An Abstract View of Programming Languages[1]

Eugenio Moggi
Lab. for Found. of Comp. Sci.
University of Edinburgh
EH9 3JZ Edinburgh, UK
em@lfcs.edinburgh.ac.uk

June 1989

[1]These Notes were produced at Stanford University as part of the course CS 359 taught in Spring Term 1989. A special thank to John Mitchell, who made my visit to Stanford possible, and to all students who attended the course and provided useful feedback.

Abstract

The aim of these course notes is to show that notions and ideas from Category Theory can be useful tools in Computer Science, by illustrating some recent applications to the study of programming languages based on the principle "notions of computation as monads". The main objective is to propose a unified approach to the denotational semantics of programming languages. The category-theoretic notions introduced in the course will be presented with specific applications in mind (so that they will not sound too abstract) and guidelines for linking abstract and concrete concepts will be stressed.

Contents

Chapter 1

Introduction

The aim of this course is to show that notions and ideas from Category Theory can be useful tools in Computer Science for **formulating definitions and theories** and **structuring complex concepts**, by illustrating some recent applications to the study of programming languages ([Mog89b, Mog89a])[1]. To motivate the use of Category Theory made in these notes, I find particularly enlightening few passages from Chapter 2 of [Mos89], where Mosses gives an overview of Denotation Semantics focusing not only on its achievements, but also on its **failures**.

Successes:

- **mathematical models** for programming languages (compare with operational and axiomatic approach)
- **canonical definition** of the meaning of programs (denotational model)
- documents the **design** of a programming language
- establishes a **standard** for implementations, but does **not** specify the techniques used in implementations
- provides a basis for **reasoning about the correctness** (but not the complexity) of programs - either directly or indirectly, like in LCF (see [GMW79])
- promotes **insight** regarding concepts underlying programming languages (design with formal semantics in mind)

[1] There is another application that I would have liked to consider in this course, namely the study data-refinement via natural transformations (see [HJ]). However, I did not feel confident enough to present this subject, where work is still in progress.

Failures:

- the denotations of simple expressions, e.g. integer expressions, might have to be **changes**, when the programming language is **extended** (see Page 70 of [Mos89])
- Denotational Semantics is **feasible** for **toy programming languages**, but does **not scale up** easily to **real programming languages** (see Page 106 of [Mos89])
- it is **not** feasible **to re-use** parts of the description of one programming language into another (see Page 106 of [Mos89]).

Mosses stresses as a major shortcoming of Denotational Semantics the lack of **modularity**. One would like to consider various features of programming languages in isolation, so that a study of them is feasible, but then one has to face the problem of putting the peaces together to form complex theories for real programming languages. In Logic it is fairly straightforward to put theories together, and Type Theory has exploited this for studying different type constructions in isolation and then combine them to form complex typed languages, however:

- types are not the only feature of programming languages, actually in simple imperative languages types are the easy part
- there are complex concepts, e.g. that of topological group, that are not obtained by putting together the concepts of group and topological space in the obvious way

There should be various ways of composing theories/concepts, and a study of such **compositions** is of great importance for using a **structured** approach to the developement of concepts.

1.1 Why Category Theory

Category Theory developed mainly from Algebraic Topology through a process of abstraction, and after some of the fundamental notions had been properly formulated it has been applied to various areas of Mathematics (for our applications the concepts used in Categorical Logic, the study of Logic through Category Theory, will be particularly useful). These applications have sometimes required/suggested new fundamental notions.

What does Category Theory offer:

- a **way of thinking** and general guidelines

- an **abstract view** of objects, which hide the internal structure
  - set $\longleftrightarrow$ object
  - function $\longleftrightarrow$ morphism

  and focus on how objects interrelate
- a **syntax independent view** of languages/theories (suggested by Categorical Logic)

What to do for using Category Theory

- **reformulate** intuitive ideas or mathematical concepts in category-theoretic term, e.g. datatype constructors as adjoint functors.
- device **languages**, **formal systems** or **logics** to provide a more friendly interface to technical achievements/concepts (category-theoretic or whatever) and to make them usable by a wider number of people, e.g. lambda calculus as formal system for cartesian closed categories (see [Sco80]).

  In order to give the user some flexibility, it is important to allow extensions of languages and additions of axioms. So formal systems should describe entailment relations rather than specific theories.

In Computer Science, like in Physics, **theories have to be tested**. Category Theory in itself cannot be tested, because it is not about Computer Science. What has to be judged is the way computer science concepts are linked to category-theoretic ones, and what advantages (if any) are obtained by going through Category Theory.

Why not other theories. There are other theories that may take the **meta-role** played by Category Theory: Universal Algebra, Logic, Type Theory. None of them reach the level of abstraction possible in Category Theory, and some of them has not accumulated enough technical tools or general methodologies to cope with a wide range of situations (spanning several fields of mathematics). However, these theories can be better than Category Theory for more specific applications.

The three levels. A concept can be viewed at three different levels:

- **intuitive**, where the concept is usually explained by examples, but has not been formalised or made precised
- **mathematical**, where the concept has been rigorously defined, so that the examples given at the lower level fit into it

- **abstract**, where the mathematical concept has been abstracted from the context in which it was made precise, and therefore can be instanciated into other contexts (not considered before)

The existence of the first two levels does not require much justification. However, the need of an abstract level (and of Category Theory) has not always been recognised in Mathematics, because it looks like delaying the solution of a problem by reformulating it in a more abstract terms. The definition of topological group, exemplifies why one need the concept of group at a more abstract level (group in a category), in order to export it from the context of sets to that of topological spaces.

1.2 Overview of the course

1. A Categorical Manifesto (see [Gog89]) with a textbook on Category Theory (e.g. [Mac71, Gol79, BW85]):

   - basic concepts of Category Theory
   - their manifestations in areas related to Computer Science
   - methodological guidelines

   The paradigm of Categorical Logic (see [KR77]) applied to programming languages

2. Feature 1: notion of computation as monad (see [Mog89b])

   - how the idea developed
   - a justification of why monads
   - a language for monads and tensorial strength
   - how to achieve modularity: monad morphisms and monad constructions
   - Denotational Semantics (see [Sch86]) revisited

3. Feature 2: the distinction types/programs and program modules (see [Mog89a])

   - the type-theoretic explanation and basic ideas ([Mac86, HM88])
   - a critique of the type-theoretic explanation: programming languages as indexed-categories
   - higher order modules

4. Composition of features, the inadequacy of the Categorical Logic paradigm and the need of the **abstract level** (see [Mog89a]): programming languages as objects of a 2-category

- category $\longleftrightarrow$ object
- functor $\longleftrightarrow$ 1-morphism
- natural transformation $\longleftrightarrow$ 2-morphism

Chapter 2

A review of some category-theoretic concepts

- textbooks: first four chapters of [Mac71] or tutorial chapter of [Pie88]
- Category Theory as methodology: [Gog89]
- an overview of Categorical Logic (see [KR77])

2.1 2-categories

In the simplest form of categorical doctrine (see [KR77]), the one for algebraic theories, the identification

- theory as category
- model as functor
- homomorphism as natural transformation

is already unsatisfactory, because we really relate theories to categories with distinguished finite products and models to functors preserving finite products on the nose, and more generally

- theory as category *with additional structure*
- model as *structure preserving* functor
- homomorphism as natural transformation ...

The vagueness of "with additional structure" demand a more *radical* step, which abstracts from the category of small categories (with some additional structures) in the same way as the definition of category abstract from its paradigmatic example: the category of sets.

The most natural outcome of this abstraction is the notion of 2-category, where the category structure is enriched with a notion of *morphism between morphisms*, mimicing the properties of natural transformations. We recall the definitions of 2-category, 2-functor and 2-natural transformation, i.e. the **Cat**-enriched analogue of category, functor and natural transformation (see [Kel82]).

Definition 2.1.1 *A* **2-category** $\mathcal{C}$ *is a* **Cat**-*enriched category, i.e.*

- *a class of objects* $\mathrm{Obj}(\mathcal{C})$
- *for every pair of objects* c_1 *and* c_2 *a category* $\mathcal{C}(c_1, c_2)$
- *for every object* c *an object* $id_c^{\mathcal{C}}$ *of* $\mathcal{C}(c, c)$ *and for every triple of objects* c_1, c_2 *and* c_3 *a functor* $comp_{c_1,c_2,c_3}^{\mathcal{C}}$ *from* $\mathcal{C}(c_1, c_2) \times \mathcal{C}(c_2, c_3)$ *to* $\mathcal{C}(c_1, c_3)$ *satisfying the associativity and identity axioms*
  - $comp(\_, comp(\_, \_)) = comp(comp(\_, \_), \_)$
  - $comp(id, \_) = \_ = comp(\_, id)$

Notation 2.1.2 An object f of $\mathcal{C}(c_1, c_2)$ is called a **1-morphism**, while an arrow σ is called a **2-morphism**. We write $\_;\_$ for $comp(\_, \_)$

$$\begin{tikzcd} c_1 \arrow[r, shift left=3, "f_1"] \arrow[r, shift right=3, "f'_1"'] \arrow[r, phantom, "\downarrow \sigma_1"] & c_2 \arrow[r, shift left=3, "f_2"] \arrow[r, shift right=3, "f'_2"'] \arrow[r, phantom, "\downarrow \sigma_2"] & c_3 & \stackrel{\_;\_}{\longmapsto} & c_1 \arrow[r, shift left=3, "f_1; f_2"] \arrow[r, shift right=3, "f'_1; f'_2"'] \arrow[r, phantom, "\downarrow \sigma_1; \sigma_2"] & c_3 \end{tikzcd}$$

and $\_ \cdot \_$ for composition of 2-morphisms

$$\begin{tikzcd} c_1 \arrow[r, shift left=6, "f_1"] \arrow[r] \arrow[r, shift right=6, "f_2"'] \arrow[r, phantom, shift left=3, "\downarrow \sigma_1"] \arrow[r, phantom, shift right=3, "\downarrow \sigma_2"] & c_2 & \stackrel{\_\cdot\_}{\longmapsto} & c_1 \arrow[r, shift left=3, "f_1"] \arrow[r, shift right=3, "f_2"'] \arrow[r, phantom, "\downarrow \sigma_1 \cdot \sigma_2"] & c_2 \end{tikzcd}$$

Example 2.1.3 The paradigmatic example of 2-category is **Cat** itself (see [Mac71]):

- the objects are categories
- the 1-morphisms are functors and $\_;\_$ is functor composition,
- the 2-morphisms are natural transformations and $\_;\_$ and $\_\cdot\_$ are respectively horizontal and vertical composition of natural transformations.

Definition 2.1.4 *A* **2-functor** *F from $\mathcal{C}_1$ to $\mathcal{C}_2$ is a mapping*

$$
\begin{tikzcd}
c \arrow[r, bend left, "f"] \arrow[r, bend right, "f'"'] \arrow[r, phantom, "\downarrow \sigma"] & c' \text{ in } \mathcal{C}_1
\end{tikzcd}
\quad \stackrel{F}{\longmapsto} \quad
\begin{tikzcd}
Fc \arrow[r, bend left, "Ff"] \arrow[r, bend right, "Ff'"'] \arrow[r, phantom, "\downarrow F\sigma"] & Fc' \text{ in } \mathcal{C}_2
\end{tikzcd}
$$

which commutes with identities, $\_;\_$ and $\_\cdot\_$.

Definition 2.1.5 *If F_1 and F_2 are 2-functors from $\mathcal{C}_1$ to $\mathcal{C}_2$, then a* **2-natural transformation** *τ from F_1 to F_2 is a family $\langle F_1 c \stackrel{\tau_c}{\rightarrow} F_2 c | c \in \mathrm{Obj}(\mathcal{C}_1)\rangle$ of 1-morphisms in $\mathcal{C}_2$ s.t.*

$$
\begin{tikzcd}
c \arrow[d, bend right] \arrow[d, bend left] \arrow[d, phantom, "\stackrel{\sigma}{\rightarrow}"] \\
c'
\end{tikzcd}
\quad \Longrightarrow \quad
\begin{tikzcd}
F_1 c \arrow[r, "\tau_c"] \arrow[d, bend right] \arrow[d, bend left] \arrow[d, phantom, "\stackrel{F_1\sigma}{\rightarrow}"] & F_2 c \arrow[d, bend right] \arrow[d, bend left] \arrow[d, phantom, "\stackrel{F_2\sigma}{\rightarrow}"] \\
F_1 c' \arrow[r, "\tau_{c'}"] & F_2 c'
\end{tikzcd}
$$

i.e. the functors $\tau_c; F_2\_$ and $F_1\_; \tau_{c'}$ from $\mathcal{C}_1(c, c')$ to $\mathcal{C}_2(F_1 c, F_2 c')$ are equal for every c and c' in $\mathrm{Obj}(\mathcal{C}_1)$.

Remark 2.1.6 There are a lot of other concepts related to 2-categories that have no counterpart for categories: modifications, pseudo-functors, lax-functors, ... (see [KS74]). Unfortunately the notation and terminology is far from standard.

