5.2) and encode CCS' operational rules as a $\rho$-rule (Section 5.3), resulting in a fully abstract semantics for CCS, and finally, in Section 5.5 we demonstrate how the semantics can be used for stream programming through some programming examples.

### *5.1 Syntax and Operational Rules of CCS*

The syntax of value-passing CCS is as follows (relabelling and conditionals of CCS (Milner, 1980) are omitted for simplicity, but they can be straightforwardly included):

$$P := \bar{c}(v).P \mid c(x).P_x \mid P_1 + P_2 \mid \mathtt{nil} \mid P_1 \parallel P_2 \mid \mathtt{rep}(P) \mid P \backslash c \qquad (5.1)$$

A CCS program, conventionally called a *process*, $\bar{c}(v).P$ *sends* a value $v \in \mathbb{V}$ to the channel $c \in Chan$ from some fixed set $Chan$ of channels, and continues as $P$. Similarly, for a family of processes $\{P_x\}_{x\in\mathbb{V}}$, the process $c(x).P_x$ receives a value—bound to the variable $x$—from the channel $c \in Chan$, and continues as $P_x$. The *sum* $P_1 + P_2$ of $P_1$ and $P_2$ can act as both $P_1$ and $P_2$, and the inactive process `nil` does nothing. The *parallel composition* $P_1 \parallel P_2$ act as $P_1$ and $P_2$ concurrently, possibly communicating with each other. The process

[1] This is a consequence of the fact that the category $B$-Alg is isomorphic to the category Id $\times$ $B$-Alg of algebras for copointed functors

$$\frac{}{\bar{c}(v).P \overset{\bar{c}(v)}{\rightsquigarrow} P} \qquad \frac{v \in \mathbb{V}}{c(x).P_x \overset{c(v)}{\rightsquigarrow} P_v} \qquad \frac{i \in \{0,1\} \qquad P_i \overset{a}{\rightsquigarrow} P}{P_0 + P_1 \overset{a}{\rightsquigarrow} P} \qquad \frac{P \overset{a}{\rightsquigarrow} P'}{P \parallel Q \overset{a}{\rightsquigarrow} P' \parallel Q}$$

$$\frac{Q \overset{a}{\rightsquigarrow} Q'}{P \parallel Q \overset{a}{\rightsquigarrow} P \parallel Q'} \qquad \frac{P \overset{c(v)}{\rightsquigarrow} P' \qquad Q \overset{\bar{c}(v)}{\rightsquigarrow} Q'}{P \parallel Q \overset{\tau}{\rightsquigarrow} P' \parallel Q'} \qquad \frac{P \overset{\bar{c}(v)}{\rightsquigarrow} P' \qquad Q \overset{c(v)}{\rightsquigarrow} Q'}{P \parallel Q \overset{\tau}{\rightsquigarrow} P' \parallel Q'}$$

$$\frac{P \overset{a}{\rightsquigarrow} P'}{\mathtt{rep}(P) \overset{a}{\rightsquigarrow} P' \parallel \mathtt{rep}(P)} \qquad \frac{P \overset{a}{\rightsquigarrow} P' \qquad a \notin \{c(v), \bar{c}(v)\}}{P \backslash c \overset{a}{\rightsquigarrow} P' \backslash c}$$

Fig. 1: Operational semantics for CCS

$\mathtt{rep}(P)$ is the *replication* of arbitrarily many copies of $P$ in parallel. Finally, $P \backslash c$ *restricts* the communication on channel $c$ between $P$ and the ambient environment.

The (small step) operational semantics of CCS is inductively defined in Figure 1 as as a ternary relation $(\rightarrow) \subseteq P \times \mathsf{Act} \times P$, where $\mathsf{Act}$ is the following set of *actions*:

$$\mathsf{Act} = \{\bar{c}(v) \mid c \in \mathit{Chan}, v \in \mathbb{V}\} \cup \{c(v) \mid c \in \mathit{Chan}, v \in \mathbb{V}\} \cup \{\tau\} \tag{5.2}$$

The intuition is that $P \overset{\bar{c}(v)}{\rightsquigarrow} Q$ means that $P$ sends a value $v$ to channel $c$ and becomes $Q$; similarly $P \overset{c(v)}{\rightsquigarrow} Q$ stands for $P$ receiving $v$ and becoming $Q$; and finally there is a special action $\tau$, called the *silent* action, for which $P \overset{\tau}{\rightsquigarrow} Q$ means that some communication succeeds inside $P$, which is not visible to the ambient environment.

