1^{st} Year Transfer Report

Mauro Jaskelioff

20th October 2006

This document is a summary of the work I have done during my first year whilst researching on the modularity of structural operational semantics. I discuss why modularity is important in semantics and the shortcomings of structural operational semantics in this regard. I review the related literature and explain what I have achieved so far. Also, I outline some possible directions for future work.

1 Introduction

1.1 Modularity in semantics

Formal semantics of programming languages are an indispensable tool. Computer scientists and programmers use them to communicate, reason about, and guide the design and implementation of programming languages. However, in any real modern programming language, one expects to find advanced features like support for concurrency, communication between processes, exceptions and the like. As the complexity of a language increases, providing formal semantics for such a language can be a very laborious and error-prone task.

The usual way to tackle the building of a complex construction is to divide it into smaller, more manageable pieces, and then assemble those pieces together to obtain the composite whole. But, how can we do this for semantics? Can we really separate the components of the different features in a programming language?

In denotational semantics, it was shown by Moggi [26] that a concept from category theory —monads— is useful for structuring semantics. This idea was so successful that the programming language Haskell actually provides a special syntax for structuring programs with the help of monads. Liang, Hudak and Jones [22] use monads and monad transformers to structure interpreters. They present a myriad of features of programming languages and interpreters that can be defined by combining any of these features. While monads and monad transformers do not solve every possible problem of modularity, they are widely accepted and have been successfully applied to real problems.

On the other hand, for operational semantics [30], the different existent approaches to modularity have not been that successful. The most widely spread approach is that of Modular SOS [27], which although inspired by monads, takes a more concrete approach. The main contribution of MSOS is providing a new syntax for writing rules. Each rule is written only specifying the particular side-effect that one is interested in; the other side-effects are left unspecified. The side-effects are limited to input, output and state (non-determinism is intrinsic to the SOS approach so it does not need to be explicitly considered). One of the goals of Mosses was to improve modularity of SOS without losing its intuitive appeal, hence the concrete approach: the focus on the syntax for writing the rules and the fixed choice of effects. While one could argue that more effects could be added to the theory if they were needed, in MSOS there is a single way two languages can be put together. In the monad transformers approach, we obtain different effects if we compose monad transformers in a different order (for example, local versus global state), whereas in MSOS this order is fixed.

Another approach is that of Lämmel's Rule Evolution Kit [19], but this work is unsatisfactory for our purposes since it manipulates the rules that conform a structural operational semantics without taking into account its semantic content, focusing exclusively on the syntax of the rules. As it is stated in its name, this is a tool for rule manipulation, not semantic manipulation. Hence, it does not give us any insight on how to reason modularly about SOS.

If we want to not only define operational semantics, but also reason about them in a modular way, we need some formal relation between the properties of the components and the combined language. Neither of the previous approaches provides any support for this, except for the notion of conservative operational extension [1], which only ensures that a program in a language $\mathcal{L}_1$ is not going to change its behaviour if considered with the semantics of language $\mathcal{L}_1 \cup \mathcal{L}_2$. Clearly, there is much room for improvement.

All this evidence suggests that obtaining a theory of modularity in operational semantics, with enough flexibility to encompass real languages, and enough semantic content to be able reason to about them, is a much desirable and hard to obtain goal. For achieving this goal, it is essential to work with an adequate framework.

1.2 A Mathematical Operational Semantics

Plotkin [31] mentions that the original SOS notes had deliberately not been written in a theoretical framework, the idea was that theory would come later. He also mentions that Robin Milner produced some (unpublished) notes on a general approach to operational semantics for a given algebraic signature, and that his ideas were close to the De Simone format [6], a syntactic format on SOS rules for ensuring certain properties in the defined language.

As the years passed, more formats were discovered. Some of those formats were aimed at ensuring that a notion of equivalence associated to a SOS was a congruence. A relevant example of such a format is GSOS [5], which guarantees that strong bisimulation is a congruence for SOS written in that format. Particularly interesting was the introduction of the bialgebraic framework [36, 35], a categorical abstraction of GSOS. This framework, not only provides a new tool for obtaining rule formats, but also, since it is expressed in categorical terms, it provides a strong metatheory. Category theory [23] pervades denotational semantics [33, 38, 18, 34], so this framework also provides a connection between these different approaches to semantics: it has made possible to speak about denotational and operational semantics in the same language.

Since the introduction of the bialgebraic framework by Turi, many research papers have been published on the subject, expanding and clarifying its underlying theory and confirming its usefulness. Section 2 provides an overview of the bialgebraic framework as introduced by Turi and Plotkin, and a summary of the ideas in the aforementioned papers that may be relevant to our research.

1.3 Objectives

If we want the definition and maintenance of semantics of programming languages of considerable size to be manageable, modularity is essential. Structural operational semantics is a widespread approach but its modularity is rather poor, and the available solutions fall short in their support for formal reasoning and automation. The objective of this research is to:

- Obtain techniques for improving modularity of SOS that are based on a solid theory.
- Analyse how to use these techniques for the calculation of modular abstract machines.
- Apply these techniques to the construction of a modular compiler which would allow us to selectively combine different features of programming languages, emulating what Liang et. al. [22] did with monad transformers for constructing modular interpreters.

2 Literature on the bialgebraic framework

Operational semantics are a popular and intuitive way of specifying semantics, particularly in the area of process algebra, where concurrency, communication between processes and more recently, mobility are fundamental aspects.

For each operational semantics there is an associated notion of process equivalence, where two processes are regarded as equal if their observable behaviour is the same. Since we may have different notions of what is observable and what is not, there are also different notions of process equivalence.

For a notion of equivalence to be useful, it should be respected by the operations of the language. That is, we would like to not be able to distinguish equivalent processes when we plug them in some arbitrary context. We want the notion of equivalence to be a *congruence*.

Verifying that equivalence is a congruence is usually difficult, so many researchers started looking for general criteria under which this can be guaranteed. This criteria is usually expressed as some syntactic restriction on the format of the operational rules. GSOS is a well-known rule format which guarantees that behavioural equivalence (in this case, bisimulation) is a congruence.

2.1 Towards a Mathematical Operational Semantics

In [36], Turi provided a category-theoretic abstract formulation of well-behaved structural operational semantics. This presentation was revised and simplified by Turi and Plotkin in [35] and is known as *bialgebraic semantics*[1]. Every bialgebraic semantics guarantees that bisimulation is a congruence. Bialgebraic semantics can be obtained from abstract operational rules, natural transformations of the form

$$\Sigma(Id \times B) \to BT$$

for signature Σ, terms generated by the signature T, and behaviour B. They proved that there is a one-to-one correspondence between finitely-branching GSOS rules and abstract operational rules for the behaviour $BX = \mathcal{P}_f(Act \times X)$ on **Set**, corresponding to finitely branching trees with labels drawn from Act on the transitions (here $\mathcal{P}_f$ is the finite powerset functor and Act is a fixed set of labels).

This abstract construction has several advantages over concrete rule formats. By generalising to other categories and other behaviour endofunctors, other rule formats can be discovered. Moreover, just by dualising abstract GSOS rules, another rule format was found: safe tree rules. This connection between two different rule formats probably would not have been noticed if we had kept insisting on working with concrete rules. By abstracting from irrelevant details more connections come to light. As the example of safe tree rules shows, using category theory as a metatheory gives us a powerful and concise way of reasoning about our abstractions.

From abstract operational rules one obtains a distributive law between a monad and a comonad. It's precisely this distributive law which informs the definition of bialgebraic semantics, which we give in section 2.1.2. If we forget about abstract operational rules, and work at the level of abstraction of distributive laws, there is even more room for generalisation. We can regard the syntax as being *any* monad, and not just a freely-generated one, and behaviours can be *any* comonad, and not just a cofreely generated one. Then, we may consider adding equations to the signature (terms are interpreted as before but quotiented by some set of equations) and working with timed processes, whose behaviours are not, in general, cofreely generated by an endofunctor.

[1] In [36], two equivalent formulations were given: *functorial operational semantics* and *functorial denotational semantics*, although none of them were more "operational" or "denotational" than the other. In [35] the name *universal semantics* was suggested, but the name that caught on is the one presented here.

2.1.1 Adequacy and bisimulation as a congruence

Bisimulation is a congruence in bialgebraic semantics as a corollary of a stronger result: every bialgebraic semantics coinduces a denotational model that is adequate, in the following sense:

If Σ is the functor corresponding to the signature of a language, a denotational model can be seen as a Σ-algebra $[\![-]\!]\colon \Sigma M \to M$. The initial Σ-algebra is given by the programs of the language (closed terms $T0$) and the unique homomorphism from the initial Σ-algebra to the denotational model yields the initial algebra semantics.

$$\begin{tikzcd}
\Sigma T0 \arrow[r, "\Sigma(\mathsf{fold}([\![-]\!]))"] \arrow[d, "\cong"'] & \Sigma M \arrow[d, "{[\![-]\!]}"] \\
T0 \arrow[r, dashed, "\mathsf{fold}([\![-]\!])"'] & M
\end{tikzcd}$$

On the other hand, if we consider the operational model of closed programs $(\!| - |\!)\colon T0 \to BT0$ for a behaviour B, the abstract behaviour of programs will be given by coinduction, that is, by the corresponding final coalgebra semantics.

$$\begin{tikzcd}
T0 \arrow[r, dashed, "\mathsf{unfold}((\!|-|\!))"] \arrow[d, "(\!|-|\!)"'] & \nu B \arrow[d, "\cong"] \\
BT0 \arrow[r, "B(\mathsf{unfold}((\!|-|\!)))"'] & B(\nu B)
\end{tikzcd}$$

If we consider denotational models on the carrier of the final coalgebra νB, we arrive to the following definition of adequacy.

Definition 1 *A denotational model is* adequate[2] *with respect to an operational one when its initial algebra semantics* $\mathsf{fold}([\![-]\!])\colon T0 \to \nu B$ *is equal to the final coalgebra semantics* $\mathsf{unfold}((\!| - |\!))\colon T0 \to \nu B$ *corresponding to the operational model.*

This notion of adequacy can be expressed diagrammatically as in figure 1.

B-bisimulations

For every behaviour functor B we have an associated notion of behaviour equivalence called B-bisimulation. Turi and Plotkin showed that under some mild conditions on B, the final B-bisimulation is equivalent to the internal notion of equality in the carrier of the final coalgebra.

[2]This notion of adequacy is stronger that the usual notion of adequacy found in the literature, which only requires the final coalgebra semantics to be 'included' in the initial algebra semantics.

$$
\begin{tikzcd}
\Sigma T0 \arrow[r, "\Sigma\mathsf{fold}(\llbracket - \rrbracket)"] \arrow[d, "\cong"'] & \Sigma(\nu B) \arrow[d, "\llbracket - \rrbracket"] \\
T0 \arrow[r, dashed, shift left, "\mathsf{fold}(\llbracket - \rrbracket)"] \arrow[r, dashed, shift right, "\mathsf{unfold}(\llparenthesis - \rrparenthesis)"'] \arrow[r, phantom, "\equiv"] \arrow[d, "\llparenthesis - \rrparenthesis"'] & \nu B \arrow[d, "\cong"] \\
BT0 \arrow[r, "B(\mathsf{unfold}(\llparenthesis - \rrparenthesis))"'] & B(\nu B)
\end{tikzcd}
$$

Figure 1: Adequacy condition for bialgebraic semantics.

Lemma 1 *If the behaviour functor B preserves weak pullbacks*[3] *then, equality of final coalgebra semantics is equivalent to equality of the greatest B-bisimulation.*

$$\mathsf{unfold}(\llparenthesis t \rrparenthesis) = \mathsf{unfold}(\llparenthesis t' \rrparenthesis) \quad \Leftrightarrow \quad t \sim_B t'$$

In theorem 5 in section 2.4.2, the condition that B should preserve weak pullbacks is shown not to be necessary for ensuring that the greatest bisimulation is a congruence.

Corollary 1 *If the initial algebra semantics and the final coalgebra semantics coincide, observational equivalence corresponding to the final coalgebra semantics is a* congruence.

Proof The proof just makes use of the fact that, by definition, every initial algebra semantics is compositional. For details, see [35, §12]. $\Box$

Adequacy revisited

We can provide a more symmetric formulation of adequacy if we consider T-algebras and D-coalgebras. If T is freely generated by Σ then T-algebras are isomorphic to Σ-algebras, and we can move to the category of T-algebras.

