An algebraic view of program composition

Pietro Cenciarelli

Ludwig–Maximilians–Universität München
cenciare@informatik.uni-muenchen.de

Abstract. We propose a general categorical setting for modeling program composition in which the call-by-value and call-by-name disciplines fit as special cases. Other notions of composition arising in denotational semantics are captured in the same framework: our leading examples are nondeterministic call-by-need programs and nonstrict functions with side effects. Composition of such functions is treated in our framework with the same degree of abstraction that Moggi's categorical approach based on monads allows in the treatment of call-by-value programs. By virtue of such abstraction, interesting program equivalences can be validated axiomatically in mathematical models obtained by means of modular constructions.

1 Introduction

In denotational semantics programs are interpreted in domains with suitable computational structure. For example, a domain for interpreting integer programs with exceptions must include (besides integers) denotations for exceptions and allow case analysis. In the categorical semantics proposed in [Mog91], the concrete structure of such domains is hidden behind the structure of a strong monad T, where TX is the domain of programs of type X. The advantage of describing program denotations in terms of the abstract structure of a monad is that a language can be extended with new computational features (e.g. a mechanism for exceptions or side-effects) and reinterpreted by just adopting a "more powerful" monad, without rewriting the old semantic equations. The *computational lambda calculus* (or computational *metalanguage*), the formal system associated in [Mog91] with this semantics, features a type constructor T and an operator let_T to compose programs of the form $A \to TB$, parametric in A, with programs of type A, which live in the domain TA.

The notion of composition implemented by let_T corresponds to a *call-by-value* parameter evaluation in that programs are modelled by morphisms of the form $A \to TB$, indexed by values in A, and they satisfy only a restricted form of substitution. On the other hand, *call-by-name* programs, which accept unevaluated expressions as inputs, are modelled by morphisms $TA \to TB$, indexed by "computations" in TA. In the metalanguage composition of such programs works according to β-reduction. Categorically, call-by-value programs compose in the Kleisli category of a monad T, while call-by-name programs compose in the base category.

Other notions of composition arise in computer science. For nondeterministic partial functions, for example, the *call-by-need* discipline differs from call-by-value in that it is nonstrict, and from call-by-name in that different occurrences of a parameter are always assigned the same value. How do call-by-need programs compose? In the computational metalanguage one has no choice but treating nondeterministic call-by-need programs as a special kind of call-by-name programs (the "additive" ones) and interpret them as morphism $\mathcal{P}A \to \mathcal{P}B$, for some power construction $\mathcal{P}$. However, a more finely grained semantics can be obtained by interpreting programs as morphisms of the form $A_\perp \to \mathcal{P}B$ and exploiting the relation between $\mathcal{P}$ and the lifting construction $(\_)_\perp$ to get an operation $let^{\perp}_{\mathcal{P}}$ for composing such morphisms just like $let_{\mathcal{P}}$ composes strict programs $A \to \mathcal{P}B$. Similar operations let^{R}_{T} are available for monads R and T when the structure of T extends, in a suitable sense to be explained below, the structure of R. Such operations and the categorical setting in which they arise are studied in this paper.

We propose a general categorical framework for modeling program composition in which the call-by-value and call-by-name disciplines fit as special cases. In view of the relation between monads and algebraic theories, different notions of composition are obtained by distinguishing the algebraic structure with respect to which programs behave as homomorphisms. This approach gives a uniform account of different strategies of parameter evaluation capturing notions of composition which do not accommodate naturally in the monadic setting of [Mog91]. Common programming constructs such as exception handlers, pipes etc. can be interpreted in the proposed framework without exposing the concrete structure of the semantic domains. The benefits are twofold: On the one hand our framework allows an axiomatic approach to validation of program equivalences in large classes of models. On the other hand it allows property-preserving reinterpretation of program constructs under model extensions, thus supporting a modular approach to denotational semantics in the spirit of [Mog90a,Cen95].

Synopsis. Section 2 discusses a motivating example. Section 3 gives a general categorical explanation of the constructions of Section 2 and presents a semantic framework which gives a uniform account of different disciplines of program composition in terms of the algebraic notion of homomorphism. The setting of Section 3 is further generalised in Section 4, where a weak theory of program composition is proposed; the theory features two operations, similar to the unit and lifting of Kleisli triples, of which simple equational properties are proven. Applications are described in Section 5 where these operations are used to define the semantics of common program constructs. Then properties of such constracts are derived axiomatically and shown to be preserved when models are suitably extended with new computational features.

2 A motivating example

The viewpoint proposed in this paper is that different strategies of parameter evaluation can be described in terms of how programs preserve computational

structure. In this section we discuss an example where semantic domains are provided by a composite monad $T = Q \circ R$. In such cases, an operation of program composition let_T^R is available, where only the structure of Q is preserved. Using such an operation for defining the denotational semantics of programs with side-effects (modeled by Q) and failure (modeled by R) we are able to validate program equivalences axiomatically. Observing that similar operations are available, with the same benefits, for monads which are not of the form $Q \circ R$, we look for the general semantic setting, subsuming monad composition, where such operations arise. This is done in the next section.