### 5.2 The choice of the behavior functor

The non-deterministic nature of the operational semantics in Figure 1 is due to the non-deterministic $(P + Q)$ and parallel $(P \parallel Q)$ operators. Thus, we need to use the finite powerset functor $\mathscr{P}_{\mathrm{fin}}(-)$ to model transitions to a finite set of programs as we did in the previous section.

The reason we used the *finite* powerset functor (or the finite lists over a type) is rather theoretical: there is no fixed-point for the powerset functor $\mathscr{P}(-)$ which we need to construct the codomain for the interpretation function[1].

Moreover, we also need to capture the action coming from the transition rules of the form $P \overset{a}{\rightsquigarrow} Q$. Thus, the obvious choice for $B$ would be $\mathscr{P}_{\mathrm{fin}}(\mathsf{Act} \times -)$.

However, consider the process $c(x).P_x$ receiving a value. Its possible transitions $\{(c(v), P_v) \mid v \in \mathbb{V}\}$ are an *infinite* subset of $\mathsf{Act} \times P$. However, $\mathscr{P}_{\mathrm{fin}}(-)$ is the *finite* subset functor and it is thus not suitable for our purposes.

To overcome this problem we consider the following behavioral functor instead:

$$\mathscr{P}_{\mathrm{fin}}((\mathit{Chan} \times \mathbb{V} \times -) + (\mathit{Chan} \times (-)^{\mathbb{V}}) + 1)$$

Intuitively, an observable behavior has three possible shapes given by the coproduct. A “send” rule produces a channel-value pair along with the target process $(\mathit{Chan} \times \mathbb{V} \times X)$.

[1] This result is due to Cantor who proved that for every set $X$, $X \ncong \mathscr{P}(X)$.

A "receive" rule produces a channel together with a *function* mapping each value $v$ to its process $P_v$ $(\mathit{Chan} \times X^{\mathbb{V}})$). Finally, a "synchronisation" between two processes is presented by the type 1.

## 5.3 The choice of the rule format

Now we need to encode the operational rules in Figure 1 in a $\rho$-rule as we did in the previous sections. However, neither the simple rule format (3.1) nor the copointed one (5.3) here are suitable for implementing the rules of CCS. In particular, a limitation of both rule formats is that the target in the conclusion must have *exactly one* constructor $\sigma'$ applied to the meta-variables, rather than multiple nested constructors.

The operational rules of arithmetic language in Section 3 and Section 4 happen to meet this requirement, but many rules of CCS in Figure 1 do not. For example, the rule for $\bar{c}(v).P \overset{\bar{c}(v)}{\rightsquigarrow} P$ has no constructors applied to $P$, but the rule for $\texttt{rep}(P) \overset{a}{\rightsquigarrow} P' \parallel \texttt{rep}(P)$ has two constructors applied to $P$ and $P'$, namely $\parallel$ and $\texttt{rep}$.

To overcome this limitation, we need to generalise the type of $\rho$-rules once again by allowing arbitrarily many constructors in the target of transitions. We do this by replacing $\Sigma$ in the target of $\rho$ with the *free monad* over the signature $\Sigma$. Ultimately, the type signature of $\rho$ should be as follows:

$$\rho :: \forall x.\ \Sigma\ (b_{\times}\ x) \to b\ (\Sigma^*\ x)$$

where $\Sigma^*$ is the free monad. We now briefly recall its operations.

### 5.3.1 Free Monads

Given a functor $\Sigma$, the free monad $\Sigma^*$ over $\Sigma$ is defined by the following datatype:

$$\mathbf{data}\ \Sigma^*\ a = \mathit{Var}\ a \mid \mathit{Op}\ (\Sigma\ (\Sigma^*\ a))$$

which comes with the following recursion scheme:

$$\begin{aligned}
&\mathit{eval} :: \mathit{Functor}\,\Sigma \Rightarrow (\Sigma\,b \to b) \to (a \to b) \to \Sigma^*\ a \to b\\
&\mathit{eval}\ \mathit{alg}\ \mathit{gen}\ (\mathit{Var}\ v) = \mathit{gen}\ v\\
&\mathit{eval}\ \mathit{alg}\ \mathit{gen}\ (\mathit{Op}\ c)\ \ = \mathit{alg}\ (\mathit{fmap}\ (\mathit{eval}\ \mathit{alg}\ \mathit{gen})\ c)
\end{aligned}$$