Definition 2 *The Eilenberg-Moore category, or category of T-algebras of a monad $T = \langle T, \eta, \mu \rangle$ in a category $\mathcal{C}$, denoted as $\mathcal{C}^T$, has as objects pairs $\langle X, h \rangle$, with X an object of $\mathcal{C}$ and $h \colon TX \to X$ an arrow of $\mathcal{C}$ such that the following diagrams commute*

$$
\begin{tikzcd}
T^2X \arrow[r, "Th"] \arrow[d, "\mu_X"'] & TX \arrow[d, "h"] \\
TX \arrow[r, "h"'] & X
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
TX \arrow[d, "h"'] & X \arrow[l, "\eta_X"'] \arrow[dl, equal] \\
X &
\end{tikzcd}
$$

[3] A universal is said to be weak when only its *existence* is guaranteed, but not its *uniqueness*.

The arrows of the category $\mathcal{C}^T$ are those arrows f of $\mathcal{C}$ making the following diagram commute

$$\begin{tikzcd} TX \arrow[r, "Tf"] \arrow[d, "h"'] & TY \arrow[d, "k"] \\ X \arrow[r, "f"'] & Y \end{tikzcd}$$

Dually, we can define the category of D-coalgebras of a comonad D.

Proposition 1 *Σ-algebras are isomorphic to T-algebras, and dually, B-coalgebras are isomorphic to D-coalgebras, where T is the free monad on Σ and D is the cofree comonad on B.*

$$\begin{tikzcd} \Sigma\text{-}Alg \arrow[r, phantom, "\cong"] \arrow[dr, "U^{\Sigma}"'] & T\text{-}Alg \arrow[d, "U^{T}"] \\ & \mathcal{C} \end{tikzcd} \qquad \begin{tikzcd} B\text{-}Coalg \arrow[r, phantom, "\cong"] \arrow[dr, "U_{B}"'] & D\text{-}Coalg \arrow[d, "U_{D}"] \\ & \mathcal{C} \end{tikzcd}$$

U^{Σ}, U^{T}, U_B, U_D, are the obvious forgetful functors.
The maps of the isomorphisms are:

$$\begin{array}{rcl} \Sigma\text{-}algebra & & T\text{-}algebra \\ \langle X, h\colon \Sigma X \to X\rangle & \mapsto & \langle X, \mathsf{fold}(id + h)\colon TX \to X\rangle \\ \langle X, k \circ \mathsf{inr}_X \circ \Sigma\eta_X \colon \Sigma X \to X\rangle & \leftarrow\!\shortmid & \langle X, k\colon TX \to X\rangle \end{array}$$

$$\begin{array}{rcl} B\text{-}coalgebra & & D\text{-}coalgebra \\ \langle X, h\colon X \to BX\rangle & \mapsto & \langle X, \mathsf{unfold}(id \times h)\colon X \to DX\rangle \\ \langle X, B\varepsilon_X \circ \pi_2 \circ k\colon X \to BX\rangle & \leftarrow\!\shortmid & \langle X, k\colon X \to DX\rangle \end{array}$$

As a consequence of proposition 1, the initial Σ-algebra is equivalent to the multiplication of the monad on the initial object $\mu_0\colon T^2 0 \to T0$ and the final B-coalgebra is equivalent to the comultiplication of the comonad on the final object $\delta_1\colon D1 \to D^2 1$ (notice that $D1 \cong 1 \times BD1 \cong BD1$, so we have $D1 \cong \nu B$.) Therefore, the diagram in figure 1 is equivalent to the following (more symmetric) diagram:

$$\begin{tikzcd} T^2 0 \arrow[r, "T\mathsf{fold}(\llbracket - \rrbracket)"] \arrow[d, "\mu_0"'] & TD1 \arrow[d, "\llbracket - \rrbracket"] \\ T0 \arrow[r, dashed, shift left, "\mathsf{fold}(\llbracket - \rrbracket)"] \arrow[r, dashed, shift right, "\mathsf{unfold}(\llparenthesis - \rrparenthesis)"'] \arrow[r, phantom, "="] \arrow[d, "\llparenthesis - \rrparenthesis"'] & D1 \arrow[d, "\delta_1"] \\ DT0 \arrow[r, "D(\mathsf{unfold}(\llparenthesis - \rrparenthesis))"'] & D^2 1 \end{tikzcd}$$

By some abuse of notation we maintain the names for the operational model and the denotational model.

In this last diagram the duality of the algebraic and coalgebraic approaches is made evident and hence, we can appreciate the elegant beauty of bialgebras.

2.1.2 Constructing Adequate Semantics

Following the definition of adequacy, to obtain adequate semantics we need to construct a category with objects $TX \to X \to DX$, and as morphisms, arrows which are both T-algebra and D-coalgebra morphisms, $T0$ should be the carrier of the initial object of this category and $D1$ the carrier of the final object. As we are about to see, for any lifting $\tilde{T}$ of the monad T to the D-coalgebras, the category of the $\tilde{T}$-algebras has exactly this structure.

In the following we make this statement more precise by providing the necessary definitions and by spelling out the structure of the category of $\tilde{T}$-algebras.

Definition 3 *Let $T = \langle T, \eta, \mu \rangle$ be a monad and D an endofunctor on the same category $\mathcal{C}$. A monad $\langle \tilde{T} = \tilde{T}, \tilde{\eta}, \tilde{\mu} \rangle$* lifts *$T$ to the D-coalgebras if the following diagram*

$$\begin{tikzcd} D\text{-}Coalg \arrow[r, "\tilde{T}"] \arrow[d, "U_D"'] & D\text{-}Coalg \arrow[d, "U_D"] \\ \mathcal{C} \arrow[r, "T"'] & \mathcal{C} \end{tikzcd}$$

commutes and U_D preserves the operations of the monad. That is, the following equations should hold.

$$\begin{array}{rcll} U_D\tilde{T} & = & TU_D & : \ D\text{-}Coalg \to \mathcal{C} \\ U_D\tilde{\eta} & = & \eta_{U_D} & : \ U_D \to TU_D \\ U_D\tilde{\mu} & = & \mu_{U_D} & : \ T^2U_D \to TU_D \end{array}$$

According to these equations, we are asking the action of $\tilde{\eta}$ and $\tilde{\mu}$ on the carriers of the coalgebras to be the same as η and μ in the unlifted monad. Diagrammatically,

$$\begin{tikzcd} X \arrow[r, "\eta_X"] \arrow[d, "k"'] & TX \arrow[d, "\tilde{T}(k)"'] & T^2X \arrow[l, "\mu_X"'] \arrow[d, "\tilde{T}^2(k)"] \\ DX \arrow[r, "D\eta_X"'] & DTX & DT^2X \arrow[l, "D\mu_X"] \end{tikzcd}$$

Let's consider the structure h of an object in the category D-Coalg$^{\tilde{T}}$, of $\tilde{T}$-algebras. This object has as carrier a D-coalgebra $\langle X, k \rangle$ with $k\colon X \to DX$.

So, in D-Coalg$^{\tilde{T}}$:

$$\langle \langle X, k \rangle, h \rangle$$

By the first condition on the structure of $\tilde{T}$-algebras, in D-Coalg we have that the following diagram commutes.

$$\begin{tikzcd}
\tilde{T}^2\langle X,k\rangle \arrow[r, "\tilde{T}h"] \arrow[d, "\mu_{\langle X,k\rangle}"'] & \tilde{T}\langle X,k\rangle \arrow[d, "h"] \\
\tilde{T}\langle X,k\rangle \arrow[r, "h"'] & \langle X,k\rangle
\end{tikzcd}$$

Since h is an arrow of D-Coalg, h is coalgebra homomorphism. Also, $\tilde{T}(k)\colon TX \to DTX$. Then, the following diagram commutes.

$$\begin{tikzcd}
TX \arrow[r, "\tilde{T}(k)"] \arrow[d, "h"'] & DTX \arrow[d, "Dh"] \\
X \arrow[r, "k"'] & DX
\end{tikzcd}$$

So, we have that if h is the structure of a $\tilde{T}$-algebra then it is also a T-algebra. One can prove that the converse also holds.

Therefore, the objects in D-Coalg$^{\tilde{T}}$ are triples $\langle X,k,h\rangle$ with $k\colon X \to DX$ and $h\colon TX \to X$ such that the previous diagram commutes.

The arrows in D-Coalg$^{\tilde{T}}$ are arrows that preserve the commutativity of this diagram. That is, they are arrows f such that

$$\begin{tikzcd}
 & TY \arrow[d, "l"] & \\
TX \arrow[ur, "Tf"] \arrow[d, "h"'] & Y \arrow[r, "m"] & DY \\
X \arrow[ur, "f"'] \arrow[r, "k"'] & DX \arrow[ur, "Df"'] &
\end{tikzcd}$$

commutes. Notice that this is the same as saying that f is both a D-coalgebra arrow and a T-algebra arrow.

As we can see in the last diagram, adequate semantics live in a category of algebras of a monad lifting a monad T to the D-coalgebras. Also, by duality, one can interpret adequacy as living in a category of coalgebras of a comonad lifting a D-comonad to the T-algebras. We should also prove that initiality and finality of the initial T-algebra and final D-coalgebra is preserved, but for this it is sufficient to consider the adjunction splitting the monad and the adjunction splitting the comonad. Since left (right) adjunctions preserve colimits (limits) we obtain the desired result.

In order to obtain a symmetric definition of bialgebraic semantics, we introduce the following

Definition 4 *Given a monad $T = \langle T, \eta, \mu \rangle$ and a comonad $D = \langle D, \epsilon, \delta \rangle$ on a category $\mathcal{C}$, a* distributive law of the monad T over the comonad D *is a natural transformation $\lambda \colon TD \to DT$ such that the operations are preserved, in the sense that the following diagrams commute.*

$$\begin{tikzcd} & DT & \\ TD \arrow[ur, "\lambda"] & & D \arrow[ll, "\eta_D"] \arrow[ul, "D\eta"'] \end{tikzcd} \qquad \begin{tikzcd} T^2D \arrow[r, "T\lambda"] \arrow[d, "\mu_D"'] & TDT \arrow[r, "\lambda_T"] & DT^2 \arrow[d, "D\mu"] \\ TD \arrow[rr, "\lambda"'] & & DT \end{tikzcd}$$

$$\begin{tikzcd} & TD \arrow[dl, "\lambda"'] \arrow[dr, "T\epsilon"] & \\ DT \arrow[rr, "\epsilon_T"'] & & T \end{tikzcd} \qquad \begin{tikzcd} D^2T & DTD \arrow[l, "D\lambda"'] & TD^2 \arrow[l, "\lambda_D"'] \\ DT \arrow[u, "\delta_T"] & & TD \arrow[ll, "\lambda"] \arrow[u, "T\delta"'] \end{tikzcd}$$

Theorem 1 *For any monad T and comonad D on the same category it is equivalent to give:*

- *A distributive law $\lambda \colon TD \to DT$ of T over D*
- *A lifting $\tilde{T}$ of T to the D-coalgebras.*
- *A lifting $\tilde{D}$ of D to the T-algebras.*

Proof Given a distributive law λ, we obtain the liftings as follows.

$$\tilde{T}(k) = \lambda_X \circ Tk \qquad \tilde{D}(h) = Dh \circ \lambda_X$$

If we now start from a lifting $\tilde{T}$, then $U_D\tilde{T} = TU_D$. The comonad D is split by the adjunction $U_D \dashv G_D$, hence $D = U_D G_D$. Then, $TD = TU_D G_D = U_D \tilde{T} G_D$. We consider the arrow $T\varepsilon_D \colon TD \to T$, which by the previous argument is $T\varepsilon \colon U_D\tilde{T}G_D \to T$. Transposing this arrow across the adjunction splitting D, we obtain an arrow $\overline{\lambda} \colon \tilde{T}G_D \to G_D T$. We define $\lambda = U_D\overline{\lambda} \colon TD \to DT$.

Dually, we obtain a distributive law λ from the lifting $\tilde{D}$. It can be proved that these constructions are indeed distributive laws by proving that they respect the operations in the monad and the comonad.

For details, see [35, §7.1] $\square$

One important consequence of the previous theorem is the following proposition:

Proposition 2 *From a distributive law between a monad T and a comonad D we can obtain*

- *a canonical operational model $\llparenthesis - \rrparenthesis = \lambda_X \circ T(0_D)$ where $0_D \colon 0 \to D0$.*
- *a canonical denotational model $[\![-]\!] = D(1^T) \circ \lambda_X$ where $1^T \colon T1 \to 1$.*

Now we give the formal definition of bialgebraic semantics.

