Available online at [www.sciencedirect.com](http://www.sciencedirect.com/)



[Electronic Notes in Theoretical Computer Science 276 (2011) 81–104](http://dx.doi.org/10.1016/j.entcs.2011.09.016)

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Towards Effects in Mathematical Operational Semantics

Faris Abou-Saleh Dirk Pattinson

*Department of Computing Imperial College London*

**Abstract**

In this paper, we study extensions of mathematical operational semantics with algebraic effects. Our starting point is an effect-free coalgebraic operational semantics, given by a natural transformation of syntax over behaviour. The operational semantics of the extended language arises by distributing program syntax over effects, again inducing a coalgebraic operational semantics, but this time in the Kleisli category for the monad derived from the algebraic effects. The final coalgebra in this Kleisli category then serves as the denotational model. For it to exist, we ensure that the the Kleisli category is enriched over CPOs by considering the monad of possibly infinite terms, extended with a bottom element.

Unlike the effectless setting, not all operational specifications give rise to adequate and compositional semantics. We give a proof of adequacy and compositionality provided the specifications can be described by evaluation-in-context. We illustrate our techniques with a simple extension of (stateless) while programs with global store, i.e. variable lookup.

*Keywords:* Operational semantics, Coalgebras, Effects, Kleisli category, Comodels

# Introduction

Operational and denotational semantics provide orthogonal descriptions of a pro- gramming language: operational semantics specifies the execution of programs, and denotational semantics associates a mathematical meaning to programs. Crucially, denotational semantics is *compositional* so that the denotation of a program can be inferred from the interpretation of its subprograms, and so facilitates the task of compositional verification, either of program properties, or of features of the language at large. The operational and denotational description of a language are connected via the notion of adequacy – operationally equivalent programs should have the same denotation, and vice-versa. The verification of adequacy is usually done on a language-by-language basis and is often tedious for non-trivial languages.

A more general approach was proposed by Turi and Plotkin [[24](#_bookmark41)] in the case of structural operational semantics, assuming that the operational behaviour of a language can be modelled coalgebraically. Then, the semantic domain can be taken as a final coalgebra, and mapping programs to their denotations provides an

1571-0661 © 2011 Elsevier B.V. Open access under [CC BY-NC-ND license](http://creativecommons.org/licenses/by-nc-nd/3.0/).

[doi:10.1016/j.entcs.2011.09.016](http://dx.doi.org/10.1016/j.entcs.2011.09.016)

adequate denotational semantics. The main result of *op.cit.* is that this semantics is compositional if the operational rules can be given in the form of a natural transformation of syntax over behaviour, i.e. they form an *abstract operational semantics*.

While this programme of mathematical operational semantics is well-suited to process-calculi-like languages, it is not yet clear how imperative languages, or in- deed any language that produces computational effects [[14](#_bookmark33),[15](#_bookmark34),[5](#_bookmark21)], can be treated adequately, and a number of difficulties present themselves. Firstly, giving an ab- stract operational semantics relies on syntax and behaviour being defined on the same category, whereas computational effects are are most naturally expressed in the Kleisli category of a monad [[12](#_bookmark31)], which is unsuitable for representating syntax. Second, many of the standard techniques for constructing final coalgebras (that play the roles of denotational domains) fail in a Kleisli category. Finally, working with a Kleisli category to obtain adequate denotational models, we need new proof techniques to show compositionality of the denotational semantics.

In this paper, we make first steps towards abstract operational semantics for languages with effects and address the difficulties outlined above. We begin with a pure, effectless language, and an operational semantics given by a natural transfor- mation. We extend the syntax with computational effects as given by an algebraic theory [[19](#_bookmark37)], and extend the operational semantics to handle these effects; this in- duces an effectful operational model by structural recursion.

To obtain the extended operational semantics, we introduce a notion of ‘dependency-support’ that records which subterms appear in the premises of the operational rules. In particular, this covers the case where operational rules have at most one premise that exhibits nontrivial behaviour.

Once it is obtained, an effectful operational model is considered as a coalgebra in the Kleisli category induced by the algebraic effects. This ensures that when the coalgebra behaviour is iterated, one accumulates the effects arising during program execution. The existence of final coalgebras (denotational models) in this setting is subject to this Kleisli-category being enriched over chain-complete partial orders; we explore the issue of satisfying this requirement in a generic way.

Lastly, unlike the effectless setting, we demonstrate that not all abstract oper- ational semantics are compositional. We give some abstract conditions that guar- antee compositionality; in particular, we show that operational semantics described by evaluation-in-context give rise to compositional semantics.

At this point, our approach has several limitations. We neglect equational speci- fications of algebraic effects which results in a too fine-grained notion of operational equivalence. Moreover, one often wishes to abstract away some aspects of program execution traces, such as the number of steps before termination; we do not do this here. We plan to address these points in future work following [[18](#_bookmark36)] along with more discussion of comodels [[16](#_bookmark35)].

*Related Work.* Algebraic effects have been considered in [[14](#_bookmark33)] in the context of PCF, which has recently been extended to account for a larger class of operational phenomena [[8](#_bookmark26)]. Both papers are mainly operational, and indeed [[14](#_bookmark33)] concludes

with the remark that ‘one would wish to reconcile this work with the co-algebraic treatment of operational semantics’ in [[24](#_bookmark41)]. The framework of abstract operational semantics itself has been extended into various directions [[9](#_bookmark28),[10](#_bookmark29)], but effects have so far been elusive. Our treatment of computational effects is inspired by [[19](#_bookmark37),[6](#_bookmark22)]. Final coalgebras in Kleisli categories were studied in [[4](#_bookmark23)].

# Introducing Effects into Syntax and Behaviour

We recall the mathematical operational semantics of Turi and Plotkin [[23](#_bookmark40),[24](#_bookmark41)] applied to multi-sorted syntax signatures in a category *C*S. In general, we assume that *C* is cartesian closed, with countable products and coproducts (*ω*-arities are needed for effects, e.g. the read-operation of [[15](#_bookmark34)]). We also assume that countable polynomial functors have initial algebras and final coalgebras.

**Definition 2.1** *If F* : *C → C is a functor, an F* -algebra *is a pair ⟨A, α⟩ where A ∈C and α* : *FA → A. Dually, an F* -coalgebra *is a pair ⟨C, γ⟩ where γ* : *C → FC. Algebras and coalgebras form categories that we denote by* Alg(*F* ) *and* Coalg(*F* )*, respectively. An* S-sorted signature Sig *over a set* S *of sorts has function symbols with (at most countable) arities, written as f* : (*si*)0*≤i<α → sf where si ∈* S*, and α ≤ ω is the arity of f. If arities are clear from the context, we write f* : (*si*) *→ sf . Every* S*-sorted signature* Sig *induces a functor* ΣSig : *C*S *→ C*S *given by*

ΣSig(*Xs*)*s∈*S = (`*f* :(*si*)*→s* Q*i<α Xsi* )*s∈*S *where α is the arity of f.*

For our running example, we take while programs in *C* = Set that we describe via an extension of an effectless language – the fragment of while without variable lookup *x* or update *x*:=*n* – with effects for global state. The base language that we call *stateless while* mainly serves the purpose of exemplifying our techniques.

**Example 2.2** Let S = *{***Num***,* **Bool***,* **Prog***}* denote numerical, boolean and pro- gram expressions, respectively. The signature of *stateless while* is given by

*N* ::=*n | N* + *N | N ∗ N |* +*n*(*N* ) *| ∗n*(*N* )

*E* ::=*b | N* = *N | N ≤ N |* =*n* (*N* ) *| ≤n* (*N* ) *| ¬E | E ∧ E*

*P* ::=skip *| P* ; *P |* while (*E*) do *{P}|* if (*E*) then *{P}* else *{P}*

where *n* is a numeral in N, and *b* is a boolean in B = *{true, false}*. The auxil- iary operators +*n*(*−*), *∗n*(*−*), etc. record partial results of evaluating expressions like *N* + *M* from left-to-rightAs explained in Section [3](#_bookmark8), these auxiliary operators are introduced to give a correct effectful operational semantics for + and *∗*. The grammar above induces a syntax functor Σ : Set3 *→* Set3 in a straightforward way.

**Remark 2.3** *If* Σ *is a syntax functor induced by a signature, the assumptions on C provide us with a free construction F E U where U* : Alg(Σ) *→ C*S *is the forgetful functor on* Σ*-algebras. We write T*Σ = *UF; intuitively, the s-component* (*T*Σ*X*)*s contains the terms of sort s over sorted variables* (*Xs*)*.*

Operational models of languages given by a signature Sig are coalgebras *T*Σ0 *→ BT*Σ0 where *B* : *C*S *→ C*S is an endofunctor describing the *behaviour* of programs and *T*Σ0 is the set of closed programs. Their operational semantics may be specified by an abstract operational semantics (abstract OS):

**Definition 2.4** *An abstract operational semantics (abstract OS) for syntax and behaviour functors* Σ*,B is a natural transformation ρX* : Σ(*X × BX*) *−→ BT*Σ*X.*

**Example 2.5** The observable behaviours of *stateless while* are deterministic tran- sitions that may produce a value (for boolean and arithmetic expressions) whereas for programs, only termination is observable. That is, we let *B*(*N, E, P* ) = (*N* + N*,E* + B*,P* + 1), or equivalently (*N, E, P* )+ (N*,* B*,* 1).

For numeric expressions *u, uj ∈ N* , we write *u → uj* to represent a transition

from *u* to *uj* – formally, *u* has behaviour inr *j √*

(*u* ) *∈ N* + N. We write *u → n* if *u*

terminates with value *n ∈* N – formally inr(*n*) *∈ N* + N. Similar notation applies

*√*

for other types. We write *p →* if program *p ∈ P* terminates.