The *incorrect* formulation of the paradigm of Categorical Logic was the motivating example for introducing 2-categories, so that it can be reformulated (more abstractly) as:

- theory as object (of a 2-category)
- model as 1-morphism
- homomorphism as 2-morphism

An useful observation is that, in a 2-category we can reformulate abstractly a substantial part of category theory, so that even in such an abstract reformulation we can still rely on category-theoretic concepts. Adjunctions, comma categories, monads, split fibrations are all examples of 2-categorical concepts, i.e. they *make sense* in any 2-category. However, other concepts cannot be captured 2-categorically: opposite categories, adjunctions with parameters (used to define exponentials), dinatural transformations. We consider other two examples where 2-categories are particularly appropriate.

Term rewriting. We have seen that an algebraic language can be viewed as a category, with terms as morphisms and substitution as composition. To consider term rewriting in this framework, one can view a rewriting rule as a morphism from the redex to the contractum, e.g.

$$add0: (0+n) \Longrightarrow n: N \to N$$

$$add1: s(m)+n \Longrightarrow s(m+n): N \times N \to N$$

Exercise 2.1.6.1 *Describe the rewriting of $s(0+n)$ to $s(n)$ as a 2-morphism.*

Intuitively vertical composition corresponds to apply a rewriting to the contractum of another rewriting. Horizontal composition amounts to take a term M in a context $C[\_]$, perform two *independent* rewritings, one of M and the other of $C[\_]$, then take as contractum of $C[M]$ the contractum of $C[\_]$ with the hole replaced by the contractum of M.

Programming languages with type-constructors and polymorphic functions. [HJ] proposes a reinterpretation of the paradigm of Categorical Logic in terms of *data-refinement*, more generally in terms of an *implementation relation* between two semantics for the same programming language:

- programming language as category
- denotational semantics as functor
- refinement as natural transformation

The paper divides a programming language in an inner part, with only basic types and operations whose interpretation has to be supplied by the user, and an outter part, which specify type constructors (as endofunctors) and polymorphic operations (as natural transformations between endofunctors) with a fixed interpretation in every denotational semantics. However, it seems natural to let the user specify also the implementation of type constructors (e.g. *List*) and polymorphic functions (e.g. *reverse*). The paradigm proposed by Hoare should be reformulated as follows:

- programming language as 2-category (with finite products)[1]
- denotational semantics as 2-functor (preserving finite products)
- refinement as *lax-natural transformation*

[1] This allows type constructors with any arity.

Intuitively, in the programming language L as a 2-category 1-morphisms are names for type constructors (with a given arity) and 2-morphisms are names for polymorphic functions. If L has only one object (for simplicity), then a denotation semantics from L to **Cat** should map the only object of L to a category $\mathcal{C}$, 1-morphisms to endofunctors on $\mathcal{C}$, 2-morphisms to natural transformation, by respecting the arities.

Finally, to justify the choice of lax-natural transformations, we have to ask how does the simple view of a language as a category fit in the more general one, of a language as a 2-category, and then check whether the general notion of refinement specialises to natural transformations.

Given a category $Path(G)$ freely generated from a graph G, we can identify it with the 2-category with finite products $Free(G)$ freely generated from an object L, 1-morphisms $a: 1 \to L$ for each node of G and 2-morphisms $f: a \Longrightarrow b$ for each edge from a to b in G.

Exercise 2.1.6.2 *For any graph G, prove that 2-functors $F: Free(G) \to$ **Cat** preserving finite products are in one-one correspondence with functors from $Path(G)$ to some category $\mathcal{C}$ (take $\mathcal{C}$ to be $F(L)$).*

*Given two 2-functors $F_1, F_2: Free(G) \to$ **Cat**, corresponding to functors $F_1': Path(G) \to \mathcal{C}_1$ and $F_2': Path(G) \to \mathcal{C}_2$, prove that lax-natural transformations from F_1 to F_2 are in one-one correspondence with pairs $\langle U, \tau \rangle$, where $U: \mathcal{C}_1 \to \mathcal{C}_2$ is a functor and $\tau: F_1; U \dot{\to} F_2$ is a natural transformation.*

Prove that 2-categories, 2-functors and lax-natural transformations are a 2-category, so the abstract paradigm for programming languages works also in this case.

Chapter 3

Notions of computation as monads

We present a *category-theoretic semantics of computations*. An application of this semantics is the *computational λ-calculus*, a modification of the typed λ-calculus (in particular of $\beta\eta$-conversion), which is **correct** for proving equivalence of programs and **independent** from any specific *notion of computation* (see [Mog89b]). A promising area of application for this semantics is the Denotational Semantics of programming languages, since it suggests new ways to structure such semantics.

We review some criticisms that can be moved to the λ-calculus and the ways it has been used for proving equivalence of programs:

- $\beta\eta$-conversion is not correct w.r.t. operational semantics
- the type constructors of LCF, an extension of the λ-calculus based on denotational semantics, are not the most natural one form the point of view of programming languages (call-by-value or call-by-name). In particular, there are problems in establishing a clear relation between denotational and operational semantics.
- modifications of the λ-calculus based on operational considerations are correct, but they lack a *completeness* result.

3.1 Calculi and programming languages

The *direct connections* between the pure λ-calculus (i.e. the subject of [Bar81]) and functional programming languages sometimes do not go beyond the binding mechanism of λ-abstraction (see [McC62, Lan64]). This is not surprising if we look at the history of the type free lambda calculus (Preface of [Bar81]): "Around 1930 the type free lambda calculus was introduced as a foundation for logic and mathematics. Due to the appearance of paradoxes, this aim was not fulfilled, however. Nevertheless a consistent part of the theory turned out to be quite successful as a theory of computations. It gave an important momentum to early recursion theory and more recently to computer science. ...As a result of these developments. the lambda calculus has grown into **a theory worth studying for its own sake**. People interested in applications may also find the pure λ-calculus useful, since these **applications are usually heuristic rather than direct**. ..."

Types arise in various applications, for instance they are a common *feature* of many programming languages, in logic they can be *identified* with formulae, in (cartesian closed) categories they *correspond* to objects. As shown in [Sco80], "nothing is lost in considering type free theories just as *special parts* of typed theories". For this reason, we give more emphasis to the typed λ-calculus (and more generally typed languages), as it provides a more flexible and structured basis for applications than the pure λ-calculus.

Call-by-value, call-by-name and the λ-calculus

In [Plo75] call-by-value and call-by-name are studied in the setting of the lambda calculus: this study exemplifies very clearly the *mismatch* between the pure λ-calculus and (some) programming languages.

From an operational point of view, a programming language is completely specified by giving the set Prog of programs and the **evaluation mechanism** $\mathrm{Eval}\colon \mathrm{Prog} \rightharpoonup \mathrm{Prog}$, i.e. a partial function mapping every program to its resulting **value** (if any).

> In the setting of the lambda calculus, programs are identified with the (closed) λ-terms (possibly with extra constants, corresponding to some *features* of the programming language). The evaluation mechanism *induces* a congruence relation $\approx$ on λ-terms, called **operational equivalence**, and a *calculus* is said to be **correct** w.r.t. $\approx$ when $M = N$ (derivable in the calculus) implies $M \approx N$.

Plotkin's intention is to study programming languages, therefore he accepts the evaluation mechanism and looks for *the corresponding calculus*.

The λ-calculus is not correct w.r.t. call-by-value operational equivalence, therefore it cannot be used to prove equivalence of programs. Starting from this observation, Plotkin introduces the λ_v-calculus and shows that it is correct w.r.t. call-by-value operational equivalence. However, to prove the Church-Rosser and Standardisation theorems Plotkin proceeds by *analogy* and re-uses some techniques already applied to the λ-calculus. In the case of call-by-name the situation is much simpler, since only the η axiom is not *correct* w.r.t. call-by-name operational equivalence.

The idea of starting from operational considerations, and then develop a calculus has been followed by several people to take account of mor complex features of programming languages like non-determinism, side-effects, control-mechanisms (see [Sha84, FFKD86, MT89]).

The partial λ-calculus

The λ_{p}-calculus is a formal system presented in [Mog86] and (with minor differences) in Chapter 4 of [Ros86], which is **sound and complete** w.r.t. interpretation in *partial cartesian closed categories*.

While the λ_{v}-calculus is *discovered* by operational considerations, and his only criterion for *judging* a calculus is its correctness w.r.t. the operational semantics (but in general there are plenty of correct calculi), the λ_{p}-calculus is *discovered* by model-theoretic considerations, namely as the **unique** calculus which is sound and complete w.r.t. a certain class of models.

The terms of the λ_{p}-calculus are typed λ-terms, and the formal system is for deriving one-sided sequents $x_1: \tau_1, \ldots, x_m: \tau_m.A_1, \ldots, A_n \Longrightarrow A_0$, where $x_1: \tau_1, \ldots, x_m: \tau_m$ is a *type environment* and the A_i are either existence statements ($\mathrm{E}(t)$) or *equivalences* ($t_1 \equiv t_2$). The (type free) λ_{p}-calculus proves more equivalences than the λ_{v}-calculus (e.g. $(\lambda x.x)(yz) \equiv yz$), nevertheless it is still correct w.r.t. call-by-value operational equivalence.

In [Mog88] various modifications of the λ_{p}-calculus are considered that are sound and complete w.r.t. other classes of models, e.g. classical type structures of (monotonic) partial functionals.

Logics for computable functions and domain theory

The use of logics for proving correctness of programs was pioneered in [Flo67, Hoa69]. The programming languages considered in these early approaches are very simple (while programs) and the formal languages of expressing properties of programs are a mixture of programming language and predicate calculus (Hoare's triples). The main feature of these logics, e.g. **Hoare's logic**, is that they are **programming language dependent** and support a **structured methodology** for proving correctness, i.e. properties are proved by induction on the structure of programs.

We are interested in a different kind of approach, based on logics for reasoning about mathematical structures that provide models for (functional) programming languages. Such a logic is **programming language independent**, but potentially it can be used to prove properties of programs (like equivalence or correctness). To exploit this potentiality the logic has to be **flexible**, so that one can describe how a particular programming language is interpreted in a certain mathematical structure. More precisely, its language should be extendible (to include the features of a programming language), it should be possible to add (non-logical) axioms (to axiomatise the properties of these features) and the logic itself should be uncommitted (so that any programming language can be *axiomatised*).

The need for a mathematical semantics of programming languages led to the Scott-Strachey approach to denotational semantics and the development of **domain theory** (see [Sco70, SS71, Sco76, Plo81]). Denotational semantics takes an abstract view of *computations*, based on **continuous functions** between certain *spaces* of *partial elements* (**cpos**), rather than partial recursive functions (on the natural numbers).

A first attempt at a logic for reasoning about cpos and continuous functions was the **logic for computable functions** (see [Sco69]), which was subsequently extended to the **polymorphic predicate λ-calculus** $PP\lambda$ (see [GMW79, Pau85]). The idea is first to consider an extension of the typed λ-calculus suitable for the (cartesian closed) category of cpos and continuous functions (the *intended model*), by adding constants for *fixed-point operators* and *least elements*, and then to axiomatise *some* properties of the intended model, like fixed point induction, in a (classical) first order logic, whose formulae are built up from inequations between λ-terms ($M \leq N$).

In $PP\lambda$ programs are *identified* by λ-terms and properties (of a programs) are ordinary first order formulae. As pointed out at Page 9 of [Sco69], $PP\lambda$ (as well as the logic of partial elements) is a theory of partial functions based on total functions: "...classical type theory supposes **total** (everywhere defined) functions, while algorithms in general produce **partial** functions. We do not wish to reject a program if the function defined is partial – because as everyone knows it is not possible to predict which programs will *loop* and which will define total functions. The solution to this problem of total vs. partial functions is to make a *mathematical model* for the theory of partial functions using ordinary total functions." More precisely, *divergence* is represented by the least element ($\perp$) of a cpo and a partial (recursive) function $f{:}\,\mathbf{N} \rightharpoonup \mathbf{N}$ becomes a *strict* function $f_\perp{:}\,\mathbf{N}_\perp \to \mathbf{N}_\perp$.

There are various criticisms that can be made to $PP\lambda$ (and domain theory), more or less related to the way partial functions are represented:

- while programs (and similar programming languages) have a simple set-theoretic semantics as partial functions from stores to stores, but their denotational semantics (based on cpos and continuous functions) is *comparatively clumsy*.
- the way of expressing *termination* (*existence*) of a program (partial element) t is *indirect* "$t \neq \perp$" and intuitionistically incorrect, according to [Sco79].
- the (choice and) interpretation of type constructors is based on the typed λ-calculus (i.e. a theory of total functions), rather than considerations about (real) programming languages. For instance, in a *lazy* programming language the *values* of type $\tau_1 \times \tau_2$ are pairs of *unevaluated expressions* and $[\![\tau_1 \times \tau_2]\!]$ should be the *lifted product* $([\![\tau_1]\!] \times [\![\tau_2]\!])_\perp$, while in a *eager* programming language the *values* of type $\tau_1 \times \tau_2$ are pairs of *values* and $[\![\tau_1 \times \tau_2]\!]$ should be the *smash product* $[\![\tau_1]\!] \otimes [\![\tau_2]\!]$.