Using *eval*, the monad instance of $\Sigma^*$ is defined as

$$\begin{aligned}
&\mathit{return} :: a \to \Sigma^*\ a && (\gg\!=) :: \mathit{Functor}\,\Sigma \Rightarrow \Sigma^*\ a \to (a \to \Sigma^*\ b) \to \Sigma^*\ b\\
&\mathit{return} = \mathit{Var} && x \gg\!= f = \mathit{eval}\ \mathit{Op}\ f\ x
\end{aligned}$$

The recursion scheme above is justified by the fact that the free monad over a signature functor $\Sigma$ is equivalent to the least-fixed point of $\Gamma X = A + \Sigma X$ satisfying $\mu\Gamma = A + \Sigma(\mu\Gamma)$.

For every $\Sigma$-algebra we can construct a $\Sigma^*$-algebra by evaluating the syntax using the identity function and the $\Sigma$-algebra and viceversa

$$\begin{aligned}
&\lfloor\cdot\rfloor_* :: \mathit{Functor}\,\Sigma \Rightarrow (\Sigma^*\ a \to a) \to \Sigma\,a \to a && \lceil\cdot\rceil_* :: \mathit{Functor}\,\Sigma \Rightarrow (\Sigma\,a \to a) \to \Sigma^*\ a \to a\\
&\lfloor f\rfloor_* = f \cdot \mathit{Op} \cdot \mathit{fmap}\ \mathit{Var} && \lceil g\rceil_* = \mathit{eval}\ g\ \mathit{id}
\end{aligned}$$

The maps $\lceil\cdot\rceil_*$ and $\lfloor\cdot\rfloor_*$ witness the fact that $\Sigma$-algebras and the $\Sigma^*$-algebras for the monad $\Sigma^*$ are in one-to-one correspondence[1].

### 5.3.2 The $\rho$-rule for CCS

As a first step we need to implement the syntax of the language using a signature functor. This will be parameterised by a type $\mathbb{C}$ of channel labels. As usual, $\mu(CCS\,l)$ will be the type of inductively generated CCS programs (5.1).

$\textbf{type}\ \mathbb{V} = \mathbb{N}$
$\textbf{type}\ \mathbb{C} = String$
$\textbf{data}\ CCS\,x = Send\ \mathbb{C}\ \mathbb{V}\ x \mid Recv\ \mathbb{C}\ (\mathbb{V} \to x) \mid Sum\,x\,x$
$\qquad\qquad\quad \mid\ Nil \mid Par\,x\,x \mid Rep\,x \mid Restrict\ \mathbb{C}\ x$

Secondly, we need to define the set of actions

$\textbf{data}\ Act\,x = ActS\ \mathbb{C}\ \mathbb{V}\ x \mid ActR\ \mathbb{C}\ (\mathbb{V} \to x) \mid Silent\,x$
$\textbf{newtype}\ Acts\,x = Acts\ \{unActs :: [Act\,x]\}$

The following two definitions are simply the liftings of concatentation ($+\!\!+$) and *filter* respectively to the type *Acts*

$bapp :: Acts\,x \to Acts\,x \to Acts\,x$
$bapp\ bs\ bs' = Acts\ (unActs\ bs +\!\!+ unActs\ bs')$
$bfilter :: (Act\,x \to 2) \to Acts\,x \to Acts\,x$
$bfilter\ p = Acts \cdot filter\ p \cdot unActs$

Now the operational rules of CCS in Figure 1 can be encoded as a $\rho$-rule of type

$\rho_{CCS} :: CCS\ (x, Acts\,x) \to Acts\ (CCS^*\ x)$

The rules for sending and receiving are as follows

$\rho_{CCS}\ (Send\ c\ v\ (x, \_)) = Acts\ [ActS\ c\ v\ (return\ x)]$
$\rho_{CCS}\ (Recv\ c\ k) \qquad\quad = Acts\ [ActR\ c\ (fmap\ (\lambda\,(x, \_) \to return\,x)\ k)]$

where the sending process has one single action, i.e. the send action along with the channel, the value and the copy of the remaining computation and the receiving process has one single action as well, i.e. the receive action with the channel and the continuation that always return the process after receiving the value $v$.