The abstract operational semantics for stateless while may be presented by stan- dard operational rules such as:

*√ √ √*

*u → u′*

*u* + *v → u′* + *v*

*√*

*u → n*

*u* + *v →* +*n*(*v*)

*v → v′*

+*n*(*v*) *→* +*n*(*v′*)

*√*

*v → m*

*√*

*n → n b → b* skip *→*

*e → e′*

*√*

*e → true*

+*n*(*v*) *→ n* + *m*

(etc*.*)

if (*e*) then *{p}* else *{q} →* if (*e′*) then *{p}* else *{q}*

if (*e*) then *{p}* else *{q} → p*

*p → p′*

*p* ; *q → p′* ; *q*

*√*

*p →*

*p* ; *q → q* while (*e*) do *{p} →* if (*e*) then *{p*; while (*e*) do *{p}}* else *{*skip*}*

and other familiar rules for the remaining operators *∗, ¬,* =*, <*= *, ∧* (see e.g. [[13](#_bookmark32)]). It can easily be verified that these rules induce a natural transformation *ρ* : Σ(*X × BX*) *→ BT*Σ*X* that distributes syntax over behaviour. To illus- trate, we define *ρ* for some operators. We suppose *X* = (*N, E, P* ) and abbreviate if (*e*) then *{p}* else *{q}* to if(*e, p, q*), and similarly for while(*e, p*):

|  |  |  |
| --- | --- | --- |
| + ((*u, bu*)*,* (*v, bv* )) | *'−→* Cases*{* | *bu* = *n* in N : +*n*(*v*)*, bu* = *u′* in *N* : (*u′* + *v*)*}* |
| +*n* ((*v, bv* )) | *'−→* Cases*{* | *bv* = *m* in N : (*n* + *m*)*, bv* = *v′* in *N* : +*n*(*v′*)*}* |
| if ((*e, be*)*,* (*p, bp*)*,* (*q, bq* )) | *'−→* Cases*{* | *be* = *false* in B : (*q*)*, be* = *true* in B : (*p*)*,* |
|  |  | *be* = *e′* in *E* : (if(*e′, p, q*))*}* |
| while ((*e, be*)*,* (*p, bp*)) | *'−→* if(*e,* | (*p* ; while(*e, p*))*,* skip) |

Theorem 5.1 of [[24](#_bookmark41)] shows how *ρ* induces operational models by structural recursion. with closed programs being the carrier of the model.

**Proposition 2.6** *Suppose ρX* : Σ(*X × BX*) *→ BT*Σ*X is a natural transformation. For every coalgebra ⟨Y, γ* : *Y → BY ⟩, there is a unique morphism, which we denote T*˜*γ, such that the following diagram commutes. Here, η and μ are the unit and multiplication of T, and ψ is the* Σ*-algebra structure of TY .*

*Y*  *ηY*  *T Y* ¸*, ψ*

Σ*TY*

*γ*

J *BηY*

*T*˜*γ*

J

Σ*⟨*id*,T*˜*γ⟩*

J

*BY*  *B TY* ¸*,* Σ(*TY × BTY* )

*BμY ◦ρTY*

To induce the desired operational model – a coalgebra structure for closed programs,

*T* 0 *→ BT* 0 – we take *Y* = 0 and *γ* =?*BY* :0 *→ BY* the initial map.

**Example 2.7** For Example [2.5](#_bookmark2), this operational model describes the transition behaviour of while programs without variable lookup or update. Coalgebraic bisimilarity is given by a map from *T*Σ0 into the final *B*-coalgebra, which is (N *×* N*,* N *×* B*,* N *×* 1) – it maps expressions of each type to the number *n ∈* N of steps-to-termination and the terminal value *v ∈* N*,* B*,* 1 for that expression. Two expressions *p, q* are then behaviourally equivalent iff they terminate in the same number of steps, and with the same terminal value.

* 1. *Effects as Syntax*

We now extend an abstract operational semantics with effectful commands, such as variable lookup/update for while programs. We extend the signature by adding constructors that describe effects such as variable lookup at every sort.

**Definition 2.8** *An* effect signature *is a single-sorted signature* Eff *representing effects of arity at most ω. The* effectful extension Sig *⊕* Eff *of a language* Sig *with effects* Eff *consists of the function symbols in* Σ*, together with a function symbol δs* : (*s*)0*≤j<α → s (i.e. all arguments of type s) for each δ ∈* Eff *of arity α and all*

*sorts s of* Sig*. If* Δ: *C → C is the signature functor induced by* Eff*, we write* Δ˜

*for*

Δ*n* : *Cn → Cn. Thus* Σ+ Δ˜

*is the signature functor induced by* Sig *⊕* Eff*.*

Key examples of effects include global state, interactive I/O, and non-determinism. We focus on global state [[15](#_bookmark34)] as required for while programs.

**Example 2.9** Given a finite set of variable-locations *L* with values in *V* , the effects for global state are ‘read’ rd : v *→ l* and ‘write’ wr : 1 *→ l×*v operations on variables, where *|L|* = *l < ω* and *|V |* = *v ≤ ω*. Thus the effect signature Eff has function symbols rd*x* of arity *v*, and wr*x,n* of arity 1, for all *x ∈ L*, *n ∈ V* .

We read rd*x*((*cn*)*n∈V* ) as the computation that reads the value *n* of *x* in the store and then executes the command *cn*. The command wr*x,n*(*c*) assigns *n* to *x* and proceeds with command *c*.

The free-algebra functor *T*Σ+Δ˜

(Remark [2.3](#_bookmark1)) describes the program terms of the

extended language, allowing effects to be freely incorporated into program syntax.

In particular, an extension of stateless while with effects for global state allows us to describe ‘standard’ while-programs in the following way.

**Example 2.10** Extending the signature Sig*sl* for *stateless while* with Eff given by global state introduces new syntax operators rd*x* and wr*x,n* of arity *ω* and 1 respectively (for all *x ∈ L* and *n ∈* N), i.e. Δ*X* = *L×Xω* + *L×* N *×X* where *L* is the

set of store locations. We can thus write numeric expressions like 3 + rd*x*(0*,* 1*,* 2*,.. .*) – corresponding to 3 + *x* – and we can express *x*:=3 as wr*x,*3(skip).

* 1. *Effectful Behaviour and the Final Coalgebra in Kl(M)*

We argue that effects are most naturally captured by moving from the underly- ing category *C*S to a Kleisli category Kl(*M* ) for a suitable monad *M* , where *MX* constructs an appropriate set of effect-trees with leaves in *X*. As described in [[8](#_bookmark26)], program execution can be understood as producing a syntactic effect-tree whose leaves are programs. If *M* constructs these effect-trees, then a suitable functor for effectful behaviour is *MB*, where *B* is as before. An effectful operational model is thus an *MB*-coalgebra with carrier *T*Σ+Δ˜ 0.

This structure is appropriate for describing the one-step evolution of programs as

they are executed. However, it does not directly yield the right notion of multi-step evaluation; iterating a coalgebra map *γ* : *X → MBX* in the underlying category

*γ*

*C* gives a map *X −→ MBX*

*MBγ*

*−→ MBMBX* that fails to accumulate the effects.

However, a distributive law *λ* : *BM → MB* allows us to accumulate effects correctly as follows:

*γ MBγ*

*MλBX*

2 2 *μB*2 *X* 2

*X −→ MBX*

*−→ MBMBX*

*−→ M B X*

*−→ MB X* (1)

The effects arising in the first two execution steps have been combined, giving a single effect tree whose leaves (in *B*2*X*) describe two effectless transition steps.

This result can be achieved naturally by moving from *C* into a Kleisli category Kl(*M* ); *MB*-coalgebras in *C* are interpreted as *B*-algebras in Kl(*M* ) for a behaviour functor *B* obtained by ‘lifting’ *B* from *C* into Kl(*M* ). The above chain of morphisms then corresponds to the iteration *Bγ ◦ γ*; we can think of Kleisli-morphisms as accumulating and propagating effects.

Recall the objects of the Kleisli category Kl(*M* ) are the same as the underly- ing category *C*, but with morphisms *fj* : *X → Y* in 1-1 correspondence with the (‘underlying’) morphisms *f* : *X → MY* in *C*. Composition *gj ◦ fj*, for *gj* : *Y → Z* and *fj* : *X → Y* , is given by the arrow *hj* : *X → Z* corresponding to the morphism

*f Mg μZ †*

*X → MY → MMZ → MZ* (its ‘overlying’ arrow). We write *g*

for the latter part,

*μZ ◦ Mg* : *MY → MZ*, and use notation *j* in the natural way – e.g. the above identity becomes *gj ◦ fj* = (*g† ◦ f* )*j*.

We write *J* for the canonical (left-adjoint) inclusion functor *C →* Kl(*M* ), identity- on-objects and sending *f* : *X → Y* to *Jf* = *ηY ◦ f* : *X → Y* [[3](#_bookmark24)]. One finds that *gj ◦ Jf* = (*g ◦ f* )*j* and *Jf ◦ hj* = (*Mf ◦ h*)*j*.

A ‘lifting’ *B* of the behaviour functor *B* into Kl(*M* ) satisfies *JB* = *BJ*. In particular, this implies that *B* is identity-on-objects – so a Kleisli-morphism *γj* : *X → BX* has underlying type *γ* : *X → MBX*, establishing the 1-1 correspondence between *MB*-coalgebras and *B*-coalgebras mentioned above.

Liftings can be described in terms of distributive laws [[21](#_bookmark39),[3](#_bookmark24)]:

**Lemma 2.11** *There is a 1-1 correspondence between liftings B of B into* Kl(*M* ) *and distributive laws λ* : *BM → MB. For B deﬁned by sums and copowers (i.e. isomorphic to BX* = *V* + *A × X for some V, A), these distributive laws exist.*

Given a distributive law *λ*, the corresponding *B* sends an arrow *fj* : *X → Y*

to the arrow *Bfj*

: *BX → BY* corresponding to *BX*

*Bf*

*→ BMY*

*λ*

*→ MBY* . Thus

*Y*

for *γj* : *X → BX*, the composition *Bγj ◦ γj* corresponds to the chain of arrows anticipated in ([1](#_bookmark4)):

(*Bγ*)*† ◦ γ* = (*λX ◦ Bγ*)*† ◦ γ* = *μBX ◦ MλX ◦ MBγ ◦ γ*

We later use the easy fact that if *f* is an *MB*-coalgebra morphism, then *Jf* is a

*B*-coalgebra morphism.