In a language with side effects and a mechanism for aborting computation, for example, a construct $handle\,(M, N)$ runs the program M and, if a failure occurs, makes a second attempt to produce a value by running N. Such a language may be interpreted in a cartesian closed category by mapping terms of type τ to elements of $((\llbracket\tau\rrbracket + 1) \times S)^S$, where S is some object of states. In particular:

$$\begin{aligned}\llbracket handle(M,N)\rrbracket = \lambda\, s : S.\ case\ \pi_0(\llbracket M\rrbracket s)\ of\ inl\,(v).\ \langle inl\,(v), \pi_1(\llbracket M\rrbracket s)\rangle \\ inr\,(u).\ \llbracket N\rrbracket(\pi_1(\llbracket M\rrbracket s)).\end{aligned}$$

By using lambda abstraction and projections, this equation exposes the concrete structure of the domains of interpretation. Hence, it works fine for a toy language but not for more realistic ones where domains of the form $((X + 1) \times S)^S$ may be inadequate to host programs. A more general presentation of the semantics of failure handling can be given by using the computational lambda calculus as metalanguage. In any model where programs are interpreted in domains of the form QRX, where Q is an arbitrary monad and $RX = X + 1$ we define:

$$\begin{aligned}\llbracket handle(M,N)\rrbracket\ =\ let_Q\ z \Leftarrow \llbracket M\rrbracket\ in\ case\ z\ of\ inl\,(v).\ val_Q(inl\,(v)) \\ inr\,(u).\ \llbracket N\rrbracket. \qquad (1)\end{aligned}$$

The case above is obtained when $QX = (X \times S)^S$. Adopting interpretation (1), one can work formally in a suitable theory of the computational lambda calculus and validate program equivalences for any model of the above class. The following equation, for example, can be easily derived from the axioms of the calculus.

$$handle\,(handle\,(L, M), N) = handle\,(L, handle\,(M, N)). \qquad (2)$$

Unfortunately, there are perfecly reasonable models for exceptions where no val_Q and let_Q operations are available to implement the handling of an exception. The monad $TX = ((X \times S) + 1)^S$, for example, models a "dramatic" form of failure, in which the state is lost upon occurrence of an exception. Equation (2) should also hold for programs of this form, but we have no formal (i.e. axiomatic) means of proving this equivalence without exposing again the concrete structure of T.

However, the monads R and T are related by two operations $val_T^R : R \to T$ and let_T^R, the latter feeding programs of the form $RA \to TB$ with arguments of type TA, which do for T what val_Q and let_Q do for the monad QR. Given $L : RA$, $M : TA$ and $N : RA \to TB$, define:

$$val_T^R(L) = \lambda s : S.\ case\ L\ of\ inl(v).\ inl\langle v, s\rangle \\ inr(u).\ inr(u)$$

$$let_T^R\ x \Leftarrow M\ in\ N(x) = \lambda s : S.\ case\ M(s)\ of\ inl\langle v, s'\rangle.\ N(inl(v))s' \\ inr(u).\quad N(inr(u))s_0,$$

where s_0 is some recovery state from which computation is resumed if a dramatic failure occurs. We can now define the semantics of dramatic exception handlers by just replacing val_Q with val_T^R and let_Q with let_T^R in (1), and the given proof of (2) goes through unchanged (see application 51). This approach is shown in Section 5 to yield a uniform interpretation of *handle* in a large class of models obtained by modular constructions: let H be an arbitrary monad and let $\mathcal{F}H$ be the monad $(\mathcal{F}H)X = (H(X \times S))^S$; assuming that suitable operations val_H^R and let_H^R are given for interpreting failure in the computational setting of H, one obtains operations $val_{\mathcal{F}H}^R$ and $let_{\mathcal{F}H}^R$ for reinterpreting failure in the more elaborate setting of $\mathcal{F}H$.

For which monads R and T can we find suitable operations val_T^R and let_T^R lifting R-computation to T-computation? What equations should one expect such operations to satisfy? Associativity seems a reasonable assumption. Moreover, in the above example, val_T^R is a *left* unit for let_T^R, that is: $let_T^R(val_T^R) = id$. On the other hand, it is not a *right* unit, that is, $let_T^R(f) \circ val_T^R = f$ does not hold. If let_T^R is to model a nonstrict form of program composition and the view is adopted that programs should form a category, this is a rather odd state of affairs. In the next section we look for a categorical picture to give us a convincing set of axioms for a general theory of program composition.

3 An algebraic view of program composition

In this section we propose an abstract categorical setting, called *extension setting*, for interpreting program composition. The underlying algebraic intuition is explained by discussing the example of non-deterministic call-by-need programs.

In the functional programming language Haskell programs are said to evaluate their parameters "by-need." Call-by-need differs from call-by-value in that application is nonstrict: A typical Haskell implementation of the Ackermann function, for example, would include a clause `ack 0 n =1`. Then, for a nonterminating program `loop`, the term `ack(0,loop)` evaluates to `1`, while it would fail to produce a result in Standard ML, where parameters are called by-value.

Call-by-need also differs from call-by-name in the presence of nondeterminism. A sequential program may exhibit nondeterministic behaviour when interacting with the operating system. For example, many programming languages,

including Haskell, feature a library function `GetTime` which returns nondeterministically the current value of the system clock. Let the call `ack(2,GetTime)` match the clause `ack n m = ack (ack (n-1) m) (m-1)`. With a call-by-name discipline, as in the Algol-like language of [Ten91], this call would result in evaluating the second argument at different times, thus producing nonsensical results. Conversely, arguments that are called by-need are evaluated only once, if ever. The discriminating notion here is *additivity*. Let `p` and `q` be programs and let `p or q` be the program which runs either `p` or `q`, nondeterministically. A program `f` is called additive when `f(p or q)`=`f(p) or f(q)`. Then, call-by-need and call-by-value programs are additive while call-by-name are not.

The above discussion suggests an "algebraic" explanation of these three calling mechanisms. Consider an interpretation in the category *Set* of small sets of a simple nondeterministic language, where programs producing values in X are interpreted as elements of the finite powerset $\mathcal{P}X$ of X. Two operations are fundamental in the finite powerset construction: binary union, which we can use to interpret `or`, and emptyset, which we can use to interpret `loop`. In this setting, one can view call-by-value programs as homomorphisms with respect to both operations, call-by-need with respect to union only, and call-by-name with respect to neither. The following interpretation of the three calling mechanisms is based upon this observation.

