Computational applications of calculi based on monads

Pietro Cenciarelli

Doctor of Philosophy
University of Edinburgh
September 1995

(Graduation date: July 1996)

A Giovanna e Renato

Abstract

This thesis studies various manifestations of monads in the mathematics of computation and presents three applications of calculi based on monads.
The view that monads provide abstract mathematical interpretations of computational phenomena led E. Moggi to use the internal language of a category with a strong monad, which he called the computational lambda calculus, for describing denotational semantics of programming languages. Moggi argued that models of complicated forms of computation could be described modularly by using semantic constructors for manipulating monads.
For the first application, we describe a theory of exceptions in the computational lambda calculus and give a computationally adequate interpretation of a fragment of ML, including the exception handling mechanism, in models of this theory. To our knowledge no other model of ML exceptions is available in the literature to date. We also show that normalization fails when exceptions are added to the simply typed lambda calculus.
Building on top of the computational lambda calculus, A.M. Pitts proposed a predicate calculus to reason about the evaluation properties of programs: the Evaluation Logic. Following the tenets of synthetic domain theory, we interpret this logic in an ambient category with set-like structure and a fully reflective subcategory of domains with a monad for interpreting computation. We establish abstract conditions under which the monad extends to the ambient category to ensure good interaction with the logical structure. We also show that a monad and first order logical structure yield suitable evaluation relations, which can be used to give a standard interpretation of Evaluation Logic when higher order structure is not available.
For the second application, we focus on side effects and investigate the use of Evaluation Logic in partial correctness reasoning. We show that, under fairly common circumstances, monads for side effects admit an extension to the ambient category which is more natural than that described for arbitrary monads and we validate special axioms for members of this class. The resulting theory of computation with side effects is then put to work on a textbook example of partial correctness specification.
For our third application, we consider Moggi's modular approach to denotational semantics. We develop the theory of this approach by determining which equations are preserved by a fairly general class of semantic constructors and which ones are reflected (conservativity). Moreover, we establish a correspondence between categories of computational models and categories of theories of the metalanguage, along the lines of Gabriel-Ulmer duality, in a type-theoretic framework. Using the Extended Calculus of Constructions, we develop a semantics for parallel composition by combining elementary notions of computation defined independently and we use LEGO to prove properties of such semantics formally.

Acknowledgements

I want to thank Rod Burstall for waking me up on the phone early one morning in Ithaca to inform me that I had been admitted to the PhD program and for giving me advice during the early stages of my work.

I am grateful for the financial support I received first from the Edinburgh University Faculty of Science and later from the EEC Jumelage project "Programming Language Semantics and Program Logics."

Thanks to Eugenio Moggi for following my work from a thousand miles away and for his hospitality during my trips to Genoa, and to Gordon Plotkin and John Power, for helping me with useful comments and suggestions during the writing-up.

I learnt a lot from discussing computers and mathematics with Rod Burstall, Marcelo Fiore, Eugenio Moggi, Gordon Plotkin, Alex Simpson, Bob Tennent and Andrew Wilson. I should specially thank Andrew for being such a pleasant office mate, for helping me patiently with English and for teaching me that most things are *be'er 'an a steen 'ahin' 'i lug.*

Alex, Aphro, Cottone, Jennie, Marcelo, Simona, Toby and Viky provided, with their friendship, the thing that Nature denied to this beautiful town: warmth. Jocelyne m'a donné la chose plus importante : son amour.

Declaration

This thesis was composed by myself. The work reported herein, unless otherwise stated, is my own.

Pietro Cenciarelli

Table of Contents

1 Introduction

Programming language semantics is an area of applied mathematics which studies the meaning of computer programs. The notion of "meaning" depends on the context of discourse [Neu44,Flo67,SS71,Plo81].

In denotational semantics, programs are explained in terms of abstract mathematical entities, called "denotations," which model their implementation-independent behaviour. In this context, *formal metalanguages* [Sco69,Plo77,Gor79,Plo85b,Abr87,Mog91b] can be used to link programs and their denotations: on one side they provide a language for addressing elements of mathematical structures, while on the other they provide a formal framework in which programs can be interpreted by means of translations.

How tightly should a metalanguage match the mathematical objects it describes and in what detail should it represent the structure of computation? In [Mog90a,Mog91b], Moggi proposed a categorical view: the use of *strong monads* to represent notions of computation *abstractly* and a formal metalanguage to match the categorical semantics. This approach goes one step towards the identification of abstract reasoning principles for computation. It also has a pragmatical significance, because abstraction is a basic ingredient to achieve *modularity*.

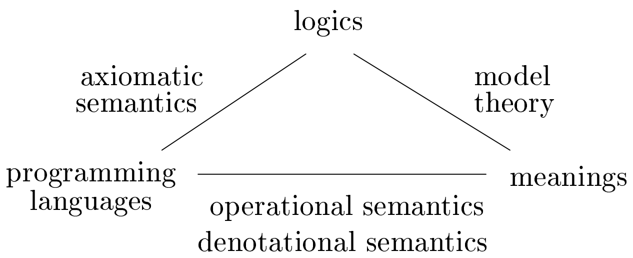
This thesis presents three applications of monads to the semantics of computation: in the first application monads provide a language for interpreting programs with exceptions; in the second, they provide a formal basis for reasoning about computations with states; in the third, they provide a structuring principle for a stepwise development of denotational models. As we shall see, sometimes practice feeds back into theory.

In this introduction, we describe the theoretical context of our exploration and, in the last section, we summarize our main results and give the outline of the thesis.

Programs, meanings and logics

Mathematical descriptions of programming languages are needed for the analysis and synthesis of computer programs. Both activities require logics to express the properties of interest and to guide the reasoning. Different kinds of semantics and logics may be appropriate for performing different tasks. Operational semantics, for example, yield notions of program equivalence which can be used for proving correctness of optimization steps, while denotational semantics may provide richer mathematical foundations for formal methods.

The use of formal systems to support reasoning about properties of computer programs is appealing because it allows one to disregard the mathematical details of the models. Such systems are available, for example, for reasoning about partial correctness [Hoa69] and noninterference [Rey81] for imperative languages, temporal properties of concurrent systems [Sti92], evaluation properties of programs with higher-order functional features [CP92,Pit91]. However, adopting the view of [Sco69] that formalism without eventual interpretation is in the end useless, we regard program logics and mathematical representations of the meaning of programs as parts of the same picture:



In the next sections we shall encounter many inhabitants of this triangle; although most of them are based on denotational semantics, many others can be found in the operational setting, where a basic example is the triad of CCS [Mil80], temporal logics and labelled transition systems.

Each of the disciplines along the edges of this diagram addresses specific problems. At the bottom, mathematical structures are matched against the programs they represent. Operational and denotational semantics can be related by *computational adequacy* results, as in [Plo85b,Abr90b,Fio94b]. In denotational semantics, the usual criteria for assessing the well-fittedness of models are *full abstraction* [Mil77,Plo77,Wad76,HA80,AJM94,HO94] and *universality* [Mey88].

On the left edge, axiomatic semantics is specially relevant to the study of formal methods for program development and verification [Hoa69,Dij76,Gri81].

Finally, the most frequently asked questions on the right edge concern soundness and completeness, both for models based on operational and on denotational descriptions. Examples of completeness results in the setting of process algebras are provided by [BA91], where it is shown how to generate complete sets of axioms for strong bisimulation automatically from SOS descriptions of programming languages, and by [Win85] which gives a complete proof system for SCCS [Mil83], whose models are based on non-well-founded sets [Acz88].

The completeness of logical theories can also be studied "functorially" (see section 6.5), by identifying theories with categories with logical structure and models with structure-preserving functors. A classical result in this setting is the *conceptual completeness* of pretopoi [MR76]: if the functor $Mod(\mathcal{T}_1) \to Mod(\mathcal{T}_2)$ between categories of models induced by composition by a logical functor $F : \mathcal{T}_2 \to \mathcal{T}_1$ between pretopoi is an equivalence, then F is an equivalence.

In the setting of domain theory, [Abr91] stresses the importance of "Stone-type" duality theorems, relating theories (viewed as categories) with the categories of their models, for understanding the relationship between semantics and program logics.

Metalanguages in denotational semantics

As remarked in [Abr91], "denotational semantics are always based, more or less explicitly, on a typed functional metalanguage." Then, where should we place metalanguages in the above picture?

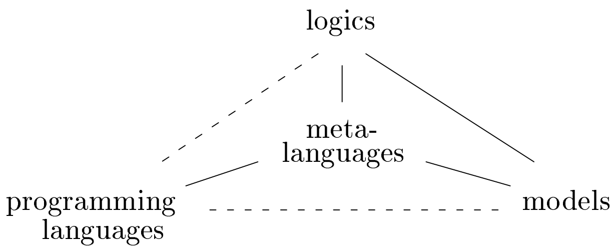
A leading example of metalanguage, at least from a historical point of view, is PCF. In 1969, D. Scott introduced a logic for computable functions, LCF [Sco69], to formalize principles of mathematical reasoning about partial recursive functions. LCF has enough power to axiomatize, for instance, integer arithmetics. The underlying term language of this logic, named "Programming language for Computable Functions" (PCF) in [Plo77], has been used to describe the semantics of non-trivial fragments of programming languages (including Pascal [AAW77]) through its standard interpretation in terms of cpos and continuous functions.

Scott remarked that PCF was not a programming language because, although allowing conditional expressions and recursive definitions, it featured neither assignments nor gotos. A different view was adopted in [Plo77], where PCF was endowed with a reduction relation in order to study the connections between operational and denotational semantics. A consequence of this approach, as advocated in [Plo91b, lecture III: Denotational Semantics], is that not only can programming languages be interpreted by translating them into a metalanguage, but also one can relate their denotational and operational semantics by exploiting a similar result proven once and for all for the metalanguage.

LCF was implemented by R. Milner in 1972 [Mil72,GMW79] and used as a mechanized logic for proving correctness of computer programs [Wey72,AA74], including interpreters and compilers [MW72,New75]. Structuring principles for LCF theories are sought in [SB83].

More elaborate type theories have been proposed as frameworks for studying more sophisticated programming language features, and even "implementational" issues such as garbage collection; the latter was addressed in [Laf88,Wad91] among other applications of linear logic to computer science. A logic for parametric polymorphism [Rey83] is proposed in [PA93], which is to System F what LCF is to PCF. However, this logic accounts for neither recursive functions nor types, which are both crucial in domain theory. Recursion is studied in FPC [Plo85a], a typed calculus which allows recursive definitions of types, as PCF allows recursive definitions of functions. In FPC, the latter can be obtained from the former as shown in [Gun92, 7.4]. Recursion is also addressed in [Plo93], where second order intuitionistic linear logic is combined with a formal theory of relational parametricity to obtain a framework in which general recursive type equations have universal solutions, with corresponding principles of induction and co-induction. The problem of finding appropriate logics for domain theory is a matter of current research. We shall return to this point later on.

In the above examples, metalanguages play the role of formal interfaces to mathematical models, that of term languages of logics and that of "target" languages for expressing denotations of programs. We can therefore place them in the centre of the triangle:



The dashed lines indicate that programs are interpreted and reasoned about *indirectly* through the metalanguage. This approach has the following advantages. On the denotational edge, interpretations of different programming languages and notions of computation can be more easily related if represented in the same formal framework. On the axiomatic edge, program logics which refer to metalanguages, where computation is represented in mathematical terms, are less likely to be affected by the idiosyncrasies of specific programming languages.

As we shall see below, these benefits are further enhanced when an abstract notion of computation is adopted. The next two sections introduce a metalanguage and program logics based on this abstraction. They are used in this thesis to develop applications.

Monads and the computational metalanguage

Denotational semantics of real programming languages are complicated and often require a considerable mathematical overhead. Adequate descriptions of even simple computational gadgets such as local variables may need sophisticated tools such as *possible worlds* [OT92]. Moreover, the mathematical structure used for describing one form of computation may be incompatible with that for describing another, so that models for several interacting computational phenomena are often studied case by case.

As noted in [Mos92], the reason why conventional programming devices require elaborate "coding" techniques to be expressed in lambda notation is that the basic operations of lambda abstraction and application give little grasp on program composition. Various modifications of the theory of $\beta\eta$-equivalence have been proposed in order to study features of programming languages. Examples are the call-by-name and call-by-value lambda calculi introduced in [Plo75], which are shown to be correct with respect to the corresponding evaluation mechanisms.

Another example is the λ_p-calculus introduced in [Mog86], which is shown to be sound and complete with respect to interpretation in partial cartesian closed categories. The approach adopted in this calculus, which was first pursued in [Plo85b], is to take partial functions as primitive in domain theory, rather than total functions on spaces of partial elements, as in the tradition of [Sco69]. Partial combinatory and lambda calculi are described in [Fef95], while "free" logics for reasoning with possibly non-denoting terms are surveyed in [Ben86].

A category $p\mathcal{C}$ of partial maps can be defined axiomatically as proposed in [Ros86, Mog86] by choosing a class of monomorphisms closed under certain operations in a category $\mathcal{C}$ of "total" maps. Intuitively, the morphisms in the chosen class represent domains of partial functions. Under certain circumstances, there is an adjunction between $\mathcal{C}$ and $p\mathcal{C}$ defining a "lifting" monad in $\mathcal{C}$ (see section 2.5), of which $p\mathcal{C}$ is the Kleisli category (see [Mac71] for definitions).

In [Mog90a,Mog91b], Moggi proposed a generalization of this scenario where *monads* are taken as abstract notions of computation and program composition is abstractly interpreted as composition in the Kleisli category. The metalanguage arising from this categorical semantics of computation, called *computational lambda calculus*, is the internal language of a cartesian closed category with a strong monad (see section 2.2). Examples of computationally relevant monads are given in section 2.5.

The main feature of the computational lambda calculus is the explicit distinction between types of values and types of programs, called *computational* types. These are formed by applying a unary type constructor T: for every type A, there is a type TA inhabited by the programs computing values of A. Of course, one can form computations of computations and so on. Interpreting types as objects in a category $\mathcal{C}$, T represents a map $obj(\mathcal{C}) \to obj(\mathcal{C})$. For instance, integer programs which may raise exceptions can be interpreted as elements of the object $T(int) = int + E$, where E interprets the type of exceptions.

Besides the type constructor T, the metalanguage features two type-indexed families of operations: $val_A : A \to TA$ and $let_{A,B} : (A \to TB) \times TA \to TB$. Members of the first family construct trivial computations from values: $val_A(a)$ is the program "`return` a," which, in the case of exceptions, corresponds to the injection $A \to A + E$. The operations of the family *let* perform program composition (while composition of functions corresponds to substitution as usual in the lambda calculus) allowing a program of type $A \to TB$, accepting as inputs *values* of type A, to apply to *computations*, that is a term of type TA. In the case of exceptions, $let_{A,B}(f, z)$ runs f on the value possibly produced by z, while, if z raises an exception e, it returns e.

These operations are equipped with a set of equations, which give a complete axiomatization of strong monads, where *val* corresponds to the unit, and *let* to an internal version of Kleisli lifting.

Other operations and constant types given by a signature Σ may also be included in the calculus, together with appropriate (equational) axioms, to make the notion of computation more concrete and represent specific programming languages. We shall call $ML_T(\Sigma)$ such a particular theory in the computational lambda calculus. Computer programs are interpreted by translations into $ML_T(\Sigma)$ mapping terms of type τ to terms of type $T(\!|\tau|\!)$, where $(\!|\tau|\!)$ is the translation of τ. This paradigm is explained in chapter 3.

By hiding the concrete structure of the domains of interpretation and by providing an abstract primitive operation of program composition, the computational metalanguage allows the intricacies of concrete models to be left out of denotational descriptions. In this way, such descriptions become *simpler*, because the mathematical structure required for interpreting one device of a programming language need not appear in the semantic equations which describe another. They also become easier to *modify*: if a new computational feature is added to the language, e.g. a mechanism for raising and handling exceptions, the domains of interpretation may have to change in order to accommodate new mathematical structure, which may cause massive rewriting in a traditional denotational description: in the case of exceptions, all semantic equations must deal with the possibility that subterms might raise exceptions. What has changed is the notion of program composition. However the equations need not change if this notion is captured by a simple *let*.

The benefits of the monadic approach have been exploited in several applications described in [Wad92], including a compiler for Haskell.

The abstraction over the notion of computation introduced by the use of monads in denotational semantics provides a basis for investigating general reasoning principles about programs and a structuring mechanism for a modular development of theories and models of computation. These topics are explored in the next two sections.

Monads and program logics

General program logics, that is logics providing general principles for reasoning about computation, may arise from an abstract understanding of program evaluation. The advantage of such logics is that programs from different languages, or from upgraded versions of the same language, can be reasoned about within the same framework.

Below we consider a few semantic scenarios for program logics based on abstract notions of computation: Fix-Logic, Evaluation Logic and modal operators in higher order logic. In the next section we shall encounter another, where *evaluation modalities*, which capture the notion of evaluation abstractly, are defined in a first order framework.

In the previous section, we saw that the idea of interpreting divergent computations in categories of partial maps can be generalized to other forms of computation by using monads. Similarly, consider the principle of Scott-induction proposed in [Sco69] to reason about recursively defined partial functions. This principle can be derived from an axiomatization of the initial algebra of a lifting monad. *Fix-Logic*, the predicate calculus proposed in [CP92], uses the computational metalanguage as term language and is interpreted in a category with a strong monad whose underlying functor has initial algebra. The assumption of such an algebra yields an abstract reasoning principle called *fix induction*.

In Fix-Logic, programs and logic are integrated by means of modalities extending predicates on values to predicates on computations. However, this extension is rather trivial as it only discriminates between computations of the form $val(v)$. In *Evaluation Logic* (EL) [Pit91], whose underlying type system is again the computational metalanguage, A. Pitts introduced more general modalities in the context of intuitionistic first order predicate calculus to match operational descriptions of programs given in natural semantics style [Kah88]. There are two "evaluation" modalities, called *necessity* and *possibility*, and written respectively:

$$[x \Leftarrow E]\ \phi(x) \quad \text{and} \quad \langle x \Leftarrow E\rangle\ \phi(x).$$

The intended meaning of these formulae is that the property $\phi(x)$ holds respectively of *any* and of *some* values x produced by the evaluation of E, where x is a variable of a type A and E is a program of type TA. Programs can be related to the values they compute by means of *evaluation relations*:

$$\Leftarrow_A \subseteq A \times TA,$$

whose intended meaning is captured by the formula $(a \Leftarrow E) \stackrel{\text{def}}{=} \langle x \Leftarrow E\rangle\,(x = a)$, that is: "$E$ can evaluate to a." Conversely, for most notions of computation, the evaluation modalities can be defined from $\Leftarrow$ as shown in section 4.8 using first order quantifiers.

The result is a powerful program logic which can be specialized by axiomatizing appropriate operations to suit specific forms of computation. Other program logics can be embedded into EL, including dynamic logic and Hoare logic (see [Pit91] and [Cen95]).

Several interpretations have been proposed of the evaluation modalities [Pit91,Mogb, Moga], of which we give a detailed account in chapter 4 (where we also propose yet another). Pitts' original interpretation was based on hyperdoctrines, over which he defined the notion of *T-modality*. Both hyperdoctrines and T-modalities are categorical structure that must be provided on top of a model of the underlying type theory in order to interpret the logic.

A different approach was taken in [Mogb], where the logical structure is sought in the same category where types are interpreted. The idea is to define modal operators by making the categorical gear for interpreting computational types act upon that for interpreting logic. We shall call this semantics "standard" because interpretation is determined only by the structure of a strong monad. The advantage of a standard semantics is that it is subject to the same structuring mechanisms, based upon manipulation of monads, which apply to the underlying metalanguage (see next section).

However, it is not immediately obvious that logic and computation should live in the same semantic framework, as one may put limitations on the other. For example, formulae of LCF are limited to conjunctions of inequalities in order to preserve chain-completeness and hence admissibility for fixed point induction. In order to extend this principle to first order logic, admissibility tests over formulae must be performed, as in Cambridge LCF [Pau87]. Similarly, Fix-Logic cannot be consistently extended with intuitionistic implication or existential quantification, as shown in [CP92]. As for EL, Moggi's standard semantics needs higher order structure to interpret possibility, which is not available in categories of domains, unless many interesting predicates, such as "f is total" (which can be expressed in full EL as $\forall\, n.\, \langle x \Leftarrow f(n)\rangle\ \mathit{true}$), are cut off.