The terminal sequence for *B* in the Kleisli category suggests that the final *B*- coalgebra in Kl(*M* ), if it exists, is a natural candidate for program denotations. Standard results guaranteeing its existence are reviewed below [[21](#_bookmark39),[4](#_bookmark23),[7](#_bookmark27)].

**Definition 2.12** *A category C is* Cppo-enriched *if all hom-sets are complete partial*

*orders with least element ⊥ and composition is ω-continuous in both arguments* *(g ◦* .*n fn* = .*n*(*g ◦ fn*) *and similarly for the ﬁrst argument). Composition in C is* left-strict *if ⊥X,Y ◦ f* = *⊥X,Z for f* : *Y → Z. A functor B* : *C → D between* Cppo*-*

*enriched categories C, D is* locally continuous *if it preserves suprema of ω-chains,*

*i.e. B*(.*n fn*)= .*n B*(*fn*)*, and* locally monotone *if Bf ≤ Bg whenever f ≤ g.*

The following is shown in [[4](#_bookmark23), Theorem 3.3 and Proposition 3.9] and readily extends componentwise to the multi-sorted setting.

**Proposition 2.13** *Let M be a monad on C,* Kl(*M* ) *a* Cppo*-enriched Kleisli category with left-strict composition, and F a locally-continuous lifting of F into* Kl(*M* )*. If the initial F-algebra ⟨D, α⟩ exists in C, then the ﬁnal F-coalgebra in* Kl(*M* ) *is given by ⟨D, Jα—*1*⟩. If C* = Set*n, F need only be locally monotonic.*

**Proof.** Follows from [[4](#_bookmark23)] Propositions 3.2 and 3.9. Note that in Proposition 3.2 the initial *F* -algebra in Kl(*M* ) is given by *η ◦ α*, where *η* is the unit of *M* ; its inverse in Kl(*C*) is easily shown to be *η ◦ α—*1, or equivalently *Jα—*1. *2*

Particular effect monads *M* may satisfy these constraints, or be modified to do so. However, ideally one would like to generate such monads from a given effect signature. We must equip the Kleisli category with a Cppo-enriched structure with left-strict composition; obtaining appropriate *B* is usually not a problem. For a given effect signature Δ without equations, defining a Cppo-enriched structure is not problematic. In Set*n*, we may define:

**Definition 2.14** *For a countably-polynomial functor* Δ˜ *the cofree* Δ˜ + *⊥-coalgebra functor on Cn where ⊥ ∼*= 1*.*

*on Cn, we deﬁne F*Δ˜

*to be*

*F*Δ˜

is indeed a monad (Theorems 2 and 4 of [[11](#_bookmark30)]), and is given component-wise

by *F*Δ: (*F*Δ˜ *X*)*s* = *F*Δ(*Xs*). The objects *F*Δ˜ may be given the natural partial order

structure. In Cppo, this structure appears as the initial Δ˜ -algebra [[2](#_bookmark25)].

By contrast, left-strictness is not so straightforward. Intuitively, this requires that if we replace all leaves of an effect-tree with *⊥*, the resulting tree is identified with *⊥*. We anticipate a better solution in Cppo, perhaps by introducing divergence

as an effect ([[5](#_bookmark21)] Example 6); for now, we give a strategy in Set for adjusting *F*Δ˜ in

Set to ensure left-strictness.

* + 1. *Left-strictness in* Set

In Set, we may achieve left-strictness by restricting *F*Δ*X* as follows:

**Definition 2.15** *We write G*Δ *for the functor with G*Δ*X ⊆ F*Δ*X consisting of the effect-trees which do not have a non-trivial subtree with all leaves equal to ⊥. On*

*morphisms, G*Δ*f is the restriction of F*Δ*f. We write G*Δ˜

*nentwise in* Set*n.*

*for G*Δ *applied compo-*

It is straightforward to show that *G*Δ is a functor on Set. An ordering *±G* on *G*Δ*X* is inherited from that of *F*Δ*X*: for *t, tj ∈ G*Δ*X*, *t ±G tj* iff *tj* is obtained from *t* by replacing some *⊥* leaves with subtrees. *G*Δ is easily defined on arrows *f* : *X → Y* as the restriction of *F*Δ*f* : *F*Δ*X → F*Δ*Y* to *G*Δ*X*, which simply relabels each leaf *x ∈ X* with *f* (*x*). (The result is a tree in *G*Δ*Y* .)

*G*Δ has a monad structure similar to that of *F*Δ: the unit *ηX* maps *x ∈ X* to the singleton tree with leaf *x* (which lies in *G*Δ*X*); the multiplication *μX* : *G*2 *X → G*Δ*X* ‘plugs in’ the trees at each leaf and then prunes any resulting all-*⊥* subtrees. The monad axioms are straightforward to verify, and the definition of multiplication ensures that *⊥†* = *μ◦G*Δ*⊥* indeed maps every tree to *⊥*, so that Kl(*G*Δ) is left-strict.

Δ

We have that *G*Δ*X ⊆ F*Δ*X* is a sub-cppo of *F*Δ*X*, as the limit of an *ω*-chain in *G*Δ*X* also lies in *G*Δ*X*. This follows from the contrapositive: if the limit *tω* of an *ω*-chain (*tn*) in *F*Δ*X* has a non-trivial all-*⊥* subtree (*tω ∈/ G*Δ*X*) then some *tn* must also have non-trivial all-*⊥* subtree (*tn ∈/ G*Δ*X*).

Thus Kl(*G*Δ) has a Cppo-enriched structure, obtained pointwise from that given by *G*Δ. To apply proposition [2.13](#_bookmark5) for behaviour functor *BX* = *V* + *A × X*, it remains to supply a locally monotonic lifting *B*, or equivalently a distributive law *λX* : *BG*Δ*X → G*Δ*BX*. This may be defined as before: *λX* (inl(*v*)) = *ηX* (inl(*v*)), and *λX* (inr(*a, t*)) = *tj* where *tj* is obtained by replacing every non-*⊥* leaf *x* with inr(*a, x*). Local monotonicity follows from the fact that for an arrow *f* : *X → G*Δ*Y* , *Bf* = *λY ◦ Bf* takes as input either a value inl(*v*) or a pair (*a, x*) *∈ A × X*; it correspondingly returns the singleton tree with leaf inl(*v*), or the tree obtained from *f* (*x*) *∈ G*Δ*Y* by replacing every non-*⊥* leaf *y* with (*a, y*).

**Corollary 2.16** *There is a lifting of BX* = *V* + *A×X to* Kl(*G*Δ)*, and* Kl(*G*Δ) *has a ﬁnal B-coalgebra as given by Proposition* [*2.13*](#_bookmark5)*.*

*n*

Lastly, this result extends componentwise to *G*Δ˜ on Set .

# From Effectless to Effectful Abstract Operational Se- mantics

Given a suitable monad *M* , we consider the problem of obtaining a suitable operational model, an *MB*-coalgebra with carrier *T*Σ+Δ˜ 0, the set of closed program syntax terms mixed with effects. Theorem [2.6](#_bookmark3) allows us to do this via structural re- cursion; it requires an operational specification – an abstract operational semantics

– for the effectfully extended language with syntax functor Σ + Δ˜

and behaviour

*M* , i.e. a natural transformation *ρ*Gff : (Σ + Δ˜ )(*X × MBX*) *→ MBT*Σ+Δ˜ *X*.

This section describes a first attempt at systematically obtaining *ρ*Gff from an abstract operational semantics *ρ* for the *effectless* fragment of the language. Note that one does not have to use *ρ*; one could directly specify *ρ*Gff for the language in question, avoiding the need for dependency-functions below, but the operational rules become more intricate; we hope to elaborate on this in future work.

**Remark 3.1** *In addition to Proposition* [*2.13*](#_bookmark5)*, we assume M is given componentwise by a monad M*0*, and has a (componentwise) strength and a natural transformation*

*φX* : Δ˜ *MX → MX. In* Set *this holds for M* = *G*Δ˜ *X, using the* Δ˜ *-part of the inverse of the* Δ˜ + *⊥* + *X coalgebra structure of F*Δ˜ *X.*

To define *ρ*Gff from *ρ*, we must handle program syntax Σ and effect syntax Δ˜ . One readily obtains a natural transformation (later called *ψ*1) for the latter, but it

is less easy to define suitable *ψ*2 : Σ(*X × MBX*) *−→ MBT*Σ*X* to handle existing

*X*

syntax. We soon show how an assumption that *ρ* is ‘dependency-supported’ allows us to define *ψ*2 correctly.

Let us attempt to describe the effectful behaviour of if (*e*) then *{p}* else *{q}* when *e, p, q* exhibit effectful behaviour. For instance, *e* might have behaviour rd*x*(*true, ej, false,.. .*) – depending on whether *x* is 0*,* 1*,* 2, *e* terminates as *true*, evolves to *ej*, terminates as *false*, and so on. We would expect the behaviour of if (*e*) then *{p}* else *{q}* to depend similarly on the value of *x* – it would respec- tively evolve to *p*, if (*ej*) then *{p}* else *{q}*, *q*, and so on. Note that we never consider the behaviour of *p* or *q*, effectful or otherwise; effects present in their be- haviour should only be considered once the condition *e* is evaluated.

Now we consider how to formalise this in Set3. Assume a tuple *X* = (*N, E, P* ) with *e ∈ E* and *p, q ∈ P* . The possible behaviours of *e* and *p, q* are the second and third components of *BX* = (*N* + N*,E* + B*,P* + 1). Recall that effectless operational rules are commonly represented by a natural transformation Σ(*X ×BX*) *→ BT*Σ*X*. For if statements, this gives a function *ρ* : (*E × BE*) *×* (*P × BP* )2 *→ BT*Σ*X* whose arguments are respectively a pair (*e, b*) of a boolean variable *e* and an (effectless) behaviour for it, and similarly for programs *p, q ∈ P* .