The case of the 0 (nil) process is just the empty set of actions

$\rho_{CCS}\ Nil = Acts\ [\,]$

The choice process $(P_0 + P_1)$ is the concatenation of the list of actions from the first process with the second process, while the restriction $(P\backslash c)$ requires some additional functions. The first is *check* implementing the condition $a \notin \{c(v), \bar{c}(v)\}$ (Figure 1) and the other is *bfilter* filtering out all the actions that satisfies the condition implemented by *check*

[1] Here we mean that algebras *for a monad* $\Sigma^*$ are isomorphic to algebras for $\Sigma$

$\rho_{CCS}\ (Sum\ (\_, b)\ (\_, b')) = fmap\ return\ b +\!\!+_B fmap\ return\ b'$
$\rho_{CCS}\ (Restrict\ c\ (\_, b)) \quad = bfilter\ check\ (fmap\ (\lambda y \to Op\ (Restrict\ c\ (return\ y)))\ b)$ **where**
$\quad check\ (ActS\ c'\ \_\ \_) = c \not\equiv c'$
$\quad check\ (ActR\ c'\ \_) \quad = c \not\equiv c'$
$\quad check\ (Silent\ \_) \qquad = True$

The parallel case (||) implements four rules in one so it is perhaps not surprising that its definition is a bit more involved

$\rho_{CCS}\ (Par\ xb\ xb') = lmerge\ xb\ xb' +\!\!+_B lmerge\ xb'\ xb$
$\quad$ **where** $lmerge\ (x, b)\ (x', b') = fmap\ (\lambda y \to Op\ (Par\ (return\ y)\ (return\ x')))\ b +\!\!+_B$
$\qquad Acts\ [Silent\ (Op\ (Par\ (return\ y)\ (return\ (k\ m))))$
$\qquad\quad |\ (ActS\ c\ m\ y) \leftarrow unActs\ b, (ActR\ c'\ k) \leftarrow unActs\ b', c \equiv c']$

here *lmerge* takes the initial abstract processes $x$ and $x'$ and their respective list of actions $b$ and $b'$ and it produces the union of the list of actions coming from a one-step reduction of $x$ and the list of silent actions coming from the coupling of a send from $x$ and a receive from $x'$. At the top level, the rule outputs the union of the results of merging the outputs of $x$ and $x'$ and their mirror case.

The rule for `rep` is simpler to implement but crucially requires the structure from the free monad as it returns two levels of syntax, the first is the *Par* constructor and the second constructor is *Rep*

$\rho_{CCS}\ (Rep\ (x, b)) = fmap\ (\lambda y \to Op\ (Par\ (return\ y)\ (Op\ (Rep\ (return\ x)))))\ b$

At this point, we need to recover the distributive law for GSOS rules from which we can construct the universal semantics.

### 5.4 The general case: recovering distributive laws

The $\rho$-rule for CCS can be generalised in the following way. For a signature functor $\Sigma$ and a behavior functor $b$ the $\rho$-rule has the following type:

$\rho :: \forall x.\ \Sigma\ (b_\times\ x) \to \Sigma^*\ (b\ x)$

This parametric function gives raise to a rule format called GSOS

$$\frac{x_1 \overset{l_1}{\leadsto} x_1' \ldots x_n \overset{l_n}{\leadsto} x_n'}{\sigma(x_1, \cdots, x_n) \overset{a}{\leadsto} t(y_1, \cdots, y_m)} \tag{5.3}$$

for meta-variables $\{y_1 \ldots, y_m\} \subseteq \{x_1, \ldots, x_n\} \cup \{x_1', \ldots, x_n'\}$ and constructors $\sigma, \in \Sigma$ and programs $t \in \Sigma^*$.