A nondeterministic program `p(x)`, with a call-by-name parameter `x`, expects an unevaluated expression as input. Therefore such programs correspond (roughly) to functions of the form $\mathcal{P}A \to \mathcal{P}B$, and `p(q)` is obtained by straight composition. On the other hand, if `p` is call-by-value, it must run on the *results* of its argument's evaluation and produce nothing if `q` produces none. Therefore, such programs correspond to morphisms of the form $A \to \mathcal{P}B$. Composition of such programs is obtained by exploiting the operation of Kleisli lifting $(\_)^{*\mathcal{P}}$ of the monad $\mathcal{P}$, which maps morphisms $A \to \mathcal{P}B$ to morphisms $\mathcal{P}A \to \mathcal{P}B$. In particular, $[\![\texttt{p(q)}]\!] = [\![\texttt{p}]\!]^{*\mathcal{P}}[\![\texttt{q}]\!]$. In the computational metalanguage this is written:

$$[\![\texttt{p(q)}]\!] \;=\; \mathit{let}_{\mathcal{P}}\; x \Leftarrow [\![\texttt{q}]\!]\; \mathit{in}\; [\![\texttt{p}]\!].$$

Morphisms of the form $f^{*\mathcal{P}}$ are strict and additive precisely because finite powersets are the free construction associated with the theory of *semilattices*. Semilattices are algebraic structures with a nullary operation 0 and a binary operation $\vee$ satisfying the following axioms:

$$\begin{aligned} x \vee x &= x \\ x \vee y &= y \vee x \\ x \vee (y \vee z) &= (x \vee y) \vee z \\ x \vee 0 &= x. \end{aligned}$$

A monad providing the free construction associated with an algebraic theory is said to *classify* the theory. The correspondence between monads and algebraic theories in enriched categories is studied in [KP93,Rob95].

To model a call-by-need $\mathtt{p}$, not only must we say how it behaves on values, but also what it can do when no value is produced in input. This can be done by interpreting $\mathtt{p}$ as a morphism $A_\perp \to \mathcal{P}B$, where $A_\perp = \{X \in \mathcal{P}A \mid \mathit{card}(X) \leq 1\}$ and $\mathit{card}(X)$ is the cardinality of X. Then, to interpret $\mathtt{p(q)}$, we look for an operation $(\_)^\star$ to return an additive, possibly nonstrict extension of $[\![\mathtt{p}]\!]$ to $\mathcal{P}A$.

The idea is to split the finite powerset monad into two constructions, one for each operation of the theory of semilattices. First we consider the theory of 0, with no axioms. The free models of this theory in Set are given by the lifting monad $(\_)_\perp$. To "finish" the construction, we cannot use the finite *nonempty* powerset monad $\mathcal{P}^\vee$ which classifies the theory of $\vee$, as $\mathcal{P}A \neq \mathcal{P}^\vee(A_\perp)$. In fact, we must consider this theory not in Set but in $\mathit{Set}^\perp$, the category of algebras of the monad $(\_)_\perp$. An algebra (A, a) for this monad consists of a set A and a distingiushed element $a \in A$. Homomorphisms from (A, a) to (B, b) are functions $f : A \to B$ such that $f(a) = b$. The free models of the theory of $\vee$ in $\mathit{Set}^\perp$ are given by the monad $\langle \mathcal{P}^+, \eta, \mu \rangle$, where $\mathcal{P}^+$ maps (A, a) to $(\{X \in \mathcal{P}A \mid a \in X\}, \{a\})$, $\eta_{(A,a)}(x) = \{x, a\}$ and $\mu_{(A,a)}(X) = \bigcup_{W \in X} W$. Clearly, writing $A_\perp$ for the free algebra $(A_\perp, \emptyset)$, the underlying set of $\mathcal{P}^+(A_\perp)$ is (isomorphic to) $\mathcal{P}A$.

In fact, $\mathcal{P}$ is the extension of the monad $\mathcal{P}^+$ along the forgetful functor $\mathit{Set}^\perp \to \mathit{Set}$ in the sense explained below. Similarly, the operation $(\_)^{*_{\mathcal{P}^+}}$, which lives in $\mathit{Set}^\perp$, extends to an operation $(\_)^\star$ in Set. In particular, $(\_)^\star$ maps functions $A_\perp \to \mathcal{P}B$ to functions $\mathcal{P}A \to \mathcal{P}B$ where $f^\star X = \bigcup \{f(x) \mid x \in X_\perp\}$. Then, call-by-need is modelled as: $[\![\mathtt{p(q)}]\!] = [\![\mathtt{p}]\!]^\star [\![\mathtt{q}]\!]$. Pretty-printing:

$$[\![\mathtt{p(q)}]\!] \;=\; \mathit{let}^\perp_{\mathcal{P}}\, x \Leftarrow [\![\mathtt{q}]\!]\ \mathit{in}\ [\![\mathtt{p}]\!].$$

Functions of the form $f^\star$ are strict only when f is strict. Moreover, they are additive because $(\_)^\star$ extends the Kleisli composition of the monad $\mathcal{P}^+$ which classifies the theory of $\vee$.

The situation just described generalises as follows. We call *extension setting* a categorical picture

$$\mathcal{C} \underset{G}{\overset{F}{\longleftarrow\!\!\longrightarrow}} \mathcal{X} \xrightarrow{M} \mathcal{X}$$

where $F \dashv G$ are adjoint functors and $M = \langle M, \eta^M, \mu^M \rangle$ is a monad on $\mathcal{X}$. Let $R = \langle GF, \eta^R, \mu^R \rangle$ be the monad induced by the adjunction $F \dashv G$ on $\mathcal{C}$, let ϵ be the counit of this adjunction and let T be the functor $GMF : \mathcal{C} \to \mathcal{C}$. The latter is the right Kan extension of GM along G. The natural transformations $\eta^T = G\eta^M F \circ \eta^R$ and $\mu^T = G\mu^M F \circ GM\epsilon MF = G(\epsilon MF)^{*_M}$ endow T with the structure of a monad. Following [Str72], we call such a monad the extension of M along G. This extension is unique in the sense of [Str72, §2].