Suppose we now have effectful behaviours for *e, p, q* – elements of *MBE* and *MBP* , rather than *BE* and *BP* . We aim to define a function analogous to *ρ*, but incorporating effectful behaviour: (*E × MBE*) *×* (*P × MBP* )2 *−→ MBT*Σ*X*. The informal argument involves ‘pulling out’ the effects present in the behaviour of *e*,

which corresponds to two applications of monad strength [[12](#_bookmark31)]:

(*E × MBE*) *×* (*P × MBP* )2 *−→ M* ((*E × BE*) *×* (*P × MBP* )2)

However, now we are stuck. We need something of form (*E×BE*)*×*(*P ×BP* )2 so that we can apply *ρ*; however, repeatedly using the monad strength would propagate the behaviours of *p* and *q*, which is incorrect as argued above. At this point, the informal argument used the fact that the behaviours of *p* and *q* are irrelevant. Thus one might simply *discard* the effectful behaviours *MBP* and replace them with arbitrary values in *BP* :

*M* ((*E × BE*) *×* (*P × MBP* )2) *→ M* ((*E × BE*) *×* (*P × BP* )2)

Now applying *Mρ* to if (*e*) then *{p}* else *{q}* gives the correct effectful behaviour. However, it is a crude way to exploit the information that ‘the behaviour of if (*e*) then *{p}* else *{q}* does not depend on that of *p* or *q*’. We express this de- pendency more systematically, assuming that *ρ* has an appropriate factorisation – deducible from the operational rules – discarding irrelevant arguments. Without loss of generality, we assume the first *n* arguments must be kept.

This gives a canonical means of defining *ψ*2 if the behaviours of every program term depends on that of *at most one subterm* – i.e. operational rules have at most one premise. However, two issues arise in the presence of multiple premises. Consider a ‘synchronous execution’ operator:

*p → pj,q → qj*

*j √ √*

*j √ √*

*p × q → pj × qj*

*p → p ,q → p →,q → q*

*p × q → pj p × q → qj*

*p →,q →*

*√*

*p × q →*

If *p* and *q* both introduce variable updates (e.g. to the same variable), obviously there can be no canonical choice for the effectful behaviour of *p × q*; one must make a choice whether to apply the variable update of *p* first, or of *q*. In the presence of equations, if the effects are commutative (e.g. non-determinism), this choice makes no difference and the extension is canonical; otherwise one is forced to put an ordering on the sub-terms specifying the order of propagation. Without loss of generality, we will suppose this ordering is simply left-to-right, and define a natural transformation comb to propagate effects in this way using monadic strength.

The second issue is that a term may not always depend on the same number of sub-terms. In while, + and *∗* are an example of this; without auxiliary operators

+*n* and *∗n*, a standard operational semantics would contain the rules

*u → uj*

*√ j √ √*

*u* + *v → uj* + *v*

*u → n,v → v*

*u* + *v → u* + *v*

*u → n,v → m*

*√*

*u* + *v → n* + *m*

In applying the first rule, we must not propagate any effects of *v*, unlike the other cases. This would require a more fine-grained approach than the one above; however, introducing auxiliaries can ensure the behaviour of every syntax constructor once

again always depends on the same sub-terms – here, we achieve this by introducing

+*n* and *∗n* as above.

For the rest of this section, we will assume that the behaviour of every program term of arity *α* depends on some number *n ≤ α* of subterms, whose effects are to be propagated from left-to-right. We make this formal:

**Definition 3.2** *For a signature* Sig*, a dependency function dep* : Sig *→* N *is a function satisfying* 0 *≤ dep*(*f* ) *<* ar(*f* ) *for every symbol f ∈* Sig*.*

**Definition 3.3** *Given a signature* Sig *with corresponding syntax functor* Σ *and a dependency function dep, for each Y in C*S*, the restriction* res*Y,X is a natural transformation in X, whose s-component is given by:*

res*Y,X* : (Σ*X*)*s* = `*f* :(*si*)*→s ∈*Sig Q0*≤i<*ar(*f* )(*X × Y* )*si* *s*

*−−−−*=*−−→*

`*f* :(*si*)*→s ∈*Sig

Q0*≤i<*ar(*f* ) *Ysi ×* Q0*≤i<*ar(*f* ) *Xsi*

`(Q *πi×*id)

*−−−−−−−→*

`*f* :(*si*)*→s ∈*Sig

Q0*≤i<dep*(*f* ) *Ysi ×* Q0*≤i<*ar(*f* )

*s*

*Xsi*

*s*

*When dep*(*f* )= 0*, we take π*0 *as the terminal map and Ys*0 *as the initial object* 1*.*

**Definition 3.4** *Given a dependency function dep, an abstract operational seman- tics ρX* : Σ(*X ×BX*) *→ BT*Σ*X is dep-*supported *if there is a natural transformation ρj such that ρX is given by the composition*

res*BX,X*

Σ*X −−−−−→*

⎛

⎝*f* :(*si*)*→s*

⎛

⎝0*≤i<dep*(*f* )

(*BX*)*si ×*

0*≤i<*ar(*f* )

⎞⎞

*Xsi* ⎠⎠

*s*

*ρ′X*

*−−−→ BT*Σ*X*

Trivially, every language is *dep*-supported if we set *dep*(*f* ) = ar(*f* ) for all *f* ; but this will not necessarily give the desired behaviour, as illustrated above.

**Example 3.5** For while programs, only the behaviour of the first argument matters – except for while statements, and constants, which have no arguments. Thus we define a dependency function by *dep*(*f* ) = 1 for all *f* except the constants (*n ∈* N*,b ∈* B, and skip), and while (*e*) do *{p}* – which have *dep*(*f* ) = 0. It is easily shown that the abstract operational semantics for While is then *dep*-supported.

To handle multiple premises, one needs a method comb of combining effect-trees. One may use the monad strength st*X,Y* , of type *X ×* (*MY* ) *−→ M* (*X ×Y* ). Here, it takes an *x* and an effect-tree with leaves in *Y* , and pairs each leaf *y* with *x*, giving an effect-tree with leaves in *X × Y* .

**Definition 3.6** *For all n < ω, given objects Y*0*,..., Yn—*1 *we inductively deﬁne an arrow* comb*Y*0*,...,Yn−*1 : Q0*≤i≤n—*1 (*MYi*) *−→ M* Q0*≤i≤n—*1 *Yi by* comb*Y*0 = id*MY*0 *, and* comb*Y*0*,...,Yn*+1 *in terms of* comb*Y*0*,...,Yn as follows:*

*MYn*

*×* Q0*≤i<n*

*MYi*

id*×*comb*Y*0*,...,Yn*

*−−−−−−−−−−→*

*MYn*

*× M* Q

0*≤i<n Yi*

st

*−−−−−→ M* (*MYn ×*

Q0*≤i<n Yi*)

*∼*=

*−−−−−−−−→ M* (

Q0*≤i<n Yi × MYn*)

*M* st

*−−−−→ M*

2 Q0*≤i<n*+1 *Yi*)

*μ*

*−−−−−−−−→ M*

Q0*≤i<n*+1 *Yi*

* 1. *Abstract Operational Semantics for Effectful Behaviour*

Now we define the *ρ*Gff from the beginning of this section, assuming *ρ* is *dep*- supported. We express it in terms of the following components:

1 : Δ˜ (*X × MBX*) *→ MBT*Σ*X ψ*2

*ψ*

*X*

*X*

: Σ(*X × MBX*) *→ MBT*Σ*X*

We may then define *ρ*Gff = *MB* inc *◦* [*ψ*1 *, ψ*2 ] where inc ‘includes’ terms given by

*X X*

*T*Σ*X* into the ‘bigger language’ *T*Σ+Δ˜ *X*. We can do this by giving *T*Σ+Δ˜ *X* the

evident Σ-algebra structure; then inc is the initial Σ-algebra map *T*Σ*X → T*Σ+Δ˜ *X*.

**Definition 3.7** *We deﬁne ψ*1

*X*

*by MBηX ◦ φBX ◦* Δ˜ *π*2*, where π*2 *is the projection*

*X × Y → Y , φBX is as given by Remark* [*3.1*](#_bookmark9)*, and η is the unit of T*Σ*.*

**Definition 3.8** *ψ*2

*X*

*is deﬁned below. In the second map, ζ isomorphically replaces*

(*MBX*)*si with M*0((*BX*)*si* ) *(Remark* [*3.1*](#_bookmark9)*) if dep*(*f* ) *>* 0*, otherwise it is the unit η*1 :1 *→ M* 1 *(recall* (*−*)*s*0 =1 *by Deﬁnition* [*3.3*](#_bookmark10)*). The fourth uses strength of T*Δ*. The ﬁfth map ‘swaps the M and the coproduct ’ as follows. For each f we*

`*f∈*Sig

*write* inj*f for the injection into the f-component of the coproduct Y →* `*g∈*Sig *Yg;*

*we apply M to* inj*f and take the coproduct* [*M* inj]*f over all f ∈* Sig*.*

res(*MBX*)*,X*

*ψ*2 : Σ(*X × MBX*)

` Q Q

*X*

*−−−−−−−→*

*s*

(` (*ζ×*id))

*f*

*s*

*−−−−−−−−→*

*f* :(*si*)*→s ∈*Sig

` Q

*f* :(*si*)*→s*

0*≤i<depf* (*MBX*)*si ×*

Q

0*≤i<depf M* (*BX*)*si ×*

0*≤i<*ar(*f* ) *Xsi*

0*≤i<*ar(*f* ) *Xsi*

*s*

(` (comb*×*id))*s*  ` Q

Q

*−−−−−−−−−−→*

*f*

*f* :(*si*)*→s M*

0*≤i<depf* (*BX*)*si ×*

0*≤i<*ar(*f* ) *Xsi*

(`*f* st)*s*  ` Q

Q

*−−−−−−−−−→*

*s*

([*M* inj*f* ]*f∈*Sig)*s*

*f* :(*si*)*→s*

`

*M* 0*≤i<depf* (*BX*)*sdep*(*f* ) *×*

Q

0*≤i<*ar(*f* ) *Xsi*

*s*

Q

*−−−−−−−−−→*

*Mρ′*

*M f* :(*si*)*→s*

0*≤i<depf* (*BX*)*sdep*(*f* ) *×*

*M* inc

0*≤i<*ar(*f* ) *Xsi*

*s*

*−−−−−−→ MBT*Σ*X*

*−−−−−−−→ MBT*Σ+Δ˜ *X*

**Lemma 3.9** *ψ*1 *and ψ*2 *are natural transformations.*

**Proof.** All the components of the definition – namely *φBX*, *ηX* , res*Y,X* , *ρj*

*X*

, inc*X* ,

comb*Xs*1 *,...,Xsn* , and st*BX,*` *Xsi* and inj*f* for any *f ∈* Sig – are natural in *X*. *2*

Thus we obtain *ρ*Gff from Definition [3.7](#_bookmark11). It is natural because it is defined in terms of natural transformations *ψ*1, *ψ*2, and inc*X* : *T*Σ*X → T*Σ+Δ˜ *X*.