Now we need to recover a distributive law for the free monad $\Sigma^*$ over $B_\times$ from $\rho$. We can do this by using the recursion scheme for the free monad *eval*:

$rhoToLambda :: \forall \Sigma\ b.\ (Functor\ \Sigma, Functor\ b) \Rightarrow (\forall x.\ \Sigma\ (b_\times\ x) \to b\ (\Sigma^*\ x))$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to (\forall x.\ \Sigma^*\ (b_\times\ x) \to b_\times\ (\Sigma^*\ x))$
$rhoToLambda\ \rho = eval\ alg\ (pmap\ return)$ **where**
$\quad alg :: \Sigma\ (b_\times\ (\Sigma^*\ x)) \to b_\times\ (\Sigma^*\ x)$
$\quad alg = (Op \cdot (fmap\ \varepsilon)) \wedge ((fmap\ (\ggg id)) \cdot \rho)$

With the distributive law *rhoToLambda* $\rho$, the universal operational *opsem* and denotational semantics *desem* induced by $\rho$ can be defined similarly to what we did in the previous sections. Only here we need to lift and downlift the algebras using the operators $\lceil\cdot\rceil_*$, $\lfloor\cdot\rfloor_*$, $\lceil\cdot\rceil_\times$ and $\lfloor\cdot\rfloor_\times$ to make the types match

$$\begin{aligned}
&\mathit{opsem} :: \forall \Sigma\, b.\ (\mathit{Functor}\,\Sigma, \mathit{Functor}\, b) \Rightarrow (\forall x.\ \Sigma\,(b_\times\, x) \to b\,(\Sigma^*\, x)) \to \mu\Sigma \to b\,(\mu\Sigma)\\
&\mathit{opsem}\ \rho = \lfloor (\!| \mathit{alg} |\!) \rfloor_\times\ \textbf{where}\\
&\quad \mathit{alg} :: \Sigma\,(b_\times\,(\mu\Sigma)) \to b_\times\,(\mu\Sigma)\\
&\quad \mathit{alg} = \lfloor \mathit{pmap}\,(\lceil \mathit{In} \rceil_*) \cdot (\mathit{rhoToLambda}\,\rho) \rfloor_*
\end{aligned}$$

and similarly for the denotational semantics:

$$\begin{aligned}
&\mathit{desem} :: \forall \Sigma\, b.\ (\mathit{Functor}\,\Sigma, \mathit{Functor}\, b) \Rightarrow (\forall x.\ \Sigma\,(b_\times\, x) \to b\,(\Sigma^*\, x)) \to \Sigma\,(\nu b) \to \nu b\\
&\mathit{desem}\ \rho = \lfloor [\![ \mathit{coalg} ]\!] \rfloor_*\ \textbf{where}\\
&\quad \mathit{coalg} :: \Sigma^*\,(\nu b) \to b\,(\Sigma^*\,(\nu b))\\
&\quad \mathit{coalg} = \lfloor \mathit{rhoToLambda}\,\rho \cdot \mathit{fmap}\,(\lceil \mathit{out} \rceil_\times) \rfloor_\times
\end{aligned}$$

### 5.4.1 *The Full Abstraction Argument*

Since $\lambda$ is distributive, the functions $(\!|\, \mathit{desem}\,\rho\, |\!)$ and $[\![\mathit{opsem}\,\rho]\!]$ always coincide (Hinze and James, 2011), and, since $[\![\mathit{opsem}\,\rho]\!]$ is an unfold it is always fully abstract. The following recursion scheme interprets a language $\mu\Sigma$ provided an abstract rule $\rho$ (as usual, we write $[\![t]\!]\rho$ for *sem* $\rho$ *t*):

$$\begin{aligned}
&\mathit{sem} :: \forall \Sigma\, b.\ (\mathit{Functor}\,\Sigma, \mathit{Functor}\, b) \Rightarrow (\forall x.\ \Sigma\,(b_\times\, x) \to b\,(\Sigma^*\, x)) \to \mu\Sigma \to \nu b\\
&\mathit{sem}\ \rho\ t = (\!|\, \mathit{desem}\,\rho\, |\!)\, t \quad \text{-- or } [\![\mathit{opsem}\,\rho]\!]\, t
\end{aligned}$$

Applying *sem* to $\rho_{CCS}$, we obtain the following function that gives semantics to CCS programs as their (global) behaviours:

$$\begin{aligned}
&\mathit{sem}_{CCS} :: \mu CCS \to \nu Acts\\
&\mathit{sem}_{CCS} = \mathit{sem}\ \rho_{CCS}
\end{aligned}$$

In the rest of this section, we continue our example of CCS with this recursion scheme, and show how it be used for functional programming.