These considerations on domain theory and $PP\lambda$ (among others) led Gordon Plotkin to take a different approach to denotational semantics, based on **con-**

tinuous partial functions, and to reformulate $PP\lambda$ using intuitionistic logic of partial elements on top of a suitable term language, called *metalanguage* (see [Plo85]).

The metalanguage is a typed functional language with a rich collection of types (products, coproducts, partial function spaces and recursive types), it has an operational semantics, based on eager evaluation, and a denotational semantics, according to which types denote cpos (possibly without a least element) and terms denote continuous partial functions. The two semantics are related by a correspondence theorem between (operational) termination and (denotational) existence, inspired by a similar result due to Martin-Löf (see [ML83] and Chapter 6 of [Abr87]).

The choice of eager evaluation makes it easier to translate programming languages into the metalanguage *correctly*, i.e. so that the operational and denotational semantics obtained via the translation are the *intended* ones. In fact, it is straightforward to translate lazy programming languages into typed eager programming languages, while the reverse translation is in general impossible (or at least very clumsy).

Since many programming languages can be *correctly* translated into the metalanguage, it is not necessary to *extend* the metalanguage for *accommodating* them, and a logic for reasoning about them can be *derived* from that for the metalanguage, while in $PP\lambda$ it has to be *axiomatised*, by creating a new theory. For the same reason, the correspondence theorem between operational and denotational semantics, unlike similar results for the λ-calculus (e.g. [Wad76]), has **direct** application to programming languages.

3.2 Notions of computation as monads

It is important to clarify what is the intuitive meaning assigned to "notions of computation", since the word computation is rather overloaded. We consider a computation as the **denotation of a program**, while a more fine grained (operational approach) considers a computation as a **possible execution sequence for a program**. In the latter case the attribute "possible" is necessary to account for non-deterministic or probabilistic programs.

We give a categorical semantics of computations, but also this claim requires some explanation, since a categorical semantics can be given at different levels of abstractions:

1. execution sequences can be viewed as (enriched) categories, where objects are events and morphisms are relations between them, (like causality or a delay, (see V. Pratt).

2. programs can be viewed as categories (with an initial state), where objects are states (or histories) and morphisms are transitions (or pieces of execution

sequences). This approach corresponds to transition systems.

3. programs or execution sequences can be viewed as objects of a category, whose morphisms can be, for instance, mappings from the events of an execution sequence s_1 to events in another execution sequence s_2 satisfying certain properties. This approach has been applied to define categorically, e.g. as products or coproducts, some operations on programs or execution sequences (see V. Pratt, J. Meseguer, ...).

4. finally programs can be viewed as morphisms of a category, whose objects are types. This is the view taken by Denotational Semantics.

We take the denotational view, and consider programs as morphisms in a category. To be precise a *complete program* should be identified with an element, i.e. a morphism with domain the terminal object, while a *program in an environment* corresponds to a morphism.

By "notion of computation" we mean a **qualitative** description of the denotations of programs in a certain programming language, rather than the interpretation function itself. Examples of notions of computations are:

1. computations with side effects, where a program denotes a map from a store to a pair, value and modified store.

2. computations with exceptions, where a program denotes either a value or an exception.

3. partial computations, where a program denotes either a value or diverges.

4. nondeterministic computations, where a program denotes a set of possible values.

3.2.1 A justification for monads

In order to justify the use of monads for modelling notions of computations we adopt the following *intuitive* understanding of programs: **a program is a function from values to computations**.

1. First we take a category $\mathcal{C}$ as a model for functions and develop on top a general understanding of values and computations. More precisely we introduce a unary operation T on the objects of $\mathcal{C}$, which map an object A, viewed as the *set of values of type* τ, to an object TA corresponding to the *set of computations of type* τ.

2. Then a program from A to B, i.e. which takes as input a value of type A and after performing a certain computation will return a value of type B, can be identified with a morphism from A to TB in $\mathcal{C}$.

3. Finally, we identify a minimum set of requirements on values and computations so that programs are the morphisms of a suitable category.

These three steps lead to Kleisli triples for modelling notions of computation and Kleisli categories for modelling categories of programs.

Definition 3.2.1 *A* **Kleisli triple** *over* $\mathcal{C}$ *is a triple* $(T, \eta, \_^*)$*, where* $T\colon \mathrm{Obj}(\mathcal{C}) \to \mathrm{Obj}(\mathcal{C})$*,* $\eta_A\colon A \to TA$*,* $f^*\colon TA \to TB$ *for* $f\colon A \to TB$ *and the following equations hold:*

- $\eta_A^* = \mathrm{id}_{TA}$
- $\eta_A; f^* = f$
- $f^*; g^* = (f; g^*)^*$

Remark 3.2.2 Intuitively η_A is the *inclusion* of values into computations and f^* is the *extension* of a function f from values to computations to a function from computations to computations, which first evaluates a computation and then applies f to the resulting value. Although it seems natural to require that η_A is a mono, we will not do so, since it causes several technical problems.

The axioms for Kleisli triples amounts exactly to say that programs form a category, the **Kleisli category** $\mathcal{C}_T$, where the set $\mathcal{C}_T(A, B)$ of morphisms from A to B is $\mathcal{C}(A, TB)$, the identity over A is η_A and composition of f followed by g is $f; g^*$. Intuitively $f; g^*$ takes a value a and applies f to produce a computation fa, then it *executes/evaluates* the computation fa to get a value b, and finally it applies g to b to produce a computation.

Exercise 3.2.2.1 *Define suitable Kleisli triples (over the category of sets) to model the following notions of computations:*

- *total computations:* $TA = A$
- *partial computations:* $TA = A_\perp$
- *non-deterministic computations:* $TA = \mathcal{P}(A)$
- *computations with side-effects:* $TA = (A \times S)^S$*, where* S *is a set of states/stores*
- *computations with exceptions:* $TA = A + E$*, where* E *is a set of exceptions*
- *computations with continuations:* $TA = R^{R^A}$*, where* R *is a set of possible results*

- *computations for communicating processes:*

$$TA = \mu P.A + 1 + (Act \times P) + (P \times P)$$

  where Act is a set of actions (with an involution operation $\bar{\alpha}$ and a silent action τ, to model communication). Intuitively an element of P, called process, can be either a value (in CCS there are no values), or Nil (which represent deadlock), or an action followed by a process, or a non-deterministic choice between two processes

- *computations with complexity: $TA = (A \times N)$, i.e. a program is interpreted by the a pair giving the final result and the time required to compute it*

What kind of Kleisli triples would you use to model partial computations with side-effects or computations with side-effects and exceptions? Try also other combinations.

For adjunctions we saw that there are several equivalent definitions, but only one of them is phrased in such that can be easily generalised to arbitrary 2-categories. For this reason we introduce monads (called also triples), which are equivalent to Kleisli triples but are defined only in terms of functors and natural transformations.

Definition 3.2.3 *A* **monad** *over $\mathcal{C}$ is a triple (T, η, μ) s.t.*

$$\begin{tikzcd} \mathcal{C} \arrow[d, bend right, "id_{\mathcal{C}}"'] \arrow[d, bend left, "T"] \arrow[d, phantom, "\xrightarrow{\eta}"] \\ \mathcal{C} \end{tikzcd} \qquad \begin{tikzcd} \mathcal{C} \arrow[d, bend right, "T;T"'] \arrow[d, bend left, "T"] \arrow[d, phantom, "\xrightarrow{\mu}"] \\ \mathcal{C} \end{tikzcd}$$

$(T;\mu) \cdot \mu = (\mu; T) \cdot \mu$ *and* $(T;\eta) \cdot \mu = \mathrm{id}_T = (\eta; T) \cdot \mu$.

Proposition 3.2.4 *There is a one-one correspondence between Kleisli triples and monads.*

Proof Given a Kleisli triple $(T, \eta, \_^*)$, the corresponding monad is (T, η, μ), where T is the extension of the function T to an endofunctor by taking $T(f\colon A \to B) = (f; \eta_B)^*$ and $\mu_A = \mathrm{id}^*_{TA}$.

Conversely, given a monad (T, η, μ), the corresponding Kleisli triple is $(T, \eta, \_^*)$, where T is the restriction of the functor T to objects and $(f\colon A \to TB)^* = Tf; \mu_B$. ∎

A general theorem about monads says that they are all induced by adjunctions, namely a monad over $\mathcal{C}$ is always of the form $(F; G, \eta, F; \epsilon; G)$, where (F, G, η, ϵ) is an adjunction from $\mathcal{C}$ to some other category $\mathcal{D}$. Actually there are two general constructions, that given a monad over $\mathcal{C}$ return an adjunction inducing that monad. We will use only the construction due to Kleisli; however, the Eilenberg-Moore construction as well as additional information on monads can be found in most textbooks on Category Theory (see [Mac71, BW85]).

Proposition 3.2.5 *Given a monad (T, η, μ) over $\mathcal{C}$, or equivalently a Kleisli triple, the quadruple $(F_T, G_T, \eta, \epsilon)$, defined below, is an adjunction from $\mathcal{C}$ to the Kleisli category $\mathcal{C}_T$ which induces the monad (T, η, μ):*

- *$F_T: \mathcal{C} \to \mathcal{C}_T$ is the functor mapping A to A and $f: A \to B$ to $f; \eta_B$*
- *$G_T: \mathcal{C}_T \to \mathcal{C}$ is the functor mapping A to TA and $f \in \mathcal{C}_T(A, B)$, i.e. $f: A \to TB$ in $\mathcal{C}$, to f^**
- *$\eta: \mathrm{Id}_C \dot{\to} F_T; G_T$ is η of the Kleisli triple, since $T = F_T; G_T$*
- *$\epsilon: G_T; F_T \dot{\to} \mathrm{Id}_{\mathcal{C}_T}$ is the natural transformation s.t. $\epsilon_A = \mathrm{id}_{TA} \in \mathcal{C}_T(TA, A)$*

3.2.2 A simple metalanguage

One may consider formal systems (for reasoning about computations) motivated by two different objectives: reasoning about programming languages and reasoning about programs (in a fixed programming language).

When reasoning about programming languages one has different monads over the same category $\mathcal{C}$ (for simplicity), one for each programming language, and the main aim is to study how they relate to each other. So it is natural to base a formal system on a *metalanguage* for $\mathcal{C}$, where monads are treated as unary type-constructors.

When reasoning about programs one has only one monad T, because the programming language is fixed, and the main aim is to prove properties of programs. In this case the obvious choice for the term language (of a formal system) is the *programming language* itself, i.e. a language for $\mathcal{C}_T$. However, the *expressiveness* of the programming language is adequate, only if the monad T satisfies the mono requirement, because then it is possible to express equality of values in terms of equality of computations and an unary predicate over computations (similar to the existence predicate in the logic of partial elements), whose extension is the set of values.

In this section we take the first approach, while [Mog89b] takes the second one. We introduce a metalanguage, whose terms denote morphisms in a category $\mathcal{C}$. The aim of this section is to focus only on the crucial ideas of the interpretation, and the language has been oversimplified (for instance terms have exactly one free variable and the only assertions are equations) in order to define its interpretation in any category with a monad (T, η, μ).

The metalanguage is parametric in a signature (i.e. a set of base types and unary function symbols), therefore its interpretation in a category with a monad is parametric in an interpretation of the symbols in the signature.

- Given an interpretation $[\![A]\!]$ for any base type A, i.e. an object of $\mathcal{C}$, then the interpretation of a type $\tau ::= A \mid T\tau$ is an object $[\![\tau]\!]$ of $\mathcal{C}$ defined in the obvious way, namely $[\![T\tau]\!] = T[\![\tau]\!]$.

- Given an interpretation $[\![f]\!]$ for any unary function f of arity $\tau_1 \to \tau_2$, i.e. a morphism from $[\![\tau_1]\!]$ to $[\![\tau_2]\!]$ in $\mathcal{C}$, then the interpretation of a well-formed term $x{:}\,\tau \vdash e{:}\,\tau'$ is a morphism $[\![x{:}\,\tau \vdash e{:}\,\tau']\!]$ from $[\![\tau]\!]$ to $[\![\tau']\!]$ in $\mathcal{C}$ defined by induction on the derivation of $x{:}\,\tau \vdash e{:}\,\tau'$ (see Table 3.1).

- On top of the term language we consider equations, whose interpretation is as usual (see Table 3.2).

Remark 3.2.6 The let-constructor is very important semantically, since it corresponds to composition in the Kleisli category $\mathcal{C}_T$. Moreover, $(\mathrm{let}_T\, x{=}e \,\mathrm{in}\, e')$ cannot be *reduced* to the more basic substitution (i.e. $e'[x{:}=e]$), which corresponds to composition in $\mathcal{C}$, without collapsing $\mathcal{C}_T$ to $\mathcal{C}$.

In the λ-calculus the let-constructor is usually treated as syntactic sugar for $(\lambda x.e')e$, and this can be done also in the λ_c-calculus, but it relies on function spaces, that are not more primitive than a notion of computation.