**Example 3.10** We describe the action of *ρ*Gff for stateless while programs extended with global state. It takes as input terms of form *σ*((*x*1*, t*1)*,* (*x*2*, t*2)*,.* ) where *σ* is

either a syntax constructor for stateless while (Example [2.2](#_bookmark0)), or effect syntax for global state, viz. rd*x* or wr*x,n*. Each *xi* is a variable (of appropriate sort), and *ti* a corresponding effectful behaviour – an effect-tree (of reads rd*x* and updates wr*x,n*) whose leaves are variables *yi* or terminal values *vi*. The output is an element of *MBT*Σ+Δ˜ *X* – an effect-tree with each leaf either an expression *t*, or a terminal

value *v* of the same sort as *σ*.

*ρ*Gff acts straightforwardly (via *ψ*1) on effect terms such as wr*x,n*((*p, tp*)), where *p* is an expression and *tp* an effectful behaviour. This term is simply mapped to wr*x,n*(*p*); similarly, rd*x*((*x*1*, t*1)*,* (*x*2*, t*2)*,.. .*) is mapped to rd*x*(*x*1*, x*2*,.. .*).

The effect on syntax terms like +((*m, tm*)*,* (*n, tn*)) follows from the definition of

*ψ*2. The restriction res discards the behaviour *tn* as it is irrelevant; the strength

comb ‘pulls’ the tree *tm* out, giving an effect tree *tj*

*m*

of the same shape as *tm*,

but whose leaves are of form +((*m, bm*)*, n*) where *bm* is a leaf of *tm*, an effectless behaviour. Finally, the standard operational semantics (Example [2.5](#_bookmark2)) is applied to

each leaf of *tj*

*m*

. As an illustration, we might have

+((*n,* rd*x*(*nj,* 5+ 2*,* 3*,.. .*))*,* (*m,* 5)) *'→* rd*x*(*nj* + *m,* +5(2) + *m,* +3(*m*)*,.. .*)

Given *ρ*Gff , Theorem [2.6](#_bookmark3) yields an operational model by structural recursion:

**Corollary 3.11** *Given an effect signature* Eff*, every dep-supported abstract oper- ational semantics ρ* : Σ(*X × BX*) *→ BT*Σ*X gives rise to an effectful operational model, an MB-coalgebra with carrier carrier T*Σ+Δ˜ 0*.*

**Example 3.12** We illustrate the operational behaviour of the program

*p* = if (*x* = 1) then *{y*:=2*}* else *{*skip*}*

where *x* is shorthand for rd*x*(0*,* 1*,* 2*,.. .*), and *y*:=2 shorthand for wr*y,*2(skip):

*—→* rd*x*(if (0 = 1) then *{y*:=2*}* else *{*skip*},* if (1 = 1) then *{y*:=2*}* else *{*skip*},*

if (2 = 1) then *{y*:=2*}* else *{*skip*},.. .*))

*—→* rd*x*(if (=0 (1)) then *{y*:=2*}* else *{*skip*},* if (=1 (1)) then *{y*:=2*}* else *{*skip*},*

if (=2 (1)) then *{y*:=2*}* else *{*skip*},.. .*))

*—→* rd*x*(*∗,* wr*y,*2(skip)*, ∗, ∗,.. .*) *—→* rd*x*(*∗,* wr*y,*2(*∗*)*, ∗, ∗,.. .*)

If *B* has a lifting to Kl(*M* ), the operational model can be seen as a *B*-coalgebra living in the Kleisli category. If in addition the final *B*-coalgebra exists, we may thus define an operational equivalence for programs in terms of the unique Kleisli- coalgebra map **[***−* **]** into the final *B*-coalgebra. For *C* = Set and *BX* = *V* + *A × X*, Proposition [2.16](#_bookmark7) implies both these conditions.

**Definition 3.13** For *C* = Set, two programs *p, q ∈ T*Σ+Δ˜ 0 are *operationally equiv- alent*, *p ∼*=*op q*, if [[*p*]]= [[*q*]] where **[***−* **]** is the unique *B*-coalgebra morphism into the final *M* -coalgebra.

**Example 3.14** Theorem [2.13](#_bookmark5) implies the carrier of the final *B*-coalgebra is the initial *B*-algebra, *D*. The underlying arrow therefore maps into *MD*. For While expressions, *D* is (N *×* N*,* N *×* B*,* N *×* 1); hence (at sort *s*) the elements of (*MD*)*s*

are effect-trees with leaves in (*n, v*) *∈ Ds*. One may show (by uniqueness) that the final *B*-coalgebra arrow ! maps each term to the overall effect-tree it produces during execution, with leaves (*n, v*) *∈ D* describing executions – *n* is the number of steps before termination, and *v* the final value of that execution. For example, the program *p* of Example [3.12](#_bookmark13) is mapped to rd*x*((3*, ∗*)*,* wr*y,*2(4*, ∗*)*,* (3*, ∗*)*,* (3*, ∗*)*,.. .*).

# Towards Adequacy for Effectful Operational Seman- tics

In the previous section, we argued that an operational model for an effectful lan- guage finds its natural place in the Kleisli-category Kl(*M* ). This section is devoted to the problem of obtaining an adequate effectful denotational semantics.

To discuss adequacy, it is convenient to change notation. From now on, we relabel the syntax functor of the extended language, Σ + Δ˜ , simply to Σ (i.e. Σ now incorporates effectful commands directly). We now write *T* for the free Σ- algebra functor. In the same vein, we forget about the effectless abstract OS *ρ* :

Σ(*X × BX*) *→ BTX* and relabel the effectful abstract OS *ρ*Gff to *ρ*.

The effectful denotational model arises by mapping the operational model into the final coalgebra *D* in the Kleisli category Kl(*M* ). In the underlying category this gives a map *T* 0 *→ MD*, so we aim to take *MD* as our denotational model.

* 1. *A Couple of Counterexamples*

There is a key difference between the effectful and effectless settings: for some effectful operational specifications (abstract OS), the operational semantics is *not* compositional, and adequacy fails.

**Example 4.1** Consider an interleaving operator *|* or a ‘partial’ (one-step) evaluator

:*>*, defined by the following rules (at any type):

*x → xj*

*x | y → y | xj*

*√*

*x → v x | y → y*

*x → xj*

*x* :*> y → y*

*√*

*x → v*

*x* :*> y → y*

These rules are single-premise, so they may be effectfully extended as in the previous section. Operationally, the effectful extension *x | y* branches according to the effects given by the *ﬁrst step* of behaviour of *x*, then by that of *y*; then more effects are introduced by the successors of *x*, then by the successors of *y*, and so on. *x* :*> y* exhibits effects from the *ﬁrst* step of *x*’s execution only.

Now consider the following two programs, *p*1 = wr*y,*1(skip; skip) and *p*2 = skip; wr*y,*1(skip). They are identified by the map into the final Kleisli coalgebra (as they both assign *y* = 1 and terminate in 3 steps); thus are operationally equivalent.

Yet if we put them in the contexts [*−*] *| q* or [*−*] :*> q*, where

*q* = rd*y*(wr*z,*0(skip)*,* wr*z,*42(skip)*,* wr*z,*42(skip)*,.. .*)

the results are not operationally equivalent; if *y* is initially 0, then *p*1 *| q* will assign *z* = 42, and *p*2 *| q* will assign *z* = 0. *p*1 :*> q* and *p*2 :*> q* are similarly different. Thus, the operational semantics is not compositional with respect to *|* or :*>*.

Compositionality fails because our denotational semantics intentionally discards information about precisely *when* effects occur during program execution. If the operational semantics of a syntax operator like *|* depends on this information, our denotational semantics clearly cannot be adequate.

Thus we must restrict *ρ* to forbid too-fine-grained operational interaction be- tween subterms. One option is a form of *evaluation-in-context*, where a subterm *xi* is evaluated until all its execution branches have terminated, before evaluating any other terms on those branches. We formalise this notion below.

* 1. *Extending The Original Semantics*

We review the adequacy proof of Turi and Plotkin in our effectful setting, in terms of Σ-algebras and *MB*-coalgebras; our denotational and operational semantics arise as a quotient of theirs.

We write *⟨D, s⟩* for the final *MB*-coalgebra – the denotational model in Turi and Plotkin’s original setting. We say *D* is a layered semantics, in that it describes all possible layerings of effects *M* interleaved with effectless transitions *B*.

Recall that *⟨D, α⟩* is the initial *B*-algebra. As before, we assume Proposition

[2.13](#_bookmark5) holds for *F* = *B*, so that *⟨D, Jα—*1 : *D → MBD⟩* is the final *B*-coalgebra. We equip the new denotational model *MD* with an *MB*-coalgebra structure, *s* = *MBηM ◦ Jα—*1. Like the final *MB*-coalgebra, this formally describes sequences of effect-layers *M* and effectless transitions *B*, except that every effect layer after the first is trivial (non-branching), as they arise from the unit *ηM* . In a sense, the effects are ‘collected’ into the first step.