## 5.5 *Programming Examples*

In the rest of this section, we demonstrate how the semantics induced by $\rho$-rules is useful through programming examples in CCS. For convenience, we define the following helper functions wrapping constructors with *In*:

$$\begin{array}{llll}
p \,||\, q & = \mathit{In}\,(\mathit{Par}\,p\,q); & \mathit{nil} & = \mathit{In}\,\mathit{Nil}\\
\mathit{send}\,c\,m\,p & = \mathit{In}\,(\mathit{Send}\,c\,m\,p); & \mathit{sum}\,p\,q & = \mathit{In}\,(\mathit{Sum}\,p\,q)\\
\mathit{rep}\,p & = \mathit{In}\,(\mathit{Rep}\,p); & \mathit{res}\,c\,p & = \mathit{In}\,(\mathit{Restrict}\,c\,p)\\
\mathit{recv}\,c\,k & = \mathit{In}\,(\mathit{Recv}\,c\,k); & \mathit{res}'\,\mathit{ls}\,p & = \mathit{foldr}\,\mathit{res}\,p\,\mathit{ls}
\end{array}$$

These are just the constructors of the language post-applied to *In* while *res*′ is the restriction of a list of channels *ls* to the process *p*.

**Example 5.1.** We can use CCS to do stream programming. The following program sends all natural numbers to the channel `"output"` one by one:

```
nats :: μCCS
nats = res "i" (send "i" 0 nil || rep iter) where
  iter = recv "i" (λi → send "output" i (send "i" (i + 1) nil))
```

A process *iter* receives the current value from channel `"i"` and sending it to the `"output"` channel. The process *iter* is replicated as infinitely many copies by *rep*, so *iter* can send $i + 1$ to another copy of itself to continue the iteration. Finally, there is a *send* `"i"` 0 *nil* to start the iteration, and *res* `"i"` restricts the communication on `"i"` inside the process, so other processes cannot meddle with the iteration, and can only observe the channel `"output"`.

Applying the semantics function to the process, we obtain $sem_{CCS}\ nats :: \nu Acts$, which is intuitively a (coinductive) tree whose edges are labelled with an action *Act*, and different paths stand for different possibilities of non-determinism. We can collect all the outputs as a list in such a tree using the following function:

```
outputs :: νActs → [𝕍]
outputs (Out° (Acts [])) = []
outputs (Out° (Acts bs)) = concat (map f bs) where
  f (ActS _ m bs') = m : outputs bs'
  f (ActR _ _)     = []
  f (Silent bs')   = outputs bs'
```

And indeed, $outputs\ (sem_{CCS}\ nats) = [0, 1, 2, 3, ...]$.

**Example 5.2.** The semantics $sem_{CCS}$ is useful for exploring the non-deterministic behaviour of concurrent systems, since all non-deterministic possibilities are recorded in the semantics $\nu Acts$. As an example, the following program implements a counter using the subprograms defined at the beginning of this section:

```
counter = res "init" (send "init" 0 nil || iter) where
  iter = rep (recv "init" (λv → sum
     (recv "rd" (\_ → send "count" v (send "init" v nil)))
     (recv "wt" (λi → send "init" i nil))))
```

Intuitively, *counter* is a mobile program storing a value called *init* and responding to read or write requests from other processes which would like to read or overwrite the contents of the store.

The cell can be read and modified by accessing the channels `"rd"`, `"wt"`, and `"count"` as follows:

```
read :: (ℕ → μCCS) → μCCS              write :: ℕ → μCCS → μCCS
read k = send "rd" 0 (recv "count" k)  write v p = send "wt" v p
```

Suppose there are two processes incrementing the counter concurrently without using a lock as shown in the following program:

$incr :: \mu CCS \to \mu CCS$
$incr\ p = read\ (\lambda v \to write\ (v+1)\ p)$
$counterTest = res'\ [\texttt{"rd"}, \texttt{"wt"}, \texttt{"count"}]\ (counter \mid\mid incr\ (incr\ nil)$
$\quad \mid\mid incr\ (incr\ (read\ (\lambda v \to send\ \texttt{"output"}\ v\ nil))))$

Then executing the semantics tells us all possible outcomes:

$$nub\ (outputs\ (sem_{CCS}\ counterTest)) = [4, 3, 2, 1]$$

where *nub* is a function which removes the duplicates from a list.