We write settings like the one above as triples $\langle F, G, M \rangle$ and indicate with R, T and ϵ respectively the monad GF, the monad GMF and the counit of the adjunction $F \dashv G$. We call $\mathcal{C}$ the *base category* of the setting.

Given an extension setting $\langle F, G, M\rangle$ on a base category $\mathcal{C}$, we intend to interpret program composition by means of a family $(\_)^\star$ of associative operations of the form:

$$(\_)^\star_{A,B} : \mathcal{C}(RA, TB) \to \mathcal{C}(TA, TB),$$

extending M-lifting along G. More formally: we require that, for all morphisms $h : FA \to MFB$, $f : RB \to TC$ and $g : RA \to TB$, the following holds:

$$(Gh)^\star = G(h^{*_M}) \tag{3}$$
$$f^\star \circ g^\star = (f^\star \circ g)^\star \tag{4}$$

When the functor G is monadic, (3) requires $(\_)^\star$ to behave like $(\_)^{*_M}$ on R-homomorphisms. Since T extends M along G, this is to say that $f^\star$ should preserve T structure whenever f preserves R structure. Note that the natural transformation $\iota = G\eta^M F : GF \to GMF$ in a setting $\langle F, G, M\rangle$ is a monad morphism from R to T, and that the equation $\iota^\star = id$ follows immediately from (3).

Example 31 *Call-by-value and call-by-name.*

Given a monad T, an interpretation $[\![\mathtt{p(q)}]\!] = [\![p]\!]^\star[\![q]\!]$ of call-by-value program composition is obtained in the setting $\langle Id, Id, T\rangle$, where R is the identity and $(\_)^\star = (\_)^{*_T}$. On the other hand, a call-by-name interpretation is obtained in any setting $\langle F, G, Id\rangle$, where $R = GF = T$ and $(\_)^\star$ is the identity. These examples are the "extreme" cases where R possesses all or nothing of the structure of T. The following examples show that intermediate cases are also interesting.

Example 32 *Call-by-need.*

Here we describe a setting, analogous to the finite powerset example developed earlier, relating the lifting and Hoare powerdomain monads in the category of cpos. A similar picture can be drawn for algebraic cpos.

Let *Cpo* be the category of possibly bottomless cpos. The Hoare powerdomain $\mathcal{P}(A)$ of such a cpo A is the set of downward closed subsets of A ordered by inclusion. Empty set and union are the universal operations on $\mathcal{P}(A)$ satisfying the theory of semilattices, together with the axiom: $x \vee y \geq x$.

One can split this construction in two steps as done for powersets. In particular, let $Cpo^\perp$ be the category of cpos with bottom element and strict continuous functions. This is the category of algebras of the lifting monad. If X is an object of $Cpo^\perp$, let $\mathcal{P}^+X$ be the cpo of nonempty downwards closed subsets of X ordered by inclusion and let $\eta_X : X \to \mathcal{P}^+X$ map x to $\{y \mid y \leq x\}$. The union operation makes of $\mathcal{P}^+X$ the free $\{\vee\}$-algebra generated by X in $Cpo^\perp$. That is: for any map $f : A \to B$ in $Cpo^\perp$, where B is endowed with an operation $\vee$ satisfying the given equations, there is a unique $\vee$-homomorphism $f^\dagger : \mathcal{P}^+A \to B$ such that $f^\dagger \circ \eta_A = f$. This gives to $\mathcal{P}^+$ the structure of a monad which extends to $\mathcal{P}$ along the forgetful functor $G^\perp = Cpo^\perp \to Cpo$. The operation $(\_)^\star$ such that $f^\star X = \bigcup\{f(x) \mid x \in X_\perp\}$ extends $(\_)^{*_{\mathcal{P}^+}}$ along $G^\perp$.

Example 33 *Composition of monads.*

Any two monads which compose give rise to an operation $(\_)^\star$. Let R and Q be monads on a category $\mathcal{C}$ and let the natural transformation $\lambda : RQ \stackrel{\cdot}{\to} QR$ be a distributive law of Q over R. The functor $\hat{Q} : \mathcal{C}^R \to \mathcal{C}^R$ mapping R-algebras α to $Q\alpha \circ \lambda$ has the structure of a monad which forms an extension setting $\langle F^R, G^R, \hat{Q}\rangle$. The monad $\hat{Q}$ is the *lifting* of Q to $\mathcal{C}^R$ associated with λ (see [Bec69]). In particular, we have $G^R\hat{Q} = QG^R$. The extension of $\hat{Q}$ along the forgetful functor G^R is the composite monad QR. In fact, we have $G^R\hat{Q}F^R = QG^RF^R = QR$. Note that the pair (F^R, λ) is a monad morphism $Q \to \hat{Q}$.

Writing $h^{*\hat{Q}} = \mu^{\hat{Q}} \circ \hat{Q}h$ and noticing that μ^Q is the underlying natural transformations of $\mu^{\hat{Q}}$, we see that the operation $(\_)^{*Q}_{RA,RB}$, which is obviously associative, extends $(\_)^{*\hat{Q}}$ along the forgetful functor G^R.

4 Notions of composition

In the previous section we developed some intuition on how a general operation $(\_)^\star$ to interpret program composition should look like and we wrote axioms to support our intuition formally. We assumed that such an operation, similar to composition in the Kleisli category of a monad, lives in an extension setting. Here we develop the theory of more general notions of composition, which need not belong to an extension setting. When they do, we prove that the equations of Section 3 are satisfied. However, it is in the more general theory that we derive the properties that we expected to hold from our earlier discussion.

Let $R : \mathcal{C} \to \mathcal{C}$ be a functor, let T be a monad on $\mathcal{C}$ and let $\sigma : R \to T$ be a natural transformation. Given $h : A \to TB$, we write $h_\sigma : RA \to TB$ the morphism

$$h_\sigma = h^{*_T} \circ \sigma.$$

Note that, when R has the structure of a monad and σ is a monad morphism, there is a forgetful functor $G_\sigma : \mathcal{C}^T \to \mathcal{C}^R$ mapping T-algebras (A, α) to R-algebras $(A, \alpha \circ \sigma)$. In this case we have $h_\sigma = G^R h^\dagger$, where $h^\dagger : F^R A \to G_\sigma F^T B$ corresponds bijectively to h by the adjunction $F^R \dashv G^R$.