By Proposition [2.6](#_bookmark3), *ρ* induces *MB*-coalgebra structures *⟨TD, T*˜*s⟩* and

*⟨TMD, T*˜*s⟩* for syntax terms over the ‘layered’ and ‘collected’ denotational models

respectively. By taking the syntax functor to be Σ, the behaviour functor to be

*MB*, and the abstract OS to be *ρ*, we obtain the following diagram (cf. [[24](#_bookmark41)], p.84).

Σ*T* 0 Σ*den*  Σ*D*

*ψT* 0

*β*

J *den* J

*T* 0

*T*˜?

*op*  *D*

*s*

J *MBop*

*MBT* 0

J

*M BD*

where we write *ψT* 0 for the Σ-algebra structure of free-syntax ters *T* 0. Proposition

[2.6](#_bookmark3) induces (via *ρ*) the coalgebra structure *T*˜? on closed terms *T* 0, where ? : 0 *→*

*MBT* 0 . The Σ-algebra structure *β* on the denotational model *MD* is obtained from the final *MB*-coalgebra morphism *βTD* from *⟨TD, T*˜*s⟩* into *D*, as follows:

Σ*ηT*

*ψD βTD*

*β* : Σ*D −→* Σ*TD −→ TD −→ D*.

The initiality of *T* 0 induces the Σ-algebra morphism *den* into *⟨D, β⟩*, giving a (necessarily) compositional map into the denotational model. *op* is the *MB*- coalgebra morphism from *T* 0 into *D* induced by finality of the latter, giving an operational characterisation of program behaviour.

Turi and Plotkin’s work implies that the operational map *op* is a Σ-algebra morphism, so that the initiality of *T* 0 implies that *den* = *op* (as there is only one such map), so that the operational and denotational maps coincide. In particular, it implies compositionality of the operational equivalence *∼*=*op* induced by *op*, as defined in [3.13](#_bookmark14) – i.e. *∼*=*op* is a congruence with respect to syntax given by Σ. It also implies that the denotational semantics is adequate with respect to *∼*=*op*.

* 1. *An Adaptation to the Kleisli Setting*

Our strategy is to map from the layered denotational model *D* into *MD*. As the former is an *MB*-coalgebra – and thus a *B*-coalgebra – we obtain a unique *B*- coalgebra morphism *cj* : *D → D* in Kl(*M* ) which effectively collects all the effects from every step of the behaviours in *D* into the first step, giving a single layer of effects and allowing the former diagram to be extended:

Σ*T* 0

Σ*op* Σ*D*

Σ*c* Σ*M D*

(2)

*ψ*0 *β* (*∗*) *β*

J *op*

J *c* J

*T* 0 *D*  *M D*

*T*˜?

J *MBop*

*s*

J(*Bc*)*†*

*Mα−*1

J

*MBT* 0

*D*  *M BD*

The Σ-algebra structure *β* is induced in a similar manner to *β*, via the final *B*-coalgebra morphism *βTMD* from *TMD* into *D*: *β* = *βTMD ◦ ψMD ◦* Σ*ηT* . This gives an algebraic structure to the denotational model, and thus by initiality of *T* 0, induces a denotational map – a Σ-algebra morphism – from *T* 0 into *MD*.

The operational map in our semantics – given by the final *B*-coalgebra morphism from the operational model *T* 0 into *D* (in Kl(*M* )) – factors through the original one, *op*, as follows. The bottom-right square commutes by definition of *cj* as a *B*- coalgebra morphism. By definition *op* is an *MB*-coalgebra morphism, hence *Jop* is also a *B*-coalgebra morphism; thus the composition *cj ◦ Jop*, equal to *c ◦ op* in the underlying category, is a *B*-coalgebra morphism. By finality of *D* as a *B*-coalgebra, *c ◦ op* is necessarily equal to the final *B*-coalgebra morphism from *T* 0 into *MD*.

Now we aim to show the top-right square (\*) commutes – i.e. that *c* is a Σ- algebra morphism. This would imply the denotational maps and operational maps coincide, giving adequacy and compositionality as for the original semantics. This is because *op* is known to be a Σ-algebra morphism, so the operational map *c ◦ op* would also be a Σ-algebra morphism, so must coincide with the denotational map.

The Σ’s in condition (\*) may be replaced with *T* ’s as follows, giving the square (+) below. (The top line is *β*, the bottom *β*.)

Σ*D* Σ*ηT* Σ*T D*  *ψD*  *T D*  *βTD*  *D*

Σ*c* Σ*Tc*

*Tc* (+) *c*

JΣ*ηT* J *ψ*

J ** J

Σ*MD*  Σ*T MD*  *MD*

*βTMD* *M D*

*TMD*

The first square is the image under Σ of the naturality of *η*. The second com- mutes because *ψX* , the Σ-algebra structure of *TX* – the free Σ-algebra over *X* – is a natural transformation. This is an easy consequence of the definition of *T* by adjunction, *T* = *UF* .

One may interpret the condition (+) as an abstract constraint on the abstract OS *ρ* as follows. *TD* describes terms over the ‘old’ denotational model, whose arguments may introduce effects at each transition step. The upper path assigns an overall effect-tree to these terms, based on their behaviour given by *ρ*. The lower path (via *Tc*) collects the effects of each argument into its *ﬁrst* execution step, giving an element of *TMD*, and then similarly assigns an overall effect-tree. Thus, (+) implies that the stage at which effects occur (in the arguments of terms *TD*) is irrelevant, as they might as well all be in the first step.

* 1. *A Condition on Cones*

One may characterise the condition (+) somewhat more concretely, in terms of cones in the Kleisli category. For simplicity we assume the initial sequence of *B* in *C* converges after *ω* steps (as it does for *BX* = *V* + *A×X*). Note that left-strictness implies *M* 0 = 1, so 0 is the final object in Kl(*M* ) ([[3](#_bookmark24), Lemma 2.3.5]).

**Definition 4.2** The *cone generated by a B-coalgebra ⟨X, γ⟩* (over the final sequence

*n*

up to *ω*) consists of the arrows (*γn* : *X → B* 0)*n<ω* obtainable by composition in

the following diagram:

*γ*  *Bγ* 2

2

*B γ*

*X*

!*X*

J!

*BX*

*B*!*X*

J *B*!

*B X*

2

J*B X*

2!

2 *B* !

*···*

0 ¸*, B*0 *B*0 ¸*, B*0 *B*

0 ¸*, B*0 *···*

The limit-colimit coincidence described in [[4](#_bookmark23)] implies that (*sn* : *D → Bn*0)*n<ω* is a limiting cone [[3](#_bookmark24)]. It is straightforward to show that the *B*-coalgebra morphism from any *⟨X, γ⟩* into *D* must coincide with the mediating morphism between their generated cones.

Revisiting the condition (+), the path *c ◦ βTD* is a composition of *B*-coalgebra morphisms, so it is also one, and must coincide with the mediating morphism from ((*T*˜*s*)*n*) to (*sn*). In the other path, *βTMD* coincides with the mediating morphism

from ((*T*˜*s*)*n*) to (*sn*), so precomposing with *Tc* gives another cone over the final

sequence ((*Tc ◦ T*˜*s*)*n*), with mediating morphism *βTMD ◦ Tc*.

Our strategy is to show this cone is the same as the cone generated by *⟨TD, T*˜*s⟩*

– i.e., that (*T*˜*s*)*n* = (*Tc ◦ T*˜*s*)*n*. This would imply the mediating morphisms are

the same, giving equality of the upper and lower paths. The situation is depicted

*n*

below; we must show the two paths *TD → B* 0 coincide for all *n*.

! *T*˜*s * *BT*˜*s *2 2 ˜

*T DTD*

*Tc*

*B TD*

*B*!*TD*

*B TD*  *B T s*  *· · ·*

2

*B* !*TD*

!

J

*TMD*

*TMD*

*T*˜*s*  *B TMD*

*B*!

*BT*˜*s*  *B* 2

2

2

*TMD B T s*

˜

*· · ·*

J *s*

*TMD* J, *7*

*B* !*TMD* J *s*

2

0 *B*0 *B* 0 *···*

We will focus on the case where *ρ* is in evaluation-in-context format, a perspective which is often helpful for specifying effectful operational semantics – (cf. [[8](#_bookmark26),[14](#_bookmark33)]):

**Definition 4.3** An abstract OS *ρ* is *in evaluation-in-context format* if it arises as an extension of effectless operational rules corresponding to the following templates:

*x → xj √ √*

1 1

*σ*(*x*1*,..., xn*) *→ σ*(*xj ,..., xn*)

1

*x*1 *→ v*

*σ*(*x*1*,..., xn*) *→ t*

*x*1 *→ v*

or

*σ*(*x*1*,..., xn*) *→ u*

or

*σ*(*xj ,..., xn*) *→ t*

1

*σ*(*xj ,..., xn*) *→ v*

1

where *t* is an arbitrary term over *v* and the *xi*.

Thus a term’s behaviour depends on at most one subterm – without loss of generality, the first. It is executed in its place until termination, at which point the term evolves to another term depending on the final value.

**Theorem 4.4** *(C* = Set*) For BX* = *V* + *X, MX* = *G*Δ˜ *X, and ρ in evaluation- in-context format, we have* (*T*˜*s*)*n* = (*Tc ◦ T*˜*s*)*n. Thus condition (+) holds, and the*

*denotational and operational semantics induced by the initial and ﬁnal horizontal (co)algebra morphisms in (*[*2*](#_bookmark17)*) coincide.*

*n***Proof.** First, note that the horizontal arrows *TX → B TX* for *X* = *D, MD* have (underlying) codomain *MBnTX*, i.e. effect-trees over *n*-step behaviour traces; their leaves may be terminal traces – ending with a value in *V* – or not, ending with a

term in *TX*. When the horizontal arrows are composed with vertical *BnX → Bn*0,

the non-terminal leaves are replaced with *⊥*. (This may be proven by induction

on *n*, using the fact that the distributive law for *G*Δ˜

inr(*M* 0) *∈ BM* 0 into *⊥*.)

and *B* maps non-terminal

Thus, for any *n*, to show the two paths *TD → B*

*n*0 agree, it is enough to show

*n*

that when applied to any term *t*(*d*1*,.. .*) *∈ TD*, the horizontal paths to *B*

*n *

*TD* and

*B TMD* produce effect-trees which share the same terminal leaves (and agree in

shape before those leaves). This is because all non-terminal leaves will be mapped to *⊥* by the ensuing vertical arrows (and any resulting all-*⊥* subtrees removed by left-strictness as given by the definition of *G*Δ˜ ).