3.2.3 Extending the metalanguage

The metalanguage introduced in the previous section is far too simple to do anything useful. We discuss the additional structure require on the category $\mathcal{C}$ in order to interpret also algebraic terms. Other extensions of the metalanguage do not present problems, e.g. it is straightforward to incorporate functional types, sums and products. The next critical extension is the introduction of dependent types (that we will not discuss).

The standard requirement on a category $\mathcal{C}$ for interpreting algebraic terms is that it must have finite products, so that the interpretation of a function symbol of arity $\overline{\tau} \to \tau$ is a morphism from $[\![\times(\overline{\tau})]\!]$ (i.e. $[\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]$) to $[\![\tau]\!]$ and similarly the interpretation of a well-formed term $x_1{:}\,\tau_1, \ldots, x_n{:}\,\tau_n \vdash e{:}\,\tau$ is a morphism from $[\![\times(\overline{\tau})]\!]$ to $[\![\tau]\!]$. However, products are not enough, to interpret $(\mathrm{let}_T\, x{=}e \,\mathrm{in}\, e')$, when e' has other free variables beside x (see [Mog89b]). The additional structure we need is given by a natural transformation which relates products and monad.

Definition 3.2.7 *A* **strong monad** *over a category $\mathcal{C}$ with finite products is a monad (T, η, μ) together with a natural transformation $\mathrm{t}_{A,B}$ from $A \times TB$ to $T(A \times B)$ s.t.*

$$\begin{tikzcd}
1 \times TA \arrow[r, "\mathrm{t}_{1,A}"] \arrow[dr, "r_{TA}"'] & T(1 \times A) \arrow[d, "Tr_A"] \\
 & TA
\end{tikzcd}$$

RULE	SYNTAX		SEMANTICS
var	$\dfrac{}{x{:}\,\tau \vdash x{:}\,\tau}$	$=$	$\mathrm{id}_{[\![\tau]\!]}$
let	$x{:}\,\tau \vdash e_1{:}\,T\tau_1$	$=$	g_1
	$x_1{:}\,\tau_1 \vdash e_2{:}\,T\tau_2$	$=$	g_2
	$x{:}\,\tau \vdash (\mathrm{let}_T\, x_1{=}e_1 \,\mathrm{in}\, e_2){:}\,T\tau_2$	$=$	$g_1; g_2{}^*$
$\mathrm{f}{:}\,\tau_1 \to \tau_2$	$x{:}\,\tau \vdash e_1{:}\,\tau_1$	$=$	g_1
	$x{:}\,\tau \vdash \mathrm{f}(e_1){:}\,\tau_2$	$=$	$g_1; [\![\mathrm{f}]\!]$
$[\_]_{\mathrm{T}}$	$x{:}\,\tau \vdash e{:}\,\tau'$	$=$	g
	$x{:}\,\tau \vdash [e]_T{:}\,T\tau'$	$=$	$g; \eta_{[\![\tau']\!]}$

Table 3.1: Terms and their interpretation

RULE	SYNTAX		SEMANTICS
eq	$x{:}\,\tau_1 \vdash e_1{:}\,\tau_2$	$=$	g_1
	$x{:}\,\tau_1 \vdash e_2{:}\,\tau_2$	$=$	g_2
	$x{:}\,\tau_1 \vdash e_1 = e_2{:}\,\tau_2$	$\iff$	$g_1 = g_2$

Table 3.2: equations and their interpretation

$$
\begin{tikzcd}
(A\times B)\times TC \arrow[rr, "\mathrm{t}_{A\times B,C}"] \arrow[d, "\alpha_{A,B,TC}"'] & & T((A\times B)\times C) \arrow[d, "T\alpha_{A,B,C}"] \\
A\times(B\times TC) \arrow[r, "\mathrm{id}_A\times \mathrm{t}_{B,C}"] & A\times T(B\times C) \arrow[r, "\mathrm{t}_{A,B\times C}"] & T(A\times(B\times C))
\end{tikzcd}
$$

and which satisfies also the following diagrams:

$$
\begin{tikzcd}
A\times B \arrow[rr, "\mathrm{id}_{A\times B}"] \arrow[d, "\mathrm{id}_A\times\eta_B"'] & & A\times B \arrow[d, "\eta_{A\times B}"] \\
A\times TB \arrow[rr, "\mathrm{t}_{A,B}"] & & T(A\times B) \\
A\times T^2B \arrow[u, "\mathrm{id}_A\times\mu_B"] \arrow[r, "\mathrm{t}_{A,TB}"] & T(A\times TB) \arrow[r, "T\mathrm{t}_{A,B}"] & T^2(A\times B) \arrow[u, "\mu_{A\times B}"']
\end{tikzcd}
$$

where r and α are the natural isomorphisms

- $r_A: 1\times A \to A$
- $\alpha_{A,B,C}: (A\times B)\times C \to A\times(B\times C)$

The tensorial strength t induces a natural transformation $\psi_{A,B}$ from $TA\times TB$ to $T(A\times B)$, namely

$$\psi_{A,B} = c_{TA,TB}; \mathrm{t}_{TB,A}; (c_{TB,A}; \mathrm{t}_{A,B})^*$$

where c is the natural isomorphism

- $c_{A,B}: A\times B \to B\times A$

The morphism $\psi_{A,B}$ has the correct domain and codomain to interpret the pairing of a computation of type A with one of type B (obtained by first evaluating the first argument and then the second). There is also a dual notion of pairing, $\tilde{\psi}_{A,B} = c_{A,B}; \psi_{B,A}; Tc_{B,A}$ (see [Koc72]), which amounts to first evaluating the second argument and then the first. With this additional structure available we can interpret algebraic terms, in particular the let-constructor (see Table 3.3).

At this point it is easy to give a sound and complete formal system for proving equality of algebraic terms interpreted in a strong monad. The inference rules for such a formal system can be partitioned in two (see Table 3.4):

- general rules of equational logic
- the inference rules for let-constructor and types of computations

RULE	SYNTAX		SEMANTICS
let			
	$\Gamma \vdash e_1 : T\tau_1$	$=$	g_1
	$\Gamma, x_1 : \tau_1 \vdash e_2 : T\tau_2$	$=$	g_2
	$\Gamma \vdash (\mathrm{let}_T\, x_1{=}e_1\, \mathrm{in}\, e_2) : T\tau_2$	$=$	$\langle \mathrm{id}_{[\![\Gamma]\!]}, g_1\rangle ; \mathrm{t}_{[\![\Gamma]\!],[\![\tau_1]\!]} ; g_2^*$

Table 3.3: Interpretation of let

We write $\_[x := e]$ for the substitution of x with e in $\_$ and $(\mathrm{let}_T\, \overline{x}{=}\overline{e}\, \mathrm{in}\, e)$ for $(\mathrm{let}_T\, x_1{=}e_1\, \mathrm{in}\, (\ldots (\mathrm{let}_T\, x_n{=}e_n\, \mathrm{in}\, e)\ldots))$, where n is the lenght of the sequence $\overline{x}$ (and $\overline{e}$). In particular, $(\mathrm{let}_T\, \emptyset{=}\emptyset\, \mathrm{in}\, e)$ stands for e.

$$\text{subst}\ \frac{\Gamma \vdash e : \tau \qquad \Gamma, x : \tau \vdash A}{\Gamma \vdash A[x := e]}$$

$=$ is an congruence relation

$$\text{ass}\ \ \Gamma \vdash (\mathrm{let}_T\, x_2{=}(\mathrm{let}_T\, x_1{=}e_1\, \mathrm{in}\, e_2)\, \mathrm{in}\, e) = (\mathrm{let}_T\, x_1{=}e_1\, \mathrm{in}\, (\mathrm{let}_T\, x_2{=}e_2\, \mathrm{in}\, e)) : T\tau$$

provided x_1 is not free in e

$$\text{let.}\beta\ \ \Gamma \vdash (\mathrm{let}_T\, x_1{=}[x_2]_T\, \mathrm{in}\, e) = e[x_1 := x_2] : T\tau$$

$$\text{T.}\eta\ \ \Gamma \vdash (\mathrm{let}_T\, x{=}e\, \mathrm{in}\, [x]_T) = e : T\tau$$

Table 3.4: Inference rules

Chapter 4

Denotational Semantics and monads

We have identified a precise abstract concept corresponding to a notion of computation, now we want to consider possible applications to the Denotational Semantics of programming languages. In the introduction we pointed out that a major shortcoming of Denotational Semantics is the lack of modularity, which prevent *gluing together* semantics for toy languages into a semantics for a complex language. Having an abstract notion of computation, rather than a bunch of examples for it, will provide the key step towards modularity, namely *parametrisation* w.r.t. a notion of computation.

To give semantics to a complex language L we propose a stepwise approach, which starts from a monad (notion of computation) corresponding to a toy sublanguage of L and then at each step applies a *monad constructor* which add one feature to the language. To make this approach precise we introduce a category $\mathrm{Mon}(\mathcal{C})$, whose objects are monads (over a given category $\mathcal{C}$), and identify monad constructors with endofunctors over $\mathrm{Mon}(\mathcal{C})$.

Definition 4.0.8 *Given two monads* (T, η^T, μ^T) *over* $\mathcal{C}$ *and* (S, η^S, μ^S) *over* $\mathcal{D}$, *a* **monad-morphism** *from the first to the second monad is a pair* (U, σ), *where* $U\colon \mathcal{C} \to \mathcal{D}$ *is a functor and* $\sigma\colon T; U \dot{\to} U; S$ *is a natural transformation s.t. :*

$$\begin{tikzcd}
UA \arrow[r, "U\eta^T_A"] \arrow[dr, "\eta^S_{UA}"'] & U(TA) \arrow[d, "\sigma_A"'] & U(T^2A) \arrow[l, "U\mu^T_A"'] \arrow[d, "\sigma_{TA}"] \\
 & S(UA) & S(U(TA)) \arrow[d, "S\sigma_A"] \\
 & & S^2(UA) \arrow[ul, "\mu^S_{UA}"]
\end{tikzcd}$$

Remark 4.0.9 We said that a programming language with (unary) type constructors and polymorphic operations can be viewed as a freely generated 2-category, and that lax natural transformations are the appropriate morphisms between models of such a language. If we take the free 2-category $L(\mathrm{Mon})$ with one monad, then there is a one-one correspondence between monads in **Cat** and 2-functors from $L(\mathrm{Mon})$ to **Cat**, this extend to a one-one correspondence between monad morphisms and lax natural transformations between the corresponding 2-functors from $L(\mathrm{Mon})$ to **Cat**.

This correspondence is very useful as guideline for finding the *correct* notion of morphism for monads with additional structure. For instance, when T is the monad for non-determinism, one can define a natural transformation $or_A: TA \times TA \to TA$, corresponding to non-deterministic choice. Then one can define the language for non-determinism as the free 2-category with one monad and a 2-morphism *or* satisfying certain equations.

As pointed out before, this 2-categorical view of programming languages can model only types and operations that are *covariant* and *natural*, so functional types and evaluation are outside its scope. Furthermore, it is arguable whether the component of the natural transformation σ should be morphisms in $\mathcal{D}$ or something different, e.g. relations in $\mathcal{D}$. What follows should be treated as a general methodology for having modularity in Denotational Semantics, with the warning that some technical details have been simplified, in particular the definition of monad morphism may not be a satisfactory formalisation of what is a relation between notions of computation (programming language semantics).

Monad morphisms not only give a functor U from $\mathcal{C}$ to $\mathcal{D}$, they also induce a functor U_* from $\mathcal{C}_T$ to $\mathcal{D}_S$, which *extends* U from functions to programs.

Proposition 4.0.10 *There is a one-one correspondence between monad morphisms (U, σ) from (T, η^T, μ^T) over $\mathcal{C}$ to (S, η^S, μ^S) over $\mathcal{D}$ and pairs of functors (U, U_*) s.t. the following diagram commutes*

$$\begin{tikzcd} \mathcal{C} \arrow[r, "U"] \arrow[d, "F_T"'] & \mathcal{D} \arrow[d, "F_S"] \\ \mathcal{C}_T \arrow[r, "U_*"'] & \mathcal{D}_S \end{tikzcd}$$

where F_T and F_S are the left adjoints given by the Kleisli construction.

Proof We sketch only the correspondence:

- given a monad morphism (U, σ), we define $U_*: \mathcal{C}_T \to \mathcal{D}_S$ as the functor mapping A to UA and $f \in \mathcal{C}_T(A, B)$, i.e. $f: A \to TB$ in $\mathcal{C}$, to $(Uf); \sigma_B$. We claim that the pair (U, U_*) satisfies the diagram, in fact

$$\begin{tikzcd}
f: A \to B \arrow[r, "U"] \arrow[d, "F_T"'] & Uf \arrow[d, "F_S"] \\
f; \eta^T_B \arrow[r, "U_*"'] & U(f; \eta^T_B); \sigma_B = (Uf); \eta^S_{UB}
\end{tikzcd}$$

where the equation follows from the axiom $U(\eta^T_B); \sigma_B = \eta^S_{UB}$ for monad morphisms.