Definition 41 *Let $R : \mathcal{C} \to \mathcal{C}$ be a functor and let T be a monad on a category $\mathcal{C}$; a* weak notion of composition *is a pair $(\iota, (\_)^\star)$, where $\iota : R \to T$ is a natural transformation and $(\_)^\star$ is a family of operations:*

$$(\_)^\star_{A,B} : \mathcal{C}(RA, TB) \to \mathcal{C}(TA, TB)$$

satisfying the following equations: for all $f : RB \to TC$, $g : RA \to TB$ and $h : A \to TB$,

$$f^\star \circ g^\star = (f^\star \circ g)^\star \tag{4}$$

$$h^\star_\iota = h^{*_T}. \tag{5}$$

We write $(\iota, (\_)^\star) : R \to T$ for a weak notion of composition as above to make R and T understood.

Proposition 42 *The operation $(\_)^\star_{A,B}$ of a weak notion of composition is natural in B.*

Proof. Let $(\iota, (\_)^\star) : R \to T$ be a weak notion of composition; naturality of $(\_)^\star_{A,B}$ in B is expressed by the equation $Tf \circ g^\star = (Tf \circ g)^\star$, where $f : B \to C$ and $g : RA \to TB$. Then,

$$Tf \circ g^\star = (\eta \circ f)^* \circ g^\star = (\eta \circ f)^\star_\iota \circ g^\star = ((\eta \circ f)^\star_\iota \circ g)^\star = (Tf \circ g)^\star.$$

Proposition 43 *Weak notions of composition $(\iota, (\_)^\star) : R \to T$ satisfy the following equations:*

$$\iota^\star = id_T \tag{6}$$
$$f^* \circ g^\star = (f^* \circ g)^\star \tag{7}$$
$$Th = (\iota \circ Rh)^\star. \tag{8}$$

Proof. Note that $\iota = \eta_\iota$. Let $f : B \to TC$, $g : RA \to TB$ and $h : A \to B$,

$$\begin{aligned} \iota^\star &= \eta_\iota^\star = \eta^* = id_T; \\ f^* \circ g^\star &= f_\iota^\star \circ g^\star = (f_\iota^\star \circ g)^\star = (f^* \circ g)^\star; \\ Th &= (\eta \circ h)^* \circ \iota^\star = ((\eta \circ h)^* \circ \iota)^\star = (Th \circ \iota)^\star = (\iota \circ Rh)^\star. \end{aligned}$$

Weak notions of composition $(\iota, (\_)^\star) : R \to T$ in which R is a monad on $\mathcal{C}$ and ι is a monad morphism, often live in an extension setting. In fact, the forgetful functor $G_\iota : \mathcal{C}^T \to \mathcal{C}^R$ induced by ι often has left adjoint. This is always the case when $\mathcal{C}$ is *Set* [BW85, 9.3]. In general, it is well known that G_ι has left adjoint when $\mathcal{C}^T$ has all coequalisers of reflexive pairs [Lin69, coroll. 1]. A sufficient condition for that to happen is that $\mathcal{C}$ has such coequalisers and T preserves them [Lin69, coroll. 3].

If G_ι has left adjoint F_ι, we obtain an extension setting $\langle F^R, G^R, M\rangle$ where M is the monad induced on $\mathcal{C}^R$ by the adjunction $F_\iota \dashv G_\iota$. In this setting, $G^R M F^R = G^R G_\iota F_\iota F^R = G^T F^T = T$.

The following theorem shows the correspondence between weak notions of composition and operations satisfying (3) and (4) as in Section 3.

Theorem 44 *Let $\langle F, G, M\rangle$ be an extension setting; a family of operations $(\_)^\star_{A,B} : \mathcal{C}(RA, TB) \to \mathcal{C}(TA, TB)$ satisfies (3) and (4) if and only if $(\iota, (\_)^\star)$ is a weak notion of composition, for some ι such that $\iota \circ \eta^R = \eta^T$.*

Proof. [(3)⇒(5)] Let $\iota = G\eta^M F$ and let $h : A \to TB$. Using the naturality of $(\_)^{*_M}$, we have:

$$\begin{aligned} h_\iota^\star &= (\mu_B^T \circ \iota_{TB} \circ Rh)^\star = (G\epsilon^{*_M}_{MFB} \circ G\eta^M_{FTB} \circ GFh)^\star = (G(\epsilon^{*_M}_{MFB} \circ \eta^M_{FTB} \circ Fh))^\star \\ &= G(\epsilon^{*_M}_{MFB} \circ \eta^M_{FTB} \circ Fh)^{*_M} = G\epsilon^{*_M}_{MFB} \circ G(\eta^M_{FTB} \circ Fh)^{*_M} \\ &= \mu_B^T \circ G((\eta^M_{FTB})^{*_M} \circ MFh) = \mu_B^T \circ Th = h^{*_T}. \end{aligned}$$

[(5)⇒(3)] Let $(\iota, (\_)^\star) : R \to T$, with $\iota \circ \eta^R = \eta^T$, be a weak notion of composition. For any $f : RA \to TB$ we have $(f \circ \eta^R)_\iota = f$ by easy calculations. Let $h : FA \to MFB$, we have:

$$\begin{aligned}(Gh)^\star &= (Gh \circ \eta^R_A)^\star_\iota = (Gh \circ \eta^R_A)^{*_T} = \mu^T \circ T(Gh \circ \eta^R_A)\\ &= G\mu^M_{FB} \circ GM\epsilon_{MFB} \circ GMFGh \circ T\eta^R_A\\ &= G\mu^M_{FB} \circ GMh \circ GM\epsilon_{FA} \circ GMF\eta^R_A\\ &= G(\mu^M_{FB} \circ Mh) \circ GM(\epsilon_{FA} \circ F\eta^R_A) = Gh^{*_M}.\end{aligned}$$

Definition 45 *A* notion of composition *is a weak notion of composition* $(\iota, (\_)^\star)$ *such that, for all* $f : RA \to TB$ *the following equation holds:*

$$f^\star \circ \iota = f. \tag{9}$$

When working with sets, the inclusions $\iota_A : A_\perp \to \mathcal{P}A$ of Section 3 do not satisfy (9) while they do in the case of cpos (example 32). Another *strictly* weak notion of composition is the pair of operation val^R_T and let^R_T defined in the introduction to model dramatic failure. On the other hand, (9) is satisfied in the models of interleaving of Application 54.