Definition 5 *A bialgebraic semantics is a distributive law between a monad T that corresponds to syntax, and a comonad D that corresponds to the observable behaviour.*

Although the liftings mentioned in theorem 1 are equivalent to a distributive law, we prefer using a distributive law as definition since, due to its symmetry, it is self-dual. This means we do not need to consider the dual case.

We have defined bialgebraic semantics as a distributive law but we have not explained how to construct such a distributive law. In [35] two ways to construct bialgebraic semantics were given. The first one is by abstract operational rules, and the second is by their dual.

2.1.3 Abstract Operational Rules

Definition 6 *An* abstract operational rule *is a natural transformation*

$$\rho\colon \Sigma(Id \times B) \to BT$$

where T is the monad freely generated by Σ.

Proposition 3 *From an abstract operational rule $\rho\colon \Sigma(Id \times B) \to BT$, we can construct a monad T_ρ lifting T to the B-coalgebras.*

Proof See [35, §4]. □

Definition 7 *The* operational monad $\tilde{T}$ *induced by some abstract operational rule $\rho\colon \Sigma(Id \times B) \to BT$, is the monad $T\rho$ corresponding to the construction in proposition 3.*

Abstract operational rules originated as an abstraction of rules in the GSOS format [5].

Definition 8 *Let A_i and B_i range over subsets of A. A GSOS-rule is a rule of the form*

$$\frac{\{x_i \xrightarrow{a} y_{ij}^a\}_{1\leqslant j\leqslant m_i^a}^{1\leqslant i\leqslant n, a\in A_i} \qquad \{x_i \stackrel{b}{\not\to}\}_{b\in B_i}^{1\leqslant i\leqslant n}}{\sigma(x_1,\ldots,x_n) \xrightarrow{c} t}$$

such that x_i and y_{ij}^a are all distinct, and those are the only variables that occur in the term t.

If there are finitely many rules for each operator σ in Σ and action c in A, then the set of rules is said to be image finite.

Proposition 4 *Image finite GSOS rules are in one-to-one correspondence with abstract operational rules of the behaviour $BX = (\mathcal{P}_f X)^A$, where $\mathcal{P}_f$ is the finite subset functor.*

2.1.4 Abstract denotational rules

Definition 9 *The dual of an abstract operational rule, called* abstract denotational rule*, is a natural transformation*

$$\varrho\colon \Sigma D \to B(Id + \Sigma)$$

By the construction dual to the one in proposition 3, we can construct a comonad D_ϱ lifting D to the Σ-algebras (where D is again the comonad cofreely generated by B.)

Definition 10 *Consider rules of the form*

$$\frac{\{z_i \xrightarrow{a_i} y_i\}_{i\in I} \quad \{v_j \not\xrightarrow{b_j}\}_{j\in J}}{\sigma(x_1,\ldots,x_n) \xrightarrow{c} t}$$

where x_k, y_i, z_i, and v_j are all variables, and I and J are countable, possibly infinite sets. The x_k and y_i should be distinct variables, and the rule should be well-founded in the sense that all backward chains in its variable dependency graph are finite.

If t is either a variable or of the form $\sigma'(x'_1,\ldots,x'_m)$ for some operator σ' of the signature and some (not necessarily distinct) variables $x'_1,\ldots,x'_m$, then the rule is a safe tree rule.

Safe tree rules are more general than GSOS since they allow for lookahead (given a transition $x \xrightarrow{a} y$, we can also use $y \xrightarrow{b} z$.) They are restricted in the form of the term in the conclusion but, in practise, this restriction can usually be sidestepped by adding auxiliary operators to the signature.

Proposition 5 *Safe tree rules are in one-to-one correspondence with abstract denotational rules.*

2.1.5 Case Studies

Turi presents a variety of concrete structural operational semantics and their bialgebraic models [37]. Two base categories are considered:

- **Set**, the same category considered in modelling GSOS, and
- $SL(\mathbf{Set})$, the category of semi-lattices and join-preserving functions. Interestingly, by working on $SL(\mathbf{Set})$, one can treat, for example, non-determinism with side-effects and explicit termination by considering $BX = (S\cdot(1+X))^S$ instead of using the behaviour $BX = (\mathcal{P}(S\cdot(1+X)))^S$. Also, working in this category means that the notion of equivalence obtained is trace-equivalence, instead of bisimulation.

In Turi's previous work he focused on the behaviour $BX = (\mathcal{P}X)^A$ of labelled transition systems. In this work other behaviour functors are studied, more precisely, he considers behaviours generated by the following grammar:

$$B ::= Id \mid 1 + B \mid S \cdot B \mid B^S \mid \mathcal{P}_f B \mid \mathcal{P}_{cf} B$$

where $\mathcal{P}_f$ is the finite power-set functor, $\mathcal{P}_{cf}$ is the commutative free semi-lattice monad, and S is a parameter which can be interpreted as a set of states or actions.

The examples in this paper are:

- A sequential composition operator in the presence of explicit termination and state in an affine SMC category.
- A parallel composition operator and a choice operator, modelled in a category $\mathcal{C}$ with powers and a commutative free semi-lattice monad $\mathcal{P}_{cf}$.

Modularity

Modularity is analysed for the previous examples by defining the behaviours to be parametric in some part of the behaviour B'. For the previous examples the behaviour $BX = B'(1+X)$ is considered. Another example is a loop construct, for which the whole behaviour B can be given as a parameter, as long as an operation $\otimes\colon X \times BX \to X \otimes BX$ is given.

What we have here is a general schema for the definition of the semantics of a given construct. We do not obtain a concrete semantics until we fix a specific behaviour, so it is not an ideal solution. Nevertheless, this scenario is similar to that of monad transformers in which the parameter is a monad, and monad transformers have proven to be useful nonetheless. Consequently, we will consider parameterised behaviours in section 3.1 as an approach to modularity.

Guarded Recursion

A system of equations

$$\begin{array}{rcl} x_0 & = & t_0 \\ x_1 & = & t_1 \\ & \vdots & \end{array}$$

where x_i is a variable in X and t_i is a term in TX, can be seen as a coalgebra $\Theta\colon X \to TX$ s.t. $x_i \mapsto t_i$.

A term is said to be *guarded* if its (one step) behaviour does not depend on the behaviour of its variables. If every t_i is guarded, then we have a guarded system of equations. Notice that guarded systems of equations do not allow for silly equations like $x = x$; every expansion of a variable corresponds to some observable behaviour.

Given a functor Σ freely generating the terms TX, we can obtain a subfunctor of Σ, $\Sigma_G\colon \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ (with insertion map into Σ given by $\Sigma_G \circ \Delta$) which correspond to the signature of guarded terms. The guarded terms are then given by a functor $GX = \mu Y.\Sigma_G(Y, TX)$.

Definition 11 *A* guard *for some abstract operational rules ρ is a span*

$$\begin{tikzcd} & G \arrow[dl, "\xi"'] \arrow[dr, "\zeta"] & \\ T & & BT \end{tikzcd} \quad \textit{such that} \quad \begin{tikzcd} & GU \arrow[dl, "\xi_U"'] \arrow[dr, "\zeta_U"] & \\ TU \arrow[rr, "T_\rho"'] & & BTU \end{tikzcd}$$

commutes.

The left leg ξ tells how to obtain terms TX from guarded terms GX and can be calculated by folding the insertion of Σ_G into Σ. The right leg ζ tells how to obtain the behaviour of guarded terms, and can be calculated by folding some abstract operational rule on guarded terms $\rho'\colon \Sigma_G\langle Id \times B, Id\rangle \to BT$. Summarising, to obtain a guard we need to provide the signature of guarded terms Σ_G, its insertion into Σ, and the semantics ρ' for guarded terms.

Given a guard (G, ξ, ζ), and a system of guarded equations $\Theta\colon X \to GX$, we can obtain the operational monad in a similar way to what we did in proposition 3, only that now instead of considering coalgebras $k\colon X \to BX$ and the composite $B\eta \circ k\colon X \to BTX$, we consider the composite $\zeta \circ \Theta\colon X \to BTX$.

2.1.6 Some Concluding Remarks

Structural Operational Semantics (SOS) might be more intuitive and more easily learnt than other semantic styles, but lacks a strong metatheory. Bialgebraic semantics gives a mathematical model of well-behaved SOS semantics. By working in a category-theoretic setting, we are able to reason about SOS focusing only on the fundamental notions of abstract syntax and behaviour.

Abstract operational rules and abstract denotational rules, are the counterpart in the bialgebraic framework of two known syntactic format for rules, and yield two constructions of bialgebraic semantics. More general syntactic formats for rules arise from other formulations of abstract rules (for examples, see next section and section 2.4.2). Consequently, when analysing modularity of SOS in this framework, we should aim to work at the most abstract level possible, so as to include in our analysis future formats that may appear.

2.2 Lenisa, Power and Watanabe

In the following we introduce the work of Lenisa, Power and Watanabe which was first presented in [20], and then expanded in [21, 32].

Their main contribution is bringing up the fact that abstract operational rules are equivalent to a distributive law $TH \to HT$ between the syntax monad T and H, the cofree copointed endofunctor on B (section 2.2.1.) Working at this level of abstraction, they presented an abstract operation for joining two languages with same behaviour (section 2.2.2), and started to analyse how to add equations to the syntax (for example, to make an operator associative by definition.) Lastly, they defined the meaning of a stream of transitions, by first defining what a two-step transition is, and then generalising to n-step transitions (section 2.2.3). They also suggested that by considering the limit of this last construction we would obtain a big-step semantics, hence relating small-step semantics and big-step semantics. The big-step semantics they construct in this way from a small-step one is equal to the canonical distributive law of a monad over a comonad.

2.2.1 Abstract rules as distributive laws

Abstract operational rules (see section 2.1.3) are equivalent to a distributive law of a monad and a copointed endofunctor.

Definition 12 *A* copointed endofunctor *on a category* $\mathcal{C}$ *is an endofunctor* $H\colon \mathcal{C} \to \mathcal{C}$ *together with a natural transformation* $\varepsilon\colon H \to Id$. *An* $\langle H, \varepsilon\rangle$*-coalgebra is an object* X *of* $\mathcal{C}$ *together with a map* $k\colon X \to HX$ *such that the following diagram commutes.*

$$\begin{tikzcd} X \arrow[r, "k"] \arrow[dr, equal] & HX \arrow[d, "\varepsilon_X"] \\ & X \end{tikzcd}$$

A map of $\langle H, \varepsilon\rangle$*-coalgebra is a coalgebra map that preserves the* ε *operation. With these definitions we obtain the category* $\langle H, \varepsilon\rangle$*-Coalg of* $\langle H, \varepsilon\rangle$*-coalgebras.*

Definition 13 *A* distributive law *of a monad* $\langle T, \eta, \mu\rangle$ *over a copointed endofunctor* $\langle H, \varepsilon\rangle$ *is a natural transformation* $\lambda\colon TH \to HT$ *such that the following diagrams commute.*

$$\begin{tikzcd} TTH \arrow[r, "T\lambda"] \arrow[d, "\mu_H"'] & THT \arrow[r, "\lambda_T"] & HTT \arrow[d, "H\mu"] \\ TH \arrow[rr, "\lambda"'] & & HT \end{tikzcd} \quad \begin{tikzcd} & H \arrow[dl, "\eta_H"'] \arrow[d, "H\eta"] \\ TH \arrow[r, "\lambda"'] & HT \end{tikzcd} \quad \begin{tikzcd} TH \arrow[r, "\lambda"] \arrow[d, "T\varepsilon"'] & HT \arrow[dl, "\varepsilon_T"] \\ T & \end{tikzcd}$$

For an endofunctor B on a category $\mathcal{C}$, if $\mathcal{C}$ has finite products, then the cofree copointed endofunctor on B is given by $\langle Id \times B, \pi_1\rangle$. The cofree comonad D on the endofunctor B agrees with the cofree comonad on the copointed endofunctor $\langle Id \times B, \pi_1\rangle$, because of the following isomorphisms:

$$\langle Id \times B, \pi_1\rangle\text{-Coalg} \cong B\text{-Coalg} \cong D\text{-Coalg}$$

We also have that the cofree comonad on the cofreely generated endofunctor exists iff the cofree comonad on B exists.

The main result of this section is the following:

Theorem 2 *To give an abstract operational rule* $\rho\colon \Sigma(Id \times B) \to BT$ *is equivalent to giving a distributive law* $\lambda\colon T(Id \times B) \to (Id \times B)T$ *of the free monad* T *over the copointed endofunctor* $\langle Id \times B, \pi_1\rangle$.