- given a pair (U, U_*) satisfies the diagram, we define $\sigma: T; U \dot{\to} U; S$ as the natural transformation s.t. $\sigma_A = U_*(\mathrm{id}_{TA})$, which is a morphism from $U(TA)$ to $S(UA)$ in $\mathcal{D}$.

∎

The definition of monad morphism given above is very general. Since we are interested only in relating monads or strong monads over some fixed category $\mathcal{C}$, we simplify the definition by requiring U to be the identity functor on $\mathcal{C}$ (as done in [BW85]).

Definition 4.0.11 *Given two strong monads $(T, \eta^T, \mu^T, \mathrm{t}^T)$ and $(S, \eta^S, \mu^S, \mathrm{t}^S)$ over the same category $\mathcal{C}$, a* **simplified strong monad morphism** *from the first to the second monad is a natural transformation $\sigma: T \dot{\to} S$ s.t. :*

$$\begin{tikzcd}
A \arrow[r, "\eta^T_A"] \arrow[d, "\mathrm{id}_A"'] & TA \arrow[d, "\sigma_A"'] & T^2A \arrow[l, "\mu^T_A"'] \arrow[d, "(\sigma;\sigma)_A"] \\
A \arrow[r, "\eta^S_A"'] & SA & S^2A \arrow[l, "\mu^S_A"]
\end{tikzcd}
\qquad
\begin{tikzcd}
A \times TB \arrow[r, "\mathrm{t}^T_{A,B}"] \arrow[d, "\mathrm{id}_A \times \sigma_B"'] & T(A \times B) \arrow[d, "\sigma_{A \times B}"] \\
A \times SB \arrow[r, "\mathrm{t}^S_{A,B}"'] & S(A \times B)
\end{tikzcd}$$

4.1 Some examples of monad constructions

In this section we fix a category $\mathcal{C}$ with enough datatypes, e.g. the category of sets or cpos, and consider the category $\mathrm{Mon}(\mathcal{C})$ of (strong) monads over $\mathcal{C}$ and simplified (strong) monad morphisms. We will define several endofunctions F on $\mathrm{Obj}(\mathrm{Mon}(\mathcal{C}))$, by abstracting from domain-theoretic constructions for giving semantics to programming languages (see [Sch86, Mos89]) and address the following questions:

- can F be extended to an endofunctor F over $\mathrm{Mon}(\mathcal{C})$?

- is there a natural transformation from $\mathrm{Id}_{\mathrm{Mon}(\mathcal{C})}$ to F? Which intuitively means that every T-computation can be viewed as an $F(T)$-computation via a monad morphism from T to $F(T)$. A monad constructor F is expected to have such property, when its action on T is to *add* some features.

- If associated to T there are some additional operations, i.e. natural transformations, is there a way of lifting them to $F(T)$?

Remark 4.1.1 The first two questions come together, since if we were not interested in establishing relations between monads, but only in constructing them, then endofunctions would be completely satisfactory. On the other hand, to establish relations among monads it is important to have canonical ways of lifting a relation between T and S to a relation between $F(S)$ and $F(T)$.

The last question is of paramount importance for a modular approach to Denotational Semantics. For instance, if we have the monad T of exceptions with additional operations for *raising* and *handling* exceptions, then when we add side-effects (by applying a monad constructor F), we would like to derive in a *canonical* way the operations for *raising* and *handling* exceptions in $F(T)$ from those defined over T, as well as introducing in a *canonical* way the operations for manipulating stores. There is a related question that we have not been able to address properly, yet. If an operation op' in $F(T)$ is derived canonically from an operation op in T, what axioms can be inferred for op' given a set of axioms satisfied by op?

We introduce some general terminology for *operations* associated to a monad, that will be used to discuss operations associated to the monad constructor for exceptions as well as lifting of operations from T-computations to T-computations with additional features.

Notation 4.1.2 We fix a countable set of type variables Γ.

- The metavariable τ range over the following sets of type expressions:

$$\tau \in Type ::= 1 \mid A \mid \gamma \mid T(\tau) \mid \tau^A \mid \tau_1 \times \tau_2$$

  where the symbols A are objects of $\mathcal{C}$, γ is a variable in Γ and T is an uninterpreted type constructor.

- Given an interpretation for T, i.e. an endofunctor S over $\mathcal{C}$, the type expression τ will denote a functor $[\![\tau]\!]^S$ from $\mathcal{C}^\Gamma$ to $\mathcal{C}$. Moreover, a natural transformation σ from S to S' induces a natural transformation σ_τ from $[\![\tau]\!]^S$ to $[\![\tau]\!]^{S'}$.

- An **operation** op of arity $\tau_1 \dot{\to} \tau_2$ on the endofunctor S is a natural transformation from $[\![\tau_1]\!]^S$ to $[\![\tau_2]\!]^S$.

- Given an operation op of arity $\tau_1 \dot{\to} \tau_2$ on S and a natural transformation $\sigma: S \dot{\to} S'$, we say that an operation op' of arity $\tau_1 \dot{\to} \tau_2$ on S' is a **lifting** of op along σ iff the following diagram commutes:

$$\begin{tikzcd}
[\![\tau_1]\!]^S \arrow[r, "op"] \arrow[d, "\sigma_{\tau_1}"'] & [\![\tau_2]\!]^S \arrow[d, "\sigma_{\tau_2}"] \\
[\![\tau_1]\!]^{S'} \arrow[r, "op'"] & [\![\tau_2]\!]^{S'}
\end{tikzcd}$$

- Given an endofunctor F on $\mathrm{Mon}(\mathcal{C})$ and for every S an operation op^S of arity $\tau_1 \dot{\to} \tau_2$ on $F(S)$, we say that the operation constructor op is **natural** iff for every monad morphisms $\sigma: S \to S'$ the following diagram commutes:

$$\begin{tikzcd}
[\![\tau_1]\!]^{F(S)} \arrow[r, "op^S"] \arrow[d, "(F\sigma)_{\tau_1}"'] & [\![\tau_2]\!]^{F(S)} \arrow[d, "(F\sigma)_{\tau_2}"] \\
[\![\tau_1]\!]^{F(S')} \arrow[r, "op^{S'}"] & [\![\tau_2]\!]^{F(S')}
\end{tikzcd}$$

- Given an endofunctor F on $\mathrm{Mon}(\mathcal{C})$ and a natural transformation $\eta^F: \mathrm{Id}_{\mathrm{Mon}(\mathcal{C})} \dot{\to} F$, we say that $\_^F$ is a **natural lifting** of operations of arity $\tau_1 \dot{\to} \tau_2$ along η^F iff for every monad S and operation $op: \tau_1 \dot{\to} \tau_2$ on S op^F is a lifting of op along η^F_S and

$$\begin{tikzcd}
[\![\tau_1]\!]^S \arrow[r, "op"] \arrow[d, "\sigma_{\tau_1}"'] & [\![\tau_2]\!]^S \arrow[d, "\sigma_{\tau_2}"] \\
[\![\tau_1]\!]^{S'} \arrow[r, "op'"] & [\![\tau_2]\!]^{S'}
\end{tikzcd}
\quad \text{implies} \quad
\begin{tikzcd}
[\![\tau_1]\!]^{F(S)} \arrow[r, "op^F"] \arrow[d, "(F\sigma)_{\tau_1}"'] & [\![\tau_2]\!]^{F(S)} \arrow[d, "(F\sigma)_{\tau_2}"] \\
[\![\tau_1]\!]^{F(S')} \arrow[r, "op'^F"] & [\![\tau_2]\!]^{F(S')}
\end{tikzcd}$$

We give a very general theorem on the existence of natural liftings for operations of very simple arities.

Proposition 4.1.3 *Let $\tau 0$ be a metavariable ranging over the set of type expressions without T, i.e.*

$$\tau 0 \in Type0 ::= \ 1 \mid A \mid \gamma \mid \tau 0^A \mid \tau 0_1 \times \tau 0_2$$

if F is an endofunctor on $\mathrm{Mon}(\mathcal{C})$ and $\eta^F: \mathrm{Id}_{\mathrm{Mon}(\mathcal{C})} \dot{\to} F$ is a natural transformation, then there is a natural lifting of operations of arity $\tau 0 \dot{\to} \tau$ along η^F.

Proof Given an operation $op\colon \tau 0 \dot{\rightarrow} \tau$ on S, we define its lifting op^F along η^F_S as the natural transformation $op^F = op; \eta^F_{S\,\tau}$. This is a lifting of op along η^F_S because $\eta^F_{S\,\tau 0}$ is the identity (as T does not occur in $\tau 0$). Moreover, it is natural because for every monad morphisms $\sigma\colon S \to S'$ the following diagram commutes:

$$\begin{tikzcd}
[\![\tau]\!]^S \arrow[r, "\eta^F_{S\,\tau}"] \arrow[d, "\sigma_\tau"'] & [\![\tau]\!]^{F(S)} \arrow[d, "(F\sigma)_\tau"] \\
[\![\tau]\!]^{S'} \arrow[r, "\eta^F_{S'\,\tau}"] & [\![\tau]\!]^{F(S')}
\end{tikzcd}$$

∎

The above result is rather unsatisfactory, because we want to consider operations that takes computations as parameters (like *handle* below). However, it seems that better results can be achieved only by relying on some special property of the monad constructor under investigation.

4.1.1 Exceptions

The monad constructor for exceptions (compare with 9.2 of [Sch86]) is particularly simple and enjoys nice mathematical properties. It is also a strong monad constructor provided coproducts distribute over products, which is always the case in a cartesian closed category.

Definition 4.1.4 *Given a monad $T = (T, \eta^T, \mu^T)$ the monad T_{excp} of T-***computations with exceptions** *is defined as follows:*

- $T_{excp}(\_) = T(\_ + E)$
- $\eta^{T_{excp}}_A(a) = [\mathrm{inl}(a)]_T$, *i.e.* $\eta^{T_{excp}}_A = \mathrm{inl}; \eta^T_{A+E}$
- $\mu^{T_{excp}}_A(c) = (\mathrm{let}_T\, x{=}c\,\mathrm{in}\,(\mathrm{case}\, x\, \mathrm{of}\; c\colon T(A+E) \Rightarrow c \,|\, e\colon E \Rightarrow [\mathrm{inr}(e)]_T))$, *i.e.* $\mu^{T_{excp}}_A = ([\mathrm{id}_{T(A+E)}, \mathrm{inr}; \eta^T_{A+E}])^*$

A monad morphism $\sigma\colon S \to T$ induces a monad morphism $\sigma_{excp}\colon S_{excp} \to T_{excp}$, namely $\sigma_{excp\,A} = \sigma_{A+E}$

For every monad T there are two monad morphisms $\eta^{excp}_T\colon T \to T_{excp}$ and $\mu^{excp}\colon T_{excp_{excp}} \to T_{excp}$ which make the monad constructor for exceptions into a monad over $\mathrm{Mon}(\mathcal{C})$

- η^{excp}_T has A-component $T(\mathrm{inl})\colon TA \to T(A+E)$
- μ^{excp}_T has A-component $T([\mathrm{id}, \mathrm{inr}])\colon T((A+E)+E) \to T(A+E)$

For every monad T there are two operations on T_{excp}, **raise**: $E \dot{\to} T(\gamma)$ and **handle**: $E \times T(\gamma) \times T(\gamma) \dot{\to} T(\gamma)$, that are natural in T:

- $raise(e)$ raises the exception e.
- $handle(e, c_1, c_2)$ evaluates c_2 and if this evaluation raises e, then it evaluates c_2. This operation can be defined only if E has a decidable equality, i.e. a morphism $eq: E \times E \to 1 + 1$ s.t. $eq(e_1, e_2) = \mathrm{inl}(*)$ iff $e_1 = e_2$.

Finally we give sufficient conditions on arities for the existence of natural liftings of operations of those arities along η^{excp}.

Proposition 4.1.5 *If τ_1 is an out-type and τ_2 is an in-type (see Table 4.1), then there is a natural lifting of operations of arity $\tau_1 \dot{\to} \tau_2$ along η^{excp}.*

Proof Let $\sigma: \Gamma \to Type$ be a substitution mapping a type variable γ either to itself or to $\gamma + E$, this induces (in the obvious way) an endofunctor $[\![\sigma]\!]$ on $\mathcal{C}^\Gamma$. Let $in: \mathrm{Id} \dot{\to} [\![\sigma]\!]$ be the natural transformation s.t. $in_f: f\gamma \dot{\to} [\![\sigma]\!](f\gamma)$ is either $\mathrm{inl}: f\gamma \to f\gamma + E$ (when $\sigma(\gamma) = \gamma + E$) or the identity of $f\gamma$ (when $\sigma(\gamma) = \gamma$).