The aim of integrating logic and computation in a unified theory of semantics is pursued in *synthetic domain theory* [Ros86,Pho90,Hyl90,Tay91,RS], where logic is interpreted in an ambient category with set-like structure, typically a topos, and computation in a full reflective subcategory of the former, with all the closure properties required to perform domain-theoretic constructions. The leading example of this setting is the *effective topos* [Hyl82], with its many full subcategories of *PERs* [FMRS90].

A dual approach can be adopted in the framework of *axiomatic domain theory* [Fio94b, FP94]. Instead of identifying inside a topos a good category of domains to interpret computation, one can look "around" such a category for an appropriate topos to interpret logic. In [Fio94a], it is shown that every small model of a given axiomatization of domains has a full and faithful representation in a model of cpos and continuous functions living in a suitable intuitionistic set theory.

If the logical structure is provided by an ambient category $\mathcal{E}$, and the computational structure by a strong monad T defined on a subcategory $\mathcal{D}$ of $\mathcal{E}$, Moggi's standard semantics of EL requires an extension of T to $\mathcal{E}$. This can be done by a general construction as shown in section 4.6, provided $\mathcal{D}$ is fully reflective in $\mathcal{E}$, as it happens in both the synthetic and the axiomatic approaches mentioned above.

T-modal operators are proposed in [Mog91b] to give a uniform account of various program logics in a higher order setting. Let T be a monad on $\mathcal{E}$, and let $\top : 1 \to \Omega$ classify the predicates of the logic; a T-modal operator is a T-algebra $\alpha : T\Omega \to \Omega$. Given such an α, a predicate $\phi : A \to \Omega$ over values of type A can be lifted to one over computations, written $\Box_\alpha \phi$, as follows: $\Box_\alpha \phi \stackrel{\text{def}}{=} \alpha \circ T\phi : TA \to \Omega$.

In some cases, the notion of T-modal operator may be too strong: interesting operators may fail to satisfy the equations which make them algebras of a *monad*. Then, algebras of the underlying *functor* of T can be considered instead.

Consider, for example, the "standard" operator α which classifies $T\top : T1 \to T\Omega$ (if T does not preserve subobjects, the *image* of this morphism can be used). The unit and associative laws for such an α are given by special conditions, respectively on the unit and the multiplication of T, viz. that they are *cartesian* (see definition 4.5.3). As shown in chapter 5, the multiplication of the monad $TX \stackrel{\mathrm{def}}{=} (X \times S)^S$ for computation with side effects does not satisfy this property and, in fact, the standard α for this monad is not associative. Still, α is an interesting one: the modality $\Box_\alpha$ maps ϕ to the predicate which holds of all programs p such that, for all states $s : S$, $p(s) = \langle x, s' \rangle$ for some state s' and value x satisfying ϕ. Indeed, this modality can be used to interpret Hoare triples.

Monads and semantic modules

The problem of finding good structuring principles for programming language semantics has been addressed in various contexts. In the context of model theory, *institutions* [BG85], which use notions from category theory, provide a general framework for combining small specifications into descriptions of complex models. In the context of algebraic specification, [ST89] proposes a method for stepwise refinement of formally specified ML modules leading to correct program implementations. In the context of denotational semantics, [Mos88,Mos90] stress the importance of *auxiliary notation* as abstraction mechanisms to hide the detailed structure of the underlying data types from the semantic equations.

Here, we concentrate on the use of monads as a structuring mechanism in denotational semantics [Mog91c,CM93,CF94,LHJ95].

By introducing a layer of abstraction between programming languages and their concrete models, the computational metalanguage plays the role of an interface: to hide the details of implementation. However, if part of the mathematical structure is hidden, some extra work is needed to get from a semantic description given in terms of the metalanguage to a concrete model. Since the domains for interpreting programs may be quite complicated, it is crucial for usability of denotational semantics to develop tools for managing such complexity. Moggi's proposal ([Mog90a,Mog91c]) is to work stepwise towards complicated models by means of *semantic constructors*, which allow computational features to be adjoined and reasoned about one at a time.

The idea is to mimic the stepwise methodology for program development: viewing theories as specifications and models as implementations, the problem of finding an implementation of a complex specification $\mathcal{L}_2$ can be reduced to that of finding an implementation of a simpler specification $\mathcal{L}_1$ and a *constructor* [ST87] mapping models of $\mathcal{L}_1$ to models of $\mathcal{L}_2$.

In our setting, a specification is a theory $ML_T(\Sigma)$ of the computational metalanguage over a signature Σ. An implementation of such a specification is a cartesian closed category with a strong monad and additional structure to interpret Σ. Calling $Mod(\Sigma)$ the class of such objects, a semantic constructor is a function $\mathcal{F} : Mod(\Sigma_1) \to Mod(\Sigma_2)$.

Starting from a model of a simple metalanguage $ML_T(\Sigma_0)$, we can incrementally construct a model of $ML_T(\Sigma_n)$ by providing a sequence of signatures $\Sigma_i \subset \Sigma_{i+1}$, with $0 \leq i < n$, and constructors $\mathcal{F}_i : Mod(\Sigma_i) \to Mod(\Sigma_{i+1})$. In general, the latter involve a reinterpretation of the type constructor T (that is, a change of monad) and of the operations in Σ_i.

The core of a semantic constructor is a *monad constructor* [Mog90a], that is a function mapping monads to monads. On top of that, structure for interpreting one signature must be "converted" into structure for interpreting another. The effort here is to prove that the structure produced by a constructor satisfies the axioms of the theory. In order to establish such a result, it is useful to know which equations are *preserved* by a monad constructor. Similarly, to study the conservativity of certain constructors, it is useful to know which equations are *reflected*.

Logical frameworks can be used to support reasoning about properties of semantic constructors, such as the ones mentioned above. Availability of several implementations of type theories running on computers [CAB$^+$86,Mag92,LP92,DFH$^+$93], makes it tempting to adopt one as a tool to develop programming language semantics. A framework combining the Extended Calculus of Constructions (XCC) [Luo82] with the metalanguage for computational monads was recently proposed in [Mog95b].

Type theories have long been used for formalizing various branches of mathematics (as in the AUTOMATH tradition [deB70]) as well as for developing a general theory of representation of formal systems (as in the ELF tradition [HHP87]). They provide self-explaining primitive notation for formalizing mathematics, be it analysis [Jut77,CH85] or metatheories of various forms of calculi [Ber91,Alt92]. Generally, implementations provide machine-assistance in managing complicated syntax, environments for proof development and libraries.

Besides these pragmatical benefits, there may also be theoretical reasons in using powerful type theories to represent the constructions of denotational semantics. One case is the following.

A correspondence is often established, in categorical type theory, between theories of some formal calculus, with the appropriate notion of translation, and mathematical models, with the appropriate notion of morphism between models. Typical examples are the simply typed lambda calculus [LS86] and the higher order polymorphic lambda calculus [See87]. This correspondence can be exploited in the framework of *functorial semantics* [KR77] to characterize certain maps between categories of models as *relative interpretations* induced by theory translations. For example, relative interpretation of finite limit theories is characterized by the Gabriel-Ulmer duality established in [GU75]. Now, while there is a direct correspondence between theories of the computational metalanguage and cartesian closed categories with a strong monad, there is no good correspondence between theory translations (at least the kind of translations one needs to represent semantic constructors) and strong monad morphisms. However, the correspondence is restored when the metalanguage and its models are embedded into a type theory such as the Extended Calculus of Constructions. In chapter 6, we explore this matter in detail.

Summary of results and outline of the thesis

This thesis stems from three experiments with monads in denotational semantics: in the first, we use the computational lambda calculus to give an adequate model of ML exceptions; in the second, we use EL for reasoning about partial correctness of `while`-programs; in the third, we develop in the Extended Calculus of Constructions (XCC, [Luo82]) examples of semantic constructors and prove properties of constructed models formally in LEGO [LP92].

While working out these applications, general questions were raised, sometimes requiring the development of a piece of theory. For example, the involvement of non-trivial mixed variance constructions in models of exceptions raised the question of whether normalization is preserved if the simply typed lambda calculus is augmented with exceptions. Evaluation relations were used to define left and right rules for necessity in EL, which make the proof system more manageable; the problem arose of finding a standard interpretation of such relations in a first order setting. Implementation in type theory suggested that most semantic constructors can be presented as theory translations, provided a general enough syntax is adopted; the natural question was how to characterize such constructors *intrinsically.*

The body of the thesis is composed of six chapters: three of them present an application; each is preceded by a chapter introducing and developing the relevant theory.

Chapter 2 contains some basic categorical definitions and introduces the computational metalanguage and its models based on monads. Examples of computationally relevant monads are provided. We claim no originality here.

Chapter 3 studies the semantics of a small but non-trivial fragment of ML, including the exception handling mechanism. To our knowledge, no other denotational description of ML exceptions has been previously worked out.

We give a theory over the computational metalanguage, with operations to construct, raise and handle exceptions. An abstract model of this theory is a bi-cartesian closed category with an object of exceptions which is a solution to a suitable recursive domain equation. Recursion comes in because ML's exceptions may have parameters of arbitrary type and, in particular, parameters may be programs raising exceptions.

An analysis of the domain-theoretic constructions required to interpret the language reveals that models include the structure to interpret diverging computations and this suggests something that is not imediately clear from the operational semantics: programs with exceptions may fail to terminate even if no recursion is used. In particular, as shown in section 3.2, one can write a fixed point combinator for arrow types in the simply typed lambda calculus with exceptions. This failure of normalization was also noticed by M. Lillibridge in a yet unpublised paper [Lil95], which was unknown to us when we obtained this result.

The programming language is interpreted by a translation into the metalangauage. This interpretation is proven to be adequate with respect to a given operational semantics: a program e evaluates to a value v if and only if the translation of e is *provably equal* to the translation of v in the theory of the metalanguage. Part of the proof consists in defining appropriate *logical relations* between terms and their denotations in a concrete model in the category of cpos. Because of the recursive nature of computational types, such relations cannot be defined by induction on the structure of types and they are obtained instead by applying a technique suggested in [Pit].

There is a general pattern in the definition of logical relations for computational languages: two intertwined families are defined; members of one family, $\leq$, relate canonical terms and *values*, while members of the other, $\preceq$, relate general terms and *computations*. Relations in the first family between canonical terms of type $A \to B$ and values in the domain $[\![A \to B]\!] = [\![A]\!] \to T[\![B]\!]$, are defined "logically" by suitable conditions involving relations of the second family between programs of type B and computations in $T[\![B]\!]$. In section 3.6, $\preceq$ is defined "by hand" from $\leq$ by looking at the concrete structure of computational types. However, we believe that $\preceq$ should be described more abstractly as a canonical lifting of $\leq$ over computations. This is a matter of further research (see chapter 8).

Except for the notions of *relational structure* and *admissible action* used in the proof of section 3.6, all material in this chapter is original.

Chapter 4 presents Evaluation Logic and addresses two questions about the interaction between computational and logical structure: how to use the former to get a standard interpretation of program logics in a synthetic domain theory setting and how to get such an interpretation when higher order structure is not available.

Our presentation of the calculus includes equality and all connectives of first order predicate calculus, while the one in [Pit91] has only equality, conjunction and disjunction, and the one in [Mogb] includes no disjunction, existential quantification and possibility. Note that the rules for necessity are different in [Pit91] and in [Mogb]. We adopt the latter and call "standard" the corresponding version of the logic. In the choice of primitive and derived rules, our presentation matches the view of strong monads as triples $\langle T, \mathit{val}, \mathit{let}\rangle$, while the one in [Mogb] matches the more familiar view including unit, multiplication and strength.

We survey Pitts' semantics based on hyperdoctrines [Pit91], Moggi's first standard semantics [Mogb] and his second [Moga], which was introduced to capture monads for which the first would not work (e.g. the Plotkin powerdomain monad). Adopting the interpretation in [Mogb], which is used in chapter 5, we describe models of the full calculus.

As explained earlier, Moggi's standard semantics (both of the above) require higher order structure to interpret possibility and this is typically found in an ambient category in which the category of domains is fully reflective. We consider a construction to extend a monad T, defined on a category $\mathcal{C}$, to a monad $\tilde{T}$ on an ambient category $\mathcal{E}$ of which $\mathcal{C}$ is a fully reflective subcategory. We learnt this construction from Moggi but ours is the observation that the category of $\tilde{T}$-algebras is equivalent to that of T-algebras. This result (theorem 4.6.3) says that $\tilde{T}$ is, in some sense, a "minimal" extension of T. We also find abstract conditions allowing the extension to $\tilde{T}$ of the strength of T.

Although models of computation do live in categories with higher order logical structure, it is not clear why this structure should be *necessary* to get a standard interpretation of possibility and of the evaluation relations, as done in [Mogb]. We show how to obtain evaluation relations just from an *endofunctor* T and first order structure. Such relations can be integrated with evaluation modalities in a logic featuring *left* and *right* rules of the form:

$$\Box\text{\_left}\ \frac{\Gamma, \phi(M) \vdash \psi \quad \Gamma \vdash x \Leftarrow E}{\Gamma, [x \Leftarrow E]\ \phi(x) \vdash \psi} \qquad \Box\text{\_right}\ \frac{\Gamma, x \Leftarrow E \vdash \phi(x)}{\Gamma \vdash [x \Leftarrow E]\ \phi(x)}\ x \notin FV(\Gamma)$$

As we argue in chapter 4, such rules would simplify the structure of proofs in EL, where each modality is instead related with each logical operator by a separate rule.

For most notions of computation the modalities of EL are *definable* from evaluation relations by using first order quantification (see section 4.8), which means that, from our interpretation of such relations, a standard model of the logic is obtained in a first order framework, e.g. in a *logos* [FS90, 1.7]. We find the abstract conditions under which our definition of the evaluation relations corresponds to the one in [Mogb].

Our original contribution in this chapter consists in the theorems on extending monads to an ambient category (section 4.6) and the part on evaluation relations (section 4.8).

Chapter 5 sets Evaluation Logic to work on partial correctness.

Partial correctness specifications are formulae involving imperative programs and *assertions* about their states. Assertions are formulae which, unlike specifications, are interpreted "in a state." We consider assertions of a very general form: they may contain any term of the programming language, including expressions with loops and side effects. There are two reasons for allowing such generality: first, the assumption made in Hoare Logic that expressions have no computational content is violated by most common programming languages, which include devices such as function procedures or block expressions. Second, this level of generality provides a good ground for testing the power of the evaluation modalities as a mechanism for integrating logic and computation.

We study the monad $TX = (X \times S)_\perp^S$ for partial computations with side effects: we show that this monad satisfies the conditions required by the standard interpretation of [Mogb] and validate the special axiom $\Box\_val^*$ of section 4.1, which is used later to give a formal account of assertions in EL.

Two things should be remarked about these proofs. First, they are corollaries to more general theorems stating that the *monad constructors* $\mathcal{F}T \stackrel{\text{def}}{=} T(\_ \times S)^S$ preserves monads satisfying the desired properties. We believe that one obtains greater insight into a notion of computation from an analysis of constructors such as $\mathcal{F}$ than from the study of the corresponding monad in isolation. Second, EL is used to reason about several interacting notions of computation. For example, if T and Q are strong endofunctors, the necessity modality for their composite can be obtained by composition as follows:

$$[x \Leftarrow z]_{TQ}\, \phi(x) \;\cong\; [y \Leftarrow z]_T\, [x \Leftarrow y]_Q\, \phi(x),$$

where the modalities are indexed by the notion of computation adopted in the evaluation. In general, the strategy we adopt for reasoning about a notion of computation, say, to validate axioms, is to look at it at a lower level of abstraction by breaking it into more elementary pieces and then to prove the desired properties formally in the combined theory of such pieces. We shall call "modular" proofs that follow this strategy.

Next, we consider a version of EL for computation with side effects, called EL*se*, whose formulae are interpreted "in a state" and adopt it as a calculus of assertions. EL*se* is a theory in the version of Evaluation Logic of [Pit91]. Rather than working in a non-standard model of this theory (e.g. by interpreting predicates on a type A as subobjects of $[\![A]\!] \times S$) we translate it into standard EL with constants for explicit manipulation of states. We introduce suitable axioms for such constants and validate them for monads of the form $\mathcal{F}T$. Again, the proof is modular, in the sense discussed above. The soundness of interpretation of EL*se* in EL is then proven by formally deriving each inference rule of the former in the latter.

EL*se* can be used for reasoning about partial correctness. We generalize the familiar formalism of Hoare triples to include a simple form of *annotated* `while`-programs, that is terms obtained by alternating commands and assertions. Partial correctness specifications of possibly annotated `while`-programs are translated into ELse and the translation is shown to preserve the theorems of Hoare Logic.

Then, we take the text-book example of a formally specified program for integer division and prove its correctness in ELse. The proof requires fixed point induction to handle the recursive computations arising from while-loops. As explained earlier, the application of this rule in the presence of all connectives of first order logic includes a test of admissibility over formulae. We propose an improvement of the test presented in [Ten91] which recognizes more admissible formulae, e.g. the double negation of an admissible formula. Our proof of correctness is acknowledged in [Ten].

All material presented in this chapter is original.

Chapter 6 presents and develops the theory of semantic constructors introduced by E. Moggi in [Mog90a,Mog91c].

We introduce the notion of Σ-homomorphism (not in Moggi's presentation), which defines morphisms in the category of models of $ML_T(\Sigma)$. Σ-homomorphisms relate the interpretations of each constant of Σ in the source and target models and hence they can be used for studying semantic constructors from the point of view of the algebraic properties that they preserve and reflect.

To this effect, we consider constructors $\mathcal{F}$ endowed with families of Σ-homomorphisms $\mathcal{M} \to \mathcal{F}\mathcal{M}$, which we call "pointed." Inspired by (but more general than) the *parametric extensions* of [Mog90b], pointed constructors include the ones for exceptions and for resumptions presented in chapter 7. We show that equations involving terms of a fairly general class are preserved by pointed constructors (theorem 6.2.5). Axioms of theories of computation can be validated in constructed models by using this result. We also consider the dual issue of *conservativity* and show that equations (again subject to restrictions) which are satisfied in a model $\mathcal{F}\mathcal{M}$ are also satisfied in $\mathcal{M}$ (theorem 6.2.7).

Then, we study the "syntactic" presentation of constructors. This work stems from a joint paper with E. Moggi [CM93], where a higher order metalanguage, HML, is used to describe models of computation and translations of HML theories, also called *relative interpretations*, are used to describe semantic constructors. Our aim, here, is to give an intrinsic characterization of the constructors that can be represented in this fashion. Besides answering a mathematically natural question, this result may help finding more general type theories to capture more general constructors.

A natural setting for studying relative interpretation is *functorial semantics* [KR77], in which theories are viewed as categories with structure and models as structure-preserving functors, generally in the category of sets. Translations $\mathcal{T}_2 \to \mathcal{T}_1$ are also structure-preserving functors. Hence, a translation maps models of $\mathcal{T}_1$ to models of $\mathcal{T}_2$ by composition. Gabriel and Ulmer [GU75] studied these maps for finite limit theories. They showed that *locally finitely presentable categories* (see definition 6.5.3) are categories of models of such theories and that functors between such categories (satisfying certain conditions) correspond to relative interpretations. We establish a similar result for theories of HML.