The following results are used in the next section:

Proposition 46 *The operation* $(\_)^\star_{A,B}$ *of a notion of composition is natural in* A*.*

Proof. Let $(\iota, (\_)^\star) : R \to T$ be a notion of composition; naturality of $(\_)^\star_{A,B}$ in A is expressed by the equation $f^\star \circ Tg = (f \circ Rg)^\star$, where $f : RA \to TB$ and $g : C \to A$. Then, from (8) and (9) we have:

$$f^\star \circ Tg = f^\star \circ (\iota_A \circ Rg)^\star = (f^\star \circ \iota_A \circ Rg)^\star = (f \circ Rg)^\star.$$

Proposition 47 *Let* $(\iota, (\_)^\star) : R \to T$ *be a weak notion of composition and let* $\iota' : S \to T$ *be a natural transformation such that* $\iota = \iota' \circ \nu$ *for some natural transformation* ν*. The pair* $(\iota', (\_)^{\star\nu})$*, where* $f^{\star\nu} = (f \circ \nu)^\star$*, is a weak notion of composition* $S \to T$*.*

Proof. The associativity of $(\_)^{\star\nu}$ is an immediate consequence of the associativity of $(\_)^\star$. Moreover, let $h : A \to TB$,

$$h^{\star\nu}_{\iota'} = (h^{*_T} \circ \iota')^{\star\nu} = (h^{*_T} \circ \iota' \circ \nu)^\star = (h^{*_T} \circ \iota)^\star = h^{*_T} \circ \iota^\star = h^{*_T}.$$

Proposition 48 *Let* $(\iota, (\_)^\star) : R \to T$ *be a notion of composition and let* $\nu : R \to S$ *be a natural transformation with a right inverse, that is a natural transformation* ν' *such that* $\nu \circ \nu' = id$*. The pair* $(\iota', (\_)^{\star\nu})$*, where* $\iota' = \iota \circ \nu'$ *and* $f^{\star\nu}$ *is as above, is a notion of composition* $S \to T$*.*

Proof. It is a weak notion of composition by the previous proposition. Moreover:

$$f^{\star\nu} \circ \iota' = (f \circ \nu)^\star \circ \iota \circ \nu' = f \circ \nu \circ \nu' = f.$$

5 Applications to modular semantics

In [Mog90a], a modular approach to denotational semantics is proposed, where mathematical models of computation are obtained by stepwise application of *monad constructors*. These are functions $\mathcal{F}$ mapping monads to monads and satisfying certain naturality conditions. Intuitively, the monad $\mathcal{F}T$ augments the structure of T with the machinery to interpret a new computational feature. For example, the constructor $\mathcal{F}$ such that $(\mathcal{F}T)A = (T(A \times S))^S$ adds to T the capability of modelling side-effects. Monad constructors are studied in [CM93,Cen95].

In [Mog90b] the notion of *uniform redefinition* is introduced to lift operations defined in a computational setting M to a new setting $\mathcal{F}M$. Let $\zeta(\_)$ be some type scheme, let $op_A : \zeta(MA)$ be an operation defined for a monad M in a category $\mathcal{C}$, let $H : \mathcal{C} \to \mathcal{C}$ be an endofunctor and let $\mathcal{F}$ be a monad constructor of the form $(\mathcal{F}T)A = THA$; op can be uniformly redefined for the monad $\mathcal{F}M$ as follows:

$$(\mathcal{F}op)_A = op_{HA}.$$

This technique is not always applicable: when either $\mathcal{F}$ or op are not of the appropriate form, ad-hoc redefinitions must be sought. The above constructor for side-effects, for example, does not fulfill the requirements. Neither does the operation $C_{A,B} : (A \to MB) \times (MA \to MB) \times MA \to MB$ used in [CM93, example 2.10] to perform case analysis on interleaving programs. In this section we propose a technique based on notions of composition which yields well behaved redefinitions of operations in both cases.

We show that two benefits derive from using notions of composition to define operations in a computational setting M: on the one hand it allows properties of the operations to be formally derived without exposing (all of) the structure of M (thus for a large class of models); on the other hand it allows the operations to be redefined in cases where uniform redefinitions are not available, and their properties automatically preserved.

Application 51 *Reinterpreting failure in state models.*

Let R be the monad $RA = A + 1$. We say that a monad $\langle H, \eta, (\_)^* \rangle$ has a *structure for failure* when it is equipped with a weak notion of composition $(\iota, (\_)^\star) : R \to H$ and with a natural transformation $\rho : H \to H$ such that:

$$\iota \circ inl = \eta \tag{10}$$
$$f^\star \circ \eta = f \circ inl \tag{11}$$
$$f^\star \circ fail = \rho \circ f \circ inr \tag{12}$$
$$f^* \circ fail = fail \tag{13}$$

where $fail_A : HA$ is the natural transformation $\iota \circ inr$. Intuitively, $\rho(N)$ is the program running N after some recovery action. For example, ρ would be the

identity for $HA = ((A+1) \times S)^S$ while it would feed its argument with some recovery state for $HA = ((A \times S)+1)^S$.