Distributive laws have the following advantages over abstract operational rules:

- They provide a more uniform and elegant formulation.
- The operational monad is obtained in a more standard way.
- They lead to new ways of generalisation. We may now consider distributive laws of any monad over any copointed endofunctor. Notice that, while this is less general than considering a distributive law of a monad over a comonad, it is also less demanding in the sense that we require less structure on the behaviours.

2.2.2 Adding operations

Given two distributive laws over the same comonad, we can obtain a combined distributive law.

Proposition 6 *For monads* T *and* T' *on* $\mathcal{C}$, *if the coproduct of monads* $T + T'$ *exists, the category of algebras* $(T + T')$*-Alg is canonically isomorphic to the pullback*

$$\begin{tikzcd} P \arrow[r] \arrow[d] & T\text{-}Alg \arrow[d, "U"] \\ T'\text{-}Alg \arrow[r, "U'"'] & C \end{tikzcd}$$

Theorem 3 *Given monads* T *and* T' *and a comonad* D, *and given distributive laws* $\lambda\colon TD \to DT$ *and* $\lambda'\colon T'D \to DT'$, *there is a canonical distributive law of* $T + T'$ *over* D *if the coproduct of monads exists.*

Proof By Theorem 1, from λ and λ' we obtain liftings to the T-algebras. By Proposition 6, this liftings yield a comonad on $T + T'$. By Theorem 1 (in the other direction), we obtain the distributive law of $T + T'$ over D. □

The previous theorem was stated in [21] and [32] for distributive laws of monads over *copointed endofunctors*. Their proof carries over to the more general case presented here.

2.2.3 Dynamics

In order to analyse the dynamics of SOS, Lenisa, Power and Watanabe introduce the idea of a two step transition. They argue that, while a transition is a coalgebra $x\colon X \to BX$, considering the composite

$$X \xrightarrow{x} BX \xrightarrow{Bx} BBX$$

is not enough since this does not record intermediate steps (It defines a transition $X \to BBX$, but we lose the path). Consequently, they consider steps based on a distributive law $\lambda\colon TH \to HT$, where one has the obvious composite

$$THH \xrightarrow{\lambda_H} HTH \xrightarrow{H\lambda} HHT$$

This composite records intermediate steps but it is too general, as they show for their lead example $H = Id \times \mathcal{P}$, where $\mathcal{P}$ is the powerset functor. They introduce an equaliser in order to make the target of the first transition agree with the source of the second one. They define the behaviour of two steps to be the the copointed endofunctor $\langle H_2, \varepsilon_2\rangle$, where H_2X is the equaliser of the maps

$$\begin{tikzcd} HHX \arrow[r, shift left, "H\varepsilon_X"] \arrow[r, shift right, "\varepsilon_{HX}"'] & HX \end{tikzcd}$$

and $\varepsilon_2\colon H_2 \to Id$ is defined by composition. Since $X \xrightarrow{x} HX \xrightarrow{Hx} HHX$ is an $\langle H_2, \varepsilon_2\rangle$-algebra, we have a process for constructing H_2 from H. This process is iterated to define H_n.

Finally, they show how to obtain a distributivity law of T over H_n, the behaviour of n steps, starting from $\lambda\colon TH \to HT$. Under the condition that certain limits exist, this process yields a distributive law of the monad T over the cofree comonad D on $\langle H, \varepsilon\rangle$, which agrees with the canonical one.

2.3 Kick, Power

In [12, 11], Marco Kick used bialgebraic semantics to model timed processes. Interestingly, he had to consider the comonad D and not a behaviour functor B, since in systems for continuous time the comonad D is not cofreely generated.

In [13], Kick and Power introduce an operation for combining behaviours, which is exactly the dual of the operation in section 2.2.2. We repeat the relevant proposition and theorem in its dual form.

Proposition 7 *For comonads D and D' on $\mathcal{C}$, if the product of comonads $D \times D'$ exists, the category of coalgebras $(D \times D')$-Coalg is canonically isomorphic to the pullback*

$$\begin{tikzcd} P \arrow[r] \arrow[d] & D\text{-}\mathit{Coalg} \arrow[d, "U"] \\ D'\text{-}\mathit{Alg} \arrow[r, "U'"'] & \mathcal{C} \end{tikzcd}$$

Theorem 4 *Given a monad T, comonads D and D', and distributive laws $\lambda\colon TD \to DT$ and $\lambda'\colon TD' \to D'T$, there is a canonical distributive law of T over $D \times D'$ if the product of the comonads $D \times D'$ exists.*

This operation would allow them to separately specify the timing behaviour and the action behaviour of a given system and then combine them. However, in the leading examples of this kind of systems, the time information and the action information interact with each other. To be able to represent these examples as modularly as possible, they proved the following proposition, which does not require a total independence between time and action information, but it is able to split the semantics into two subproblems.

Proposition 8 *Given comonads D and D' on a category $\mathcal{C}$ such that the product of $D \times D'$ exists, and given a monad T on $\mathcal{C}$, to give a lifting of T to a monad on $(D \times D')$-Coalg is equivalent to giving natural transformations $\lambda\colon T(D \times D') \to DT$ and $\lambda'\colon T(D \times D') \to D'T$ such that the operations on the monad and the different comonads are respected. For details, see the original paper [13].*

Besides the separation of action and time information, no other application for these operations has been suggested in the literature.

The dual of the operation in proposition 8, has not been mentioned elsewhere, but seems to be useful. Given natural transformations $TD \to D(T+T')$ and $T'D \to D(T+T')$ which respect the operations of the monads and the comonad, we can obtain a distributive law $(T+T')D \to D(T+T')$. A possible application is the following. Consider two constructs whose semantics are mutually recursive. Using the operation, we could split the task of defining the semantics of these two constructs into two smaller problems.

2.4 Other work on Bialgebraic Semantics

2.4.1 Klin

The core of Klin's PhD thesis [17] analyses how to incorporate other notions of equivalence besides bisimulation in the bialgebraic framework. This part of the thesis is summarised in [15] and the pragmatics explained in [16]. The other part, which is also presented in [14] concerns describing recursion by equations, rather than by structural rules. Klin follows Plotkin's suggestion [29] of considering recursion as a matter of syntax. That is, a fixed-point operator is regarded as syntactic sugar for the unwinding of a term. Since the kind of equations considered can introduce divergence, the category **Set** has no longer enough structure, and he needs to move to an order-enriched category. So, in this development, all structural rules are non-recursive, and recursive constructs are expressed by a class of equations which he calls *regular unfolding rules.*

2.4.2 Bartels

Bartels analyses probabilistic systems in the bialgebraic framework, and the rule formats that can be obtained [2, 4]. In previous work on bialgebraic semantics, the relation between natural transformations (like abstract operational rules) and rule formats was only sketched. In his work he makes this relation more precise by a providing derivation process, which includes decomposing the natural transformation into simpler ones, and then deriving concrete representations.

Another of the contributions of his thesis [4], also published in [3], it's the use of the bialgebraic framework for solving systems of equations.

Particularly relevant to our research is the following theorem, in which the condition on behaviours for bisimulation to be a congruence is relaxed. Previously, the behaviour functor was required to preserve weak-pullbacks; in the following theorem this restriction is lifted.

Theorem 5 *Let λ be a distributive law of the monad T over the comonad D. The greatest bisimulation $R \subseteq P \times Q$ between any two λ-bialgebras $\langle P, \beta_P, \alpha_P \rangle$ and $\langle Q, \beta_Q, \alpha_Q \rangle$ is a congruence.*

2.4.3 Watanabe

Working in a 2-category setting, Watanabe [39] provides two different definitions of morphisms between distributive laws. He argues that one of the morphisms can be useful to interpret conservative operational extensions, while the other provides behaviour-preserving translations between languages.

2.4.4 Variable Binding

Fiore, Plotkin and Turi explained how to give initial algebra semantics to languages with variable binding [7] by working in a category of presheaves. Later, Fiore and Turi [9] showed how this idea works for modelling the operational semantics of languages with variable binding using abstract operational rules.

3 Operations on SOS

In the bialgebraic framework, operational semantics are given by well-defined formal constructions: abstract operational rules or, alternatively, distributive laws. We can approach the problem of modularity in SOS by considering operations on these formal constructions. This approach has several advantages:

- Generality. One can work at an abstract level, without having to rely on specific sets of side-effects.
- Flexibility. There are different ways of combining two different semantics. By choosing the operations and the order in which they are applied, the language designer is in control of the way components are put together.

- It provides a well-defined relation between the components and the composed semantics, as given by the operations. One can prove preservation of properties by proving that the operations involved preserve the property.

The operations can be defined at the level of abstract operational rules, distributive laws of monads over copointed endofunctors or distributive laws of monads over comonads. It is not yet clear to us which of these levels of abstraction is the best approach. In the following, I first consider the operations on abstract operational rules, and give an example of a problem that can be solved with these operations. Next, I do the same for distributive laws.

3.1 Operations on abstract operational rules

We will define some operations on abstract operational rules, but first we will define an example problem that will serve as motivation. Then, we will analyse what operations would be needed to solve the problem and define them.

3.1.1 An example modularity problem

Consider the following language of arithmetic expressions

$$a \ ::= \mathsf{Con}\ n \mid \mathsf{Add}\ a\ a$$

where n ranges over $\mathbb{Z}$.
The operational semantics of this language is

$$\frac{}{\mathsf{Con}\ x \Downarrow x} \qquad \frac{t \Downarrow x \quad u \Downarrow y}{\mathsf{Add}\ t\ u \Downarrow (x+y)}$$

We want to extend this language with an exceptions language:

$$e \ ::= \ \mathsf{Throw} \mid \mathsf{Catch}\ e\ e$$

$$\frac{}{\mathsf{Throw} \Downarrow \mathsf{Nothing}} \qquad \frac{t \Downarrow \mathsf{Just}\ x}{\mathsf{Catch}\ t\ u \Downarrow \mathsf{Just}\ x} \qquad \frac{t \Downarrow \mathsf{Nothing} \quad u \Downarrow y}{\mathsf{Catch}\ t\ u \Downarrow y}$$

The combined language is

$$ae \ ::= \ \mathsf{Con}\ n \mid \mathsf{Add}\ ae\ ae \mid \mathsf{Throw} \mid \mathsf{Catch}\ ae\ ae$$

$$\frac{}{\mathsf{Con}\ x \Downarrow x} \qquad \frac{t \Downarrow x \quad u \Downarrow y}{\mathsf{Add}\ t\ u \Downarrow (x+y)}$$

$$\frac{}{\mathsf{Throw} \Uparrow} \qquad \frac{t \Downarrow x}{\mathsf{Catch}\ t\ u \Downarrow x} \qquad \frac{t \Uparrow \quad u \Downarrow y \quad t \Uparrow \quad u \Uparrow}{\mathsf{Catch}\ t\ u \Downarrow y \mathsf{Catch}\ t\ u \Uparrow}$$

$$\frac{t \Uparrow}{\mathsf{Add}\ t\ u \Uparrow} \qquad \frac{u \Uparrow}{\mathsf{Add}\ t\ u \Uparrow}$$

The semantics for the previous languages were given in big-step style. In section 3.1.4 we present the semantics in small step style.

We see that the big-step relation of the extended language has to distinguish between normal termination (with $\Downarrow$) and exceptional termination (with $\Uparrow$). Also we need to add rules to specify the behaviour of the addition operator when one of its subexpressions is an exception. The rules for the exceptions sub-language do not suffer any changes; we will see that this takes a significant role later on.

While this new semantics looks similar to the previous ones, it is a different semantics. Nevertheless, it seems obvious that a connection must exist. What we want to do is to expose this connection.

3.1.2 Transforming and combining operational semantics

We will focus just on the concepts of syntax and behaviour. A given SOS will be characterised by its syntax (represented by a signature Σ) and behaviour B, and we will write $\mathcal{R}(\Sigma, B)$.

We start with the operational semantics of the arithmetics language $\mathcal{R}(\Sigma_A, B_A)$ and the exceptions language $\mathcal{R}(\Sigma_E, B_E)$.