We prove this by induction on *n*. The *n* = 0 case is immediate as both vertical arrows map into the final object 0 in Kl(*M* ); now assume the inductive hypothesis for *n−* 1. The action of both horizontal paths consist of iterating *B*-coalgebra maps induced by the derivation rules of *ρ*, applied to syntax terms over *D* or *MD*. By assumption, *ρ* is obtained from effectless rules, which are single-premise and the premise must be the first argument.

When applying *ρ* to a syntax term *σ*(*x*1*,.. .*), there are several cases to consider. If *σ* is an effectless command, and the behaviour of *x*1 in *MBX* is relevant (*dep*(*σ*)= 1), the effectless rules are applied to each leaf of *x*1’s behaviour (in *BX*); the rule conclusions become the leaves of the behaviour of *σ*(*x*1*,.. .*). As the rule conclusions are effectless, so are any further deductions.

If there is no dependency on *x*1 (*dep*(*σ*) = 0), the behaviour of *σ*(*x*1*,.. .*) is given directly by the rule conclusion. If *σ* is an effect, a layer of branching will be introduced, and *σ*(*x*1*,.. .*) will evolve to some *xi* on each branch.

As a result, we may class any effectless derivation – as occurs at the leaves of the behaviour of a general term *t*(*x*1*,.. .*) – into the following four types. (The conventions for *j* will become clearer later.)

*x*1 *→ xj*

*σ*(*x ,.. .*) *→ t* (*y ,.. .*)

1 1 0 1

. .

.

*t*(*x*1*,.. .*) *→ t*(*xj ,.. .*)

.

*t*(*x ,.. .*) *→ tj*(*yj ,.. .*) or *u*

1 1 1

*x*1 *→ v*

.

*t*(*x*1*,.. .*) *→ u*

*x*1 *→ v*

.

*t*(*x*1*,.. .*) *→ tj*(*yj ,.. .*)

1

By assumption on *ρ*, the top-left is the only form of derivation possible from a premise *x*1 *→ xj* ; all other arguments of *t* are fixed throughout. If the premise is *x*1 *→ v* for some *v*, the rule conclusions may similarly indicate termination (bottom- left) or give rise to a new term with a maybe-new first argument, *tj*(*yj ,.. .*) (bottom- right). Similarly, effect syntax and premiseless terms give rise to terms *tj*(*yj ,.. .*),

1

1

1

but the former extends the effect-tree.

Such deductions define the coalgebra structure-maps of *TD* and *TMD*; we con- sider how these maps are iterated in terms of these deductions. Take a term *t*(*d*1*,.. .*)

*n*

in *TD*, and consider the action of the horizontal map *TD → B TD*. We may rep-

resent *d*1 schematically as follows:

*d*1

*j*

1

*m*1

*d*

*vj*

*jj*

1

*m*2

*d*

*vjj*

*jjj*

1

*m*3

*d*

*vjjj*

*···*

Here, *m*1 represents the effectful behaviour of *d*1 *∈ D*. Its leaves, elements of *BD*,

are either elements of *D* – collectively represented by the label *dj*

1

– or terminal

values represented by *v*. Likewise, *m*2 represents the effectful behaviour of each leaf

*dj* , and so on. (Each *dj* has its own effect-tree *m*2, but to keep diagrams simple, we

1 1

do not attempt to represent this information.)

We first focus on the case where the behaviour of the term *t*(*d*1*,.. .*) depends on its first argument. Its effectful behaviour resembles that of *d*1 – ‘the same branches’ – but with each leaf *l* derived via the operational rules, with premises given by the corresponding leaf *r* of *d*1. Depending on whether they are terminal values *v* or other elements of *D*, *l* will take one of the forms listed above (with *x, y* relabelled *d, e*). Thus we may partially represent the behaviour of *t*(*d*1*,.. .*) in terms of that of *d*1 as follows, omitting continuations which depend on other arguments *ej*, *ejj*, etc:

*t*(*d*1*,.. .*)

*t*(*dj ,.. .*)

*tj*(*ej ,.. .*) *uj*

*m*1

*m*2

1

1

*t*(*djj,.. .*)

*tjj*(*ejj,.. .*) *ujj*

*m*3

1

1

*t*(*djjj,.. .*) *···*

*tjjj*(*ejjj,.. .*) *ujjj*

1

1

(3)

Note that the terms *tj* and denotations *ej* depend only on *t* and the corresponding leaf *vj* of *d*1, and similarly for *tjj*, *ejj* and so on. Now let us consider the other

‘horizontal’ path *TD →Tc TMD → B*

*n *

*TMD*. We write *f*1 for the result in *MD* of

applying *c* to (*d*1), and similarly *g*1 = *c*(*e*1), etc. It may be represented thus:

*f*1

etc*.* (2*, vjjj*)

(1*, vjj*) *vj*

*m*3

*m*2

*m*1

where *mi* represent the effect-trees occurring in the behaviours of the *ith* successors of *d*1; they have been combined into a single layer of effects. We write (*m, v*) for the element of the initial algebra *D* which terminates in *m* steps with value *v*.

Now let us consider the same term *t*(*f*1*,.. .*) as before, but with arguments *di* replaced by their images under *c*. As before, the behaviour of *t*(*f*1*,.. .*) is obtained by applying the effectless rules to each leaf of *f*1, retaining the conclusions in an effect-tree of the same shape.

Writing (*m*) for *m* primes*j···j*, first note that for every *m*, and every leaf *v*(*m*) of the *mth* successor of *d*(*m*) of *d*1, there is a corresponding leaf (*m−* 1*, v*) of *f*1 (where we identify (0*, v*) with *v*). Now for any series of effectful transitions *t*(*d*1*,.. .*) *→*

1

*... → t*(*d*(*m—*1)*,.. .*) *→*  *u*(*m*) or *t*(*m*)(*e*(*m*)*,.. .*), the premises of their derivations

1 1

*√*

(*m—*1)

must respectively be *d*1 *→ dj* , *dj*

*→ djj*, *.. .*, *d*

*→ v*(*m*) (for some successor *dj*

1 1 1 1 1

of *d*1, and so on). In addition, by naturality of *ρ* the arguments *e*(*m*) of *t*(*m*) must come from those of *t*(*d*1*,.. .*). By the previous remark, there is a corresponding leaf

*i*

(*m −* 1*, v*(*m*)) of *f*1

with the same transitions: (*m −* 1*, v*(*m*)

*√*

) *→ ... → v*

(*m*). This

allows us to derive the following transitions:

*t*((*m −* 1*, u*(*m*))) *→ ... → t*(1*, u*(*m*)) *→ u*(*m*) or *t*(*m*)(*g*(*m*)*,.. .*)

1

This implies the behaviour of *t*(*f*1*,.. .*) is as follows, where the *u*(*m*) match the cor- responding leaves of ([3](#_bookmark19)), *t*(*m*) match the corresponding syntax-terms, and *f* (*m*)*, g*(*m*) are the image under *c* of *d*(*m*)*, e*(*m*).

etc*.*

*m*3

*m*2

*m*1

*t*(*fj ,.. .*) *→ t*(*fjj,.. .*) *→ tjjj*(*gjjj,.. .*)

1 1 1

*t*(*fj ,.. .*) *→ t*(*fjj,.. .*) *→ ujjj*

1 1

*t*(*fj ,.. .*) *→ tjj*(*gjj,.. .*)

*f*1

1 1

*t*(*fj ,.. .*) *→ ujj*

1

*tj*(*gj ,.. .*)

1

*uj*

Now we show the required induction step. First, consider the case where the behaviour of *t*(*d*1*,.. .*) depends on *d*1. The diagram ([3](#_bookmark19)) shows that after iterating the behaviour of *t*(*d*1*,.. .*) for *n* steps, the resulting terminal leaves can come from two sources – either from the terminal leaves *v*(*m*) of *d*1 (for *m < n*), resulting in terminal leaves *u*(*m*) in the behaviour of *t*(*d*1*,.. .*); or from the unshown execution of successor terms, *t*(*m*)(*e*(*m*)*,.. .*).

1

The above diagram shows that for every terminal leaf arising from the former case, there is a matching leaf in the behaviour of *t*(*f*1*,.. .*), and vice versa. For the latter case, it also shows that the successor terms *t*(*m*)(*e*(*m*)*,.. .*) reachable in *m* steps by *t*(*d*1*,.. .*) correspond with terms *t*(*m*)(*g*(*m*)*,.. .*) reachable by *t*(*f*1*,.. .*) in *m* steps. Any terminal leaves arising in *n* steps via these successors must arise in less than *n* steps when evaluating them directly; so the inductive hypothesis implies both horizontal paths agree at those leaves too.

1

1

Finally, if the behaviour of term *t*(*d*1*,.. .*) does not depend on *d*1, the derivation of its behaviour *t*(*d*1*,.. .*) *→ tj*(*ej ,.. .*) may be repeated exactly for the term obtained by applying *Tc*: *t*(*f*1*,.. .*) *→ tj*(*gj ,.. .*), introducing the same effects. The inductive hypothesis applied to these successor terms tells us that for *t*(*d*1*,.. .*) and *t*(*f*1*,.. .*), the terminal leaves produced by *n* steps of behaviour agree.

1

1

*2*

**Example 4.5** By inspection of the operational rules for stateless while, we see the abstract OS for the full while language are in evaluation-in-context format; Theorem

[4.4](#_bookmark18) implies compositionality of the effectful operational semantics illustrated by Examples [3.10](#_bookmark12), [3.12](#_bookmark13) and [3.14](#_bookmark15). Adequacy follows from the fact that two programs are operationally equivalence *p ∼*=*op q* iff they are identified by the final coalgebra map, which coincides with the denotational map; hence denotational equivalence implies operational equivalence.