We define models of HML, which are fibrations of a special form, called *$\lambda\overline{\omega}$-categories*. The soundness and completeness of HML with respect to interpretation in these models is proven. Then, we show that the category of HML theories is equivalent to the category $\lambda\overline{\omega}$-Cat of $\lambda\overline{\omega}$-categories and that interpretation of such theories correspond to morphisms in $\lambda\overline{\omega}$-Cat. Hence, we obtain a functorial semantics of HML, where the syntax disappears and theories are abstractly represented as $\lambda\overline{\omega}$-categories. Next, we show that the category $\mathit{Mod}(\mathcal{T})$ of models of a theory $\mathcal{T}$ is equivalent to the category $[T, \mathit{Sets}]_{lex}$ of set valued models of a finite limit theory T, which is locally finitely presentable. Moreover, we show that the functors $\mathit{Mod}(\phi) : \mathit{Mod}(\mathcal{T}_1) \to \mathit{Mod}(\mathcal{T}_2)$ induced by a relative interpretation $\phi : \mathcal{T}_2 \to \mathcal{T}_1$ satisfy the conditions required by the Gabriel-Ulmer duality. This result is used in theorem 6.8.12 to give a simple characterization of functors of the form $\mathit{Mod}(\phi)$.

The definition of semantic constructor is from [Mog90b]. Sections 6.3 and 6.4 use material from [CM93]. Section 6.5 surveys standard results in functorial semantics. Section 6.6 reviews some notions of fibred category theory used for modelling HML. All the rest, including what is described above, is our own contribution.

Chapter 7 develops in LEGO the semantic constructors for exceptions and for resumptions described in [CM93] and uses the logic provided by the underlying type theory, the Extended Calculus of Constructions, to prove properties of such constructors formally.

The type universe of XCC provides an intuitionistic set theory in which models of computation are embedded, as in synthetic domain theory. "Domains" are inhabitants of a type `Dom` and the embedding is provided by an externalization map `E:Dom->Type`. From this data one can define suitable types and terms whose interpretation in a model of the type theory determines a full internal subcategory Dom in the ambient category of types. All the constructions developed in the applications refer to this category of discourse.

Categorical structure in Dom is represented by suitable types whose inhabitants we call *structures*. The computational metalanguage is encoded in the type theory and a formal interpretation function is defined, mapping the syntax to Dom. This function is parametric in the appropriate structure, that is: products, exponentials and a monad.

Semantic constructors are represented as functions returning structure for interpreting one theory of the metalanguage from structure for interpreting another. We present two constructors from [CM93], one for exceptions and one for resumptions. The correctness of the first is proven in LEGO by showing that it maps monads to monads. The second is used to define a model of parallel computation with interleaving.

Resumptions are constructed on top of a model featuring operators of fixed point and nondeterministic choice. The constructor redefines these operations to adapt them to the new notion of computation, where other operations for manipulating resumptions become available. From these operations, we define an operator of parallel composition of which we give a formal proof of commutativity. The proof uses properties of the abovementioned operations, e.g. the *uniformity* of fixed points (see section 7.2). Hence, we prove lemmas showing that such properties are preserved by the constructor of resumptions. Finally, we make sure that all the pieces of structure being used are consistent with each other by looking for them in a concrete model of the type theory based on PERs.

The definitions of the exception and resumption constructors, and of the operator of parallel composition are based on similar definitions in HML due to Moggi. All the rest in this chapter is original.

Chapter 8 contains directions for further research.

Prerequisites.

- Category theory.
- Basic logic and type theory.
- Some feeling for LEGO (for chapter 7 and the appendices only).

2 The Computational Lambda Calculus

In this chapter we present a metalanguage $ML_T(\Sigma)$, called *computational lambda calculus* [Mog91b], featuring computational types and polymorphic operations (á la ML) in a signature Σ to describe specific notions of computation. The first two sections contain definitions and discussion of the relevant mathematical concepts, in particular that of *strong monad* upon which the semantics of the metalanguage is based. $ML_T(\Sigma)$ is introduced in section 2.3, while section 2.4 describes its interpretation in a cartesian closed category with a strong monad and additional Σ structure. In the last section we present some examples of computationally relevant monads.

2.1 Strong endofunctors

Definition 2.1.1 *A* monoidal category $\mathcal{C} = \langle \mathcal{C}, \otimes, 1, \alpha, \lambda, \rho \rangle$ *consists of a category* $\mathcal{C}$ *with a bifunctor* $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ *which is associative up to a natural isomorphism* $\alpha : (\_\otimes\_)\otimes\_ \cong \_\otimes(\_\otimes\_)$ *and an object* 1 *of* $\mathcal{C}$ *which is left and right unit for* $\otimes$ *up to natural isomorphisms* $\lambda : 1\otimes\_ \cong \_$ *and* $\rho : \_\otimes 1 \cong \_$*, subject to the coherence conditions described in [EK66, II.1].*

Definition 2.1.2 *A monoidal category* $\mathcal{C}$ *is* closed *if each functor* $\_ \otimes B : \mathcal{C} \to \mathcal{C}$ *has a specified right adjoint* $[B, \_\,]$.

We write $\eta^B_X : X \to [B, X\otimes B]$ for the unit of such an adjunction, $\epsilon^B_X : [B, X]\otimes B \to X$ for the counit (i.e. evaluation) and $\Lambda f : X \to [B, Y]$ for the transpose of $f : X \otimes B \to Y$. If $\mathcal{C}$ is *symmetric* and closed, it is *biclosed*, that is also every $B \otimes \_$ has a right adjoint.

Let $\mathcal{C}$ be a monoidal closed category. The bijection $\ulcorner\_\urcorner : \mathcal{C}(A,B) \cong \mathcal{C}(1,[A,B])$ is defined as follows: the *name* $\ulcorner f \urcorner : 1 \to [A,B]$ of a morphism $f : A \to B$ is the element $\Lambda(f \circ \lambda_A)$. Conversely, the *arrow* $\urcorner g \ulcorner : A \to B$ of an element $g : 1 \to [A,B]$ is the morphism $(\Lambda^{-1} g) \circ \lambda_A^{-1}$. Then $\urcorner\ulcorner f \urcorner\ulcorner = f$ and $\ulcorner\urcorner g \ulcorner\urcorner = g$. We write $L_{A,B,C} : [B,C] \otimes [A,B] \to [A,C]$ for the transpose of $[B,C] \otimes [A,B] \otimes A \xrightarrow{id \otimes \epsilon} [B,C] \otimes B \xrightarrow{\epsilon} C$. Then, $L(\ulcorner f \urcorner, \ulcorner g \urcorner) = \ulcorner f \circ g \urcorner$.

Definition 2.1.3 *A* strong endofunctor $T = \langle T, st \rangle$ *on a monoidal closed category* $\mathcal{C}$ *consists of a map on objects* $T : obj(\mathcal{C}) \to obj(\mathcal{C})$ *and a family of morphisms* $st_{A,B} : [A,B] \to [TA,TB]$*, the* functorial strength *of* T*, satisfying the following equations:*

$$\ulcorner id \urcorner = st \circ \ulcorner id \urcorner \tag{2.1}$$

$$st \circ L = L \circ (st \otimes st). \tag{2.2}$$

The *underlying functor* $T : \mathcal{C} \to \mathcal{C}$ of a strong endofunctor $T = \langle T, st \rangle$ on $\mathcal{C}$ maps A to TA and $f : A \to B$ to $\urcorner (st_{A,B} \circ \ulcorner f \urcorner) \ulcorner : TA \to TB$.

Every monoidal closed category can be *enriched* over itself by taking the internal homs $[A,B]$ as hom-objects, names of identities $\ulcorner id_A \urcorner : 1 \to [A,A]$ as identity elements and L as composition [EK66, Theorem 5.2]. The following is well known:

Proposition 2.1.4 *Every strong endofunctor on a monoidal closed* $\mathcal{C}$ *can be viewed as a* $\mathcal{C}$*-functor on the category* $\mathcal{C}$ *enriched over itself and viceversa. The underlying functor of the* $\mathcal{C}$*-functor, as defined in [Kel82], is the same as that of the strong functor, as above.*

Now we consider a different notion of strength and describe the correspondence with the one above in the more general setting of a $\mathcal{V}$-category $\mathcal{C}$, for $\mathcal{V}$ monoidal closed. $\mathcal{C}$ is said to be *tensored* over $\mathcal{V}$ if for every $X \in \mathcal{V}$ and $A \in \mathcal{C}$ there exists a *tensor product* $X \otimes A \in \mathcal{C}$ and $\mathcal{V}$-natural isomorphisms

$$\mathcal{C}(X \otimes A, \_) \cong [X, \mathcal{C}(A, \_)].$$

Note that such isomorphisms give rise to an adjunction $\_ \otimes A \dashv \mathcal{C}(A, \_) : \mathcal{V} \rightharpoonup \mathcal{C}_0$ between $\mathcal{V}$ and the *underlying* category of $\mathcal{C}$, defined by

$$\mathcal{C}_0(X \otimes A, B) \stackrel{\text{def}}{=} \mathcal{V}(1, \mathcal{C}(X \otimes A, B)) \cong \mathcal{V}(1, [X, \mathcal{C}(A, B)]) \cong \mathcal{V}(X, \mathcal{C}(A, B)).$$

The above adjunction yields natural isomorphisms $\alpha_{X,Y,A} : X \otimes (Y \otimes A) \to (X \otimes Y) \otimes A$ in $\mathcal{C}_0$ as implied by Yoneda and the following natural bijections:

$$\frac{\dfrac{\dfrac{\dfrac{X \otimes (Y \otimes A) \longrightarrow B}{X \longrightarrow \mathcal{C}(Y \otimes A, B)}}{X \longrightarrow [Y, \mathcal{C}(A, B)]}}{X \otimes Y \longrightarrow \mathcal{C}(A, B)}}{(X \otimes Y) \otimes A \longrightarrow B}$$

Theorem 2.1.5 ([Koc72, 1.3]) *Let $\mathcal{V}$ be a symmetric monoidal closed category, let $\mathcal{A}$ and $\mathcal{B}$ be $\mathcal{V}$-categories tensored over $\mathcal{V}$ and let $\mathcal{A}_0$ and $\mathcal{B}_0$ be their underlying categories. There is a one-to-one correspondence between $\mathcal{V}$-functors $\mathcal{A} \to \mathcal{B}$ and functors $T : \mathcal{A}_0 \to \mathcal{B}_0$ equipped with a natural family of morphisms $t_{X,B} : X \otimes TB \to T(X \otimes B)$ satisfying the following diagrams:*

$$\begin{tikzcd} 1 \otimes TB \arrow[r, "t"] \arrow[d, "\lambda"'] & T(1 \otimes B) \arrow[dl, "T\lambda"] \\ TB & \end{tikzcd} \qquad \begin{tikzcd} X \otimes (Y \otimes TB) \arrow[r, "id \otimes t"] \arrow[d, "\alpha"'] & X \otimes T(Y \otimes B) \arrow[r, "t"] & T(X \otimes (Y \otimes B)) \arrow[d, "T\alpha"] \\ (X \otimes Y) \otimes TB \arrow[rr, "t"] & & T((X \otimes Y) \otimes B) \end{tikzcd}$$

In the light of the above result, t is often called the *tensorial* strength of T. We can see the correspondence between tensorial and functorial strengths through Yoneda:

$$\frac{\dfrac{\dfrac{[B, C] \longrightarrow [TB, TC]}{\mathcal{V}(X, [B, C]) \longrightarrow \mathcal{V}(X, [TB, TC])}}{\mathcal{C}_0(X \otimes B, C) \longrightarrow \mathcal{C}_0(X \otimes TB, TC)}}{X \otimes TB \longrightarrow T(X \otimes B)}$$

The constructions are as follows: morphisms $st_{A,B} : [A,B] \to [TA,TB]$ are obtained from t by transposing $[A,B] \otimes TA \xrightarrow{t} T([A,B] \otimes A) \xrightarrow{T\epsilon} TB$. Vice versa, $t_{X,B}$ is obtained from st as $X \otimes TB \xrightarrow{\eta^B \otimes id} [B, X \otimes B] \otimes TB \xrightarrow{\Lambda^{-1} st} T(X \otimes B)$. We refer to the aforementioned paper for the proofs that the relevant diagrams commute.

From the above discussion we obtain an alternative definition of strong endofunctor $T = \langle T, t\rangle$ on a symmetric monoidal closed category $\mathcal{C}$ as consisting of a (conventional) functor $T : \mathcal{C} \to \mathcal{C}$ and a natural transformation $t_{A,B} : A \otimes TB \to T(A \otimes B)$ satisfying the commutative diagrams of 2.1.5. In the following definition, as in most of this thesis, we consider strengths defined in *cartesian closed* categories, where the isomorphisms α, λ and ρ are the canonical ones.

Definition 2.1.6 ([Mog90b]) *Objects of the* category of strong endofunctors *are triples $\langle \mathcal{C}, T, t\rangle$, where $\langle T, t\rangle$ is a strong endofunctor on a cartesian closed category $\mathcal{C}$. Morphisms $\langle \mathcal{C}, T, t^T\rangle \to \langle \mathcal{D}, S, t^S\rangle$ are pairs (U, σ), where the functor $U : \mathcal{C} \to \mathcal{D}$ preserves the cartesian closed structure on the nose and $\sigma : UT \dot{\to} SU$ is a natural transformation making the following commute:*

$$
\begin{array}{ccc}
UA \times UTB & \xrightarrow{Ut^T} & UT(A \times B) \\
{\scriptstyle id \times \sigma}\downarrow & & \downarrow{\scriptstyle \sigma} \\
UA \times SUB & \xrightarrow[t^S]{} & S(UA \times UB)
\end{array}
$$

The equation expressing the diagram above in terms of the functorial strength is: $[UTA, \sigma_B] \circ Ust^T = [\sigma_A, SUB] \circ st^S$, that is, $\sigma_B(Ust_{A,B}(f,w)) = st_{UA,UB}(f, \sigma_A w)$. This is an instance of diagram 2.6 of section 2.2 and it expresses the fact that σ is $\mathcal{C}$-enriched. The condition that U preserves the universal structure *on the nose* is only to simplify the treatment, so that, for example, $UA \times UTB = U(A \times TB)$.

Remark. Strong endofunctors are the objects of a 2-category the 2-cells of which, $\phi : (U, \sigma) \Rightarrow (V, \rho) : \langle \mathcal{C}, T, t^T \rangle \to \langle \mathcal{D}, S, t^S \rangle$, are natural transformations $\phi : U \dot{\to} V$ such that $S\phi \circ \sigma = \rho \circ \phi T$.

Remark. Two strong endofunctors $T = \langle T, st^T \rangle$ and $S = \langle S, st^S \rangle$ defined on the same monoidal closed category $\mathcal{C}$ have an obvious composite: $TS = \langle TS, st^T \circ st^S \rangle$, while their tensorial strengths are composed as follows to obtain:

$$t^{TS} \stackrel{\text{def}}{=} X \otimes TSB \xrightarrow{t^T_{X,SB}} T(X \otimes SB) \xrightarrow{Tt^S_{X,B}} TS(X \otimes B).$$

2.2 Strong monads

In section 2.1 we showed an equivalence between two definitions of strong endofunctor: the first involving a map of objects T and a family st of morphisms, the second involving a functor T and a natural transformation t. The former is more syntactic, in the sense that T may be viewed as a type constructor and st as a family of uniformly typed operations in the internal language of a category. Note that a map on objects T plus the operations of t do not suffice to make T a functor.

In this section we follow the same approach for *strong monads* on which Moggi's computational metalanguage is based. The minimal version of this calculus described in [Mog91b] requires only the structure of a monad; however, a strength is needed to interpret terms with more than one free variable. First we introduce strong monads T as made of type constructors and operations defined in cartesian closed categories $\mathcal{C}$. This definition is adopted in chapter 6 to represent monads in a type theory. Then we show in the internal language of $\mathcal{C}$ (with "signature" T) that strong monads correspond to monads *over* $\mathcal{C}$, which look more familiar to the category theorist. In the following section, we present Moggi's computational metalanguage as a sugared version of the language of strong monads.

Definition 2.2.1 *A* strong monad $M = \langle T, val, let \rangle$ *on a cartesian closed category* $\mathcal{C}$ *consists of a map on objects* $T : obj(\mathcal{C}) \to obj(\mathcal{C})$ *and two families of morphisms* $val_A : A \to TA$ *and* $let_{A,B} : [A, TB] \times TA \to TB$ *such that, for any* $f : A \to TB$ *and* $g : B \to TC$*, the following diagrams commute:*

$$
\begin{tikzcd}
1 \arrow[r, "\ulcorner val \urcorner"] \arrow[d, "\ulcorner id \urcorner"'] & {[A, TA]} \arrow[dl, "\Lambda let"] \\
{[TA, TA]} &
\end{tikzcd}
\qquad
\begin{tikzcd}
1 \times A \arrow[r, "\lambda"] \arrow[d, "\ulcorner f \urcorner \times val"'] & A \arrow[d, "f"] \\
{[A, TB] \times TA} \arrow[r, "let"'] & TB
\end{tikzcd}
$$

$$
\begin{tikzcd}
1 \times (1 \times TA) \arrow[r, "\ulcorner g \urcorner \times (let \circ (\ulcorner f \urcorner \times id))"] \arrow[d, "\Lambda(let \circ (\ulcorner g \urcorner \times f)) \times \lambda"'] & {[B, TC] \times TB} \arrow[d, "let"] \\
{[A, TC] \times TA} \arrow[r, "let"'] & TC
\end{tikzcd}
$$

Note that *let* is an internal version of the lifting operation defined for *Kleisli triples* (another presentation of monads; see [BW85]).

In the following, we sometimes use lambda notation to address morphisms of a cartesian closed category $\mathcal{C}$. This is fairly standard practice in category theory, where one speaks of the *internal language* of $\mathcal{C}$, as in [LS86, I.10.6]. In particular, our category is endowed with families of morphisms $val_A : A \to TA$ and $let_{A,B} : (A \to TB) \times TA \to TB$ satisfying the following equations: for all $x : A$, $z : TA$, $f : A \to TB$ and $g : B \to TC$,

$$let_{A,A}\ val_A\ z \;=\; z \tag{2.3}$$

$$let_{A,B}\ f\ (val_A\ x) \;=\; fx \tag{2.4}$$

$$let_{B,C}\ g\ (let_{A,B}\ f\ z) \;=\; let_{A,C}\ (\lambda\, x : A.\, let_{B,C}\ g\ (fx))\ z \tag{2.5}$$

where "$let\ f\ z$" stands for $let(f,\, z)$. Moreover we shall write "$let\ f$" for $(\lambda\, z.\, let\ f\ z)$ and, in general, assume left associativity in unbracketed expressions. These equations are just the Curry-ed version of the diagrams in definition 2.2.1.

Proposition 2.2.2 *A strong monad $M = \langle T, val, let \rangle$ has an* underlying strong endofunctor *with the same map T on objects and strength*

$$st_{A,B} : [A, B] \xrightarrow{[A,val]} [A, TB] \xrightarrow{\Lambda let} [TA, TB].$$

Proof. The above diagrammatical description of st translates in the internal language as $st_{A,B}\ f \stackrel{\text{def}}{=} let_{A,B}\,(\lambda\, x : A.\, val_B(fx))$. Then, equation 2.1 follows immediately from axiom 2.3: $st\,(\lambda\, x.\, x) \stackrel{\text{def}}{=} let\,(\lambda\, x.\, val\, x) = let\ val = \lambda\, z.\, z$.