- The syntax of the combined language is the disjoint union of the syntax of the two components, so we will need to join the two syntaxes $\mathcal{R}(\Sigma_A + \Sigma_E, B_{AE})$.
- The behaviour of the arithmetics language is just the set of final values: an integer $\mathbb{Z} = B_A$, while in the combined language the behaviour is $B_{AE} = \mathsf{Maybe}\ \mathbb{Z}$. We will need a *lifting* operation to go from $\mathcal{R}(\Sigma_A, \mathbb{Z})$ to $\mathcal{R}(\Sigma_A, \mathsf{Maybe}\ \mathbb{Z})$.
- We saw that in the combined language that rules for the exceptions sub language did not change. This suggests that its behaviour can be combined with any other behaviour B': $B_E(B') = \mathsf{Maybe}\ B'$. In particular, we are interested in $B_E(\mathbb{Z}) = B_{AE}$.

We can express the combined language as the result of applying the previous operations to the arithmetics and exceptions semantics.

$$\dfrac{\dfrac{\mathcal{R}(\Sigma_A, K_{\mathbb{Z}})}{\mathcal{R}(\Sigma_A, \mathsf{Maybe}\ K_{\mathbb{Z}})}\ (B\text{-lift}) \qquad \dfrac{\mathcal{R}(\Sigma_E, \mathsf{Maybe}(B))}{\mathcal{R}(\Sigma_E, \mathsf{Maybe}(K_{\mathbb{Z}}))}\ (B\text{-Inst})}{\mathcal{R}(\Sigma_A + \Sigma_E, \mathsf{Maybe}\ K_{\mathbb{Z}})}\ (\Sigma\text{-Join})$$

3.1.3 Definition of the operations

Having given the intuition on the operations we now proceed to define them. Here, operational semantics $\mathcal{R}(\Sigma, B)$ are considered to be abstract operational rules $\Sigma(Id \times B) \to BT_\Sigma$.

- Joining syntax

  This operation is the one described in section 2.2.2 on page 16, specialised to abstract operational rules.

  Given two abstract rules with the same behaviour functor,

  $$\rho\colon \Sigma(Id \times B) \Rightarrow BT_\Sigma \qquad \text{and} \qquad \rho'\colon \Sigma'(Id \times B) \Rightarrow BT_{\Sigma'}$$

  we can join ρ and ρ' into a new operational semantics :

  $$\rho''\colon (\Sigma + \Sigma')(Id \times B) \Rightarrow BT_{\Sigma+\Sigma'}$$

  We can express the operation as rule

  $$\frac{\mathcal{R}(\Sigma_1, B) \quad \mathcal{R}(\Sigma_2, B)}{\mathcal{R}(\Sigma_{1+2}, B)}\ (\Sigma\text{-join})$$

  We obtain the new rule by calculating:

  $$\begin{array}{lll} \rho'' & : & (\Sigma + \Sigma')(Id \times B) \\ & = & \quad \{\text{Coproduct of Functors}\} \\ & & \Sigma(Id \times B) + \Sigma'(Id \times B) \\ & \to & \quad \{\rho + \rho'\} \\ & & BT_\Sigma + BT_{\Sigma'} \\ & = & \quad \{\text{Functors}\} \\ & & B(T_\Sigma + T_{\Sigma'}) \\ & \hookrightarrow & \quad \{\text{ Embedding}\} \\ & & B(T_{\Sigma+\Sigma'}) \end{array}$$

  Note that in **Mon**, the category of monads, $T_\Sigma + T_{\Sigma'} \cong T_{\Sigma+\Sigma'}$, but our coproduct in the calculation above is in the base category (i.e. is a coproduct of functors, not of monads).

- Lifting a Rule

  In the combined language AE we needed to write rules to propagate the exceptions through the arithmetic operations. Those propagation rules can be seen as going from a behaviour B to a behaviour FB. That is, we want to lift an abstract rule $\rho\colon \Sigma(Id \times B) \Rightarrow BT_\Sigma$ to

$$\rho^F : \Sigma(Id \times FB) \Rightarrow FBT_\Sigma.$$

Given that F is strong, and that we have a distributive law $\Sigma F \Rightarrow F\Sigma$, we can lift ρ.

$$\begin{array}{rl} \rho^F & : \quad \Sigma(Id \times FB) \\ & \rightarrow \quad \quad \{\text{strength of } F\} \\ & \quad\ \ \Sigma F(Id \times B) \\ & \rightarrow \quad \quad \{\text{distributive law}\} \\ & \quad\ \ F\Sigma(Id \times B) \\ & \rightarrow \quad \quad \{F\rho\} \\ & \quad\ \ FBT_\Sigma \end{array}$$

In rule format, for strong F:

$$\frac{\mathcal{R}(\Sigma, B) \quad \Sigma B \rightarrow B\Sigma}{\mathcal{R}(\Sigma, FB)} \ (B\text{-lift})$$

The distributive law can be obtained automatically in most cases as an instance of the distributive law of traversable functors over applicative functors [24]. Behaviours are usually applicative (in fact, they are usually monads), and we expect the signature of a language to be traversable.

- Transforming behaviours

  Sometimes, we can obtain abstract operational rules which are parameterised by a behaviour. They have some similarity to monads transformers which are parameterised by a monad. The behaviour B used as parameter is applied to a behaviour transformer B' (a functor in a functor category) which captures the interesting part of the observable behaviour of the operations in Σ.

  $$\Sigma B'(B) \rightarrow B'(B)T_\Sigma$$

  Applying a concrete behaviour to a behaviour transformer gives us an instantiation rule:

  $$\frac{\mathcal{R}(\Sigma, B'(b))}{\mathcal{R}(\Sigma, B'(B))} \ (B\text{-Inst})$$

- Obtaining abstract operational rules from big-step semantics.

  Big-step semantics evaluate to a final value. Consequently, no more steps are needed and we don't need all the structure in an abstract operational rule: it is sufficient to consider natural transformations $\Sigma M \to M$. Given a big-step semantics $\tau \colon \Sigma M \to M$, we can generate an abstract operational rule parameterised by a behaviour.

$$
\begin{array}{rcl}
\rho_\tau & : & \Sigma(Id \times MB) \\
 & \to & \quad \{\Sigma\pi_2\} \\
 & & \Sigma MB \\
 & \to & \quad \{\tau_B\} \\
 & & MB \\
 & \to & \quad \{MB\eta\} \\
 & & MBT_\Sigma
\end{array}
$$

More examples The following examples can also be represented as abstract operational rules parameterised by a behaviour, and could also be combined with the arithmetics language A, or with the composed language AE, following the same procedure that we used before: lift one of the languages to the combined behaviour, instantiate the other language to the desired behaviour and then apply the syntax joining operation.

- A State language.

$$s \ ::= \ \mathsf{Tick}\ s$$

$$\frac{\langle t, s\rangle \Downarrow \langle t', s'\rangle}{\langle \mathsf{Tick}\ t, s\rangle \Downarrow \langle t', 1 + s'\rangle}$$

  - A Trace language.

$$t \ ::= \ \mathsf{Trace}\ String\ t$$

$$\frac{t \Downarrow^{r} t'}{\mathsf{Trace}\ s\ t \Downarrow^{s +\!\!+ r} t'}$$

Interestingly, all these examples are typically seen as examples of monad transformers. The approaches are similar in the sense that both of them stack up effects (monads or, in our case, behaviours), and the known remarks about the order in which monad transformers are applied could also be made here about the order in which the parameterised behaviours are organised.

3.1.4 Small step semantics

Consider the same languages as before, but now with small-step semantics.

The deterministic small-step operational semantics of the arithmetics language are

$$\frac{}{\mathsf{Con}\ x \downarrow x} \qquad \frac{t \downarrow x \quad u \downarrow y}{\mathsf{Add}\ t\ u \downarrow (x+y)}$$

$$\frac{t \to t'}{\mathsf{Add}\ t\ u \to \mathsf{Add}\ t'\ u} \qquad \frac{t \downarrow x \quad u \to u'}{\mathsf{Add}\ t\ u \to \mathsf{Add}\ t\ u'}$$

Here, the predicate $t \downarrow v$ means that t is a final value v. In the literature this predicate is usually denoted by underlining a variable $\underline{v}$. Here we have to chosen this notation to make the translation to bialgebraic semantics more transparent.

The small-step semantics for exceptions are:

$$\frac{}{\mathsf{Throw} \uparrow} \qquad \frac{t \to t'}{\mathsf{Catch}\ t\ u \to t'}$$

$$\frac{t \uparrow}{\mathsf{Catch}\ t\ u \to u} \qquad \frac{t \downarrow x}{\mathsf{Catch}\ t\ u \downarrow x}$$

Where the predicate $t \uparrow$ means that t throws an exception.

Bialgebraic semantics For the arithmetic language the behaviour is $B_A = K_{\mathbb{Z}} + -$, which means that in each step the computation either reachs a final value or it has to do more computations. For the exception language the behaviour paramterised by the final values: $B_E(V) = 1 + K_V + -$, where a step is an exception, a final value in V, or more computations.

To obtain the combined semantics, we take $V = \mathbb{Z}$, lift the arithmetics behaviour with the functor $\mathsf{Maybe} = 1 + -$, and then join both abstract operational rules.

$$\dfrac{\dfrac{\mathcal{R}(\Sigma_A, B_A)}{\mathcal{R}(\Sigma_A, \mathsf{Maybe}\ B_A)}\,(B\text{-lift}) \qquad \dfrac{\mathcal{R}(\Sigma_E, B_E(V))}{\mathcal{R}(\Sigma_E, B_E(\mathbb{Z}))}\,(B\text{-Inst})}{\mathcal{R}(\Sigma_{A+E}, B_E(\mathbb{Z}))}\,(\Sigma\text{-join})$$

3.2 Operations on distributive laws

We can also define operations at the level of distributive laws of monads over copointed endofunctors.

3.2.1 Joining syntax

The operation is the one described in section 2.2.2. If we consider, as with the corresponding operation on abstract operational rules, monads freely generated by a signature, the operation is trivial.

3.2.2 Composing behaviours

If we have two distributive laws with same syntax monad T, but different copointed endofunctors H and H', we can obtain a new distributive law by composing behaviours

$$\frac{TH \xrightarrow{\lambda} HT \qquad TH' \xrightarrow{\lambda'} H'T}{THH' \xrightarrow{\lambda_{H'}} HTH' \xrightarrow{H\lambda'} HH'T}$$

There are problems with this rule, since we are not composing behaviours but copointed endofunctors[4] (possibly cofreely generated by a behaviour).

Take the example in section 3.1, where we wanted to compose Maybe and the constant behaviour $K_{\mathbb{Z}}$. The cofreely generated copointed endofunctors for these two behaviours and their composed behaviour $\mathsf{Maybe} \circ K_{\mathbb{Z}}$ are, respectively:

$$\begin{aligned} H &= Id \times \mathsf{Maybe} \\ H' &= Id \times K_{\mathbb{Z}} \\ H^{\circ} &= Id \times (\mathsf{Maybe} \circ K_{\mathbb{Z}}) \end{aligned}$$

However $HH' \neq H^{\circ}$.

3.2.3 Lifting a Rule

The lifting rule which corresponds to the lifting rule for abstract operational rules is just the application of the previous operation for composing behaviours, with one of the operands being canonically generated.

For abstract operational rules we needed to provide a distributive law of the signature Σ over the behaviour B, which we expected to be generated automatically using the traversability of Σ. In this case we need to automatically generate a distributive law of a monad over a copointed endofunctor.

3.3 Big-step versus small-step

When writing big-step operational semantics all behaviours are of the form K_V, where K is the constant functor and V are the values the big-step relation evaluates to. This is true even if we have side-effects, because of the isomorphism

$$MK_V \cong K_{MV}$$

[4]Composition of copointed endofunctors yields a copointed endofunctor.

Notably, the constant functor is not pointed, hence it's not monadic nor applicative. This may be problematic, as many of the previous operations rely on the behaviour functor being applicative for automatically obtaining distributive laws. Moreover, we may want to combine languages written in big-step style with languages written in small-step style. For the former problem, we need to obtain an abstract operational rule with an applicative functor. For the latter, we need to make behaviours written in different styles compatible.

One possible solution is to transform every big-step semantics into small-steps. When mixing big-step and small-step styles this transformation is preferable, rather than transforming small-step into big-step, since it is known that small-step is more expressive than big-step. Also, is the only sensible direction for obtaining applicative behaviours.

Papaspyrou [28] uses the free monad on a monad to obtain the resumptions of that monad. Resumptions, in a non-deterministic context, allow him to consider interleaving semantics. We were inspired by this into thinking that one way to achieve the transformation from big-step to small-step is by calculating the free monad on the behaviour. The free monad on a constant functor K_V is a functor $X + K_V$, where the extra "X" accounts for the possibility of making a small step without reaching a final value. The introduced variables stand for values not yet calculated. We already had an idea of how to implement this transformation, but the algorithm is not general enough, so there is more work to be done in this regard.