If τ is an out-type, then there is a natural transformation out_τ s.t. the following diagram commutes

$$\begin{tikzcd}
[\![\tau]\!]^S \arrow[r, "(\eta^F_S)_\tau"] \arrow[dr, "in; [\![\tau]\!]^S"'] & [\![\tau]\!]^{S_{excp}} \arrow[d, "out_\tau"] \\
& [\![\tau[\sigma]]\!]^S
\end{tikzcd}$$

Conversely, If τ is an in-type, then there is a natural transformation in_τ in the opposite direction. Therefore, given an operation $op: \tau_1 \dot{\to} \tau_2$ on S, we can define $op^{excp}: \tau_1 \dot{\to} \tau_2$ on S_{excp} to be $out_{\tau_1} \cdot ([\![\sigma]\!]; op) \cdot in_{\tau_2}$. ∎

Remark 4.1.6 The monad constructor for exceptions is very well-behaved in comparison to those for side-effects and continuations. We have chosen to introduce it first and highlight its properties in order to set an ideal standard.

4.1.2 Side-effects

We present a very general monad for side-effects, which will be instanciated to monads that are more appropriate for giving semantics to programming languages. This construction depends on a set S. For simplicity we work in the category of sets.

Definition 4.1.7 *Given a monad $T = (T, \eta^T, \mu^T)$ the monad T_{seff} of T-***computations with side-effects** *is defined as follows:*

	PROPERTIES			
case τ of	$E \to \tilde{\tau}$	$\tilde{\tau} \to \tau^E + E$	$\tilde{\tau} \to \tau^E$	$\tau^E \to \tilde{\tau}$
			$in-type$	$out-type$
1	yes	yes	yes	yes
A	no	yes	yes	yes
$\sigma(\gamma) = \gamma$	no	yes	yes	yes
$\sigma(\gamma) = \gamma + E$	yes	yes	no	yes
$T(\tau)$	yes	$\tilde{\tau} \to \tau^E + E$	$\tilde{\tau} \to \tau^E + E$	$\tau^E \to \tilde{\tau} \wedge E \to \tilde{\tau}$
$\tau_1 \times \tau_2$	$\wedge_i E \to \tilde{\tau}_i$	$\wedge_i \tilde{\tau}_i \to \tau_i^E + E$	$\wedge_i \tilde{\tau}_i \to \tau_i^E$	$\wedge_i \tau_i^E \to \tilde{\tau}_i$
τ^A	$E \to \tilde{\tau}$	$\tilde{\tau} \to \tau^E$	$\tilde{\tau} \to \tau^E$	$\tau^E \to \tilde{\tau}$

We write $\tilde{\tau}$ for the functor $[\![\tau[\sigma]]\!]^T$ (where $\tau[\sigma]$ is the σ-substitution instance of τ) and τ^E for the functor $[\![\tau]\!]^{T_E}$.
To determine whether the type expression τ is an in-type one should look at the entry in row τ and column in-type, if it is yes or the immediate subexpressions of τ satisfy the properties specified in the entry, then τ is an in-type, otherwise it may not be.
The auxiliary properties $E \to \tilde{\tau}$ and $\tilde{\tau} \to \tau^E + E$ are needed to test whether $T(\tau)$ is an in-type or out-type.

Table 4.1: Inductive definition of in-type and out-type

- $T_{seff}(\_) = (T(\_ \times S))^S$
- $\eta_A^{T_{seff}}(a) = (\lambda s\colon S.[\langle a, s\rangle]_T)$
- $\mu_A^{T_{seff}}(c) = (\lambda s\colon S.(\mathrm{let}_T\, \langle c_1, s_1\rangle{=}cs \,\mathrm{in}\, c_1 s_1))$

A monad morphism $\sigma\colon S \to T$ induces a monad morphism $\sigma_{seff}\colon S_{seff} \to T_{seff}$, namely $\sigma_{seff\,A} = \sigma^S_{A\times S}$

For every monad T there is one monad morphism $\eta_T^{seff}\colon T \to T_{seff}$

- η_T^{seff} has A-component $(\eta_T^{seff})_A(c) = (\lambda s\colon S.(\mathrm{let}_T\, a{=}c \,\mathrm{in}\, [\langle a, s\rangle]_T))$

Remark 4.1.8 There is no simple way to turn the monad constructor for side-effects into a monad over $\mathrm{Mon}(\mathcal{C})$.

We give sufficient conditions on arities for the existence of natural liftings of operations of those arities along η^{seff}.

Proposition 4.1.9 *If $\tau 1$ is a type expression without nesting of T, i.e.*

$$\tau 1 \in Type1 ::= \ 1 \mid A \mid \gamma \mid T(\tau 0) \mid \tau 1^A \mid \tau 1_1 \times \tau 1_2$$

then there is a natural lifting of operations of arity $\tau 1 \dot{\to} T(\gamma)$ along η^{seff}.

Proof Let $\sigma\colon \Gamma \to Type$ be the substitution mapping a type variable γ to $\gamma \times S$, this induces (in the obvious way) an endofunctor $[\![\sigma]\!]$ on $\mathcal{C}^\Gamma$. For every $\tau 0$ there is a natural transformation $in_{\tau 0}\colon \tau 0 \times S \dot{\to} \tau 0[\sigma]$ (exercise).

Given an operation $op\colon \tau 1 \dot{\to} T(\gamma)$ on S (for simplicity we take $\tau 1$ to be $\tau 0_1 \times T(\tau 0_2)$ which is paradigmatic) we can define $op^{seff}\colon \tau 1 \dot{\to} T(\gamma)$ on S_{seff} to be

$$op^{seff}(x, c) = (\lambda s\colon S.op_\sigma(in_{\tau 0_1}(x, s), T(in_{\tau 0_2})(fs)))$$

∎

In order to introduce operations on T_{seff} natural in T, we have to specialise the parameter S:

- stores $S = U^L$ (compare with Table 2.26 of [Mos89]). This monad model programs using a fixed set L of locations (for simplicity we have not placed any type restriction on the values storable in a given location, any value of type U can be stored in any location). Before executing a program these locations have already a value, either an input data or some garbage. There are two operations associated with this monad **lookup**: $L \to TU$ and **update**: $L \times U \to T1$

- $lookup(l)$ return the contents of location l.
- $update(l, u)$ updates the contents of location l with u.

- input $S = U^*$ (compare with Table 2.31 of [Mos89]). This monad model programs that can input (but not output) data. This semantics is called *batch* in [Mos89], because it does not capture the interleaving between internal computations and input of data. There is one operation associated with this monad **read**: $1 \to TU$
  - $read(*)$ read the first character in the input file and removes it from such file. To define *read* when the input is empty we can either raise an exception (if T allows that) or return a default value in U.

- output $S = U^*$ (compare with Table 2.31 of [Mos89]). This monad model programs that can output (but not input) data. Also this semantics is called *batch*, because it does not capture the interleaving between internal computations and output of data. There is one operation associated with this monad **write**: $U \to T1$
  - $write(u)$ append the character u to the output file.

4.1.3 Continuations

Continuations are a very general technique used in Denotational Semantics, based on the idea that if you tell to a *piece of program* what to do next, then it will tell you what is the result of the program. This construction takes as parameter a set R of possible results, or better what can be observed of an entire program, and it makes sense to take a very small R, e.g. the terminal object 1.

Definition 4.1.10 *Given a monad* $T = (T, \eta^T, \mu^T)$ *the monad* T_{cont} *of* T**-computations with continuations** *is defined as follows:*

- $T_{cont}(\_) = (TR)^{(TR)^{\_}}$
- $\eta_A^{T_{cont}}(a) = (\lambda k{:}\,(TR)^A.ka)$
- $\mu_A^{T_{cont}}(c) = (\lambda k{:}\,(TR)^A.c(\lambda h{:}\,(TR)^{(TR)^A}.hk))$

Remark 4.1.11 There is no way to make the monad constructor for continuations into an endofunctor, because T_{cont} is both covariant and controvariant in T. Because of this one should apply the monad constructor for continuations with some constraint. In fact, if S and T are related via a monad morphism, there is no relation between S_{cont} and T_{cont} which mirrors the one between S and T.

For every monad T there is one monad morphism $\eta_T^{cont}{:}\, T \to T_{cont}$

- η_T^{cont} has A-component $(\eta_T^{cont})_A(c) = (\lambda k\colon (TR)^A.(\mathrm{let}_T\, a{=}c\, \mathrm{in}\, ka))$

Remark 4.1.12 It does not make sense to ask whether η^{cont} is a natural transformation, because the monad constructor for continuations is not a functor.

We give a simple guideline for using the monad constructor for continuations:

- start with a simple monad T, e.g. partial computations or non-determinism
- add continuations, so that we have T_{cont} and a morphism η_T^{cont} from T to T_{cont}
- add other features modelled by an endofunctor F, so that we can lift η_T^{cont} to a morphism from $F(T)$ to $F(T_{cont})$

We do not address the issue of lifting operations on T along η_T^{cont}.

Exercise 4.1.12.1 *Prove that any operation of arity $\tau 0 \times (T\gamma)^A \dot{\to} (T\gamma)$ on T can be lifted along η_T^{cont}.*

4.1.4 Variable sets and dynamic allocation

The idea of variable set, i.e. an object of a functor category $\mathbf{Set}^K$, has been proposed by Reynolds, Oles (and others) for capturing the *variability* of the set of locations in programming languages like (idealised) ALGOL. However, variables sets arises in many other situations. For instance, an ML-program, during its execution, may allocate memory or declare new exceptions. So the set of memory cells and the set of declared exceptions grow with the number of execution steps. Similarly, to execute a *call/cc* or $note(c).e$ (see Talcott) one has to declare a new name and bind it to the current continuation. A reasonable choice for K would be the partial order on natural numbers, or natural numbers (viewed as sets) with injective maps as morphisms. Intuitively, an element of $k \in K$ tells us how many locations there are at that stage. In this section we consider a strong monad for *allocating a new element of a variable set*, which can be used as building block to define more complex monads. This construction depends on the following parameters:

- a category K, an endofunctor $F\colon K \to K$, a natural transformation $\sigma\colon \mathrm{Id}_K \dot{\to} F$
- a variable set L, i.e. a functor $L\colon K \to \mathbf{Set}$, a global element new of $F;L$, i.e. a natural transformation $new\colon 1 \dot{\to} F;L$

The objects of K should be thought as stages k, providing some information on the computation, in particular on the number of locations created so far. Intuitively, F and σ describe the change caused by the allocation of a new element, namely if we start from a stage k allocation causes a transition σ_k from k to a new stage Fk. At the level of $\mathbf{Set}^K$ the allocation of a new element transforms a variable set A into the variable set $F;A$ and the natural transformation $(\sigma;A): A \dot{\rightarrow} F;A$ tell us what are the elements of A after allocation (i.e. $F;A$) that were in A before. Finally, *new* corresponds to the newly allocated element.

Remark 4.1.13 We have considered the category **Set** of sets, but what follows applies to a wide class of categories. An intrinsic limitation of the model above, suggested by the intuitive explanation, is that the only *variability* we can capture is *monotonic* (a typical feature of Intuitionistic Logic and its Kripke Semantics).

There are some extra requirements on L and *new*, that seem appropriate if L have to be understood as a set of *names* with a *decidable equality* and *new* has to be a really new name:

- if $f: k \to k'$ in K, then $L(f)$ must be an injective map, so that if two names are different at stage k they stay different also at later stages
- the element new_k of $L(Fk)$ should not be in the image of $L(k)$ along $\sigma; L$, i.e. new_k is different from any element existing before allocation.

These assumptions are not necessary to define the strong monad below, but they are important to define operations like *update* or *handle*.

Definition 4.1.14 *The strong monad* $T = (T, \eta^T, \mu^T, \mathrm{t}^T)$ *for* **allocation** *over the category* $\mathbf{Set}^K$ *is defined as follows:*

- $(TA)(k) = \Sigma n{:}\, N.A(F^n k)$
  if $f: k \to k'$, *then* $(TA)(f)(\langle n, a\rangle) = \langle n, A(F^n f)a\rangle$
- $\eta^T_A(k)(a) = \langle 0, a\rangle$
- $\mu^T_A(k)(\langle m, \langle n, a \in A(F^{m+n}k)\rangle\rangle) = \langle m+n, a\rangle$
- $\mathrm{t}^T_{A,B}(k)(\langle a, \langle n, b \in B(F^n k)\rangle\rangle) = \langle n, \langle (\sigma^n; A)_k(a), b\rangle\rangle$

There is one operation associated with this monad **alloc**: $1 \to TL$

- $alloc(k)(*) = \langle 1, new_k(*)\rangle$, intuitively *alloc* makes one F-step in K, from k to Fk, and picks up the newly allocated element in $L(Fk)$.