Similarly, let $g : A \to B$ and $f : B \to C$; since $L\, f\, g = \lambda\, x.\, f(gx)$, equation 2.2 follows from 2.4 and 2.5:

$$\begin{aligned}
& L\,(st_{B,C}\, f)\ (st_{A,B}\, g)\, z \\
& = st_{B,C}\, f\ (st_{A,B}\ g\ z) \\
& = let_{B,C}\ (\lambda\, y : B.\, val_C\ (fy))\ (let_{A,B}\,(\lambda\, x : A. val_B\,(gx))\, z) \\
& = let_{A,C}\ (\lambda\, x : A.\, let_{B,C}\ (\lambda\, y : B.\, val_C\ (fy))\ (val_B\ (gx)))\, z \\
& = let_{A,C}\ (\lambda\, x : A.\ val_C\ (f(gx)))\, z \\
& = st_{A,C}\,(\lambda\, x : A.\, f(gx))\, z \\
& = st_{A,C}\ (L\, f\, g)\, z.
\end{aligned}$$

□

Remember that a $\mathcal{C}$-natural transformation $\sigma : (T, st^T) \to (S, st^S)$ is a family of morphisms $\sigma_A : TA \to SA$ such that:

$$\begin{tikzcd}
{[A,B]} \arrow[r, "st^T"] \arrow[d, "st^S"'] & {[TA,TB]} \arrow[d, "{[TA,\sigma_B]}"] \\
{[SA,SB]} \arrow[r, "{[\sigma_A,SB]}"'] & {[TA,SB]}
\end{tikzcd} \tag{2.6}$$

Proposition 2.2.3 *Let $\mathcal{C}$ be a cartesian closed category; there is a one-to-one correspondence between strong monads $\langle T, val,\ let \rangle$ on $\mathcal{C}$ as defined in 2.2.3 and $\mathcal{C}$-monads $\langle T, st, \eta, \mu \rangle$.*

Proof. We sketch the direction from strong monads to $\mathcal{C}$-monads. From the following constructions: $\eta_A \stackrel{\text{def}}{=} val_A : A \to TA$ and $\mu_A \stackrel{\text{def}}{=} let_{TA,A}(\lambda z : TA.\, z) : T^2A \to TA$, unit and associativity laws as well as naturality follow from routine calculations. We show $\mathcal{C}$-enrichment. In the case of η, diagram 2.6 corresponds to:

$$\begin{tikzcd}
{[A,B]} \arrow[r, "{[A,val]}"] \arrow[d, "{[A,val]}"'] & {[A,TB]} \\
{[A,TB]} \arrow[r, "\Lambda let"'] & {[TA,TB]} \arrow[u, "{[val,TB]}"']
\end{tikzcd}$$

which commutes because $[val, TB] \circ \Lambda let = id_{[A,TB]}$, that is, in the internal language, $\lambda x : A.\, let_{A,B}\, f\, (val_A\, x) = f$. Similarly, 2.6 specializes for μ as follows:

$$\begin{tikzcd}
{[A,B]} \arrow[r, "st^T"] \arrow[d, "st^T"'] & {[TA,TB]} \arrow[r, "st^T"] & {[T^2A,T^2B]} \arrow[d, "{[T^2A,\mu_B]}"] \\
{[TA,TB]} \arrow[rr, "{[\mu_A,TB]}"'] & & {[T^2A,TB]}
\end{tikzcd}$$

which, for $f : A \to B$ and $w : T^2A$ amounts to:

$$\begin{aligned}
&\mu_B(st_{TA,TB}\,(st_{A,B}\, f)\, w) \\
&= let_{TB,B}(\lambda z : TB.\, z)\,(let_{TA,TB}\,(\lambda z : TA.\, val_{TB}\,(st_{A,B}\, f\, z))\, w) \\
&= let_{TA,B}(\lambda z : TA.\, let_{TB,B}\,(\lambda z : TB.\, z)\,(val_{TB}\,(st_{A,B}\, f\, z)))\, w \\
&= let_{TA,B}\,(st_{A,B}\, f)\, w \\
&= let_{TA,B}\,(\lambda z : TA.\, let_{A,B}\,(\lambda x : A.\,(val_B(fx))\, z)\, w \\
&= let_{A,B}\,(\lambda x : A.\, val_B\,(fx))\,(let_{TA,A}\,(\lambda z : TA.\, z)\, w) \\
&= st_{A,B}\, f\,(\mu_A w).
\end{aligned}$$

$\square$

From the correspondence established by Theorem 2.1.5 we get an alternative view of a strong monad in a cartesian closed category $\mathcal{C}$ as a 4-tuple $\langle T, t, \eta, \mu\rangle$ consisting of a strong endofunctor $\langle T, t\rangle$ and a monad $\langle T, \eta, \mu\rangle$ such that:

$$
\begin{tikzcd}
A \times B \arrow[r, "id \times \eta"] \arrow[d, "\eta"'] & A \times TB \arrow[dl, "t"] \\
T(A \times B) &
\end{tikzcd}
\qquad
\begin{tikzcd}
A \times T^2B \arrow[r, "t"] \arrow[d, "id \times \mu"'] & T(A \times TB) \arrow[r, "Tt"] & T^2(A \times B) \arrow[d, "\mu"] \\
A \times TB \arrow[rr, "t"] & & T(A \times B)
\end{tikzcd}
\tag{2.7}
$$

Such diagrams follow from 2.3, 2.4 and 2.5, for η and μ defined as in the proof of Proposition 2.2.3 and $t_{A,B} \stackrel{\text{def}}{=} A \times TB \stackrel{(\Lambda val)\times id}{\longrightarrow} [B, T(A \times B)] \times TB \stackrel{let}{\longrightarrow} T(A \times B)$, that is: $t(x, w) \stackrel{\text{def}}{=} let\,(\lambda\, y.\, val(x, y))\, w$.

Definition 2.2.4 *Objects of the* category of strong monads *are 5-tuples* $\langle \mathcal{C}, T, t, \eta, \mu\rangle$, *where* $\langle T, t\rangle$ *is a strong endofunctor on a cartesian closed category* $\mathcal{C}$ *and* $\langle T, \eta, \mu\rangle$ *is a monad on* $\mathcal{C}$ *such that* t, η *and* μ *satisfy 2.7. Morphisms from* $\langle \mathcal{C}, T, t^T, \eta^T, \mu^T\rangle$ *to* $\langle \mathcal{D}, S, t^S, \eta^S, \mu^S\rangle$ *are morphisms* (U, σ) *of the underlying strong endofunctors such that:*

$$
\begin{tikzcd}
UA \arrow[r, "\eta^S"] \arrow[d, "U\eta^T"'] & SUA \\
UTA \arrow[ur, "\sigma"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
UT^2B \arrow[r, "\sigma"] \arrow[d, "U\mu^T"'] & SUTB \arrow[r, "S\sigma"] & S^2UB \arrow[d, "\mu^S"] \\
UTB \arrow[rr, "\sigma"] & & SUB
\end{tikzcd}
$$

In [Mog90b] strong monad morphisms are presented in terms of *val* and *let*. In this case, given $x : UX$, $f : UX \to UTY$ and $w : UTX$, the equations are: $val\, x = \sigma(U val\,(x))$ and $\sigma_Y(U let_{X,Y}\; f\; w) = let_{UX,UY}\; (\sigma_Y \circ f)\; (\sigma_X w)$.

Remark. Monad morphisms (U, σ) are similarly defined in [Str72], but with σ reversed. We adopt the above version because it simplifies the theory of semantic constructors discussed in section 6.2.

Remark. Strong monads are objects of a 2-category where 2-cells $\phi : (U, \sigma) \Rightarrow (V, \rho)$ are the same as those of the underlying 2-category of strong endofunctors.

Remark. Unlike strong endofunctors, strong monads compose only occasionally. Two monads $\langle T, \eta^T, \mu^T \rangle$ and $\langle S, \eta^S, \mu^S \rangle$ on a category $\mathcal{C}$ give rise to a composite $\langle TS, \eta^{TS}, \mu^{TS} \rangle$ on $\mathcal{C}$ when a *distributive law* λ satisfying suitable diagrams is provided (see [BW85, 9.2]). In that case, $\eta^{TS} = T\eta^S \circ \eta^T$ and $\mu^{TS} = T\mu^S \circ \mu^T \circ T\lambda S$. The same situation holds for *strong* monads, for which λ is also required to be $\mathcal{C}$-natural.

2.3 The computational lambda calculus

In [Mog91b], Moggi proposed the use of monads for modeling computations and a formal metalanguage, the computational lambda calculus $ML_T(\Sigma)$, to describe denotations of programs. The signature Σ, in which the metalanguage is parametric, contains operations for describing specific notions of computation; in chapter 3, for example, we present a metalanguage with operations for raising and handling exceptions. Singling out such operations is convenient in view of the modular approach to denotational semantics that we explore in in chapter 6. Here, we present $ML_T(\Sigma)$ as a special form of lambda calculus and in the next section we define its interpretation in a cartesian closed category equipped with a strong monad and Σ-structure.

First we extend the usual definition of typed lambda calculus to capture a family of languages whose signatures may include *type constructors* and *polymorphic operations.* We write $\mathcal{L}(\Sigma)$ for a generic member of this family with signature Σ.

Let Σ_τ be a collection of constant type symbols K, each associated with a natural number $\alpha_\tau(K)$ called its *arity.* Let χ be a collection of type variables. We call Σ_τ*-polytypes*, dropping "Σ_τ" when possible, the expressions freely generated from the following rules:

A1. 1 is a polytype;

A2. $X \in \chi$ is a polytype;

A3. if $K \in \Sigma_\tau$, $\alpha_\tau(K) = n$ and $\tau_1, \dots \tau_n$ are polytypes, then $K(\tau_1, \dots \tau_n)$ is a polytype (brackets are omitted when $n = 0$, in which case K is called "nullary");

A4. if σ and τ are polytypes, then so is $\sigma \times \tau$;

A5. if σ and τ are polytypes, then so is $\sigma \to \tau$.

By writing a polytype as $\tau(X_1, \dots X_n)$, we mean that *all* type variables of τ are in $X_1, \dots X_n$. By Σ_τ-*types* we mean Σ_τ-polytypes with no variables. We call *closed type scheme* a nonempty list of polytypes $\tau_1(X_1, \dots X_n), \dots \tau_m(X_1, \dots X_n), \tau(X_1, \dots X_n)$, written $\forall X_1, \dots X_n.\, \tau_1, \dots \tau_m \longrightarrow \tau$ ($\forall X_1, \dots X_n.\, \tau$ when $m = 0$).

A *signature* of a typed lambda calculus $\mathcal{L}(\Sigma)$ is a 4-tuple $\Sigma = (\Sigma_\tau, \alpha_\tau, \Sigma_\epsilon, \alpha_\epsilon)$ where Σ_τ and α_τ are as above, Σ_ϵ is a collection of operation symbols and α_ϵ associates with each $op \in \Sigma_\epsilon$ a closed type scheme called its *arity*. We write $op : \forall X_1, \dots X_n.\, \tau_1, \dots \tau_m \longrightarrow \tau$ when $\alpha_\epsilon(op) = \forall X_1, \dots X_n.\, \tau_1, \dots \tau_m \longrightarrow \tau$.

A typed lambda calculus $\mathcal{L}(\Sigma)$ consists of collections of *types*, typed *terms* and *equations* between terms, each satisfying the closure conditions specified below. Let $\Sigma = (\Sigma_\tau, \alpha_\tau, \Sigma_\epsilon, \alpha_\epsilon)$; the types of $\mathcal{L}(\Sigma)$ are Σ_τ-types as defined above. Let χ_τ be a collection of varibles, one for each type τ. The terms of $\mathcal{L}(\Sigma)$ are freely generated by the following rules: writing $M : \tau$ for "M is a term of type τ,"

B1. $* : 1$;

B2. if $x \in \chi_\tau$, then $x : \tau$;

B3. if $op : \forall X_1, \dots X_n.\, \tau_1, \dots \tau_m \longrightarrow \tau$ is an operation in Σ_ϵ, $\sigma_1, \dots \sigma_n$ are types, and $M_j : \tau_j(\sigma_1, \dots \sigma_n)$ for $j = 1, \dots m$, then $op_{\sigma_1, \dots \sigma_n}(M_1, \dots M_m) : \tau(\sigma_1, \dots \sigma_n)$;

B4. if $M : \sigma$ and $N : \tau$, then $\langle M, N\rangle : \sigma \times \tau$;

B5. if $M : \sigma \times \tau$, then $\pi_1(M) : \sigma$ and $\pi_2(M) : \tau$;

B6. if $M : \sigma \to \tau$ and $N : \sigma$, then $MN : \tau$;

B7. if $x \in \chi_\sigma$ and $M : \tau$, then $\lambda\, x : \sigma.\, M : \sigma \to \tau$.

When the types are understood we shall drop the indices of the constant operations. We shall also drop the parentheses when the precedence is understood. The usual notions of free and bound variables, and of substitution extend immediately to $\mathcal{L}(\Sigma)$. Rather than decorating variables with their type, we shall use *typing contexts* to type-check terms. A typing context Γ is a list $\Gamma = (x_1, \ldots x_n)$ of variables, where no x_i is repeated. Such a list is written $(x_1 : \tau_1, \ldots x_n : \tau_n)$, with $x_i \in \chi_{\tau_i}$, to make the type of each variable explicit. We write $\Gamma \vdash M : \tau$ when M is a term of type τ in $\mathcal{L}(\Sigma)$ whose free variables are in context Γ and call Γ a context for M.

Note that there are no polymorphic terms in $\mathcal{L}(\Sigma)$: although, as we shall see in the next section, operations in the signatures we consider may have a polymorphic intended meaning, polytypes and type schemes live only in the metatheory. Formally, we decorate operations with type indices so as to type-check terms such as nil_A where $\mathit{nil} : \forall X.\mathit{list}(X)$. Informally, we drop indices when we can.

Equations have the form $M =_\Gamma N$, where M and N are terms of same type in context Γ. The collection of equations of $\mathcal{L}(\Sigma)$ is called the *theory* of $\mathcal{L}(\Sigma)$. Such a theory is required to be closed under the usual inference rules of typed lambda calculi with unit and product types (see [LS86]). Theories include a special set of *axioms*, typically involving constants in Σ, such that anything in the theory derives from the axioms via the inference rules. When $M =_\Gamma N$ is in the theory of $\mathcal{L}(\Sigma)$, we write $\Gamma \vdash M = N$.

Equations are indexed by contexts in order for theories to make sense of possible *empty* types. For example, let $\mathcal{L}(\Sigma)$ contain no closed terms of type $\emptyset \in \Sigma_\tau$ and let the following be an axiom:

$$x : \tau, y : \tau \vdash \lambda\, z : \emptyset.\, x = \lambda\, z : \emptyset.\, y.$$

From this equation, $x : \tau, y : \tau, z : \emptyset \vdash x = y$ follows. Interpreting $\emptyset$ in *Sets* as the empty set, the theory equates any two elements of an arbitrary set τ *provided* an element of the empty set can be found. If the indices were not there, the proviso would be lost, and soundness lost with it.

Let Σ_τ be a collection of constant type symbols not including T and let α_τ be defined on $\Sigma_\tau \cup \{T\}$, with $\alpha_\tau(K) = 0$ for $K \in \Sigma_\tau$ and $\alpha_\tau(T) = 1$. Let Σ_ϵ be a collection of operation symbols not including *val* and *let*, let α_ϵ map them to closed type schemes formed from polytypes over $\Sigma_\tau \cup \{T\}$ and let $\bar{\alpha}_\epsilon$ extend α_ϵ to $\Sigma_\epsilon \cup \{val, let\}$ with $val : \forall X.\, X \longrightarrow TX$ and $let : \forall X, Y.\, (X \to TY), TX \longrightarrow TY$.

Definition 2.3.1 *A* computational lambda calculus*, or* computational metalanguage*, $ML_T(\Sigma_\tau, \Sigma_\epsilon, \alpha_\epsilon)$ is a typed lambda calculus $\mathcal{L}(\Sigma_\tau \cup \{T\}, \alpha_\tau, \Sigma_\epsilon \cup \{val, let\}, \bar{\alpha}_\epsilon)$ with Σ_τ, Σ_ϵ, α_ϵ, α_τ, and $\bar{\alpha}_\epsilon$ as above and with axiom schemes including equations 2.3, 2.4 and 2.5.*

By abuse of notation we use "Σ" also to range over signatures $(\Sigma_\tau, \Sigma_\epsilon, \alpha_\epsilon)$ of computational metalanguages. We write ML_T for the theory of $ML_T(\emptyset)$ including no other axioms than 2.3, 2.4 and 2.5. In [Mog91b], ML_T is presented in a sugared version, with the operator *let* written in the form:

$$\frac{\Gamma \vdash M : T\tau \quad \Gamma, x : \tau \vdash N : T\sigma}{\Gamma \vdash let\ x \Leftarrow M\ in\ N : T\sigma}$$

where x is bound in $(let\ x \Leftarrow M\ in\ N)$. In the following we shall sometimes adopt the above syntax which we take as an abbreviation for $let\ (\lambda\, x : \tau.\, N)\ M$. In particular, 2.3, 2.4 and 2.5 can be rewritten as:

$$\begin{array}{rl} T.\eta & let\ x \Leftarrow M\ in\ val(x) = M \\ T.\beta & let\ x \Leftarrow val(M)\ in\ N = N[M/x] \\ T.assoc & let\ y \Leftarrow (let\ x \Leftarrow L\ in\ M)\ in\ N = let\ x \Leftarrow L\ in\ (let\ y \Leftarrow M\ in\ N). \end{array}$$

The following ξ rules are derivable from β-reduction and the ξ rules for lambda abstraction and application. Note that this would not be possible had we taken $let\ x \Leftarrow M\ in\ N$ as primitive notation.

$$val.\xi \qquad \frac{M = N}{val(M) = val(N)}$$

$$let.\xi \qquad \frac{M = M' \qquad N = N'}{(let\ x \Leftarrow M\ in\ N) = (let\ x \Leftarrow M'\ in\ N')}$$

The converse of $val.\xi$ expresses a *mono requirement*: given an interpretation of $ML_T(\Sigma)$ in a strong monad T (as shown in the next section), this corresponds to the unit η of T being a monomorphism:

$$mono \qquad \frac{val(M) = val(N)}{M = N}$$

Such a property is required of "computational models" (λ_c-*models*) in [Mog91b], as it allows a view of η as an *existence* predicate, like in a logic of partial elements. The availability of such a predicate makes it possible to reconstruct the base category from the Kleisli category of a monad. In particular, there is a correspondence between categories with a monad satisfying *mono* and theories (of programming languages) with *equivalence* and *existence* predicates. We shall use the mono requirement to prove Proposition 3.7.4.

2.4 Structures and interpretation

In [Rey83], Reynolds suggested that satisfactory models of polymorphic languages should exclude ad hoc polymorphism. The concept of parametric polymorphism, usually credited to Strachey, expresses the informal requirement that instances of polymorphic entities are assigned uniform meaning.

This can be made more precise by considering *relational parametricity* [Rey83]. Assume some notion of relation $A \leftrightarrow B$ between objects A and B in a category $\mathcal{C}$. Let $\rho_1, \rho_2 : \chi \to Obj(\mathcal{C})$ be assignments to the type variables in χ; the members of a type-indexed family $\mathcal{R}$ of relations $\mathcal{R}_\tau : [\![\tau]\!]_{\rho_1} \leftrightarrow [\![\tau]\!]_{\rho_2}$ are called *logical* if $\mathcal{R}$ satisfies certain closure conditions (see [Plo73,Mit90] for definitions). In particular, if $\mathcal{C}$ has exponentials, a logical relation $\mathcal{R}_{\sigma\to\tau}$ is defined from $\mathcal{R}_\sigma$ and $\mathcal{R}_\tau$ as follows:

$$\mathcal{R}_{\sigma\to\tau} = \{f, g \mid \forall\, a, b.\, \mathcal{R}_\sigma(a, b) \supset \mathcal{R}_\tau(fa, gb)\}.$$

A polymorphic operator ω is called *relationally parametric* if its instances satisfy any logical relation. More precisely, we say that an Obj^n-indexed family of morphisms $\omega_{X_1,\ldots X_n} : [\![\sigma]\!]_{X_1,\ldots X_n} \to [\![\tau]\!]_{X_1,\ldots X_n}$ is relationally parametric if $\omega_{A_1,\ldots A_n} \mathcal{R}_{\sigma\to\tau}\, \omega_{B_1,\ldots B_n}$ for all type assignments $A_1, \ldots A_n$ and $B_1, \ldots B_n$, and logical relations in a family $\mathcal{R}$. This can be expressed diagrammatically as:

$$\begin{tikzcd}
[\![\sigma]\!]_{A_1,\ldots A_n} \arrow[r, "\omega_{A_1,\ldots A_n}"] \arrow[d, leftrightarrow, "\mathcal{R}_\sigma"'] & [\![\tau]\!]_{A_1,\ldots A_n} \arrow[d, leftrightarrow, "\mathcal{R}_\tau"] \\
[\![\sigma]\!]_{B_1,\ldots B_n} \arrow[r, "\omega_{B_1,\ldots B_n}"'] & [\![\tau]\!]_{B_1,\ldots B_n}
\end{tikzcd}$$

Example. Let $\rho_1, \rho_2 : \chi \to obj(\mathcal{C})$ be type assignments, and let $h : \rho_1 \to \rho_2$ be a collection of morphisms $h_X : \rho_1(X) \to \rho_2(X)$, indexed by χ; h induces a logical relation

$\simeq$, where $a \simeq_X b$ if and only if $b = h_X a$. Using this notion of logical relation, an operator Y of arity $\forall X. (X \to X) \longrightarrow X$ is relationally parametric when, for all $h : A \to B$, $f : A \to A$ and $g : B \to B$, if $hf = gh$ then $Y_B g = h(Y_A f)$. A "uniform" fixed-point operator Y in *Cpo*, the category of cpos, is defined by Plotkin as one that satisfies the above property restricted to *strict* maps h.