An operation $handle_A : HA \times HA \to HA$ running its first argument and handling a possible failure with its second can be defined as follows:

$$handle\,(M,N) = [\eta, N]^\star \circ M.$$

The following equations are satisfied by *fail* and *handle* in any structure for failure:

$$\begin{aligned} handle\,(\eta, N) &= \eta \\ handle\,(fail\,, N) &= \rho(N) \\ handle\,(L, handle\,(M,N)) &= handle\,(handle\,(L,M), N). \end{aligned}$$

In fact, $handle\,(\eta,N) = [\eta,N]^\star \circ \eta = [\eta,N] \circ inl\, = \eta$ and similarly for the other equations.

The signature $HA \times HA \to HA$ is indeed of the form $\zeta(HA)$, which makes *handle* qualify for uniform redefinition. Not so however for the monad constructor $\mathcal{F}$ mapping H to $(\mathcal{F}H)A = (H(A \times S))^S$. This constructor, however, extends to structures for failure as follows:

$$\begin{aligned} (\mathcal{F}\rho)w &= \lambda\, s.\, \rho(w(s_0)) \\ (\mathcal{F}\iota)z &= \lambda\, s.\, let_H\ a \Leftarrow \iota(z)\ in\ \ val_H\langle a, s\rangle \\ f^{\mathcal{F}\star}w &= \lambda\, s.\, let^R_H\ z \Leftarrow w(s)\ in\ \ case\ \ z\ \ of\ inl\langle a, s'\rangle.\, f(inl(a))s' \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad inr(u). \quad f(inr(u))s_0. \end{aligned}$$

These operations are easily shown to satisfy the axioms (10-13). Thus, by suitably extending the action of the constructor $\mathcal{F}$ to weak notions of composition $R \to H$, operations such as *fail* and *handle* are automatically redefined in models of computation with side-effects, in such a way that the relevant properties are also preserved.

Remark. The operations val^R_T and let^R_T of Section 1 are obtained by applying the constructuion just described to the identity notion of composition $R \to R$.

Application 52 Inwards *monad constructors.*

Here we describe a class of monad constructors $\mathcal{F}$ which have a canonical lifting of (weak) notions of composition $R \to T$ to $\mathcal{F}R \to \mathcal{F}T$. In the next application we use this construction to obtain a reinterpretation interleaving in models of exceptions.

We call *inwards* a monad constructor $\mathcal{F}$ such that:

$$\begin{aligned} &(\mathcal{F}M)A = M(HA) \ \text{ for some functor } H, \text{ and} \\ &\quad \mu^{\mathcal{F}M} = \mu^M H \circ Mp \ \text{ for some natural transformation } p : HMH \to MH. \end{aligned}$$

Remark. The above condition on $\mu^{\mathcal{F}M}$ arises when composing monads with functors. Let H be a functor, let $\eta^H : Id \to H$ be a natural transformation and let η^{MH} be $\eta^M H \circ \eta^H$. There is a one to one correspondence between natural transformations $p : HMH \to MH$ satisfying

$$\begin{aligned} p \circ \eta^H MH &= id \\ p \circ H\eta^{MH} &= \eta^M H \\ p \circ H\mu^{MH} &= \mu^{MH} \circ pMH, \end{aligned}$$

where μ^{MH} is the natural transformation $\mu^M H \circ Mp$, and monads $\langle MH, \eta^{MH}, \mu\rangle$ such that $\mu \circ \mu^M HMH = \mu^M H \circ M\mu$ (see [JD93]).

Proposition 53 *Let the monad constructor $(\mathcal{F}M)A = M(HA)$ be inwards, and let $(\iota, (\_)^\star) : R \to T$ be a (weak) notion of composition. The pair of operations $(\iota H, (\_)^{\star H})$, where $(\_)^{\star H}_{A,B} = (\_)^\star_{HA,HB}$, is a (weak) notion of composition $\mathcal{F}R \to \mathcal{F}T$.*

Proof Associativity (and right unit) are inherited immediately form $(\iota, (\_)^\star)$. Let $f : A \to THB$, noticing that $f^{*_{TH}} = (p \circ Hf)^{*_T}$, we have:

$$\begin{aligned} f^{\star H}_{\iota H} &= (f^{*_{TH}} \circ \iota)^\star = ((p \circ Hf)^{*_T} \circ \iota)^\star = (p \circ Hf)^{*_T} \circ \iota^\star \\ &= \mu^T \circ Tp \circ THf = \mu^{TH} \circ THf = f^{*_{TH}}. \end{aligned}$$

Application 54 *Reinterpreting interleaving in models of exceptions.*

In [CM93], the semantics of computation with interleaving is described in terms of the "resumptions monad" $TA = \mu X.Q(A + X)$ and two families of operations

$$\begin{aligned} \tau_A &: TA \to TA \\ C_{A,B} &: (A \to TB) \times (TA \to TB) \times TA \to TB. \end{aligned}$$

from which interesting programming constructs can be defined, such as the operator *pand* of parallel composition described in [Cen95, 7.3]. The operations τ and C arise from a notion of composition $(\iota, (\_)^\star) : Id + T \to T$. Let α_A be the isomorphism $Q(A + TA) \to TA$ and let γ_A be its inverse. We define:

$$\begin{aligned} \iota(z) &= \mathit{case}\ z\ \mathit{of}\ \mathit{inl}(a).\ \alpha(\mathit{val}_Q(\mathit{inl}(a))) \\ &\qquad\qquad \mathit{inr}(u).\alpha(\mathit{val}_Q(\mathit{inr}(u))) \\ f^\star(w) &= \alpha(\mathit{let}_Q\ z \Leftarrow \gamma(w)\ \mathit{in}\ \gamma(f(z))) \end{aligned}$$