Example 4.1.15 The definition of *alloc* may look suspicious, because it returns always the *same* new element, so one may expect that the denotation of

$$e_i \equiv (\mathrm{let}_T\, l_1{=}alloc\, \mathrm{in}\, (\mathrm{let}_T\, l_2{=}alloc\, \mathrm{in}\, [l_i]_T))$$

does not depend on i. However, the tensorial strength t^T renames the locations created before the last allocation (see requirements on L and *new* stated in Remark refadd-prop). We give a simple example of strong monad for allocation, where the expression above have different interpretations for different i (exercise):

- take K to be the category of natural numbers (viewed as sets, i.e. $n = \{0, \ldots, n-1\}$) with injective maps as morphisms
- take F to be the endofunctor s.t. $F(n) = n + 1$ and given an injection $i: m \hookrightarrow n$, then the injection $F(i): m+1 \hookrightarrow n+1$ extends i to $n+1$ by mapping m to n
- take σ to be the natural transformation s.t. $\sigma_n: n \hookrightarrow n+1$ is the inclusion of n into $n+1$
- take $L: K \to \mathbf{Set}$ to be the inclusion functor of K into $\mathbf{Set}$ and $new_n = n$

Prove that in this model $L \stackrel{\sigma;L}{\to} F; L \stackrel{new}{\leftarrow} 1$ is a coproduct diagram (in particlar new_n is not in the image of $L(\sigma_n)$) and that $[\![e_i]\!]_n = \langle 2, n+i \rangle$. On the other hand, if we take K to be the partial order of natural numbers and $F(n) = n+1$ (σ_n must be the unique morphism from n to $n+1$), then there is no way to choose L and new so that new_n is not in the image of $L(\sigma_n)$.

Instead of defining a strong monad we should have defined a strong monad constructor, which takes a strong monad T over the category $\mathcal{C}$ and returns a strong monad T_{allc} over $\mathcal{C}^K$.

Exercise 4.1.15.1 *Let* $(T_{allc}A)(k) = T(\Sigma n: N.A(F^n k))$*, prove that this can be extended to a functor from* $\mathrm{Mon}(\mathcal{C})$ *to* $\mathrm{Mon}(\mathcal{C}^K)$*. Investigate whether there are other possibilities for* T_{allc} *which can be extended to a functor.*

Prove that a monad T *over* $\mathcal{C}$ *induces a monad* T^K *over* $\mathbf{Set}^K$ *(hint: define a 2-endofunctor that does the job).*

Discuss lifting of operations on T *for the monad constructors* T_{alloc} *and* T^K*.*

The monad for allocation is very simple, but if we apply to it the appropriate monad constructor for stores, namely $S = U^L$ (L is the variable set equipped with the operation *alloc*), then we can models dynamic memory allocation. Similarly, to model dynamic declaration of exceptions (as in ML), we have simply to apply the monad constructor for exceptions (with $E = L$).

Exercise 4.1.15.2 *Reynolds and Oles have been considering functor categories to model a stack discipline for memory allocation. this means that we are not allowed dynamic allocation of memory that once created cannot be deallocated. Consider an operation* $\mathbf{block}: U \times (TA)^L \dot{\to} TA$*. Intuitively* $block(u, (\lambda l.e))$ *is a block, which declares a new variable* l *initialised with* u*, computes a value of type* A *and at the end deallocate* l*. Can you define block on the monad of stores over the functor category* $\mathbf{Set}^K$*? What kind of assumptions do you need on* U *and* A *for defining such operation? Hint: we consider the strong monad* $TA = (A \times S)^S$ *with* $S = U^L$ *and require* U *and* A *to be constant sets (i.e.* $U = \bar{U}^K$ *for some set* $\bar{U}$*). Note that the endofunctor* $G = \mathbf{Set}^F$ *preserves limits and colimits and* $\sigma; A: A \to F; A$ *is a natural transformation from the identity functor on* $\mathbf{Set}^K$ *to* G*. Define the following auxiliary operations*

- $eval_{new}\colon B^L \to GB$
- $sub_{new}\colon B^L \times B \to GB$
- $incl\colon G(C^B) \to (GC)^{GB}$

To define block amounts to give a morphism from $U^L \times U \times ((A \times U^L)^{U^L})^L$ *to* $A \times U^L$*, or equivalently to* $GA \times (GU)^L$ *(because* A *and* U *are constant), namely:*

$$(sub_{new} \times eval_{new}); G(\mathrm{eval}); (GA \times (incl; (GU)^{\sigma;L}))$$

4.1.5 Communication and interleaving

In this section we consider monad constructions that use covariant domain equations. We remind the universal property characterising the least solution to a covariant domain equation in a category $\mathcal{C}$.

Definition 4.1.16 *If* F *is an an endofunctor on* $\mathcal{C}$*, then a least fix point for* F *is a morphism* $\alpha\colon FA \to A$ *s.t. for every* $\beta\colon FB \to B$ *there exists unique* $f\colon A \to B$ *making the following diagram commutes:*

$$\begin{tikzcd}
FA \arrow[r, "\alpha"] \arrow[d, "Ff"'] & A \arrow[d, "f"] \\
FB \arrow[r, "\beta"] & B
\end{tikzcd}$$

Because of its universal property $\alpha\colon FA \to A$ is actually an isomorphism. It may well be the case that an endofunctor does not have a least fix point, for instance the powerset functor does not (because of a cardinality argument), but the finite powerset functor (where only finite subsets are allowed) does have a least fix point. A sufficient condition for existence of least fix points is that the category $\mathcal{C}$ must have an initial object and colimits of ω-chains, and the endofunctor F must preserve such ω-colimits.

The general pattern followed so far for defining a (strong) monad constructor F is to express the (strong) monad $F(T)$ in the metalanguage for T. To keep this pattern we add a type constructor for recursive types (see the BNF for type expressions below) and corresponding term constructors (see Table 4.2):

$$\tau \in Type ::= \ 1 \mid A \mid \gamma \mid T(\tau) \mid (\mu\gamma.\tau) \mid \tau^A \mid \tau_1 \times \tau_2$$

Remark 4.1.17 Under fairly general assumptions on the category $\mathcal{C}$ and the functor T, every type expression τ induces a functor, denoted by τ, from $\mathcal{C}^\Gamma$ to $\mathcal{C}$. Actually, if T is a strong monad, then there is a canonically defined tensorial strength $\mathrm{t}^{\tau}_{X,\Gamma}: X \times \tau \to \tau[\sigma^X]$ (where σ^X is the substitution mapping γ to $X \times \gamma$) for each functor τ. This tensorial strenght is used to interpret the metalanguage in Table 4.2, and to define a stronger requirement on a least fix point $\alpha: FA \to A$ for a strong endofunctor F, namely that for every $\beta: X \times FB \to B$ there exists unique $f: X \times A \to B$ making the following diagram commutes:

$$\begin{tikzcd}
X \times FA \arrow[r, "X \times \alpha"] \arrow[d, "\langle \pi_1, F[X]f \rangle"'] & X \times A \arrow[d, "f"] \\
X \times FB \arrow[r, "\beta"] & B
\end{tikzcd}$$

where $F[X](f: X \times A \to B)$ is the morphism $\mathrm{t}^F_{X,A}; Ff$ from $X \times FA$ to FB.

Interactive input and output

In the section on side-effects we introduced two monads for batch input and output, here we define their interactive counterpart (compare with Table 2.33 of [Mos89]).

Definition 4.1.18 *Given a monad $T = (T, \eta^T, \mu^T)$ the monads T_{iI} and T_{iO} of* ***T-computations with interactive input*** *and* **output** *are defined as follows:*

- $T_{iI}(A) = \mu\gamma.T(A + (\gamma^U))$
- $T_{iO}(A) = \mu\gamma.T(A + (U \times \gamma))$

Intuitively, a computation with interactive input may either compute a value or wait for some input and then resume the computation, while a computation with interactive output may either compute a value or output some data and then continue the computation.

Exercise 4.1.18.1 *Complete the definition of T_{iI} and T_{iO} and prove that these monad constructors are endofunctors. Prove that $(T_{iI})_{iO}(A)$ and $(T_{iO})_{iI}(A)$ are isomorphic.*

Define monad morphisms from T to T_{iI} and T_{iO}. What can you say about lifting of operations for T along these monad morphisms?

There is one operation **read**: $1 \to TU$ associated with the monad T_{iI} and one operation **write**: $U \to T1$ associated with the monad T_{iO}:

- $read(*) = \mathrm{intro}([\mathrm{inr}(\lambda u{:}U.[\mathrm{inl}(u)]_T)]_T)$
- $write(u) = \mathrm{intro}([\mathrm{inr}(\langle u, [\mathrm{inl}(*)]_T \rangle)]_T)$

Exercise 4.1.18.2 *Define a monad morphism from T_{iO} to the monad for batch output $T_{bO} = (T(A \times O))^O$, where O is the set U^* of finite sequences of elements in U. How are the two write operations related?*

	GEN – RULE	
var	$\dfrac{}{\Gamma \vdash x \colon \tau}$	
f: $\tau_1 \to \tau_2$	$\dfrac{\Gamma \vdash e_1 \colon \tau_1}{\Gamma \vdash \mathrm{f}(e_1) \colon \tau_2}$	
TYPE	INTRO – RULE	ELIM – RULE
$T(\tau)$	$\dfrac{\Gamma \vdash e \colon \tau'}{\Gamma \vdash [e]_T \colon T\tau'}$	$\dfrac{\Gamma \vdash e_1 \colon T\tau_1 \quad \Gamma, x_1 \colon \tau_1 \vdash e_2 \colon T\tau_2}{\Gamma \vdash (\mathrm{let}_T\ x_1{=}e_1\ \mathrm{in}\ e_2) \colon T\tau_2}$
1	$\dfrac{}{\Gamma \vdash * \colon 1}$	
$\tau_1 \times \tau_2$	$\dfrac{\Gamma \vdash e_1 \colon \tau_1 \quad \Gamma \vdash e_2 \colon \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle \colon \tau_1 \times \tau_2}$	$\dfrac{\Gamma \vdash e \colon \tau_1 \times \tau_2}{\Gamma \vdash \pi_i(e) \colon \tau_1}$
τ^A	$\dfrac{\Gamma, a \colon A \vdash e \colon \tau}{\Gamma \vdash (\lambda a \colon A.e) \colon \tau^A}$	$\dfrac{\Gamma \vdash e_a \colon A \quad \Gamma \vdash e \colon \tau^A}{\Gamma \vdash e(e_a) \colon \tau}$
$(\mu\gamma.\tau)$	$\dfrac{\Gamma \vdash e \colon \tau[\gamma := (\mu\gamma.\tau)]}{\Gamma \vdash \mathrm{intro}(e) \colon \mu\gamma.\tau}$	$\dfrac{\Gamma, x \colon \tau[\gamma := \sigma] \vdash e \colon \sigma \quad \Gamma \vdash e_\mu \colon (\mu\gamma.\tau)}{\Gamma \vdash rec(x.e, e_\mu) \colon \sigma}$

Table 4.2: Term expressions

Interleaving semantics for CCS

In this section we give a monad constructor for CCS with value passing (compare with 12.4 of [Sch86]). To justify the domain equation used in the definition of T_{ccs} we recall the syntax for CCS terms with value passing and the intended meaning of each operation:

$$p \in \mathit{Process} ::= \ \mathit{Nil} \mid \tau.p \mid \alpha!u.p \mid \alpha?x.p(x) \mid p_1 \mathit{or}\, p_2 \mid p_1|p_2 \mid p \setminus \alpha$$

where α range over the set *Port* of port identifiers (unlike the set *Act* of actions, there is no involution operation or silent action for *Port*) and u range over a set U of transmittable values. Intuitively, a process, can be:

- Nil the deadlocked process
- $\tau.p$ a process which makes an internal indivisible action and becomes p
- $\alpha!u.p$ a process which output u on the port α and becomes p
- $\alpha?x.p(x)$ a process which wants to input a value from port α and become $p(x)$. Note that $p(x)$ may contain x free, like $\alpha!x.\mathit{Nil}$, and that x get bounded by ?. One may also consider additional operations that use values to define processes.
- $p_1 \mathit{or}\, p_2$ the process which may become either p_1 or p_2
- $p_1|p_2$ the parallel composition of p_1 and p_2. This involves not only arbitrary interleaving of actions from p_1 with actions from p_2, but also communication of values between the two processes.
- $p \setminus \alpha$ the process p with port α hidden, i.e. α can be used only for internal communications.

The domain equation below consider the first five operations as constructors for defining processes, while the remaining one have to be defined (by induction on the structure of $T_{ccs}(A)$).

Definition 4.1.19 *Given a monad $T = (T, \eta^T, \mu^T)$ the monads T_{ccs} of T-***computations with CCS** *is defined as follows:*

- $T_{ccs}A = \mu\gamma.T(A + 1 + \gamma + (\mathit{Port} \times \gamma^U) + (\mathit{Port} \times U \times \gamma) + (\gamma \times \gamma))$

Exercise 4.1.19.1 *Complete the definition of T_{ccs}. Prove that T_{ccs} can be defined by adding incrementally the following features to T-computations (show also that the order in which they are added is irrelevant):*

- *deadlock* $T_{nil}(A) = T(A + 1)$

- *internal indivisible action* $T_{iia}(A) = \mu\gamma.T(A + \gamma)$
- *interactive input* $T_{in}(A) = \mu\gamma.T(A + (Port \times \gamma^U))$
- *interactive output* $T_{out}(A) = \mu\gamma.T(A + (Port \times U \times \gamma))$
- *non-deterministic choice* $T_{or}(A) = \mu\gamma.T(A + (\gamma \times \gamma))$

Define monad morphisms from T to each of the monads for the five incremental steps and discuss lifting of operations for T along these monad morphisms.