□

Properties such as uniformity are not algebraic, that is they cannot be enforced on models by means of equations. Therefore, models of the computational metalanguage may have to be provided with built-in parametricity. In chapter 7 we use a uniform fixed point operator Y to define structures for denotational semantics. There, we adopt a more powerful metalanguage, the Extended Calculus of Constructions [Luo82], in which domain theoretic properties can be axiomatized.

Remark. The notion of *naturality* can be related to parametricity. Let *list* be the unary type constructor of LISP. The operator *cons*, of arity $\forall X. X, \mathit{list}(X) \longrightarrow \mathit{list}(X)$, satisfies the condition that, for all $f : A \to B$,

$$\mathit{list}(f)(\mathit{cons}_A\ a\ l) = \mathit{cons}_B\ f(a)\ (\mathit{list}(f)\ l),$$

where $\mathit{list}(f)$ is often written $\mathtt{map}(f)$ (see remark in section 6.3). Similarly, the equation $\mathit{list}(f)\ \mathit{nil}_A = \mathit{nil}_B$ holds, where *nil* is a polymorphic operator of arity $\forall X. \mathit{list}(X)$. These conditions express the *naturality* of *cons* and *nil*. In order to express naturality in terms of parametricity, one should explain how a relation $\mathcal{R}_{list(A)}$ is defined *logically* from $\mathcal{R}_A$. The notion of *action* of a functor on a relational structure (see section 3.6), can be used to this extent.

□

Let $\mathcal{C}$ be a cartesian closed category and let Σ_τ be a collection of type constructors, with arities given by a map α_τ. A Σ_τ*-structure* on $\mathcal{C}$ is a collection of functions $[\![K]\!] : \mathit{obj}(\mathcal{C})^n \to \mathit{obj}(\mathcal{C})$, with $n = \alpha_\tau(K)$, one for each $K \in \Sigma_\tau$.

An interpretation $[\![\tau]\!]$ of a Σ_τ-polytype $\tau(X_1, \ldots X_n)$ in a Σ_τ-structure is a map $obj(\mathcal{C})^n \rightarrow obj(\mathcal{C})$ defined from the interpretation of the constants in Σ_τ and from the obvious interpretations of 1, $\times$ and $\rightarrow$. When τ is a type, $[\![\tau]\!]$ is an object of $\mathcal{C}$. Interpretation of types extends to *contexts*, with $[\![x_1 : \tau_1, \ldots x_n : \tau_n]\!] = [\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]$.

Lemma 2.4.1 *Let $\tau(X_1, \ldots X_n)$ be a polytype and let σ_i, $i = 1, \ldots n$, be types; then,* $[\![\tau(\sigma_1, \ldots \sigma_n)]\!] = [\![\tau]\!]_{[\![\sigma_1]\!], \ldots [\![\sigma_n]\!]}$.

Proof. By trivial induction on the structure of τ. □

An interpretation $[\![op]\!]$ of an operation $op : \forall X_1, \ldots X_n. \tau_1, \ldots \tau_m \longrightarrow \tau$ is an $Obj(\mathcal{C})^n$-indexed collection of morphisms $[\![op]\!]_{A_1, \ldots A_n} : [\![\tau_1]\!]_{A_1, \ldots A_n} \times \ldots [\![\tau_m]\!]_{A_1, \ldots A_n} \rightarrow [\![\tau]\!]_{A_1, \ldots A_n}$. Such a collection may be required to satisfy parametricity to obtain well behaved models. A Σ_ϵ-*structure* on $\mathcal{C}$ consists of an interpretation $[\![op]\!]$ for each $op \in \Sigma_\epsilon$. A Σ-*structure* on $\mathcal{C}$ consists of a Σ_τ and a Σ_ϵ structure.

An interpretation $[\![\_]\!]$ of $\mathcal{L}(\Sigma)$ in a Σ-structure on $\mathcal{C}$ maps types to objects of $\mathcal{C}$ and terms M of type τ to morphisms $[\![M]\!]_\Gamma : [\![\Gamma]\!] \rightarrow [\![\tau]\!]$, parametrically in a context Γ such that $\Gamma \vdash M : \tau$. When using lambda-notation to address morphisms in $\mathcal{C}$, the free variables of $[\![M]\!]_\Gamma$ are the ones in Γ. The interpretation $[\![M]\!]_\Gamma$ is defined by structural induction on M as follows:

C1. $[\![*]\!]_\Gamma = ! : [\![\Gamma]\!] \rightarrow 1$;

C2. $[\![x_i]\!]_{(x_1:\tau_1, \ldots x_n:\tau_n)} = \pi_i : [\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!] \rightarrow [\![\tau_i]\!]$;

C3. $[\![op_{\sigma_1, \ldots \sigma_n}(M_1, \ldots M_m)]\!]_\Gamma = [\![op]\!]_{[\![\sigma_1]\!], \ldots [\![\sigma_n]\!]}([\![M_1]\!]_\Gamma, \ldots [\![M_m]\!]_\Gamma)$;

C4. $[\![\langle M, N \rangle]\!]_\Gamma = \langle [\![M]\!]_\Gamma, [\![N]\!]_\Gamma \rangle$;
$[\![\pi_i(M)]\!]_\Gamma = \pi_i [\![M]\!]_\Gamma$;

C5. $[\![M\ N]\!]_\Gamma = [\![M]\!]_\Gamma [\![N]\!]_\Gamma$;
$[\![\lambda\, x : \sigma.\, M]\!]_\Gamma = \lambda\, x : [\![\sigma]\!].\, [\![M]\!]_{\Gamma, x:\sigma}$.

In the above definition we used the fact that, for all terms M but lambda abstractions, a context for M is also a context for its immediate subterms. Moreover, if Γ is a context for $\lambda\, x : \sigma.\, M$, then $(\Gamma, x : \sigma)$ is a context for M. In the clause for lambda abstraction, we defined interpretation at a context Γ by using interpretation at a bigger context $(\Gamma, x : \sigma)$. However, this does not break the induction since the latter is applied to a smaller term. In C3 we used lemma 2.4.1.

Definition 2.4.2 *A* model *of $\mathcal{L}(\Sigma)$, or* Σ-model, *is a pair $(\mathcal{C}, \mathcal{A})$, where $\mathcal{C}$ is a cartesian closed category and $\mathcal{A}$ is a Σ-structure on $\mathcal{C}$ satisfying the axioms of $\mathcal{L}(\Sigma)$.*

Example. Any theory $\mathcal{L}(\Sigma)$ has a syntactic model $\mathcal{T}(\Sigma)$. The underlying category of $\mathcal{T}(\Sigma)$ is the cartesian closed category whose objects are Σ_τ-types and whose morphisms are freely generated from morphisms $[M] : 1 \to \tau$, where $[M]$ is the equivalence class of closed terms N such that $\vdash M = N$. This category has an obvious Σ_τ structure and Σ_ϵ structure assigning to every $op : \forall X_1, \ldots X_n.\, \tau_1, \ldots \tau_m \longrightarrow \tau$ and types $\sigma_1, \ldots \sigma_n$ a morphism $[\![op]\!]_{\sigma_1,\ldots\sigma_n} = [\lambda x_1 : \tilde{\tau}_1, \ldots x_m : \tilde{\tau}_m.\, op\,(x_1, \ldots x_m)](x_1, \ldots x_m)$, where $\tilde{\tau}_i$ stands for $\tau_i(\sigma_1, \ldots \sigma_n)$.

$\square$

Proposition 2.4.3 (Soundness) *All theorems of ML_T are true in a cartesian closed category with a strong monad.*

Proof. Let Σ consist of $\Sigma_\tau = \{T\}$ and $\Sigma_\epsilon = \{val, let\}$ of appropriate arities; ML_T is of the form $\mathcal{L}(\Sigma)$. Any monad $\langle T, val, let \rangle$ is a Σ-structure on $\mathcal{C}$ and, by definition, it satisfies axioms 2.3, 2.4 and 2.5. Then, the result follows from soundness of typed lambda calculi with products and unit type with respect to interpretation in cartesian closed categories. $\square$

Proposition 2.4.4 (Completeness) *ML_T is complete with respect to interpretation in cartesian closed categories with a strong monad.*

Proof. The argument in the proof of soundness can be reversed: any model of ML_T is a cartesian closed category with a strong monad. Completeness follows from the existence of a syntactic model where equations are satisfied only if provable. We do not make this proof any more formal as it just repeats a similar one given in section 6.7 for a more powerful type theory. □

Models of $\mathcal{L}(\Sigma)$ are made of universal structure and "additional" structure to interpret the operations in Σ. In this chapter, we restricted the former to cartesian closedness, as universality was only meant for the unit type, products and exponentials. Metalanguages with a richer set of universal types can also be considered, for example, including sums, recursive types, (strong) natural numbers and so on. However, it may be impossible to interpret certain computational features in categories that are too rich in universal structure. A typical example is the inconsistency of coproducts, which are used in the next chapter to interpret exceptions, with fixed points in cartesian closed categories [HP90].

2.5 Examples of computationally relevant monads

Lifting. A lifting monad can be used to give an account of partiality. Let A be a set and let $A_\perp = \{X \subseteq A \mid |X| \leq 1\}$. There is a bijection between *partial* functions $A \rightharpoonup B$ and *total* functions $A \to B_\perp$, natural in A and B, mapping $f : A \rightharpoonup B$ into $f^\dagger$ such that $f^\dagger(a) = \{b \mid f(a) \text{ is defined and equal to } b\}$. The adjunction given by this bijection defines a monad in the category of sets, which we call the *lifting monad*:

$$TA \stackrel{\text{def}}{=} A_\perp$$

where $\eta = id^\dagger$ satisfies the mono requirement and μ maps $\emptyset \mapsto \emptyset$ and $\{X\} \mapsto X$. Such a monad is strong, with $(st\ f)\emptyset = \emptyset$ and $(st\ f)\{a\} = \{f(a)\}$. Note that the above "internal" definition of lifting generalizes to the category of cpos and also to topoi.

An appropriate operation to include in a metalanguage for nontermination is $\perp_X : TX$, which yields the always diverging computation for each type X. This choice is justified as follows. The lifting monad on the category of sets arises from an adjunction $\mathit{Sets} \rightharpoonup \mathit{PSets}$, where PSets is the category of sets A with a "point" $\perp_A : 1 \to A$; morphisms are point preserving functions. The right adjoint is the obvious forgetful functor $\mathit{PSets} \to \mathit{Sets}$. This functor is monadic. In fact, any pointed set $(A, \perp_A)$ is a Σ-algebra on A and every $A_\perp$ is the free language over Σ generated by A. Similar constructions can be carried out for finitary monads on arbitrary categories (with appropriate structure), among which the finite powerset monad $\mathcal{P}_{fin}$ and the monad for exceptions $TX = X + E$ (see below for both cases).

Remark. In general, studying the Kleisli category of a monad T (in the example above, the category of partial functions) helps understanding the way in which programs compose. On the other hand, the study of T-algebras may provide some insight as to which operations to include in a metalanguage for T-computations.

$\square$

The above definition of lifting can be made more general. A *class of admissible monos* (Rosolini) in a category $\mathcal{C}$ is a class of monos containing all the identities and closed under composition and pullback along arbitrary morphisms (see definition 4.3.8). Let $\mathcal{N}$ be a class of admissible monos in $\mathcal{C}$. A $\mathcal{N}$-partial map $[m, f] : A \rightharpoonup B$, is an equivalence class of pairs (m, f) with *domain* $m : X \rightarrowtail A \in \mathcal{N}$ and $f : X \to B$. We drop "$\mathcal{N}$" when understood. Partial maps compose in the category $p\mathcal{C}$ of partial maps: $[m, f] \circ [n, g]$ is $[n \circ g^{-1}m, f \circ m^{-1}g]$. A *lifting monad* $\langle (\_)_\perp, \eta, \mu \rangle$ in $\mathcal{C}$ is given by an adjunction $\mathcal{C} \rightharpoonup p\mathcal{C}$ in which the inclusion $\mathcal{C} \to p\mathcal{C}$ sending f into $[id, f]$ is left adjoint. Above we wrote $(\_)^\dagger$ for the natural isomorphism given by such an adjunction. Note that there can be no two lifting monads $\langle (\_)_\perp, \eta, \mu \rangle$ and $\langle (\_)_\perp, \eta, \mu' \rangle$ with $\mu \neq \mu'$, so that $\langle (\_)_\perp, \eta \rangle$ is good enough notation for lifting.

The transpose $h^\dagger$ of a partial map is sometimes said to *classify* h. More generally, a partial map classifier for an object A is an object $\tilde{A}$ together with a mono $A \rightarrowtail \tilde{A}$ such that, for any $\mathcal{N}$-partial map $[m, f] : X \rightharpoonup A$ there exists a unique total morphism $\chi[m, f] : X \to \tilde{A}$ forming the following pullback:

$$\begin{tikzcd} {} \arrow[r, "f"] \arrow[d, tail, "m"'] \arrow[dr, phantom, "\lrcorner", very near start] & A \arrow[d, tail] \\ X \arrow[r, "{\chi[m, f]}"'] & \tilde{A} \end{tikzcd}$$

The monad $\langle (\_)_\perp, \eta \rangle$ provides partial map classifiers $\eta_A : A \rightarrowtail A_\perp \in \mathcal{N}$ for all objects A of $\mathcal{C}$. Conversely, partial map classifiers for all objects give a lifting monad whose Kleisli category is $p\mathcal{C}$.

If $\mathcal{C}$ has initial object $\emptyset$ and $?_1 : \emptyset \to 1$ is admissible, then $\perp_A : 1 \to A_\perp$ is obtained by transposing $[?_1, ?_A]$. However, in order to validate interesting axioms involving such operations, additional assumptions may be necessary. For example, if $\emptyset$ is *strict*, that is if any morphism into $\emptyset$ is an isomorphism, then $(\mathit{let}\ f \perp_A) = \perp_B$ for any $f : A \to B_\perp$. This is easy to see by writing $(\mathit{let}\ f)$ as $\mu_B \circ f_\perp$ and using lemma 5.1.6 stating that any map $h : A \to B$ is the pullback of $h_\perp$ along η_B.

Nondeterminism. The bijection $\dagger$ described above, which involves partial functions, can be extended to arbitrary relations: there is a natural bijection between relations $r : A \leftrightarrow B$ and functions $r^\dagger A \to \mathcal{P}B$ such that $r^\dagger(a) = \{b \,|\, r(a, b)\}$. The resulting adjunction yields the *powerset monad*:

$$TA \stackrel{\text{def}}{=} \mathcal{P}A$$

in the category of sets, in which $\eta \stackrel{\text{def}}{=} \mathit{id}^\dagger$ satisfies the mono requirement and $\mu(H)$ is $\bigcup_{X \in H} X$. Again, there is a strength $\mathit{st}\ f\ X \stackrel{\text{def}}{=} \bigcup_{x \in X} f(x)$.

The obvious operation to include in a metalanguage for nondeterministic computations is $\mathit{or} : \forall X. TX, TX \longrightarrow TX$, with $[\![\mathit{or}]\!](X, Y) = X \bigcup Y$.

Note that the *finite* powerset monad on *Sets* arises as the classifying monad for the theory of semilattices, that is as the monad whose algebras are semilattices. As for lifting, the operation *or* (the join of a semilattice) associated with this monad is suggested by an analysis of the category of algebras.

Exceptions. Many programming languages feature mechanisms for "giving up" computations when exceptional situations arise. If a recovery action is to be attempted, it should be possible to perform a case analysis of programs, branching on whether or not an exception has occurred. Hence, designating an object E to interpret the outcome of aborted computations, the object

$$TX \stackrel{\text{def}}{=} X + E$$

provides suitable denotations for programs of type X in a category with sums. In particular, this T is the object map of a strong monad $\langle T, \mathit{val}, \mathit{let}\rangle$ where $\mathit{val} = \mathit{inj}_1$ and $(\mathit{let}\ N\ M) = \mathit{case}\,(M, N, \mathit{inj}_2)$. This yields a multiplication $\mu(w) = \mathit{case}\,(w, \mathit{id}, \mathit{inj}_2)$ and a strength $t(x, z) = \mathit{case}\,(z, \lambda\, y.\, \mathit{inj}_1(x, y), \mathit{inj}_2)$.

Let's consider this monad from the point of view of the operations. Given an object E in a category $\mathcal{C}$, the slice category $E/\mathcal{C}$ has morphisms with domain E in $\mathcal{C}$ as objects and commuting triangles $f \circ a = b$ as morphisms $f : a \to b$. The category $\mathcal{C}$ has sums of the form $(\_) + E$ if and only if the forgetful codomain functor $E/\mathcal{C} \to \mathcal{C}$ has left adjoint. Indeed, $E/\mathcal{C}$ is the Eilenberg-Moore category of the above monad T. T-algebras $X + E \to X$ can be viewed as *handlers* $E \to X$, while in the next chapter, where exceptions are studied in detail, we shall see that suitable operations to *raise* exceptions arise from suitable assumptions on the structure of E (see end of section 3.5).

Resumptions have long been used in denotational semantics to capture the phenomenon of interruption in program evaluation. Clearly enough, the meaning of the program running P_1 and P_2 in parallel cannot be described in terms of nondeterministic functions $[\![P_1]\!]$ and $[\![P_2]\!]$ since it must take into account interleaved executions of the two programs. Hence, it should be possible to single out the atomic steps of a computation and the interpretation of a program of type A should be an object producing a value of type A after a finite number of atomic steps. For example, reading "$\mathit{inj}_1(v)$" as "return v" and "$\mathit{inj}_2(\ldots)$" as "make one step and $(\ldots)$," the program $\mathit{inj}_2(\mathit{inj}_2(\mathit{inj}_1(v)))$ returns v after two (rather boring) atomic steps. Such programs can be interpreted as elements of the object

$$TA \stackrel{\text{def}}{=} \mu X.\, A + X$$

in a category $\mathcal{C}$, where $\mu(F)$, written $\mu X.F(X)$, is a "fixed point" operator on endofunctors $F : \mathcal{C} \to \mathcal{C}$, that is $F\mu(F) \cong \mu(F)$. *Inductive types* (see section 6.3) such as $\mu(F)$ are interpreted in categories $\mathcal{C}$ in which every (suitable) functor F has an initial algebra $F\mu(F) \to \mu(F)$. In that case, the above equation defines the object map of the strong monad of *resumptions*, where $val = inj_1$, $let\,f\,(inj_1 a) = f(a)$ and $let\,f\,(inj_2 z) = let\,f\,z$.

Programs of the kind described above are not so interesting because nothing much happens during an atomic step. In section 7.2 we show how to apply the simple resumption mechanism described above to computationally more elaborate situations, such as when programs sharing the same memory run in parallel, and we present operations appropriate for describing the mathematics of resumptions.

Remark. Lambda calculi of the form $\mathcal{L}(\Sigma)$ lack the syntactic means to describe inductive types. Signatures Σ_τ, for example, are not defined to include *higher order* constructors of the form $\mu : (type \to type) \to type$.