4 Research Directions

In this final section, we outline some possible directions for deepening and extending the research done so far.

- We need to find more examples of modularity problems to test the usefulness and limitations of the operations on SOS. These limitations might suggest the need for more expressive or different operations. We are also interested in finding a minimal set of operations to take as our core theory. Working with more examples might also shed some light on the right level of abstraction for the operations.

- One of the features missing in the languages we have considered is variable binding. It would be interesting to see how well our operations perform if we work on Fiore's categories of presheaves [7, 9, 8]. This would imply an added complexity which is not needed for non-binding constructs. Consequently, another interesting question is to analyse if we can define binding constructs in a category of presheaves and combine them with not-binding constructs defined in **Set**. In this way, the added complexity would only be present where necessary.

- Recursion has been considered in [37] by means of guarded equations and [14] by working in a CPO-enriched category. However, preliminary work based on coalgebraic solutions to recursion schemes [10, 25] suggest that there is no need to add anything to bialgebras to handle recursion. Formal proofs and examples are still needed to support this conjecture.

- Modularity of operational semantics can be applied to the construction of modular compilers. So, the analysis of compilation to abstract machines in a bialgebraic setting is of interest. Watanabe [39] did some research on behaviour-preserving translation of languages which might be relevant here.

- There is more work to be done on the transformation of big-step semantics to small-step semantics. Also, it would be desirable to obtain a good abstraction of evaluation order, so as to change from, say, left-to-right evaluation order to non-deterministic evaluation. Papaspyrou [28] defines an operation called *exhaust*, which evaluates a given term atomically. This operation also appears when defining the semantics of STM, so it would be appealing to be able to represent it in the bialgebraic framework.

- If we want to communicate the semantics obtained after the application of several operations, we should be able to print it as a traditional SOS. We might also want to execute a program step by step to test a given semantics. This means that we can no longer represent our semantics as functions in our implementation language (as in our current implementation).We should represent the semantics as a datatype and do a symbolic evaluation.

- While the bialgebraic framework was developed for representing the dynamic semantics of programming languages, it may also be used to model static semantics. So, we may find another source of examples and applications in static semantics, like type inference systems and secure information flow systems.

References

[1] Luca Aceto, Wan Fokkink, and Chris Verhoef. Conservative extension in structural operational semantics. In *Current Trends in Theoretical Computer Science*, pages 504–524. 2001.

[2] Falk Bartels. GSOS for probabilistic transition systems (extended abstract). In Lawrence S. Moss, editor, *Proc. Coalgebraic Methods in Computer Science (CMCS 2002)*, volume 65 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.

[3] Falk Bartels. Generalised coinduction. *Mathematical Structures in Computer Science*, 13(2):321–348, Apr 2003.

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

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

[6] Robert de Simone. Higher-level synchronising devices in meije-sccs. *Theor. Comput. Sci.*, 37:245–267, 1985.

[7] Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding (extended abstract). In *Proc. 14^{th} LICS Conf.*, pages 193–202. IEEE, Computer Society Press, 1999.

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

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

[10] N Ghani, C Lüth, and de Marchi. Solving algebraic equations using coalgebra. *Journal of Theoretical Informatics and Applications*, 37:301–314, 2003.

[11] Marco Kick. Bialgebraic modelling of timed processes. In P. Widmayer, F. Triguero, R. Morales, M. Hennessy, S. Eidenbenz, and R. Conejo, editors, *Proceedings ICALP'02*, volume 2380 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

[12] Marco Kick. Rule formats for timed processes. *Electr. Notes Theor. Comput. Sci.*, 68(1), 2002.

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

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

[15] Bartek Klin. A coalgebraic approach to process equivalence and a coinduction principle for traces. In *Proc. CMCS 2004*, volume 106 of *ENTCS*, 2004.

[16] Bartek Klin. From bialgebraic semantics to congruence formats. In *Proc. SOS 2004*, volume 128 of *ENTCS*, 2005.

[17] Bartosz Klin. *An Abstract Approach to Process Equivalence for Well-Behaved Operational Semantics.* PhD dissertation, BRICS, Aarhus University, 2004.

[18] J. Lambek. From lambda calculus to cartesian closed categories. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 375–402. Academic Press, 1980.

[19] Ralf Lämmel. Evolution of rule-based programs. *J. Log. Algebr. Program.*, 60-61:141–193, 2004.

[20] Marina Lenisa, John Power, and Hiroshi 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.

[21] Marina Lenisa, John Power, and Hiroshi Watanabe. Category theory for operational semantics. *Theor. Comput. Sci.*, 327(1-2):135–154, 2004.

[22] 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, NY, USA, 1995. ACM Press.

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

[24] Conor McBride and Ross Paterson. Applicative programming with effects, 2006. Accepted by the Journal of Functional Programming.

[25] Stefan Milius and Lawrence S. Moss. The category theoretic solution of recursive program schemes. In José Luiz Fiadeiro, Neil Harman, Markus Roggenbach, and Jan J. M. M. Rutten, editors, *CALCO 2005*, volume 3629 of *Lecture Notes in Computer Science*, pages 293–312. Springer, 2005.

[26] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

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

[28] Nikolaos S. Papaspyrou. Denotational semantics of evaluation order in expressions with side effects. In N. Mastorakis, editor, *Recent Advances in Information Science and Technology*, pages 87–94. World Scientific, Singapore, 1998.

[29] Gordon Plotkin. Bialgebraic semantics and recursion. In Marina Lenisa Andrea Corradini and Ugo Montanari, editors, *Proc. Coalgebraic Methods in Computer Science (CMCS 2001)*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

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

[31] Gordon D. Plotkin. The origins of structural operational semantics. *J. Log. Algebr. Program.*, 60-61:3–15, 2004.

[32] John Power. Towards a theory of mathematical operational semantics. *Electr. Notes Theor. Comput. Sci.*, 82(1), 2003.

[33] Dana Scott. Continuous lattices. In F. W. Lawvere, editor, *Toposes, Algebraic Geometry, and Logic*, number 274 in Lecture Notes in Mathematics, pages 97–136. Springer-Verlag, 1972.

[34] Michael B. Smyth and Gordon D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Comput.*, 11(4):761–783, 1982.

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

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

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

[38] Mitchell Wand. On the recursive specification of data types. In E. Manes, editor, *Category Theory Applied to Computation and Control*, volume 25 of *Lecture Notes in Computer Science*, pages 214–217. Springer-Verlag, Berlin, Heidelberg, and New York, 1975.

[39] Hiroshi Watanabe. Well-behaved translations between structural operational semantics. *Electr. Notes Theor. Comput. Sci.*, 65(1), 2002.

A Implementation in Haskell

A.1 Free monad on a signature

```
data T s x = Var x
           | Term{ term :: s (T s x) }
tfold :: Functor s ⇒ (x → z) → (s z → z) → T s x → z
tfold f g (Var x)  = f x
tfold f g (Term x) = g $ fmap (tfold f g) x
instance Functor s ⇒ Functor (T s) where
  fmap f (Var x)   = Var $ f x
  fmap f (Term fx) = Term $ fmap (fmap f) fx
instance Functor s ⇒ Monad (T s) where
  return = Var
  x ≫= f = tfold f Term x
```

Canonical transformation from a signature to its free monad.

$sigma2monad :: \mathsf{Functor}\ s \Rightarrow s\ a \rightarrow \mathsf{T}\ s\ a$
$sigma2monad = \mathsf{Term} \cdot fmap\ \mathsf{Var}$

A.2 Sum Functor

$\textbf{data}\ \mathsf{Sum}\ f\ g\ a = \mathsf{Inl}\ (f\ a) \mid \mathsf{Inr}\ (g\ a)$

$\textbf{instance}\ (\mathsf{Functor}\ s1, \mathsf{Functor}\ s2) \Rightarrow \mathsf{Functor}\ (\mathsf{Sum}\ s1\ s2)\ \textbf{where}$
$\quad fmap\ f\ (\mathsf{Inl}\ a) \ = \mathsf{Inl}\ (fmap\ f\ a)$
$\quad fmap\ f\ (\mathsf{Inr}\ a) \ = \mathsf{Inr}\ (fmap\ f\ a)$

$copair :: (s1\ a \rightarrow b) \rightarrow (s2\ a \rightarrow b) \rightarrow \mathsf{Sum}\ s1\ s2\ a \rightarrow b$
$copair\ f\ g\ (\mathsf{Inl}\ a) = f\ a$
$copair\ f\ g\ (\mathsf{Inr}\ b) = g\ b$

A.3 Copointed endofunctors

$\textbf{class}\ \mathsf{Functor}\ f \Rightarrow \mathsf{Copointed}\ f\ \textbf{where}$
$\quad copoint :: f\ a \rightarrow a$

Composition of copointed endofunctors is copointed.

$\textbf{instance}\ (\mathsf{Copointed}\ f, \mathsf{Functor}\ g, \mathsf{Copointed}\ g) \Rightarrow$
$\qquad \mathsf{Copointed}\ (\mathsf{Comp}\ f\ g)\ \textbf{where}$
$\quad copoint = copoint \cdot copoint \cdot comp$

A.3.1 Free Copointed endofunctors

$\textbf{newtype}\ \mathsf{Freecop}\ f\ a = \mathsf{Cop}\{\ decop :: (a, f\ a)\}$

$\textbf{instance}\ \mathsf{Functor}\ f \Rightarrow \mathsf{Functor}\ (\mathsf{Freecop}\ f)\ \textbf{where}$
$\quad fmap\ g\ (\mathsf{Cop}\ (a, fx)) = \mathsf{Cop}\ (g\ a, fmap\ g\ fx)$

$\textbf{instance}\ \mathsf{Functor}\ f \Rightarrow \mathsf{Copointed}\ (\mathsf{Freecop}\ f)\ \textbf{where}$
$\quad copoint\ (\mathsf{Cop}\ (x, fx)) = x$

One projection is given by copoint. The other projection is

$forgetcopoint\ (\mathsf{Cop}\ (x, fx)) = fx$

A.4 Abstract Operational Rules and Distributive Law

Abstract operational rules are

$\textbf{type}\ \mathsf{OpRule}\ s\ b = \forall\ a \cdot s\ (a, b\ a) \rightarrow b\ (\mathsf{T}\ s\ a)$

A CopRule should respect the copoint of h.

$\textbf{type}\ \mathsf{CopRule}\ s\ h = \forall\ a \cdot s\ (h\ a) \rightarrow h\ (\mathsf{T}\ s\ a)$

freecop translates from abstract operational rules to CopRules over a cofreely generated copointed endofunctor on the behaviour b.