To deserve the name T_{ccs} the monad constructor for CCS must make available all the operations on processes:

- $Nil: 1 \dot{\to} T(\gamma)$ deadlock
- $\tau: T(\gamma) \dot{\to} T(\gamma)$ internal indivisible action
- $!: Port \times U \times T(\gamma) \dot{\to} T(\gamma)$ output
- $?: Port \times (T\gamma)^U \dot{\to} T(\gamma)$ input
- $or: T(\gamma) \times T(\gamma) \dot{\to} T(\gamma)$ non-deterministic choice
- $|: T(\gamma_1) \times T(\gamma_2) \dot{\to} T(\gamma_1 \times \gamma_2)$ parallel composition
- $\backslash: Port \times T(\gamma) \dot{\to} T(\gamma)$ hidding of ports

Exercise 4.1.19.2 *Give the interpretation for the seven operations on T_{ccs} specified above. Consider two additional operations on processes $seq(p_1, p_2)$ and $alternate(p_1, p_2)$. Intuitively, $seq(p_1, p_2)$ is sequential composition, i.e. it executes p_1 and then (when p_1 has finished) p_2, while $alternate(p_1, p_2)$ alternates one action of p_1 with one of p_2 (starting from p_1). Give an interpretation for the operations $seq: T(\gamma_1) \times T(\gamma_2) \dot{\to} T(\gamma_2)$ and $alternate: T(\gamma_1) \times T(\gamma_2) \dot{\to} T(\gamma_1 \times \gamma_2)$ on T_{ccs} (hint: seq is an instance of let. alternate is underspecified, i.e. there are several possible definitions, and it is already definable on T_{iia}).*

Bibliography

[Abr87] S. Abramsky. *Domain Theory and the Logic of Observable Properties.* PhD thesis, University of London, 1987.

[Bar81] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North Holland, 1981.

[BW85] M. Barr and C. Wells. *Toposes, Triples and Theories.* Springer Verlag, 1985.

[FFKD86] M. Felleisen, D.P. Friedman, E. Kohlbecker, and B. Duba. Reasoning with continuations. In *1st LICS Conf.* IEEE, 1986.

[Flo67] R. Floyd. Assigning meaning to programs. In Schwartz, editor, *Proc. Symp. in Applied Math.*, 1967.

[GMW79] M.J.C. Gordon, R. Milner, and C.P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science.* Springer Verlag, 1979.

[Gog89] J. Goguen. A categorical manifesto. to appear as Oxford PRG technical report, 1989.

[Gol79] R. Goldblatt. *Topoi, The Categorical Analysis of Logic.* North Holland, 1979.

[HJ] C.A.R. Hoare and H. Jifeng. Natural transformation and data refinement. draft December 7, 1988.

[HM88] R. Harper and J. Mitchell. The essence of ML. In *15th POPL.* ACM, 1988.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *C. ACM*, 12, 1969.

[Kel82] G.M. Kelly. *Basic Concepts of Enriched Category Theory.* Cambridge University Press, 1982.

[Koc72] A. Kock. Strong functors and monoidal monads. *Archiv der Mathematik*, 23, 1972.

[KR77] A. Kock and G.E. Reyes. Doctrines in categorical logic. In J. Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic*. North Holland, 1977.

[KS74] G.M. Kelly and R.H. Street. Review of the elements of 2-categories. In A. Dold and B. Eckmann, editors, *Category Seminar*, volume 420 of *Lecture Notes in Mathematics*. Springer Verlag, 1974.

[Lan64] P.J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4), 1964.

[Mac71] S. MacLane. *Categories for the Working Mathematician*. Springer Verlag, 1971.

[Mac86] D. MacQueen. Using dependent types to express modular structures. In *13th POPL*. ACM, 1986.

[McC62] J. McCarthy. *The LISP 1.5 Programmers' Manual*, 1962.

[ML83] P. Martin-Löf. The domain interpretation of type theory. In P. Dybjer, B. Nordström, K. Petersson, and J. Smith, editors, *Workshop on the Semantics of Programming Languages*. Chalmers University of Technology, Göteborg, Sweden, 1983.

[Mog86] E. Moggi. Categories of partial morphisms and the partial lambda-calculus. In *Proceedings Workshop on Category Theory and Computer Programming, Guildford 1985*, volume 240 of *Lecture Notes in Computer Science*. Springer Verlag, 1986.

[Mog88] E. Moggi. *The Partial Lambda-Calculus*. PhD thesis, University of Edinburgh, 1988.

[Mog89a] E. Moggi. A category-theoretic account of program modules. In *Proceedings of the Conference on Category Theory and Computer Science, Manchester, UK, Sept. 1989*, volume 389 of *Lecture Notes in Computer Science*. Springer Verlag, 1989.

[Mog89b] E. Moggi. Computational lambda-calculus and monads. In *4th LICS Conf.* IEEE, 1989.

[Mos89] P. Mosses. Denotational semantics. Technical Report MS-CIS-89-16, Dept. of Comp. and Inf. Science, Univ. of Pennsylvania, 1989. to appear in North Holland Handbook of Theoretical Computer Science.

[MT89] I. Mason and C. Talcott. Programming, transforming, and proving with function abstractions and memories. In *16th Colloquium on Automata, Languages and Programming*. EATCS, 1989.

[Pau85] L.C. Paulson. Interactive theorem proving with cambridge lcf: a user's manual. Technical Report 80, Univ. of Cambridge, Computer Laboratory, 1985.

[Pie88] B.C. Pierce. A taste of category theory for computer scientists. Technical Report CMU-CS-88-203, Carnegie-Mellon Univ., Dept. of Comp. Sci., 1988.

[Plo75] G.D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1, 1975.

[Plo81] G.D. Plotkin. Postgraduate lecture notes in domain theory (incorporating the "pisa notes"). Edinburgh Univ., Dept. of Comp. Sci., 1981.

[Plo85] G.D. Plotkin. Denotational semantics with partial functions. Lecture Notes at C.S.L.I. Summer School, 1985.

[Ros86] G. Rosolini. *Continuity and Effectiveness in Topoi.* PhD thesis, University of Oxford, 1986.

[Sch86] D.A. Schmidt. *Denotational Semantics: a Methodology for Language Development.* Allyn & Bacon, 1986.

[Sco69] D.S. Scott. A type-theoretic alternative to CUCH, ISWIM, OWHY. Oxford notes, 1969.

[Sco70] D.S. Scott. Outline of a mathematical theory of computation. Technical Report PRG-2, Oxford Univ. Computing Lab., 1970.

[Sco76] D.S. Scott. Data types as lattices. *SIAM Journal of Computing*, 5, 1976.

[Sco79] D.S. Scott. Identity and existence in intuitionistic logic. In M.P. Fourman, C.J. Mulvey, and D.S. Scott, editors, *Applications of Sheaves*, volume 753 of *Lecture Notes in Mathematics*. Springer Verlag, 1979.

[Sco80] D.S. Scott. Relating theories of the λ-calculus. In R. Hindley and J. Seldin, editors, *To H.B. Curry: essays in Combinarory Logic, lambda calculus and Formalisms*. Academic Press, 1980.

[Sha84] K. Sharma. Syntactic aspects of the non-deterministic lambda calculus. Master's thesis, Washington State University, September 1984. available as internal report CS-84-127 of the comp. sci. dept.

[SS71] D.S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. Technical Report PRG-6, Oxford Univ. Computing Lab., 1971.

[Wad76] C. Wadsworth. The relation between computational and denotational properties for scott's D_∞-models of the lambda calculus. *SIAM Journal of Computing*, 5, 1976.

Course exercises

1. Express η_A, μ_A and $T(f: A \to B)$ in the metalanguage with base types A and B and unary operation $f: A \to B$.

   Given two monads S and T over the category C, express that σ is a simplified monad morphism from S to T via a set of equations in the metalanguage with two unary type constructors S and T, one base type for each object of $\mathcal{C}$ and one unary function $\sigma_\tau: S\tau \to T\tau$ for each type expression τ. Try to make the equations as simple as possible, no more than **two** let or $[\_]$ in each equation. Given two strong monads S and T over the category C, What additional axioms do you need (if any) to express that σ is a strong monad morphism?

2. Prove that any operation of arity $\tau 0 \times (T\gamma)^A \dot{\to} (T\gamma)$ on T can be lifted along $\eta_T^{cont}: T \to T_{cont}$. Discuss possible improvements to lifting of operations along η^{cont} and η^{seff}.

3. Given a category K, and an endofunctor $F: K \to K$ define two functors from $\mathrm{Mon}(\mathcal{C})$ to $\mathrm{Mon}(\mathcal{C}^K)$. If T is a monad over $\mathcal{C}$, then

   - $T^K(A)(k) = T(A(k))$ (hint: define a 2-endofunctor on **Cat** that does the job for any K).
   - $(T_{allc}A)(k) = T(\Sigma n: N.A(F^n k))$ (we assume that $\mathcal{C}$ has countable coproducts)

   complete the definition of T^K and T_{allc} and define a monad morphism from T^K to T_{allc}.

   Investigate whether there are other possibilities for T_{allc} which take a monad T in $\mathcal{C}^K$ instead.

   Discuss lifting of operations on T for the monad constructors T_{alloc} and T^K (hint: is there a monad morphism in the generalized sense from T to these two monads on $\mathcal{C}^K$)

4. Reynolds and Oles have been considering functor categories to model a stack discipline for memory allocation. this means that we are not allowed dynamic allocation of memory that once created cannot be deallocated. Consider an

operation **block**: $U \times (TA)^L \dot{\to} TA$. Intuitively $block(u, (\lambda l.e))$ is a block, which declares a new variable l initialized with u, computes a value of type A and at the end deallocate l. Can you define $block$ on the monad of stores over the functor category $\mathcal{C}^K$ by choosing a suitable K and L? What kind of assumptions do you need on U and A for defining such operation? (hint: consider the restrictions on a programming language, that make possible a stack discipline for memory allocation).

5. Complete the definition of the monad constructor for CCS

$$T_{css}A = \mu\gamma.T(A + 1 + \gamma + (Port \times \gamma^U) + (Port \times U \times \gamma) + (\gamma \times \gamma))$$

   Prove that T_{ccs} can be defined by adding incrementally the following features to T-computations (show also that the order in which they are added is irrelevant):

   - deadlock $T_{nil}(A) = T(A + 1)$
   - internal indivisible action $T_{iia}(A) = \mu\gamma.T(A + \gamma)$
   - interactive input $T_{in}(A) = \mu\gamma.T(A + (Port \times \gamma^U))$
   - interactive output $T_{out}(A) = \mu\gamma.T(A + (Port \times U \times \gamma))$
   - non-deterministic choice $T_{or}(A) = \mu\gamma.T(A + (\gamma \times \gamma))$

   Define monad morphisms from T to each of the monads for the five incremental steps and discuss lifting of operations for T along these monad morphisms.

6. Prove that $(T_{in})_{out}(A)$ and $(T_{out})_{in}(A)$ (see previous esercise) are isomorphic. Define a simplified version, $T_{bO} = (T(A \times O))^O$, of the monad for batch output, where O is the set U^* of finite sequences of elements in U. Define a monad morphism from T_{out} to T_{bO}, then consider how the interpretations of the operation $write: U \to T1$ in the two monad are related.

7. Give the interpretation for the following operations on T_{css}:

   - $Nil: 1 \dot{\to} T(\gamma)$ deadlock
   - $\tau: T(\gamma) \dot{\to} T(\gamma)$ internal indivisible action
   - $!: Port \times U \times T(\gamma) \dot{\to} T(\gamma)$ output
   - $?: Port \times (T\gamma)^U \dot{\to} T(\gamma)$ input
   - $or: T(\gamma) \times T(\gamma) \dot{\to} T(\gamma)$ non-deterministic choice
   - $|: T(\gamma_1) \times T(\gamma_2) \dot{\to} T(\gamma_1 \times \gamma_2)$ parallel composition
   - $\backslash: Port \times T(\gamma) \dot{\to} T(\gamma)$ hidding of ports

Consider two additional operations on processes $seq(p_1, p_2)$ and $alternate(p_1, p_2)$. Intuitively, $seq(p_1, p_2)$ is sequential composition, i.e. it executes p_1 and then (when p_1 has finished) p_2, while $alternate(p_1, p_2)$ alternates one action of p_1 with one of p_2 (starting from p_1). Give an interpretation for the operations $seq: T(\gamma_1) \times T(\gamma_2) \dot{\rightarrow} T(\gamma_2)$ and $alternate: T(\gamma_1) \times T(\gamma_2) \dot{\rightarrow} T(\gamma_1 \times \gamma_2)$ on T_{ccs} (hint: seq is an instance of let. $alternate$ is underspecified, i.e. there are several possible definitions, and it already definable on T_{iia}).