## 6 Related Work

From the categorical point of view, the theory of coalgebraic systems have been thoroughly studied (Rutten and Turi, 1993; Turi and Plotkin, 1997; Jacobs and Rutten, 2012; Rutten, 2000; Staton, 2011) and their connection with Structural Operational Semantics (SOS) (Plotkin, 2004) have been thorougly expounded by Klin (2011).

In functional programming, Turi and Plotkin's distributive laws (Turi and Plotkin, 1997) have been proven to have wide range of applications: to prove the unique fixed-point principle correct (Hinze and James, 2011); to define operational semantics modularly (Jaskelioff et al., 2011); and to prove equality of sorting algorithms (Hinze et al., 2012). Hutton's work (Hutton, 1998) aimed a popularising the use of folds and unfolds for program semantics and the razor has been used throughout the literature to explain key ideas of programming languages design, semantics and compilers (Hutton, 1998, 2021; Bahr and Hutton, 2015; Hutton, 2021). In our work we take a slightly different approach. We tweak the razor to fit into the idea that the *denotational semantics is an algebra* on the semantic domain defined as unfold and the *operational semantics is a coalgebra* defined as a fold on the syntax. Under this view the fold over the denotational semantics is equal to the unfold over the operational semantics and, by using distributive laws, full abstraction falls out for free. The proofs of these facts have been thoroughly spelled out by Hinze and James (2011) and Hinze et al. (2012) which, on the other hand, do not discuss on the full abstraction results.

## 7 Discussion

Recursion schemes were invented to ensure recursive definition are well-defined mathematically. However, recursion schemes can go even further. While folds are compositional interpretations, unfolds are fully abstract interpretations. In particular, they can "run" a transition system (or an automata) and, for a state, construct its trace such that equal traces correspond to operationally equivalent states.

When a transition system such as an operational semantics and a denotational semantics arise from a distributive law, the fold over the denotational semantics is equal to the unfold over the operational semantics.

## References

Bahr, P. & Hutton, G. (2015) Calculating correct compilers. *J. Funct. Program.* **25**.

Danielsson, N. A., Hughes, J., Jansson, P. & Gibbons, J. (2006) Fast and loose reasoning is morally correct. Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006. ACM.

Hinze, R. (2008) Functional pearl: streams and unique fixed points. Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008. ACM.

Hinze, R. & James, D. W. H. (2011) Proving the unique fixed-point principle correct: an adventure with category theory. Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011. ACM.

Hinze, R., James, D. W. H., Harper, T., Wu, N. & Magalhães, J. P. (2012) Sorting with bialgebras and distributive laws. Proceedings of the 8th ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2012, Copenhagen, Denmark, September 9-15, 2012. ACM. pp. 69–80.

Hutton, G. (1998) Fold and unfold for program semantics. Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998. ACM. pp. 280–288.

Hutton, G. (2021) It's easy as 1,2,3. Unpublished Manuscript.

Jacobs, B. & Rutten, J. (2012) An introduction to (co)algebra and (co)induction. In *Advanced Topics in Bisimulation and Coinduction*, Sangiorgi, D. & Rutten, J. J. M. M. (eds). vol. 52 of *Cambridge tracts in theoretical computer science*. Cambridge University Press. pp. 38–99.

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

Klin, B. (2011) Bialgebras for structural operational semantics: An introduction. *Theor. Comput. Sci.*

Meijer, E., Fokkinga, M. M. & Paterson, R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings. Springer. pp. 124–144.

Milner, R. (1980) *A Calculus of Communicating Systems*. vol. 92 of *Lecture Notes in Computer Science*. Springer.

Plotkin, G. D. (2004) A structural approach to operational semantics. *J. Log. Algebraic Methods Program.*

Rutten, J. J. M. M. (2000) Universal coalgebra: a theory of systems. *Theor. Comput. Sci.*

Rutten, J. J. M. M. & Turi, D. (1993) Initial algebra and final coalgebra semantics for concurrency. A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium, Noordwijkerhout, The Netherlands, June 1-4, 1993, Proceedings. Springer.

Staton, S. (2011) Relating coalgebraic notions of bisimulation. *Log. Methods Comput. Sci.*

Turi, D. & Plotkin, G. D. (1997) Towards a mathematical operational semantics. Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, June 29 - July 2, 1997. IEEE Computer Society.