Interactive input is similar to resumption in that evaluation may have to be interrupted to input a value. The strong monad for interactive input features:

$$TA \stackrel{\text{def}}{=} \mu X.\, A + X^I$$

where I is the type of the input values. Again, the unit η is inj_1, while $\mu(inj_1 z) = z$ and $\mu(inj_2 w) = inj_2(\mu \circ w)$. For the strength, let $f : A \to B$, $st\ f\,(inj_1 a) = inj_1(fa)$ and $st\ f\,(inj_2 z) = inj_2((st\ f) \circ z)$. From these data one obtains $let\,f \stackrel{\text{def}}{=} \mu \circ Tf$, that is: $let\,f\,(inj_1 a) = f(a)$ and $let\,f\,(inj_2 z) = inj_2((let\,f) \circ z)$.

Given the above monad, an obvious operation to include in the metalanguage is $input : TI$, to be interpreted as $inj_2(inj_1)$. This operation can be used to input values on the fly: let I interpret the type of integers,

$$[\![n + input]\!] = let\ x \Leftarrow input\ in\ val(n + x) = inj_2(\lambda\, x : I.\, inj_1(n + x)).$$

Others. A detailed discussion of computationally relevant monads can be found in [Mog90a]. Here we present a few examples.

$TA = A \times M$	*complexity*, for M a monoid, see [Gur91].
$TA = (A \times S)^S$	*side effects*, discussed in chapter 5.
$TA = \mu X.\, A + (O \times X)$	*interactive output*, see [Mog90a].
$TA = R^{(R^A)}$	*continuations*, ibid.
$(TA)k = \sum_{n:N} A(F^n k)$	*dynamic allocation*, ibid.

The latter is defined in a functor category $\mathcal{C}^K$, where the objects of K intuitively represent stages of memory allocation and $F : K \to K$ increments the memory with one new location. A natural transformation $\sigma : id_K \dot{\to} F$ is also required, where $A\sigma_k(x) : AFk$ represents $x : Ak$ after a new allocation.

3 Application: exceptions

In this chapter, the computational metalanguage is used to study the semantics of a fragment of Standard ML, called TMLE (Tiny ML with Exceptions), including the exception handling mechanism. To our knowledge, no denotational description of ML exceptions has been given before.

The fragment that we consider is nontrivial because ML's exceptions have the capability of passing parameters to exception handlers and this has interesting consequences both on the metatheory of TMLE and on its semantics. One consequence is that, although the language provides no syntactic facilities for declaring either recursive functions or recursive types, programs may still fail to terminate. In fact, we discovered that, just like when adding references to the simply typed lambda calculus, it is possible to write a fixed point combinator in TMLE for any functional type. This failure of normalization has been noticed before in [Lil95]. However, the result, which is yet unpublished, was rediscovered by us independently.

As mentioned in section 2.5, models of nontermination generally involve a lifting monad. Even before realizing that TMLE programs could loop, we were led to lift computational types because a recursive domain equation arises in the interpretation of the type of exceptions. In fact, on the one hand the type of exceptions must be included in all computational types, as programs may produce exceptions, while on the other it must include computational types, as exceptions can carry programs as parameters.

Modelling the language in the cartesian closed category of complete posets, there are two places in which one can look for solutions to recursive equations: one is the subcategory of objects with least element and strict maps, and the other is the category of partial maps. We prefer the second approach to the first because it allows us to adopt the categorical sums of the (total) category of cpos in the interpretation of computational types. The universal property of such sums allows exceptions to be captured by a case analysis of the outcome of a computation. The universality of sums is crucial in the soundness proof of section 3.5.

In section 3.1, we define the language TMLE and its operational semantics. Then, we give an axiomatic account of exceptions by presenting a theory $ML_T(\Sigma)$ of the computational lambda calculus featuring exception constructors and operations of raise and handle (section 3.3). In section 3.5 we give an interpretation $[\![\_]\!]_{ML_T(\Sigma)}$ of this metalanguage in a cartesian closed category with appropriate structure and prove it sound. In section 3.4, TMLE is interpreted via a translation $(\!|\_|\!)$ into $ML_T(\Sigma)$, while the denotational semantics obtained by composing $[\![\_]\!]_{ML_T(\Sigma)}$ with $(\!|\_|\!)$ is shown to be adequate with respect to the given operational semantics in section 3.7.

The proof of adequacy is based on the definition of a suitable family of logical relations [Mit90]. Because of the circularity in the type of exceptions, such a family cannot be defined by induction on the structure of types and a proof of its existence is required. We follow the trace of a similar proof given by A. Pitts in [Pit] for a fragment of ML with one recursive type. Our case, however, is more involved than Pitts' because the notion of computation itself upon which interpretation is based (that is the monad $TX = (X+E)_\perp$) carries the seed of recursion (viz. the type E). A lesser difference with [Pit] is that we solve our recursive equations in a category of partial maps, rather than in one of strict maps.

The monadic setting in which adequacy is proven highlights the existence of two intertwined families of logical relations, one between semantic values and (strongly) canonical TMLE terms, and the other extending the first to computations and arbitrary expressions of the language. This seems to be a general pattern when proving computational adequacy with logical relations and it can be exploited to develop such proofs modularly. The idea is to make *semantic constructors* (section 6.2) work not only on monads T but also on the *actions* of T on relational structures (section 3.6).

3.1 Tiny ML with Exceptions

Like in ML, TMLE exceptions can carry values, which we call arguments. We assume a possibly infinite set $\mathcal{E} = \{\epsilon_1, \epsilon_2 \ldots\}$ of exception names and a function ζ from $\mathcal{E}$ to the types of TMLE, mapping each ϵ into the type of its argument. To give programs something to do to, besides raising and handling exceptions, we assume a type `nat` of natural numbers, a constant `n` for each natural number n and a binary function symbol `Op` to perform some arithmetical operation *op*. These are the formation rules for types and expressions of TMLE:

$$(\text{nat}) \quad \frac{}{\vdash \texttt{nat}\ \mathit{type}}$$

$$(\text{exc}) \quad \frac{}{\vdash \texttt{exc}\ \mathit{type}}$$

$$(\text{arrow}) \quad \frac{\vdash \sigma_1\ \mathit{type} \quad \vdash \sigma_2\ \mathit{type}}{\vdash \sigma_1 \to \sigma_2\ \mathit{type}}$$

$$(\text{var}) \quad \frac{}{\mathtt{x}_1 : \sigma_1, \ldots \mathtt{x}_n : \sigma_n \vdash \mathtt{x}_i : \sigma_i} \quad 1 \leq i \leq n$$

$$(\text{n}) \quad \frac{}{\vdash \mathtt{n} : \texttt{nat}}$$

$$(\text{Op}) \quad \frac{\Gamma \vdash e_1 : \texttt{nat} \quad \Gamma \vdash e_2 : \texttt{nat}}{\Gamma \vdash \texttt{Op}(e_1, e_2) : \texttt{nat}}$$

$$(\text{if}) \quad \frac{\Gamma \vdash e : \texttt{nat} \quad \Gamma \vdash e_1 : \sigma \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash \texttt{if}\ e = 0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \sigma}$$

$$\text{(fn)} \quad \frac{\Gamma, \mathtt{x} : \sigma \vdash e : \tau}{\Gamma \vdash \mathtt{fn}\ \mathtt{x} : \sigma \Rightarrow e : \sigma \to \tau}$$

$$\text{(appl)} \quad \frac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 e_2 : \tau}$$

$$(\epsilon) \quad \frac{\Gamma \vdash e : \zeta(\epsilon)}{\Gamma \vdash \epsilon(e) : \mathtt{exc}}$$

$$\text{(raise)} \quad \frac{\Gamma \vdash e : \mathtt{exc}}{\Gamma \vdash \mathtt{raise}_\tau\ e : \tau}$$

$$\text{(handle)} \quad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma, \mathtt{x} : \zeta(\epsilon) \vdash e_2 : \tau}{\Gamma \vdash e_1\ \mathtt{handle}\ \epsilon(\mathtt{x}) \Rightarrow e_2 : \tau}$$

This is the intuitive meaning of the operations `raise` and `handle`: an exception named ϵ is raised by $\mathtt{raise}_\tau\ \epsilon(e)$, where the program e, of type $\zeta(\epsilon)$, is meant to produce a value to be passed to a handler. The typing rule for `raise` in ML is

$$\text{(ML\_raise)} \quad \frac{\Gamma \vdash e \Rightarrow \mathtt{exc}}{\Gamma \vdash \mathtt{raise}\ e \Rightarrow \tau}$$

where the symbol "$\Rightarrow$" relates what in ML literature are called *phrases* and *semantic objects*. Since τ does not occur in the premise, this rule allows raised exceptions to have arbitrary type. Instead, we index the term constructor `raise` with the types of the language to avoid polymorphic type checking.

In $e_1\ \mathtt{handle}\ \epsilon(\mathtt{x}) \Rightarrow e_2$, if e_1 raises an exception $\epsilon(v)$, the handler e_2 is executed with $\mathtt{x}$ bound to v; otherwise the value of e_1 is returned. Exceptions propagate throughout their context until they are handled. The evaluation of exception raising is *eager*: for example, $\mathtt{raise}_{\mathtt{nat}}\ \epsilon(\mathtt{raise}_{\mathtt{nat}}\ \epsilon(4))\ \mathtt{handle}\ \epsilon(\mathtt{x}) \Rightarrow \mathtt{x}$ evaluates to 4 rather than (lazily) to $\texttt{raise}_{\texttt{nat}}\ \epsilon(4)$. On the other hand, conditional expressions must be lazy in each branch, since we want $\texttt{if}\ 0 = 0\ \texttt{then}\ b\ \texttt{else}\ \texttt{raise}_{\texttt{nat}}\ \epsilon(e)$ to raise no exception.

We distinguish between *canonical* and *strongly canonical* terms, the former ranged over by c and the latter by b:

$$\begin{array}{rcl} b & ::= & \texttt{n} \mid \epsilon(b) \mid \texttt{fn}\ \texttt{x}\!:\!\sigma \Rightarrow e \\ c & ::= & b \mid \texttt{raise}_{\sigma}\ b. \end{array}$$

The idea is that, among the canonical terms, the strong ones represent values that programs compute while the rest is computational gear. In TMLE, the gear is just raised exceptions. However in ML, one has semantic judgements such as $s, \Gamma \vdash \mathit{phrase} \Rightarrow A, s'$ includeing an environment Γ, states s and s', and a semantic object A, which may be, for instance, the constant FAIL, which is neither a value nor an exception, but rather the operational means of expressing a matching failure.

The operational semantics of TMLE is a relation $\rightsquigarrow$ between the closed terms and the closed canonical terms of the language. This relation is defined by the following rules:

$$\text{(b)} \quad \frac{}{b \rightsquigarrow b}$$

$$\text{(Op1)} \quad \frac{e_1 \rightsquigarrow \texttt{n}_1 \qquad e_2 \rightsquigarrow \texttt{n}_2}{\texttt{Op}(e_1, e_2) \rightsquigarrow \texttt{m}} \quad m = n_1\ op\ n_2$$

$$\text{(Op2)} \quad \frac{e_1 \rightsquigarrow \texttt{raise}_{\texttt{nat}}\ b}{\texttt{Op}(e_1, e_2) \rightsquigarrow \texttt{raise}_{\texttt{nat}}\ b}$$

$$\text{(Op3)} \quad \frac{e_1 \rightsquigarrow \texttt{n} \qquad e_2 \rightsquigarrow \texttt{raise}_{\texttt{nat}}\ b}{\texttt{Op}(e_1, e_2) \rightsquigarrow \texttt{raise}_{\texttt{nat}}\ b}$$

$$\text{(if1)} \quad \frac{e \rightsquigarrow 0 \qquad e_1 \rightsquigarrow c}{\texttt{if}\ e = 0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow c}$$

$$\text{(if2)}\quad \frac{e \rightsquigarrow \mathtt{n} \qquad e_2 \rightsquigarrow c}{\mathtt{if}\ e = 0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \rightsquigarrow c} \quad n \neq 0$$

$$\text{(if3)}\quad \frac{e \rightsquigarrow \mathtt{raise}_{\mathtt{nat}}\ b \quad \vdash e_1, e_2 : \sigma}{\mathtt{if}\ e = 0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \rightsquigarrow \mathtt{raise}_{\sigma}\ b}$$

$$\text{(app1)}\quad \frac{e_1 \rightsquigarrow \mathtt{fn}\ \mathtt{x}{:}\sigma \Rightarrow e \quad e_2 \rightsquigarrow b \quad [b/\mathtt{x}]e \rightsquigarrow c}{e_1 e_2 \rightsquigarrow c}$$

$$\text{(app2)}\quad \frac{e_1 \rightsquigarrow \mathtt{raise}_{\sigma \to \tau}\ b}{e_1 e_2 \rightsquigarrow \mathtt{raise}_{\tau}\ b}$$

$$\text{(app3)}\quad \frac{e_1 \rightsquigarrow b' \quad e_2 \rightsquigarrow \mathtt{raise}_{\sigma}\ b \quad \vdash e_1 : \sigma \to \tau}{e_1 e_2 \rightsquigarrow \mathtt{raise}_{\tau}\ b}$$

$$(\epsilon 1)\quad \frac{e \rightsquigarrow b}{\epsilon(e) \rightsquigarrow \epsilon(b)}$$

$$(\epsilon 2)\quad \frac{e \rightsquigarrow \mathtt{raise}_{\zeta(\epsilon)}\ b}{\epsilon(e) \rightsquigarrow \mathtt{raise}_{exc}\ b}$$

$$\text{(rs1)}\quad \frac{e \rightsquigarrow b}{\mathtt{raise}_{\sigma}\ e \rightsquigarrow \mathtt{raise}_{\sigma}\ b}$$

$$\text{(rs2)}\quad \frac{e \rightsquigarrow \mathtt{raise}_{exc}\ b}{\mathtt{raise}_{\sigma}\ e \rightsquigarrow \mathtt{raise}_{\sigma}\ b}$$

$$\text{(hnd1)}\quad \frac{e_1 \rightsquigarrow b}{e_1\ \mathtt{handle}\ \epsilon(\mathtt{x}) \Rightarrow e_2 \rightsquigarrow b}$$

$$\text{(hnd2)}\quad \frac{e_1 \rightsquigarrow \mathtt{raise}_{\sigma}\ \epsilon'(b)}{e_1\ \mathtt{handle}\ \epsilon(\mathtt{x}) \Rightarrow e_2 \rightsquigarrow \mathtt{raise}_{\sigma}\ \epsilon'(b)} \quad \epsilon' \neq \epsilon$$

$$\text{(hnd3)} \qquad \frac{e_1 \rightsquigarrow \texttt{raise}_\sigma\ \epsilon(b) \quad [b/\texttt{x}]e_2 \rightsquigarrow c}{e_1\ \texttt{handle}\ \epsilon(\texttt{x}) \Rightarrow e_2 \rightsquigarrow c}$$

Remark. In the operational semantics of TMLE, every judgement has at most one derivation.

Remark. Exploiting the call-by-value discipline in function application, one can define sequential composition $(e_1; e_2)$ as $(\texttt{fn x}:\sigma \Rightarrow e_2)e_1$, where e_2 does not contain free occurrences of $\texttt{x}$.

3.2 Exceptions and termination

Let α be an arbitrary TMLE type. Given an exception constructor ϵ with parameter of type $\zeta(\epsilon) = \texttt{exc} \to \alpha$, it is possible to write a looping program $\perp_\alpha$ of type α: let $K : \texttt{exc} \to \alpha$ be the term $\texttt{fn x}:\texttt{exc} \Rightarrow ((\texttt{raise}_\alpha\, \texttt{x})\ \texttt{handle}\ \epsilon(\texttt{y}) \Rightarrow \texttt{y(x)})$, then

$$\perp_\alpha \stackrel{\text{def}}{=} K\epsilon(K).$$

Using the same technique, it is possible to define a fixed point combinator $Y_{\alpha\to\beta}$ for arrow types. Let ϵ have parameter of type $\zeta(\epsilon) = \texttt{exc} \to (\alpha \to \beta)$. Given any term $\texttt{p}$ of type $(\alpha \to \beta) \to (\alpha \to \beta)$, let $K_\texttt{p}$ of type $\texttt{exc} \to (\alpha \to \beta)$ be the term $\texttt{fn x}:\texttt{exc} \Rightarrow ((\texttt{raise}_{\alpha\to\beta}\, \texttt{x})\ \texttt{handle}\ \epsilon(\texttt{y}) \Rightarrow \texttt{p}\,(\texttt{y}(\texttt{x})))$; then,

$$Y_{\alpha\to\beta} \stackrel{\text{def}}{=} \texttt{fn p}:(\alpha \to \beta) \to (\alpha \to \beta) \Rightarrow (\texttt{fn z}:\alpha \Rightarrow K_\texttt{p}\,\epsilon(K_\texttt{p})\,\texttt{z}).$$

The reason why TMLE programs can loop is that $\texttt{handle}$ treats exception constructors as such by providing a sort of case analysis on $\texttt{exc}$. Note that the same would happen even if exceptions were not first class objects. Indeed, we would still be able to write a fixed point combinator for arrow types if the language included no type `exc` at all and the construction and raising of exceptions were grouped together by a single term constructor:

$$(\text{raise\_}\epsilon) \qquad \frac{\Gamma \vdash e : \zeta(\epsilon)}{\Gamma \vdash \texttt{raise}_\tau\ \epsilon(e) : \tau}$$

Remark. In [dG95], a lambda calculus with an ML-like exception handling mechanism is presented, in which the evaluation of well typed expressions cannot give rise to uncaught exceptions. In this calculus, all terms strongly normalise.

Remark. In an unpublished manuscript [Lil95], Mark Lillibridge noticed that adding exceptions to the simply typed lambda calculus makes normalization fail. In that paper, the untyped lambda calculus is encoded in a typed language with exceptions to conclude that exceptions are strictly more powerful than *callcc*. The argument uses results obtained in [HL93] proving strong normalization for an extension of F^ω with *callcc* and *abort*. However, the results in [HL93] only concern particular evaluation strategies, and it is not clear to us how that generalizes to arbitrary strategies.

Remark It is possible to mimic the raise and handling of exceptions with *callcc* by using a global stack to store handlers. Here is the ML code:

```
val stack = ref(nil): (unit -> unit) list ref;
fun push = ...
fun pop = ...

fun raise () = pop stack ();

infix handle;
fun A handle B =
```

```
callcc(fn k =>
  ((push stack (fn x => throw k (B())));
   (let val res = A() in ((pop stack);res) end)));
```

3.3 A theory of exceptions

In this section we present the equational theory of a metalanguage $ML_T(\Sigma)$ for computations with exceptions, which we use to give the denotations of TMLE programs. As for TMLE, we assume a function ζ from the set $\mathcal{E} = \{\epsilon_1, \epsilon_2 \ldots\}$ of exception names into the types of the metalanguage, mapping each exception name into the type of its argument. The signature Σ includes constant type symbols $\underline{\mathfrak{N}}$ and $\underline{\mathrm{E}}$, with arities $\alpha_\tau(\underline{\mathfrak{N}}) = \alpha_\tau(\underline{E}) = 0$, and the following constant operation symbols:

$$\begin{aligned}
&\underline{n} : \underline{\mathfrak{N}},\\
&\underline{op} : \underline{\mathfrak{N}}, \underline{\mathfrak{N}} \longrightarrow \underline{\mathfrak{N}},\\
&cond : \forall X.\, \underline{\mathfrak{N}}, X, X \longrightarrow X,\\
&\underline{\epsilon} : \zeta(\epsilon) \longrightarrow \underline{E},\\
&raise : \forall X.\, \underline{E} \longrightarrow TX,\\
&handle_\epsilon : \forall X.\, TX, (\zeta(\epsilon) \to TX) \longrightarrow TX.
\end{aligned}$$

The constant $\underline{\mathfrak{N}}$ represents the type of natural numbers. As in TMLE, we assume a constant $\underline{n}$ for each natural number n and a binary function $\underline{op}$ representing some arithmetical operation op. The conditional $cond$ branches on the value zero of its first argument.