However, the induced denotational semantics is more fine-grained than the tra- ditional semantics for While, in two respects. Firstly, we distinguish expressions like 0 and 0 + 0 + 0 because they respectively terminate in 0 and 2 steps, even though

the final value is the same; this is usually not the desired semantics.

Secondly, in the absence of equations relating effect-trees, denotations such as wr*x,*2(1*, ∗*) and wr*x,*2(wr*x,*2(1*, ∗*)) are inappropriately distinguished. However, the semantics does abstract away from the timing of effects provided they stay in the right order, as programs *p*1 and *p*2 of Example [4.1](#_bookmark16) demonstrate.

# Discussion and Future Work

In summary, we started with premise-supported operational specifications for multi- sorted, effectless languages, and demonstrated how to extend them with purely syntactic effects. This gives an operational model as a coalgebra in a Kleisli category, if the behaviour functor has a lifting. Under certain conditions, the final Kleisli- coalgebra exists, and we define an operational semantics by the unique Kleisli- coalgebra morphism into the final Kleisli-coalgebra. We may take this coalgebra to be the denotational model, by giving it an algebra structure.

By mapping from the final coalgebra in the underlying category into the final Kleisli-coalgebra, we showed that our semantic maps are as a quotient of those in Turi and Plotkin’s framework, and that our denotational semantics is adequate when the effectful operational semantics is in the evaluation-in-context format.

However there are several key omissions that we would like to address. Firstly, we have not considered equations on effects. Equations for finite effect-trees *T*Δ˜ *X* amounts to quotienting the algebra; in a Lawvere theory, they correspond to sketches [[1](#_bookmark20)]. A less syntax-driven approach would be to represent effect-trees via the free

model of the Lawvere theory in the base category. This would mean discarding the ‘final Δ-coalgebra’ approach of Definition [2.14](#_bookmark6) and taking *M* as the free Δ- algebra functor in Cppo; with equations, this corresponds to a free model functor on Cppo. This approach may also clarify how one might guarantee left-strictness of the resulting Kleisli category in a more canonical way.

Secondly, a more conceptual, abstract adequacy proof would allow easier gen- eralisation. It may help to have a more semantic characterisation of evaluation- in-context, possibly in terms of the operational specifications *ρ* rather than on the induced operational models *TD, TMD*.

Also, it is not yet clear how useful dependency-functions will be for obtain- ing effectful operational semantics *ρ*Gff from effectless ones. They work well for single-premise languages like while, and the multi-premise situation seems more straightforward when the effects are commutative. Otherwise, it may be easier to specify them directly; more examples will clarify this situation.

Lastly, our denotational semantics is sensitive to the number of steps-to- termination, which may be too fine-grained for some applications. However, in a Kleisli category it may be possible to to discard this information.

Another key question we did not have space to discuss is the relationship of the above semantics with comodels [[16](#_bookmark35)], which implement effects and are suitable for e.g. global state and interactive I/O [[20](#_bookmark38)]. However, they do not account for non-determinism, unless we exclude equations. By contrast, the conventional form

of operational semantics for nondeterminism is obtained by imposing equations on the effect-trees in the operational model we obtain in this paper. For each comodel with carrier *C* there is a notion of ‘comodel-induced operational model’, this time a coalgebra in the Kleisli category of the side-effect monad with bottom,

*SCX* = (*⊥* + *C × X*)*C*. Analogously to Kl(*G*Δ˜ ), one may give a Cppo-enrichment – so that there is a final Kleisli-coalgebra which we might take as a denotational model, and the target of an operational equivalence.

Given the effectful operational model – a *G*Δ˜ *B*-coalgebra – one may obtain a comodel-induced operational model, an *SCB*-coalgebra, via a natural transforma-

tion *ϵ* : *G*Δ˜

*→ SC* allowing comodels to ‘traverse’ trees and produce a result *x* in

*X*, as well as a new comodel state.

**Example 5.1** The comodels for global state are transition systems with implemen- tations of variable lookup and update; the cofree comodel is the canonical example, the standard implementation *S* of global store with carrier N*L*. Given a comodel, applying *ϵ* to the operational model for While in Example [3.12](#_bookmark13), we obtain an arrow which, transposed, is of form *T* 0 *× C → BT* 0 *× C*. This closely corresponds to the

*j j √ j*

standard transition-form of While programs: *⟨p, c⟩ → ⟨p ,c ⟩* or *⟨p, c⟩ → c*

each *c* in *C* is a state of the comodel – the ‘store’.

where

Consider *p* = if (*x* = 1) then *{y*:=2*}* else *{*skip*}* as in example [3.12](#_bookmark13), with the same shorthand notation. Suppose the store consists of two variables and is initially *c* = [*x* : 1*,y* : 0]:

|  |  |  |
| --- | --- | --- |
|  | *⟨*[*x* : 1*,y* : 0]*,* | if (*x* = 1) then *{y*:=2*}* else *{*skip*}⟩* |
| *→* | *⟨*[*x* : 1*,y* : 0]*,* | if (1 = 1) then *{y*:=2*}* else *{*skip*}⟩* |
| *→* | *⟨*[*x* : 1*,y* : 0]*,* | if (=1 (1)) then *{y*:=2*}* else *{*skip*}⟩* |
| *→* | *⟨*[*x* : 1*,y* : 0]*,* | *y*:=2*⟩* |
| *→* | *⟨*[*x* : 1*,y* : 2]*,* | *∗⟩* |

The map ! into the final *SC* -coalgebra is of type *T* 0 *→* (*⊥* + *C × D*)*C*, where *D* = (N *×* N*,* N *×* B*,* N *×* 1) as explained after Example [3.12](#_bookmark13). By uniqueness, one may check that it gives for each program *p* and initial comodel state *c*, either (a.) a tuple (*cj, n, v*) giving the final comodel state, steps to termination, and output value; or (b.) *⊥* if *p* diverges in state *c*. Indeed, one finds

!(*p*)([*x* : 1*,y* : 0]) = ([*x* : 1*,y* : 2]*,* 4)

in agreement with the above behaviour.

It may be that adequacy of the effectful semantics in this paper implies adequacy of the comodel-based semantics; if the combination of effect-trees comb is by monad strength, it may be possible to exploit the fact that *ρ*Gff is defined ‘in the same way for each monad’ in terms of monad morphisms.

# References

1. M. Barr and C. Wells. Toposes, triples, and theories. *Reprints in Theory and Applications of Categories no.1*, pages 1–289, 2005.
2. J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *J. ACM*, 24:68–95, January 1977.
3. I. Hasuo. *Tracing Anonymity with Coalgebras*. PhD thesis, Radboud University, Nijmegen, March 2008.
4. I. Hasuo, B. Jacobs, and A. Sokolova. Generic trace semantics via coinduction. *Logical Methods in* *Computer Science*, 3(4), 2007.
5. M. Hyland, G. Plotkin, and J. Power. Combining effects: sum and tensor. *Theor. Comput. Sci.*, 357:70–99, 2006.
6. M. Hyland and J. Power. Discrete lawvere theories and computational effects. *Theor. Comput. Sci.*, 366(1-2):144–162, 2006.
7. B. Jacobs. Trace semantics for coalgebras. *Electr. Notes Theor. Comput. Sci.*, 106:167–184, 2004.
8. P. Johann, A. Simpson, and J. Voigtl¨ander. A generic operational metatheory for algebraic effects. In

*Proc. LICS 2010*, pages 209–218. IEEE Computer Society, 2010.

1. B. Klin. Bialgebraic methods and modal logic in structural operational semantics. *Inf. Comput.*, 207(2):237–257, 2009.
2. B. Klin and V. Sassone. Structural operational semantics for stochastic process calculi. In R. M. Amadio, editor, *Proc. FoSSaCS 2008*, volume 4962 of *Lecture Notes in Computer Science*, pages 428–

442. Springer, 2008.

1. R. Matthes and T. Uustalu. Substitution in non-wellfounded syntax with variable binding. *Theor. Comput. Sci.*, 327(1-2):155–174, 2004.
2. E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.
3. H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley Professional Computing. Wiley, 1992.
4. G. D. Plotkin and J. Power. Adequacy for algebraic effects. In *FoSSaCS ’01: Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures*, pages 1–24, London, UK, 2001. Springer-Verlag.
5. G. D. Plotkin and J. Power. Notions of computation determine monads. In *FoSSaCS ’02: Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures*, London, UK, 2002. Springer-Verlag.
6. G. D. Plotkin and J. Power. Tensors of comodels and models for operational semantics. *Electr. Notes Theor. Comput. Sci.*, 218:295–311, 2008.
7. M. B. Smyth and G. D. Plotkin. The Category-Theoretic Solution of Recursive Domain Equations In

*SIAM J. Comput.*, 11(4):761–83, 1982.

1. J. Power. Enriched lawvere theories. *Theory and Applications of Categories*, 6(7):83–93, 1999.
2. J. Power. Countable lawvere theories and computational effects. *Electr. Notes Theor. Comput. Sci.*, 161:59–71, 2006.
3. J. Power and O. Shkaravska. From comodels to coalgebras: State and arrays. *Electron. Notes Theor.* *Comput. Sci.*, 106, 2004.
4. J. Power and D. Turi. A coalgebraic foundation for linear time semantics. In *In Category Theory and Computer Science*. Elsevier, 1999.
5. J. J. M. M. Rutten and D. Turi. Initial algebra and final coalgebra semantics for concurrency. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Proc. REX School/Symposium 1993*, volume 803 of *Lecture Notes in Computer Science*, pages 530–582. Springer, 1994.
6. D. Turi. *Functorial Operational Semantics and its Denotational Dual*. PhD thesis, Free University, Amsterdam, June 1996.
7. D. Turi and G. D. Plotkin. Towards a mathematical operational semantics. In *LICS*, pages 280–291, 1997.