The associativity of $(\_)^\star$ follows easily from the associativity of $(\_)^{*_Q}$. Similarly, $f^\star \circ \iota = f$ follows from $f^{*_Q} \circ \eta^Q = f$. As for (5), note that, for $h : A \to TB$, we have:

$$\begin{aligned} h_\iota(z) \ =\ & \mathit{case}\ z\ \mathit{of}\ \mathit{inl}(a).\ h(a) \\ & \qquad\qquad \mathit{inr}(u).\alpha(\mathit{val}_Q(\mathit{inr}(h^{*_T}u))) \end{aligned}$$

and hence

$$\begin{aligned} h^\star_\iota(w) &= \alpha(let_Q\ z \Leftarrow \gamma(w)\ in\ case\ z\ of\ inl(a).\ \gamma(h(a)) \\ &\qquad\qquad\qquad\qquad\qquad\qquad inr(u).\ val_Q(inr(h^{*_T}u))) \\ &= h^{*_T}w. \end{aligned}$$

Now, the operations τ and C of [CM93] can be defined as follows:

$$\begin{aligned} \tau(w) &= \iota(inr(w)) \\ C(f,g,w) &= (\lambda z.\ case\ z\ of\ inl(a).\ f(a) \\ &\qquad\qquad\qquad\qquad inr(u).\ g(u))^\star w. \end{aligned}$$

Noticing that $\eta^T = \iota \circ inl$, from (6) and (9) one can easily derive the equations

$$\begin{aligned} C(f,g,\eta) &= f \\ C(f,g,\tau) &= g \\ C(\eta,\tau,w) &= w \end{aligned}$$

showing that τ and C behave respectively as right injection and case analysis. The commutativity of the operator *pand* of [Cen95, 7.3] can be easily derived from these equation and from the commutativity of an operation of nondeterministic choice.

Let H be the functor $HA = A + E$, where E is some object "of exceptions." Given a monad M there is a *unique* monad $\langle MH, \eta, \mu\rangle$ such that $\eta = \eta^M H \circ inl$ and $\mu \circ \mu^M HMH = \mu^M H \circ M\mu$. This follows from the remark in 52. We write $\mathcal{F}$ the monad constructor for exceptions, mapping monads M to MH.

The constructor $\mathcal{F}$ is inwards. Hence, applying Proposition 53 to the notion of composition $Id + T \to T$ defined above, we get a notion of composition $H + TH \to TH$ satifying (9). Then, noticing that the natural transformation $[id + \eta^T H \circ inr, inr] : H + TH \to Id + TH$ has a right inverse $inl + id$, we obtain, by Proposition 48, a notion of composition $Id + TH \to TH$ to interpret interleaving in models constructed by $\mathcal{F}$. Again, C and τ are automatically reinterpreted in such models and the relevant properties are preserved.

6 Conclusions

We proposed a general categorical setting for modeling program composition in which the call-by-value and call-by-name disciplines fit as special cases. Call-by-need is also captured in this framework for nondeterministic programs; it is an interesting question whether call-by-need programs with side effects can be captured similarly. The proposed theory of program composition features two operations ι and $(\_)^\star$, reminiscent of the unit and lifting of Kleisli triples, of which only weak properties are assumed. These are however enough to derive simple equational properties of common program constructs such exception

handling and parallel composition. The paper argues that, by defining program constructs in terms of ι and $(\_)^\star$, not only can one validate program equivalences axiomatically for large classes of models, but also reinterpret the constructs when models are extended, preserving the truth of the relevant axioms. Since we are able to do this in cases where the uniform redefinition proposed in [Mog90b] are not available, our technique makes one step forward towards a modular approach to denotational semantics. The proposed technique is applied in [Cen98] to the semantics of Java, where we seek a modular proof of computational adequacy with respect to the operational semantics of [CKRW98].

References

[Bec69] Jon Beck. Distributive laws. In B. Eckman, editor, *Seminar on triples and categorical homology theory*, pages 119–140, Berlin, 1969. Springer LNM 80.

[BW85] M. Barr and C. Wells. *Toposes, triples and theories*. Springer-Verlag, New York, 1985.

[Cen95] P. Cenciarelli. *Computational applications of calculi based on monads*. PhD thesis, Department of Computer Science, University of Edinburgh, 1995. CST-127-96. Also available as ECS-LFCS-96-346.

[Cen98] P. Cenciarelli. Objects and computation. Presented at the Dagstuhl seminar on "The Semantic Challenge of Object-Oriented Programming", Schloss Dagstuhl, Wadern, Germany, 28/6 - 3/7 1998.

[CKRW98] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An Event-Based Structural Operational Semantics of Multi-Threaded Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, 1523 LNCS. Springer, 1998.

[CM93] P. Cenciarelli and E. Moggi. A syntactic approach to modularity in denotational semantics. In *Proceedings of 5th Biennial Meeting on Category Theory and Computer Science*. CTCS-5, 1993. CWI Tech. Report.

[JD93] M.P. Jones and L. Duponcheel. Composing monads. Technical Report YALEU/DCS/RR-1004, Department of Computer Science, Jale University, 1993.

[KP93] G.M. Kelly and A.J. Power. Adjunctions whose counits are coequalizers, and presentations of finitary monads. *Journal of Pure and Applied Algebra*, 89:163–179, 1993.

[Lin69] F.E.J. Linton. Coequalizers in Categories of Algebras. In *Seminar on Triples and Categorical Homology Theory*, pages 75–90. Springer LNM 80, 1969.

[Mog90a] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, University of Edinburgh, Comp. Sci. Dept., 1990.

[Mog90b] E. Moggi. Modular approach to denotational semantics. Unpublished manuscript, November 1990.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[Rob95] E.P. Robinson. Note on the presentation of enriched monads. Unpublished manuscript, available by ftp from `theory.doc.ic.ac.uk`, 1995.

[Str72] R. Street. The formal theory of monads. *Journal of Pure and Applied Algebra*, 2:149–168, 1972.

[Ten91] R.D. Tennent. *Semantics of programming languages*. Prentice Hall, 1991.