The constant $\underline{E}$ is the type of exceptions, whose constructors are the operations $\underline{\epsilon}$, one for each exception name. The operation $raise_\tau$ allows exceptions to be raised in any context $C[\_]$ in which "_" holds the place of a term of type τ. The portion of a program following the raise of an exception is disregarded. In TMLE, this could be phrased: $(\mathtt{raise}; N) = \mathtt{raise}$, while, of course, $(N; \mathtt{raise}) = \mathtt{raise}$ should not hold.

The first argument to $handle_\epsilon$ is the body whose possible exceptions are to be handled, while the second is a handler. There is nothing to be handled when the body is a value. Moreover, only an exception whose name is ϵ should be captured by $handle_\epsilon$.

Formalizing the above description of the theory of exceptions, the axioms of $ML_T(\Sigma)$ are the following:

$$
\begin{array}{rl}
(op) & \underline{op}(\underline{n},\underline{m}) = \underline{n\ op\ m}, \\
(cond.0) & cond(\underline{0}, M, N) = M, \\
(cond.n) & cond(\underline{n}, M, N) = N \qquad (n \neq 0), \\
(exception.\eta) & let\ x \Leftarrow raise(U)\ in\ N = raise(U), \\
(handle.\eta) & handle_\epsilon(val(M), H) = val(M), \\
(handle.\beta 1) & handle_\epsilon(raise(\underline{\epsilon}\ L), H) = HL, \\
(handle.\beta 2) & handle_\epsilon(raise(\underline{\epsilon}'\ L), H) = raise(\underline{\epsilon}'\ L) \qquad (\epsilon' \neq \epsilon).
\end{array}
$$

From the β and the ξ-rule for lambda application, the ξ-rules for $\underline{\epsilon}$, *raise* and *handle* are derivable:

$$
(exception.\xi) \qquad \frac{L_1 = L_2}{\underline{\epsilon}(L_1) = \underline{\epsilon}(L_2)}
$$

and similarly for the others.

Remark. Some of the axioms above are not "robust," meaning that they may need to be rewritten in order to describe the behaviour of exceptions in more complicated computational settings than TMLE. For example, $handle.\eta$ is rather weak when exceptions are considered in combination with side effects. In that case, we would like it to hold not only of *values*, but also of nonexceptional computations which alter the state. In the next chapter we introduce a family of *evaluation relations* $\Leftarrow_X \subseteq X \times TX$, where $v \Leftarrow M$ is read "M evaluates to v." These predicates can be used to write more general axioms such as:

$$
v \Leftarrow M \supset (v \Leftarrow handle_\epsilon(M, H)).
$$

3.4 Interpretation of the language

TMLE is interpreted via a translation $(\!|\ |\!)$ into the computational metalanguage $ML_T(\Sigma)$ for exceptions, that is $[\![e]\!]_{TMLE} \stackrel{\text{def}}{=} [\![(\!|e|\!)]\!]_{ML_T(\Sigma)}$. Types are translated as follows:

$$
\begin{aligned}
(\!|\texttt{nat}|\!) &= \underline{\mathfrak{N}},\\
(\!|\texttt{exc}|\!) &= \underline{E},\\
(\!|\sigma_1 \to \sigma_2|\!) &= (\!|\sigma_1|\!) \to T(\!|\sigma_2|\!).
\end{aligned}
$$

The translation of types extends to typing contexts in the obvious way. We assume that TMLE and $ML_T(\Sigma)$ share the same set $\mathcal{E}$ of exception names and that $(\!|\zeta(\epsilon)|\!) = \zeta(\epsilon)$. Terms are translated as follows:

$$
\begin{aligned}
(\!|\texttt{x}|\!) &= \mathit{val}(x),\\
(\!|\texttt{n}|\!) &= \mathit{val}(\underline{n}),\\
(\!|\texttt{Op}(e_1, e_2)|\!) &= \mathit{let}\ x_1 \Leftarrow (\!|e_1|\!)\ \mathit{in}\ \mathit{let}\ x_2 \Leftarrow (\!|e_2|\!)\ \mathit{in}\ \mathit{val}(\underline{op}(x_1, x_2)),\\
(\!|\texttt{if}\ e = 0\ \texttt{then}\ e_1\ \texttt{else}\ e_2|\!) &= \mathit{let}\ x \Leftarrow (\!|e|\!)\ \mathit{in}\ \mathit{cond}(x, (\!|e_1|\!), (\!|e_2|\!)),\\
(\!|\texttt{fn x}\!:\!\sigma \Rightarrow e|\!) &= \mathit{val}(\lambda x : (\!|\sigma|\!).\, (\!|e|\!)),\\
(\!|e_1 e_2|\!) &= \mathit{let}\ f \Leftarrow (\!|e_1|\!)\ \mathit{in}\ \mathit{let}\ x \Leftarrow (\!|e_2|\!)\ \mathit{in}\ f x,\\
(\!|\epsilon(e)|\!) &= \mathit{let}\ x \Leftarrow (\!|e|\!)\ \mathit{in}\ \mathit{val}(\underline{\epsilon}\, x),\\
(\!|\texttt{raise}_\sigma\ e|\!) &= \mathit{let}\ x \Leftarrow (\!|e|\!)\ \mathit{in}\ \mathit{raise}_{(\!|\sigma|\!)}(x),\\
(\!|e_1\ \texttt{handle}\ \epsilon(\texttt{x}) \Rightarrow e_2|\!) &= \mathit{handle}_\epsilon((\!|e_1|\!), \lambda x : \zeta(\epsilon).\, (\!|e_2|\!)).
\end{aligned}
$$

All strongly canonical terms $b : \sigma$ translate into terms $(\!|b|\!) = \mathit{val}(|b|)$, where $|b| : (\!|\sigma|\!)$ is the translation of b as a value rather than as a computation. Terms that are mapped to *let* expressions are *strict* when it comes to raised exceptions. Therefore, if exceptions are to be handled, the subterms of `handle` must not be filtered by a *let*, but passed immediately to *handle*. Similarly, as required from the operational semantics, `if_then_else` is strict in its first argument but not in the others. Note that $(\!|e_1 e_2|\!)$ gives a call-by-value interpretation of function application, according with the operational semantics.

Proposition 3.4.1 (Static adequacy) *TMLE terms of type τ are interpreted as terms of type $T(\!|\tau|\!)$, that is, $\Gamma \vdash e : \tau$ is derivable in TMLE if and only if $(\!|\Gamma|\!) \vdash (\!|e|\!) : T(\!|\tau|\!)$.*

Proof. By induction on the derivation of the typing judgements. For example, a derivation of $\Gamma \vdash \mathtt{raise}_\sigma\ e : \sigma$ in TMLE must contain a subderivation of $\Gamma \vdash e : \mathtt{exc}$. Then, using the inductive hypothesis,

$$\dfrac{(\!|\Gamma|\!) \vdash (\!|e|\!) : T\underline{E} \qquad \dfrac{(\!|\Gamma|\!), x : \underline{E} \vdash x : \underline{E}}{(\!|\Gamma|\!), x : \underline{E} \vdash raise_{(\!|\sigma|\!)}(x) : T(\!|\sigma|\!)}}{\dfrac{(\!|\Gamma|\!) \vdash let\ x \Leftarrow (\!|e|\!)\ in\ raise_{(\!|\sigma|\!)}(x) : T(\!|\sigma|\!)}{(\!|\Gamma|\!) \vdash (\!|\mathtt{raise}_\sigma\ e|\!) : T(\!|\sigma|\!).}}$$

□

In section 3.7, where we prove that the above interpretation is adequate with respect to the operational semantics, we shall use the following substitution lemma, whose proof is easy enough to be left out.

Lemma 3.4.2 $(\!|[b/x]e|\!) \equiv [|b|/x](\!|e|\!)$.

3.5 Interpretation of the metalanguage

In this section we describe a categorical interpretation of $ML_T(\Sigma)$ and prove its soundness. The proof only relies on an axiomatic description of models: first we define the interpretation in terms of an abstract categorical structure, and only after proving its soundness do we look for a concrete category that exhibits such a structure.

The metalanguage is interpreted in a cartesian closed category $\mathcal{C}$ with initial object $\emptyset$ and coproducts. The injections are written $inj_i^{X_1,X_2} : X_i \to X_1+X_2$, $i \in \{1,2\}$, dropping the superscripts when types are understood. When using lambda notation, we write $case(f,g,x) : Y$ for the morphism mediating $f : X_1 \to Y$ and $g : X_2 \to Y$ applied to $x : X_1 + X_2$. As usual, we write $case(f,g)$ for $\lambda x.\, case(f,g,x)$. Other structure is also needed in $\mathcal{C}$; we introduce it while spelling out the interpretation.

We assume a strong natural numbers object [LS86, I.9] $1 \xrightarrow{z} \mathfrak{N} \xrightarrow{s} \mathfrak{N}$ to interpret $\underline{\mathfrak{N}}$. Note that, since we have sums, this is the same as assuming an initial algebra for the functor $FX = 1 + X$. Let $a : 1 \to A$ and $f : A \to A$, we write $[a,f] : \mathfrak{N} \to A$ for the unique morphism such that $[a,f] \circ z = a$ and $[a,f] \circ s = f \circ [a,f]$. Arithmetical operations can be defined as usual. For example, writing n for the morphism $s^n(z)$, we can interpret $\underline{op}$ as $plus(m,n) \stackrel{\text{def}}{=} [m,s]n$.

Let $K : \underline{\mathfrak{N}}$, $L : \underline{\mathfrak{N}}$, $M : \sigma$ and $N : \sigma$ in a context Γ,

$$\begin{aligned}
&[\![\, \underline{n} \,]\!]_{\emptyset} = s^n(z),\\
&[\![\, \underline{op}(K,L)]\!]_{\Gamma} = op\,([\![K]\!]_{\Gamma}, [\![L]\!]_{\Gamma}),\\
&[\![cond\,(L,M,N)]\!]_{\Gamma} = [\, [\![M]\!]_{\Gamma,x:1}\,,\, [\![N]\!]_{\Gamma,x:\sigma}\,]\, [\![L]\!]_{\Gamma}.
\end{aligned}$$

As for computational types, since they must accommodate nonterminating programs, we assume a lifting monad $\langle (\_)_\perp, val_\perp, let_\perp \rangle$ in the category $\mathcal{C}$. Then, choosing an object E in $\mathcal{C}$ to interpret $\underline{E}$, we use the following strong monad to model computational types:

$$\begin{array}{rcl} TX & = & (X+E)_\perp, \\ val(v) & = & val_\perp(inj_1(v)), \\ let(f) & = & let_\perp(case(f\,,\,raise)), \end{array}$$

where $raise_X \stackrel{\text{def}}{=} val_\perp \circ inj_2 : E \to TX$.

Remark. The above definition makes sense for any strong monad in the place of $(\_)_\perp$. This is, in fact, a first example of *monad constructor* (see chapter 6), that is, an operation $\mathcal{F}$ that, given a monad T, returns a new monad $(\mathcal{F}T)X = T(X+E)$ to interpret T-computations with exceptions. In section 7.2, we prove in LEGO that, if T is a strong monad, so is $\mathcal{F}(T)$.

$\square$

The interpretation of $\underline{\epsilon}$, *raise* and $handle_\epsilon$ is defined in terms of two $\mathcal{E}$-indexed families of morphisms $\epsilon : [\![\zeta(\epsilon)]\!] \to E$ and $\overline{\epsilon} : E \to [\![\zeta(\epsilon)]\!] + E$ satisfying the following equations:

$$\overline{\epsilon}(\epsilon\, d) = inj_1(d) \tag{3.1}$$

$$\overline{\epsilon}(\epsilon'\, d) = inj_2(\epsilon'\, d) \qquad (\epsilon' \neq \epsilon). \tag{3.2}$$

Intuitively, $\overline{\epsilon}$ extracts the argument of any exception with name ϵ while it returns the others unaltered. Let $U : \underline{E}$, $L : \zeta(\epsilon)$, $M : T\tau$ and $H : \zeta(\epsilon) \to T\tau$ in a context Γ,

$$\begin{array}{l} [\![\epsilon\ L]\!]_\Gamma = \epsilon\ [\![L]\!]_\Gamma, \\ [\![raise_\sigma\ U]\!]_\Gamma = raise_{[\![\sigma]\!]}[\![U]\!]_\Gamma = val_\perp(inj_2[\![U]\!]_\Gamma), \\ [\![handle_\epsilon(M,H)]\!]_\Gamma = let_\perp(case(val\,,\,\overline{H}))\,[\![M]\!]_\Gamma, \end{array}$$

where $\overline{H} = \lambda x : E.\ case([\![H]\!]_\Gamma\,,\,raise\,,\,\overline{\epsilon}x) : E \to T[\![\tau]\!]$ is a function that leaves exceptions whose name is not ϵ unchanged while running the handler on the argument of the others.

Summarizing the structure described so far, a model of $ML_T(\Sigma)$ consists of:

universal structure: a cartesian closed category $\mathcal{C}$ with sums, initial object $\emptyset$ and a strong natural numbers object $1 \xrightarrow{z} \mathfrak{N} \xrightarrow{s} \mathfrak{N}$;

additional structure: a lifting monad $(\_)_\perp$ on the category $\mathcal{C}$, an object E and operations $\epsilon : [\![\zeta(\epsilon)]\!] \to E$ and $\overline{\epsilon} : E \to [\![\zeta(\epsilon)]\!] + E$ satisfying equations (3.1) and (3.2).

Theorem 3.5.1 (Soundness) $\vdash M = N \supset [\![M]\!] = [\![N]\!]$.

Proof. It is required to prove that the interpretation satisfies the axioms of section 3.3. We provide details only for the ones involving the operations on exceptions, since (op), $(cond.0)$ and $(cond.n)$ follow from routine calculations. Let $x : \sigma$, $N(x) : T\tau$, $U : \underline{E}$, $L : \zeta(\epsilon)$, $M : \tau$ and $H : \zeta(\epsilon) \to T\tau$,

$$(exception.\eta)\quad [\![let\ x \Leftarrow raise(U)\ in\ N(x)]\!] = let\ [\![N]\!]\ [\![raise(U)]\!] =$$
$$let_\perp(case([\![N]\!],\ raise))\ val_\perp(inj_2[\![U]\!]) =$$
$$case([\![N]\!],\ raise\ ,\ inj_2[\![U]\!]) = raise[\![U]\!] = [\![raise\ U]\!];$$

$$(handle.\eta)\quad [\![handle_\epsilon(val(M), H)]\!] = let_\perp(case(val, \overline{H}))\ val_\perp(inj_1[\![M]\!]) =$$
$$case(val\ ,\ \overline{H}\ ,\ inj_1[\![M]\!]) = val[\![M]\!] = [\![val(M)]\!];$$

$$(handle.\beta_1)\quad [\![handle_\epsilon(raise(\underline{\epsilon}\ L), H)]\!] = let_\perp(case(val, \overline{H}))\ val_\perp(inj_2(\epsilon[\![L]\!])) =$$
$$\overline{H}(\epsilon[\![L]\!]) = (\lambda\, x : E.\ case([\![H]\!],\ raise\ ,\ \overline{\epsilon}x))\ \epsilon[\![L]\!] =$$
$$case([\![H]\!],\ raise\ ,\ \overline{\epsilon}(\epsilon[\![L]\!])) = case([\![H]\!],\ raise\ ,\ inj_1[\![L]\!]) =$$
$$[\![H]\!]\ [\![L]\!] = [\![H L]\!];$$

$$(handle.\beta_2)\quad [\![handle_\epsilon(raise(\underline{\epsilon}'\ L), H)]\!] = \text{ (as above)}\ldots\ \overline{H}(\epsilon'\,[\![L]\!]) =$$
$$\text{(as above)}\ldots\ case([\![H]\!],\ raise\ ,\ \overline{\epsilon}_i(\epsilon'\,[\![L]\!])) =$$
$$case([\![H]\!],\ raise\ ,\ inj_2(\epsilon'\,[\![L]\!]) = raise(\epsilon'\,[\![L]\!]) = [\![raise(\epsilon'\ L)]\!].$$

□

In the proof of lemma 3.7.3 we shall see that *adequate* models of the metalanguage should also satisfy the following additional properties:

1. the pullback of $1 \xrightarrow{z} \mathfrak{N} \xleftarrow{s} \mathfrak{N}$ is $1 \leftarrow \emptyset \rightarrow \mathfrak{N}$ and s is a mono;

2. for all objects X in $\mathcal{C}$, the pullback of $X \xrightarrow{inj_1} X + E \xleftarrow{inj_2} E$ is $X \leftarrow \emptyset \rightarrow E$.

Proposition 3.5.2 *In any nontrivial category satisfying the requirement (1) above, if $n \neq m$, then $s^n(z) \neq s^m(z)$.*

Proof. Suppose $s^n(z) = s(s^{n+m}z)$. Since s is a mono, it must be $z = s(s^m z)$. Therefore, since $1 \times_{\mathfrak{N}} \mathfrak{N} = \emptyset$, there is a morphism $1 \rightarrow \emptyset$ mediating the pair $1 \xleftarrow{id} 1 \xrightarrow{s^m z} \mathfrak{N}$. $\square$

Proposition 3.5.3 *Let $\Gamma \vdash M : \tau$ and $\Gamma \vdash U : \underline{E}$; in any nontrivial category satisfying the requirement (2) above, $[\![val(M)]\!]_\Gamma(d_1) \neq [\![raise_\tau U]\!]_\Gamma(d_2)$ for all $d_1, d_2 : 1 \rightarrow [\![\Gamma]\!]$.*

Proof. If $val_\perp(inj_1[\![M]\!](d_1)) = [\![val(M)]\!](d_1) = [\![raise_\tau U]\!](d_2) = val_\perp(inj_2[\![U]\!](d_2))$, since $val_\perp$ is a mono, $inj_1[\![M]\!](d_1) = inj_2[\![U]\!](d_2)$. Hence there is a morphism $1 \rightarrow \emptyset$ mediating the pair $[\![M]\!](d_1) : 1 \rightarrow [\![\tau]\!]$ and $[\![U]\!](d_2) : 1 \rightarrow E$. Therefore, the category is trivial. $\square$

We find the above semantic structure in *Cpo*, the category of cpos (complete posets; no least element required) and continuous functions. This category has the universal structure required of models of $ML_T(\Sigma)$. In particular, $(X_1, \sqsubseteq_1) + (X_2, \sqsubseteq_2) = (X_1 \uplus X_2, \sqsubseteq)$, where $\uplus$ is the operation of disjoint union and, for $i, j \in \{1, 2\}$, $inj_i(x) \sqsubseteq inj_j(y)$ if and only if $i = j$ and $x \sqsubseteq_i y$. However we choose E, the injections inj_1 and inj_2 satisfy the additional condition (2). *Cpo* has a strong natural numbers object $(\omega, \sqsubseteq_{id})$, where $\sqsubseteq_{id}$ is the discrete ordering. The obvious z and s satisfy the additional condition (1).

Lifting is defined by adjoining a least element. This operation is the object map of a lifting monad defined by an adjunction $Cpo \rightharpoonup pCpo$, as described in section 2.5. The category *pCpo* has cpos as objects and partial continuous functions as morphisms. The domain of such a function $f : A \rightharpoonup B$ is a Scott-open subcpo of A. We write "$f(d) \downarrow$" to mean that d is in the domain of f. If $f(d) \downarrow$ and $f(d) = c$ we write "$f(d) \downarrow c$." The category $pCpo$ is Cpo-enriched, with $f \sqsubseteq g : A \rightharpoonup B$ if, for all $d \in A$, $f(d) \downarrow c_1$ implies $g(d) \downarrow c_2$ for some c_2 such that $c_1 \sqsubseteq c_2$.

The information carried by an exception is its name ϵ and its argument of type $\zeta(\epsilon)$. Since the handling of an exception requires a case analysis over the name, it makes sense to think of exceptions as elements of an $\mathcal{E}$-indexed sum

$$E \cong \sum_{\epsilon \in \mathcal{E}} [\![\zeta(\epsilon)]\!]. \tag{3.3}$$

This equation is in fact recursive: arguments of exceptions can be arbitrarily typed and, in particular, they can be programs raising exceptions, so that E may again pop out of any of the summands $[\![\zeta(\epsilon)]\!]$. Following the approach in [Fio94b], we look for a solution to (3.3) in the category of partial maps ($pCpo$). This category has only *partial* exponentials, that is objects of the form $A \to B_\perp$. However, this is not a problem, since we assumed that $\zeta(\epsilon) = (\![\zeta(\epsilon)]\!)$ and hence all exponentials in (3.3) are of the form $A \to TB$.