$pair :: (a \rightarrow b) \rightarrow (a \rightarrow c) \rightarrow a \rightarrow (b, c)$
$pair\ f\ g\ a = (f\ a, g\ a)$

$$
\begin{aligned}
&\mathit{freecop} :: (\mathsf{Functor}\ s, \mathsf{Functor}\ b) \Rightarrow \\
&\qquad \mathsf{OpRule}\ s\ b \rightarrow \mathsf{CopRule}\ s\ (\mathsf{Freecop}\ b) \\
&\mathit{freecop}\ f = \mathsf{Cop} \cdot \mathit{pair}\ \mathit{point}\ (f \cdot \mathit{fmap}\ \mathit{decop}) \\
&\qquad \textbf{where}\ \mathit{point} = \mathsf{Term} \cdot \mathit{fmap}\ (\mathsf{Var} \cdot \mathit{copoint})
\end{aligned}
$$

op2dist, given a CopRule, it yields a distributive law of a monad over a copointed endofunctor.

$$
\begin{aligned}
&\mathit{cop2dist} :: (\mathsf{Functor}\ h, \mathsf{Functor}\ s) \Rightarrow \\
&\qquad \mathsf{CopRule}\ s\ h \\
&\quad \rightarrow \quad (\forall\ a \cdot \mathsf{T}\ s\ (h\ a) \rightarrow h\ (\mathsf{T}\ s\ a)) \\
&\mathit{cop2dist}\ \mathit{op}\ (\mathsf{Var}\ \mathit{ha}) = \mathit{fmap}\ \mathsf{Var}\ \mathit{ha} \\
&\mathit{cop2dist}\ \mathit{op}\ (\mathsf{Term}\ \mathit{stha}) = \mathit{fmap}\ \mathit{join}\ (\mathit{op}\ (\mathit{fmap}\ (\mathit{cop2dist}\ \mathit{op})\ \mathit{stha}))
\end{aligned}
$$

A.5 The operational monad

$$
\begin{aligned}
&\mathit{opmonad} :: (\mathsf{Functor}\ s, \mathsf{Functor}\ h) \Rightarrow \\
&\qquad \mathsf{CopRule}\ s\ h \rightarrow (a \rightarrow h\ a) \rightarrow \mathsf{T}\ s\ a \rightarrow h\ (\mathsf{T}\ s\ a) \\
&\mathit{opmonad}\ \mathit{op}\ k = \mathit{tfold}\ (\mathit{fmap}\ \mathsf{Var} \cdot k)\ (\mathit{fmap}\ \mathit{join} \cdot \mathit{op}) \\
&\mathit{opreturn} :: (\mathsf{Functor}\ s, \mathsf{Functor}\ h) \Rightarrow \\
&\qquad \mathsf{CopRule}\ s\ h \rightarrow (a \rightarrow h\ a) \rightarrow \mathsf{T}\ s\ a \rightarrow h\ (\mathsf{T}\ s\ a) \\
&\mathit{opreturn}\ \mathit{op}\ k = \mathit{opmonad}\ \mathit{op}\ k \\
&\mathit{opmu} :: (\mathsf{Functor}\ s, \mathsf{Functor}\ h) \Rightarrow \\
&\qquad \mathsf{CopRule}\ s\ h \rightarrow \\
&\qquad (\mathsf{T}\ s\ (\mathsf{T}\ s\ a) \rightarrow h\ (\mathsf{T}\ s\ (\mathsf{T}\ s\ a))) \rightarrow \\
&\qquad \mathsf{T}\ s\ a \rightarrow h\ (\mathsf{T}\ s\ a) \\
&\mathit{opmu}\ \mathit{op}\ \mathit{tta}\ \mathit{ta} = \mathit{fmap}\ \mathit{join}\ (\mathit{tta}\ (\mathit{return}\ \mathit{ta}))
\end{aligned}
$$

A.6 Running programs

A Program is a closed term.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Zero} \\
&\textbf{type}\ \mathsf{Program}\ s = \mathsf{T}\ s\ \mathsf{Zero} \\
&\mathit{exec} :: (\mathsf{Functor}\ b, \mathsf{Functor}\ s, \mathsf{Traversable}\ s) \Rightarrow \\
&\qquad \mathsf{OpRule}\ s\ b \rightarrow \mathsf{Program}\ s \rightarrow \mathsf{Program}\ (\mathsf{Freecop}\ b) \\
&\mathit{exec}\ \mathit{op} = \mathit{unfold}\ (\mathit{opmonad}\ (\mathit{freecop}\ \mathit{op})\ \bot) \\
&\mathit{unfold} :: \mathsf{Functor}\ s \Rightarrow (b \rightarrow s\ b) \rightarrow b \rightarrow \mathsf{Program}\ s \\
&\mathit{unfold}\ g = \mathsf{Term} \cdot \mathit{fmap}\ (\mathit{unfold}\ g) \cdot g
\end{aligned}
$$

A.7 Operations

- Joining syntax

$$
\begin{aligned}
&\mathit{joinOS} :: (\mathsf{Functor}\ \mathit{s1}, \mathsf{Functor}\ \mathit{s2}, \mathsf{Functor}\ b) \Rightarrow \\
&\qquad \mathsf{OpRule}\ \mathit{s1}\ b \\
&\qquad \rightarrow \mathsf{OpRule}\ \mathit{s2}\ b
\end{aligned}
$$

$$\rightarrow \mathsf{OpRule}\ (\mathsf{Sum}\ s1\ s2)\ b$$

$$joinOS\ op1\ op2 = copair\ (extendR\ op1)\ (extendL\ op2)$$

$$extendR :: (\mathsf{Functor}\ s1, \mathsf{Functor}\ b) \Rightarrow \mathsf{OpRule}\ s1\ b \rightarrow (\forall\ a \cdot s1\ (a, b\ a) \rightarrow b\ (\mathsf{T}\ (\mathsf{Sum}\ s1\ s2)\ a))$$

$$extendR\ op = fmap\ (tfold\ \mathsf{Var}\ (\mathsf{Term} \cdot \mathsf{Inl})) \cdot op$$

$$extendL :: (\mathsf{Functor}\ s2, \mathsf{Functor}\ b) \Rightarrow \mathsf{OpRule}\ s2\ b \rightarrow (\forall\ a \cdot s2\ (a, b\ a) \rightarrow b\ (\mathsf{T}\ (\mathsf{Sum}\ s1\ s2)\ a))$$

$$extendL\ op = fmap\ (tfold\ \mathsf{Var}\ (\mathsf{Term} \cdot \mathsf{Inr})) \cdot op$$

- Lift operation

$$liftOS :: (\mathsf{Applicative}\ fx, \mathsf{Functor}\ s, \mathsf{Traversable}\ s, \mathsf{Functor}\ b) \Rightarrow \mathsf{OpRule}\ s\ b \rightarrow \mathsf{OpRule}\ s\ (\mathsf{Comp}\ fx\ b)$$

$$liftOS\ op = \mathsf{Comp} \cdot fmap\ op \cdot dist \cdot fmap\ strength \cdot fmap\ (pair\ fst\ (comp \cdot snd))$$

A.8 Rule Transformers

$$\textbf{type}\ \mathsf{Model}\ s\ b = \forall\ a \cdot s\ (b\ a) \rightarrow b\ a$$

Any model can be turned into an abstract operational rule.

$$model2aor :: (\mathsf{Functor}\ s, \mathsf{Functor}\ b) \Rightarrow \mathsf{Model}\ s\ b \rightarrow \mathsf{OpRule}\ s\ b$$

$$model2aor\ model = fmap\ \mathsf{Var} \cdot model \cdot fmap\ snd$$

We can lift a model to the right and to the left

$$liftModelR :: (\mathsf{Functor}\ s, \mathsf{Functor}\ h, \mathsf{Functor}\ j) \Rightarrow \mathsf{Model}\ s\ h \rightarrow \mathsf{Model}\ s\ (\mathsf{Comp}\ h\ j)$$

$$liftModelR\ bf = \mathsf{Comp} \cdot bf \cdot fmap\ comp$$

$$liftModelL :: (\mathsf{Functor}\ s, \mathsf{Traversable}\ s, \mathsf{Applicative}\ j) \Rightarrow \mathsf{Model}\ s\ h \rightarrow \mathsf{Model}\ s\ (\mathsf{Comp}\ j\ h)$$

$$liftModelL\ ot = \mathsf{Comp} \cdot fmap\ ot \cdot dist \cdot fmap\ comp$$

A.9 Big-Step semantics

$$\textbf{newtype}\ \mathsf{Const}\ b\ a = \mathsf{Const}\{\ deconst :: b\ \}$$

$$\textbf{instance}\ \mathsf{Show}\ b \Rightarrow \mathsf{Show}\ (\mathsf{Const}\ b\ a)\ \textbf{where}$$

$$show\ (\mathsf{Const}\ b) = \texttt{"Const "} +\!\!+\ show\ b$$

```
instance Functor (Const b) where
  fmap _ (Const b) = Const b
```

Moving effects out and into the constant functor

```
fxk2kfx :: Functor fx ⇒
              Comp fx (Const b) a
         →    Const (fx b) a
fxk2kfx = Const · fmap deconst · comp
kfx2fxk :: Functor fx ⇒
              Const (fx b) a
         →    Comp fx (Const b) a
kfx2fxk = Comp · fmap Const · deconst
extractFxs :: (Functor s, Functor fx) ⇒
                CopRule s (Freecop (Const (fx b)))
              → CopRule s (Freecop (Comp fx (Const b)))
extractFxs op = Cop · pair copoint (kfx2fxk · forgetcopoint)
                · op
                · fmap (Cop · pair copoint (fxk2kfx · forgetcopoint))
```

A.10 Applicative Functors

```
infixl 4 ⊛
class Functor f ⇒ Applicative f where
  pure     :: a → f a
  (⊛)      :: f (a → b) → f a → f b
  strength :: (b, f a) → f (b, a)
  strength (b, fa) = pure (, ) ⊛ pure b ⊛ fa
class Traversable t where
  traverse :: Applicative f ⇒ (a → f b) → t a     → f (t b)
  dist     :: Applicative f ⇒              t (f a) → f (t a)
  dist     = traverse id
newtype Comp f g a = Comp{ comp :: f (g a)}
instance (Applicative f, Applicative g) ⇒
         Applicative (Comp f g) where
  pure x                = Comp $ pure $ pure x
  Comp fs ⊛ Comp xs = Comp $ pure (⊛) ⊛ fs ⊛ xs
instance (Functor f, Functor g) ⇒
         Functor (Comp f g) where
  fmap h (Comp fga) = Comp $ fmap (fmap h) fga
instance (Traversable s1, Traversable s2) ⇒
         Traversable (Sum s1 s2) where
  traverse f (Inl s1) = pure Inl ⊛ traverse f s1
  traverse f (Inr s2) = pure Inr ⊛ traverse f s2
instance Applicative f ⇒
```

```
            Applicative (Freecop f) where
   pure x = Cop (x, pure x)
   Cop (f, fs) ⊛ Cop (x, xs) = Cop (f x, fs ⊛ xs)
instance Applicative Identity where
   pure = return
   (⊛) = ap
instance Applicative Maybe where
   pure = return
   (⊛) = ap
```

A.11 Showing fixpoints

```
class PreservesShow f where
   preservesShow′ :: (x → String) → f x → String
   preservesShow :: Show x ⇒ f x → String
   preservesShow = preservesShow′ show
instance (PreservesShow s1, PreservesShow s2) ⇒
            PreservesShow (Sum s1 s2) where
   preservesShow′ show (Inl x) = preservesShow′ show x
   preservesShow′ show (Inr x) = preservesShow′ show x
instance Show Zero where
instance PreservesShow f ⇒ PreservesShow (Freecop f) where
   preservesShow′ show (Cop (a, f)) = preservesShow′ show f
instance (PreservesShow s, Functor s, Show a) ⇒ Show (T s a) where
   show (Var x)     = "Var (" ++ show x ++ ")"
   show (Term sa) = "Term (" ++ preservesShow sa ++ ")"
instance (PreservesShow f, Functor f, PreservesShow g) ⇒
            PreservesShow (Comp f g) where
   preservesShow′ sh (Comp a) = preservesShow′ (preservesShow′ sh) a
```

A.12 Freely generated behaviours

The following code is a first attempt at the transformation from big-step to small-step (it's not correct).

```
freebehaviour :: (Functor s, Functor h, Applicative h, Traversable s) ⇒
                        (∀ a · s (h a)
                     → h (s a))
                     → (∀ a · s (T h a)
                     → T h (s a))
freebehaviour op t
   | gen_all isVar t = (Term · fmap return
                              · op · fmap change) t
   | otherwise          = (Term · fmap (freebehaviour op)
```

$\cdot\ op \cdot fmap\ extract\_layer)\ t$
where $extract\_layer$ (Var x) $= (pure \cdot return)\ x$
$extract\_layer$ (Term x) $= x$
$isVar$ (Var _) = True
$isVar$ _ = False
$change$ (Var x) $= pure\ x$

And the machinery for a generic fold.

class Monoid o **where**
$zero :: o$
$plus :: o \rightarrow o \rightarrow o$
newtype Accy $o\ a$ = Acc{ $acc :: o$ }
instance Functor (Accy o) **where**
$fmap\ f$ (Acc x) = Acc x
instance Monoid $o \Rightarrow$ Applicative (Accy o) **where**
$pure$ _ = Acc $zero$
Acc $o1 \circledast$ Acc $o2$ = Acc ($plus\ o1\ o2$)
$accumulate$ :: (Traversable s, Monoid o) $\Rightarrow$
$(a \rightarrow o) \rightarrow s\ a \rightarrow o$
$accumulate\ f = acc \cdot traverse$ (Acc $\cdot f$)
newtype Musty = Must{ $must$ :: Bool }
instance Monoid Musty **where**
$zero$ = Must True
$plus$ (Must x) (Must y) = Must ($x \wedge y$)
gen_all :: Traversable $s \Rightarrow (a \rightarrow$ Bool) $\rightarrow s\ a \rightarrow$ Bool
$gen\_all\ p = must \cdot accumulate$ (Must $\cdot\ p$)

B Examples

B.1 A Small Arithmetics Language

data Arith a = Con Int | Add $a\ a$
type ArithVal = Int
instance Functor Arith **where**
$fmap\ f$ (Con x) = Con x
$fmap\ f$ (Add $t\ u$) = Add ($f\ t$) ($f\ u$)
instance Traversable Arith **where**
$traverse\ g$ (Con x) = $pure$ \$ Con x
$traverse\ g$ (Add $t\ u$) = $pure$ Add $\circledast\ g\ t \circledast g\ u$

B.1.1 Behaviour for small step

The behaviour is either a (final) value or a computation.

```
data VC value a = K value | C a
instance Functor (VC value) where
  fmap f (K x) = K x
  fmap f (C c) = C (f c)
instance Monad (VC value) where
  return      = C
  (K v) >>= f = K v
  (C c) >>= f = f c
```

B.1.2 Small Step Semantics

```
detsmallArith :: OpRule Arith (VC ArithVal)
detsmallArith (Con x)   = K x
detsmallArith (Add t u) = case snd t of
  K v  → case snd u of
           K v′ → K (v + v′)
           C u′ → C (Term (Add (Var (fst t)) (Var u′)))
  C t′ → C (Term (Add (Var t′) (Var (fst u))))
```

B.1.3 Big Step Semantics

```
modelArith :: Model Arith (Const ArithVal)
modelArith (Con x)   = Const x
modelArith (Add t u) = Const (deconst t + deconst u)
natopArith :: OpRule Arith (Const ArithVal)
natopArith = model2aor modelArith
```

B.2 Exceptions Language

```
data Exc a = Throw | Catch a a
instance Functor Exc where
  fmap f Throw       = Throw
  fmap f (Catch a b) = Catch (f a) (f b)
instance Traversable Exc where
  traverse g Throw       = pure $ Throw
  traverse g (Catch a b) = pure Catch ⊛ g a ⊛ g b
```

B.2.1 Small Step Semantics

```
detsmallExc :: OpRule Exc (Comp Maybe (VC v))
detsmallExc Throw       = Comp Nothing
detsmallExc (Catch t u) = Comp $ Just $ case comp (snd t) of
  Nothing      → C $ Var (fst u)
  Just (K v)   → K v
  Just (C t′)  → C (Term (Catch (Var t′) (Var (fst u))))
```

B.2.2 Big Step Semantics

$$
\begin{aligned}
&\mathit{modelExc} :: \mathsf{Model\ Exc\ (Comp\ Maybe}\ b)\\
&\mathit{modelExc}\ \mathsf{Throw} \qquad\quad = \mathsf{Comp\ Nothing}\\
&\mathit{modelExc}\ (\mathsf{Catch}\ t\ u) = \mathsf{Comp}\ (\textbf{case}\ (\mathit{comp}\ t)\ \textbf{of}\\
&\qquad\qquad\qquad\qquad\quad \mathsf{Nothing} \to \mathit{comp}\ u\\
&\qquad\qquad\qquad\qquad\quad \mathsf{Just}\ x \quad \to \mathsf{Just}\ x)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{natopExc} :: \mathsf{Functor}\ b \Rightarrow \mathsf{OpRule\ Exc\ (Comp\ Maybe}\ b)\\
&\mathit{natopExc} = \mathit{model2aor}\ \mathit{modelExc}
\end{aligned}
$$

B.3 A language with state

$\textbf{newtype}\ \mathsf{Tick}\ a = \mathsf{Tick}\ a$

$$
\begin{aligned}
&\textbf{instance}\ \mathsf{Functor\ Tick}\ \textbf{where}\\
&\quad \mathit{fmap}\ f\ (\mathsf{Tick}\ a) = \mathsf{Tick}\ (f\ a)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance}\ \mathsf{Traversable\ Tick}\ \textbf{where}\\
&\quad \mathit{traverse}\ g\ (\mathsf{Tick}\ a) = \mathit{pure}\ \mathsf{Tick} \circledast g\ a
\end{aligned}
$$

$\textbf{type}\ \mathsf{MyState} = \mathsf{Int}$

$$
\begin{aligned}
&\textbf{instance}\ \mathsf{Applicative\ (State}\ b)\ \textbf{where}\\
&\quad \mathit{pure} = \mathit{return}\\
&\quad (\circledast)\ \ = \mathit{ap}
\end{aligned}
$$

B.3.1 Big Step Semantics

$$
\begin{aligned}
&\mathit{modelState} :: \mathsf{Model\ Tick\ (Comp\ (State\ MyState)}\ b)\\
&\mathit{modelState}\ (\mathsf{Tick}\ t) = \mathsf{Comp}\ (\textbf{do}\ s \leftarrow \mathit{get}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{put}\ (s+1)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{comp}\ t\\
&\quad)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{natopState} :: \mathsf{Functor}\ b \Rightarrow \mathsf{OpRule\ Tick\ (Comp\ (State\ MyState)}\ b)\\
&\mathit{natopState} = \mathit{model2aor}\ \mathit{modelState}
\end{aligned}
$$

B.4 A Trace Language

$\textbf{newtype}\ \mathsf{Trace}\ a = \mathsf{Trace}\ (a, \mathsf{String})$

$$
\begin{aligned}
&\textbf{instance}\ \mathsf{Functor\ Trace}\ \textbf{where}\\
&\quad \mathit{fmap}\ f\ (\mathsf{Trace}\ (a, o)) = \mathsf{Trace}\ (f\ a, o)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance}\ \mathsf{Traversable\ Trace}\ \textbf{where}\\
&\quad \mathit{traverse}\ g\ (\mathsf{Trace}\ (a, o)) = \mathit{pure}\ \mathsf{Trace} \circledast \mathit{fmap}\ (\mathit{flip}\ (,)\ o)\ (g\ a)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance}\ \mathsf{Applicative\ (Writer\ String)}\ \textbf{where}\\
&\quad \mathit{pure} = \mathit{return}\\
&\quad (\circledast)\ \ = \mathit{ap}
\end{aligned}
$$

B.4.1 Big Step Semantics

$$
\begin{array}{l}
\mathit{modelTrace} :: \mathsf{Model\ Trace\ (Comp\ (Writer\ String)}\ b) \\
\mathit{modelTrace}\ (\mathsf{Trace}\ (t, o)) = \mathsf{Comp}\ (\mathit{tell}\ o \gg \mathit{comp}\ t) \\
\mathit{natopTrace} :: \mathsf{Functor}\ b \Rightarrow \mathsf{OpRule\ Trace\ (Comp\ (Writer\ String)}\ b) \\
\mathit{natopTrace} = \mathit{model2aor\ modelTrace}
\end{array}
$$

B.5 Some Test programs

We define the semantics of arithmetics with exceptions, both for big-step and small step semantics, and some test programs.

$$
\begin{array}{l}
\mathit{opAE}\ \ = \mathit{joinOS}\ (\mathit{liftOS\ natopArith})\ \mathit{natopExc} \\
\mathit{opAE}' = \mathit{joinOS}\ (\mathit{liftOS\ detsmallArith})\ \mathit{detsmallExc} \\
\mathit{testA1} :: \mathsf{Program\ Arith} \\
\mathit{testA1} = \mathsf{Term\ (Add\ (Term\ (Con}\ 3)) \\
\qquad\qquad\qquad\qquad (\mathsf{Term\ (Add\ (Term\ (Con}\ 2)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{Term\ (Con}\ 1))))) \\
\mathit{testAE1} :: \mathsf{Program\ (Sum\ Arith\ Exc)} \\
\mathit{testAE1} = \mathit{mycatch}\ (\mathit{add}\ (\mathit{con}\ 3)\ \mathit{throw}) \\
\qquad\qquad\qquad\qquad (\mathit{add}\ (\mathit{add}\ (\mathit{con}\ 2)\ (\mathit{con}\ 2))\ (\mathit{con}\ 5)) \\
\mathit{con}\qquad\quad\ \ = \mathsf{Term} \cdot \mathsf{Inl} \cdot \mathsf{Con} \\
\mathit{add}\ x\ y\qquad = \mathsf{Term}\ \$\ \mathsf{Inl}\ \$\ \mathsf{Add}\ x\ y \\
\mathit{mycatch}\ x\ y = \mathsf{Term}\ \$\ \mathsf{Inr}\ \$\ \mathsf{Catch}\ x\ y \\
\mathit{throw}\qquad\ \ = \mathsf{Term}\ \$\ \mathsf{Inr}\ \$\ \mathsf{Throw}
\end{array}
$$

To execute a program in AE with opAE:

$$
\mathit{run} = \mathit{exec\ opAE\ testAE1}
$$

We define the semantics of arithmetics with state.

$$
\begin{array}{l}
\mathit{opAS} = \mathit{joinOS}\ (\mathit{liftOS\ natopArith})\ \mathit{natopState} \\
\mathit{testAS} :: \mathsf{Program\ (Sum\ Arith\ Tick)} \\
\mathit{testAS} = \mathit{tick}\ \$\ \mathit{add}\ (\mathit{con}\ 3)\ (\mathit{tick}\ \$\ \mathit{add}\ (\mathit{con}\ 4)\ (\mathit{con}\ 5)) \\
\mathit{tick}\quad\ = \mathsf{Term} \cdot \mathsf{Inr} \cdot \mathsf{Tick}
\end{array}
$$

Finally, we define the semantics of arithmetics with a trace, and arithmetics with a trace and exceptions.

$$
\begin{array}{l}
\mathit{opAO} = \mathit{joinOS}\ (\mathit{liftOS\ natopArith})\ \mathit{natopTrace} \\
\mathit{opAEO} = \mathit{joinOS}\ (\mathit{liftOS\ opAE})\ \mathit{natopTrace} \\
\mathit{testAO1} :: \mathsf{Program\ (Sum\ Arith\ Trace)} \\
\mathit{testAO1} = \mathit{trAdd}\ \texttt{" + "}\ (\mathit{trCon}\ 3) \\
\qquad\qquad\quad (\mathit{trAdd}\ \texttt{" + "}\ (\mathit{trCon}\ 4) \\
\qquad\qquad\qquad\qquad\qquad (\mathit{trCon}\ 5)) \\
\mathit{testAO2} :: \mathsf{Program\ (Sum\ Arith\ Trace)}
\end{array}
$$

```haskell
testAO2 = trAdd "a1" (trAdd "a2" (con 1)
                                 (trAdd "a4" (con 4) (con 4)))
                     (trAdd "a3" (con 2) (con 6))

mytrace s x = Term $ Inr $ Trace (x, s)
trAdd s x y = mytrace s (add x y)
trCon x     = mytrace (show x) (con x)

testAEO1 :: Program (Sum (Sum Arith Exc) Trace)
testAEO1 = mycatch' (add' (con' 3) (mytrace "Argh!" throw'))
            (add' (add' (con' 2) (con' 2)) (con' 5))

con'          = Term . Inl . Inl . Con
add' x y      = Term $ Inl $ Inl $ Add x y
mycatch' x y  = Term $ Inl $ Inr $ Catch x y
throw'        = Term $ Inl $ Inr $ Throw
```

B.6 Many Show instances

```haskell
instance PreservesShow Arith where
  preservesShow' sh (Con i) = "Con " ++ show i
  preservesShow' sh (Add a b) = sh a ++ " + " ++ sh b

instance PreservesShow Trace where
  preservesShow' sh (Trace (a, s)) = "Trace (" ++ sh a ++ ", " ++ s ++ ")"

instance PreservesShow Exc where
  preservesShow' sh Throw       = "Throw"
  preservesShow' sh (Catch t u) = "Catch " ++ sh t ++ " " ++ sh u

instance PreservesShow Maybe where
  preservesShow' sh Nothing = "Nothing"
  preservesShow' sh (Just x) = "Just (" ++ sh x ++ ")"

instance Show a => PreservesShow (Const a) where
  preservesShow' sh = show

instance Show a => Show (State MyState a) where
  show f = "value: " ++ show x ++ ", count: " ++ show s
    where (x, s) = runState f 0

instance PreservesShow (State MyState) where
  preservesShow' sh f = "value: " ++ sh x ++ ", count: " ++ show s
    where (x, s) = runState f 0

instance Show a => Show (Writer String a) where
  show w = trace ++ show a where (a, trace) = runWriter w

instance Show s => PreservesShow (Writer s) where
  preservesShow' sh (Writer (a, w)) = show w ++ " : " ++ sh a

instance Show val => PreservesShow (VC val) where
  preservesShow' sh (K a) = "K " ++ show a
  preservesShow' sh (C t) = "C (" ++ sh t ++ ")"
```