It is possible to solve recursive domain equations $X \cong F(X, X)$ in $pCpo$ when the functor $F : pCpo^{op} \times pCpo \to pCpo$ is locally continuous (the idea of using categories of partial maps for denotational semantics is originally from [Plo85a] and was recently developed in [Fio94b], to which we refer for details). Using this feature, we obtain the object E to interpret exceptions as the minimal invariant [Fre90] of a suitable functor defined below. As we shall see, the invariance of E yields a canonical family of operations ϵ and $\bar{\epsilon}$, while minimality yields a universal property to be used in the adequacy proof of section 3.7.

Let F be the family of functors $F_\sigma = pCpo^{op} \times pCpo \to pCpo$, indexed by the types of TMLE, defined by simultaneous induction as follows:

$$F_{\mathsf{nat}}(X, Y) \quad = \quad \mathfrak{N},$$

$$\begin{array}{rcl} F_{\mathtt{exc}}(X,Y) & = & Y, \\ F_{\sigma\to\tau}(X,Y) & = & F_\sigma(Y,X) \to (F_\tau(X,Y)+Y)_\perp, \end{array}$$

where $(\_) \to (\_)_\perp$ is partial exponentiation. All functors in F are locally continuous and, hence, so is $F_{\mathcal{E}}(X,Y) \stackrel{\mathrm{def}}{=} \sum_{\epsilon\in\mathcal{E}} F_{\zeta(\epsilon)}(X,Y)$. Let E be the minimal invariant object of $F_{\mathcal{E}}$. It is easy to verify that $[\![\sigma]\!]_{TMLE} = F_\sigma(E,E)$. Then,

$$E \cong \sum_{\epsilon\in\mathcal{E}} F_{\zeta(\epsilon)}(E,E) = \sum_{\epsilon\in\mathcal{E}} [\![\zeta(\epsilon)]\!]_{TMLE} = \sum_{\epsilon\in\mathcal{E}} [\![\zeta(\epsilon)]\!]_{ML_T(\Sigma)}.$$

Let $\alpha : F_{\mathcal{E}}(E,E) \to E$ be the above isomorphism; we call $\epsilon : [\![\zeta(\epsilon)]\!] \to E$ the components of α, that is, $\alpha(\epsilon, d) = \epsilon(d)$. Moreover, since E contains only elements of the form $\epsilon(d)$, the equations (3.1) and (3.2) define $\overline{\epsilon}$ completely.

Remark. Unlike ML, TMLE has no facilities for declaring exceptions. Of course, the monad $TX = (X + E)_\perp$ could not model dynamic creation of exceptions, as E would have to change dynamically in order to accomodate new exceptions. To capture such a situation, a "possible world" semantics can be adopted, like in [OT92] for modelling dynamic allocation.

3.6 Semantic approximation of TMLE programs

Composing the translation $(\!|\_|\!)$ of section 3.4 with the interpretation of the metalanguage defined in section 3.5, we obtain an interpretation $[\![\_]\!]_{TMLE}$ of TMLE. To make sure that $[\![\_]\!]_{TMLE}$ agrees with the operational semantics (computational adequacy), one would like to show that, if a program denotes a natural number n, then it evaluates to `n`. Proving such a statement is not easy, since straight induction on the structure of the program fails in the case of function application. Hence, the inductive hypotheses must be strenthened and this leads to proof techniques involving *logical relations* (see section 2.4).

Here we show the existence of a family of logical relations $\leq$, which we call formal approximation (see below) between semantic objects and TMLE programs. This family is used in the next chapter to show that $[\![\_]\!]_{TMLE}$ is computationally adequate. We believe that $\leq$ is also interesting in its own right, as it provides an example of a logical relation involving computational types.

In [Plo91a], a family of relations $\leq_\sigma$ between semantic values in $[\![\sigma]\!]$ and canonical terms of type σ is defined for a metalanguage with recursive types. Using such relations, one can write a statement $\mathfrak{M} \preceq_\sigma M$ implying that the program M terminates if the mathematical expression $\mathfrak{M}$ *denotes*. Then, computational adequacy is proven by showing that $[\![M]\!] \preceq_\sigma M$. Sometimes the argument is complicated by the fact that $\leq_\sigma$ cannot be defined by induction on σ, and hence a lengthy proof is required of the existence of such a relation. In the case of Plotkin's metalanguage, the problem arises from recursive types; for TMLE, it arises from the recursive type of exceptions.

We call Exp_σ the set of all closed TMLE terms of type σ and $Can_\sigma \subseteq Exp_\sigma$ the set of all *strongly* canonical terms. Following [Plo91a], we look for a type-indexed family $\leq$ of relations of the form $\leq_\sigma \subseteq [\![\sigma]\!]_{TMLE} \times Can_\sigma$ to provide an appropriate notion of semantic approximation of TMLE expressions. We require that the members of such a family satisfy the following conditions:

A1. for all $b \in Can_\sigma$, the set $\{d \mid d \leq_\sigma b\}$ is closed under taking least upper bounds of countable chains;

A2. $n \leq_{\mathtt{nat}} \mathtt{n}$;

A3. $\epsilon(d) \leq_{\mathtt{exc}} \epsilon(b)$ if and only if $d \leq_{\zeta(\epsilon)} b$;

A4. $f \leq_{\sigma\to\tau} (\mathtt{fn}\ \mathtt{x}:\sigma \Rightarrow e)$ if and only if $f(d) \preceq_\tau [b/\mathtt{x}]e$ for all d and b such that $d \leq_\sigma b$,

where $\mathfrak{M} \preceq_\tau e$ stands for the conjunction of the two statements:

B1. if $\mathfrak{M} = val(d)$, then $e \rightsquigarrow b$, for some b such that $d \leq_\tau b$;

B2. if $\mathfrak{M} = raise_\tau(d)$ then $e \rightsquigarrow \texttt{raise}_\tau\ b$, for some b such that $d \leq_{\texttt{exc}} b$.

Considering B1, the relation $\preceq_\tau \subseteq T[\![\tau]\!] \times Exp_\tau$ can be viewed as extending $\leq_\tau$ over computations and programs.

We call the members of a family satisfying clauses A1-4 *formal approximation relations*. Considering the mutual recursion in A3 and A4, such a family cannot be defined by a simple induction on types. Moreover, let *Rel* be the complete lattice of type-indexed families of relations $R_\sigma \subseteq [\![\sigma]\!]_{TMLE} \times Can_\sigma$ ordered by type-wise inclusion; the operation $\phi : Rel \to Rel$ obtained by reading clauses A2-4 as a system of equations $R_\sigma = \phi_\sigma(R)$ is not monotone, as R occurs both positively and negatively in the right hand side of A4. Hence, $\leq$ cannot be obtained by applying standard techniques of fixed point construction to ϕ.

Note that this same problem arises when computations use a memory where values of arbitrary type can be stored. The following example generalizes the Tiny-ML example considered in [Pit91], where memory locations could only contain integers.

Example. Let $\leq_\rho$ be the relation between semantic and syntactic states derived from a family $\leq_\sigma$, where ρ assigns to each memory location l the type $\rho(l)$ of value stored in it. In the case of state computations, the statement $\mathcal{M} \preceq_\sigma e$ spells out as: if $\mathcal{M} \downarrow d$ then, for all a, S, S', s, if $d(S) \downarrow (a, S')$ and $S \leq_\rho s$, then there exist c and s' such that $s, e \rightsquigarrow s', c$ and $a \leq_\sigma c$ and $S' \leq_\rho s'$. Note that the definition of $\leq_\sigma$ depends on $\leq_\rho$ which may involve types more complicated than σ.

$\square$

Following the approach proposed in [Pit93], we obtain a family of formal approximation relations from a suitably defined invariant relation on the recursive object E of exceptions. The argument follows the trace of the proof of adequacy given in [Pit] for a small fragment of ML with a single recursive datatype declaration. However, some definitions and results obtained in that paper must be rephrased to apply to our setting, as they refer to the category $Cppo_\perp$ of cpos with least element and strict continuous functions.

Definition 3.6.1 ([Pit, 4.1]) *A* relational structure $\mathcal{R}$ *on a category* $\mathcal{C}$ *is specified by the following data:*

- *for each object A of $\mathcal{C}$, a set $\mathcal{R}(A)$ of 'relations on A;'*
- *for each morphism $f : A \to B$ in $\mathcal{C}$, a binary relation between elements $R \in \mathcal{R}(A)$ and elements $S \in \mathcal{R}(B)$, written $f : R \subset_{\mathcal{R}} S$ (or just $f : R \subset S$). These binary relations are required to satisfy the following properties:*
  - *$id_A : R \subset R$, for all R in $\mathcal{R}(A)$;*
  - *if f and g are composable, $f : R \subset S$ and $g : S \subset T$, then $g \circ f : R \subset T$.*

If $\mathcal{R}$ is a relational structure on $pCpo$, a relation $S \in \mathcal{R}(B)$ is said to be *admissible* [Pit, 4.3] when, for all $R \in \mathcal{R}(A)$, the subset

$$[R, S] \stackrel{\text{def}}{=} \{f \mid f : R \subset S\}$$

of $pCpo(A, B)$ contains the always undefined function and is closed under taking least upper bounds of countable chains (chain-complete).

Let $F : pCpo^{op} \times pCpo \to pCpo$ be a functor; an *admissible action* of F on a structure $\mathcal{R}$ on $pCpo$ is a family of operations $F_{A,B} : \mathcal{R}(A) \times \mathcal{R}(B) \to \mathcal{R}(F(A, B))$ such that

C1. if $S \in \mathcal{R}(B)$ is admissible, so is $F_{A,B}(R, S)$, for any $R \in \mathcal{R}(A)$;

C2. if $f : A_2 \rightharpoonup A_1$ is such that $f : R_2 \subset R_1$ and $g : B_1 \rightharpoonup B_2$ is such that $g : S_1 \subset S_2$, with S_2 admissible, then $F(f, g) : F_{A_1,B_1}(R_1, S_1) \subset F_{A_2,B_2}(R_2, S_2)$.

We usually drop the subscripts in $F_{A,B}$. Of course, a functor may have more than one admissible action on a relational structure, so it is actually by an abuse of notation that we call an admissible action by the name of its corresponding functor.

Let Can be the set $\{c : \sigma \mid c \in Can_\sigma\}$, that is, the set of all strongly canonical terms tagged with their types; below, we use the following relational structure $\mathcal{R}$ on $pCpo$:

- a relation on $\mathcal{R}(A)$ is a subset of $A \times Can$;

- $f : R \subset S$ if and only if, for all $(d, b : \sigma) \in R$, $f(d) \downarrow d'$ implies $(d', b : \sigma) \in S$.

Usually, we drop the type tag. It is easy to see that a relation S in $\mathcal{R}$ is admissible if and only if, for all b, the set $\{d \mid (d, b) \in S\}$ is chain-complete. Admissibility is used in the proof of theorem 3.6.2.

Let A be a cpo; we call $\mathcal{R}_{adm}(A)$ the set of all admissible relations in $\mathcal{R}(A)$. Clearly $\mathcal{R}_{adm}(A)$ is closed under taking arbitrary intersections and so it is a complete lattice ordered by inclusion.

The *inverse image* of a relation $R \in \mathcal{R}(A)$ along a partial function $f : B \rightharpoonup A$ is defined as $f^*R = \{(d, b) \mid \text{if } f(d) \downarrow d', \text{ then } (d', b) \in R\} \in \mathcal{R}(B)$. Clearly, $f : f^*R \subset R$ and, if $f : S \subset R$, then $id : S \subset f^*R$. The operation $f^* : \mathcal{R}(A) \to \mathcal{R}(B)$ is monotone and it satisfies $id_A^* = id_{\mathcal{R}(A)}$ and $(f \circ g)^* = g^* \circ f^*$. Moreover, it is easy to verify that, since the domain of f is Scott-open, f^* preserves admissibility.

Theorem 3.6.2 *Let $F : pCpo^{op} \times pCpo \to pCpo$ be a locally continuous functor with an admissible action on $\mathcal{R}$ and let $\alpha : F(D, D) \cong D$ be its minimal invariant object. There exists an admissible relation $\Delta \in \mathcal{R}(D)$ such that*
- $\alpha : F(\Delta, \Delta) \subset \Delta$ *and*
- $\alpha^{-1} : \Delta \subset F(\Delta, \Delta)$.

An admissible relation Δ satisfying these conditions is called F*-invariant*. In [Pit] it is shown that an invariant relation for a locally continuous $F : Cppo_\perp^{op} \times Cppo_\perp \to Cppo_\perp$, if it exists, is unique; the proof also adapts to $pCpo$.

To prove theorem 3.6.2, we need the following:

Lemma 3.6.3 *Let F, D and α be as above; a relation $\Delta \in \mathcal{R}(D)$ is F-invariant if and only if $\Delta = (\alpha^{-1})^* F(\Delta, \Delta)$.*

Proof. (If) Using $f : f^* R \subset R$,

- $\alpha : F(\Delta, \Delta) = (\alpha^{-1} \circ \alpha)^* F(\Delta, \Delta) = \alpha^*((\alpha^{-1})^* F(\Delta, \Delta)) = \alpha^* \Delta \subset \Delta$ and
- $\alpha^{-1} : \Delta = (\alpha^{-1})^* F(\Delta, \Delta) \subset F(\Delta, \Delta)$.

(Only if) Noticing that $R \subseteq S$ if and only if $id : R \subset S$,

- composing $\alpha : F(\Delta, \Delta) \subset \Delta$ and $\alpha^{-1} : (\alpha^{-1})^* F(\Delta, \Delta) \subset F(\Delta, \Delta)$ we obtain $id = \alpha \circ \alpha^{-1} : (\alpha^{-1})^* F(\Delta, \Delta) \subset \Delta$, and
- since $f : R \subset S$ implies $id : R \subset f^* S$,
  $\alpha^{-1} : \Delta \subset F(\Delta, \Delta)$ implies $id : \Delta \subset (\alpha^{-1})^* F(\Delta, \Delta)$.

□

Using the lemma, we prove theorem 3.6.2 by showing that there exists a fixed point of the operation $\phi : \mathcal{R}_{adm}(D) \to \mathcal{R}_{adm}(D)$ defined as

$$\phi(R) = (\alpha^{-1})^* F(R, R).$$

We follow the trace of [Pit, section 4], to which we refer for more detail. First, we separate and negative occurrences of the variable R in $\phi(R)$ and obtain the functor $\psi : \mathcal{R}_{adm}(D)^{op} \times \mathcal{R}_{adm}(D) \to \mathcal{R}_{adm}(D)$, where $\psi(R, S) = (\alpha^{-1})^* F(R, S)$, from which $\phi(R)$ can be recovered as $\psi(R, R)$. "Dualising" ψ, one obtains a monotone function $\psi^\dagger : \mathcal{R}_{adm}(D)^{op} \times \mathcal{R}_{adm}(D) \to \mathcal{R}_{adm}(D)^{op} \times \mathcal{R}_{adm}(D)$, i.e. $\psi^\dagger(R, S) \stackrel{\text{def}}{=} (\psi(S, R), \psi(R, S))$. Then, the Knaster-Tarski fixed point theorem yields a pair of admissible relations $(\Delta^-, \Delta^+) = \psi^\dagger(\Delta^-, \Delta^+) = (\psi(\Delta^+, \Delta^-), \psi(\Delta^-, \Delta^+))$. From the universal property of this object it follows easily that $\Delta^+ \subseteq \Delta^-$.

The opposite inclusion also holds. In fact: Let $\delta : pCpo\,(D, D) \to pCpo\,(D, D)$ be defined as $\delta(f) = \alpha \circ F(f, f) \circ \alpha^{-1}$. Since F is a locally continuous functor, δ is continuous. Since $\Delta^- = \psi(\Delta^+, \Delta^-)$, we have $id : \Delta^- \subset (\alpha^{-1})^* F(\Delta^+, \Delta^-)$. Composing this inclusion with $\alpha^{-1} : (\alpha^{-1})^* F(\Delta^+, \Delta^-) \subset F(\Delta^+, \Delta^-)$, we obtain $\alpha^{-1} : \Delta^- \subset F(\Delta^+, \Delta^-)$. Similarly, one obtains $\alpha : F(\Delta^-, \Delta^+) \subset \Delta^+$. Given $f : \Delta^- \subset \Delta^+$, the admissible action of F yields $F(f,f) : F(\Delta^+, \Delta^-) \subset F(\Delta^-, \Delta^+)$, which, composed with the above inclusions, yields $\delta(f) = \alpha \circ F(f,f) \circ \alpha^{-1} : \Delta^- \subset \Delta^+$. Hence, δ restricts to $[\Delta^-, \Delta^+] \to [\Delta^-, \Delta^+]$. Since $[\Delta^-, \Delta^+]$ contains the always undefined function and it is chain-complete (because Δ^+ is admissible), the least fixed point of δ is in $[\Delta^-, \Delta^+]$. Since, by the universal property of D, this fixed point is the identity, $id : \Delta^- \subset \Delta^+$ yields $\Delta^- \subseteq \Delta^+$. Therefore, $\Delta^- = \Delta^+$ is a fixed point of ϕ as required. This concludes the proof of theorem 3.6.2.

□

Let F be the family of functors defined in section 3.5; a type-indexed family $\tilde{F}$, whose elements $\tilde{F}_\sigma$ are families of operations $(\tilde{F}_\sigma)_{A,B} : \mathcal{R}(A) \times \mathcal{R}(B) \to \mathcal{R}(F_\sigma(A,B))$ is defined as follows:

$$\begin{array}{rcl}
\tilde{F}_{\mathtt{nat}}(R,S) & = & \{(n, \mathtt{n} : \mathtt{nat}) \mid n \in \mathfrak{N}\}, \\
\tilde{F}_{\mathtt{exc}}(R,S) & = & \{(d, b : \mathtt{exc}) \mid (d, b : \mathtt{exc}) \in S\}, \\
\tilde{F}_{\sigma\to\tau}(R,S) & = & \{(f, \mathtt{fn\ x} : \sigma \Rightarrow e : \sigma \to \tau) \mid \text{for all } (d, b : \sigma) \in \tilde{F}_\sigma(S,R), \\
& & \quad \text{- if } f(d) \downarrow val(d') \text{ then } [b/\mathtt{x}]e \rightsquigarrow b' \text{ and } (d', b' : \tau) \in \tilde{F}_\tau(R,S); \\
& & \quad \text{- if } f(d) \downarrow raise_\tau(d') \text{ then } [b/\mathtt{x}]e \rightsquigarrow \mathtt{raise}_\tau(b') \text{ and } (d', b' : \mathtt{exc}) \in S\}.
\end{array}$$

Proposition 3.6.4 *For all TMLE types σ, $\tilde{F}_\sigma$ is an admissible action of F_σ on $\mathcal{R}$.*

Proof. We prove condition C1 by induction on the structure of σ, simultaneously for all members of $\tilde{F}$. The cases where σ is `nat` or `exc` are trivial. As for $\sigma \to \tau$, let $S \in \mathcal{R}(B)$ be admissible, let R be any relation in $\mathcal{R}(A)$ and let $\langle f_i \rangle$ be a countable chain of elements of $F_{\sigma\to\tau}(A,B)$ such that, for all $i \in \omega$, $(f_i, \mathtt{fn\ x} : \sigma \Rightarrow e) \in \tilde{F}_{\sigma\to\tau}(R,S)$; we show that $(f_\omega, \mathtt{fn\ x} : \sigma \Rightarrow e) \in \tilde{F}_{\sigma\to\tau}(R,S)$, where $f_\omega = \sqcup\langle f_i \rangle$. Let $(d,b) \in \tilde{F}_\sigma(S,R)$. If $f_\omega(d) \downarrow$, there must exist an n such that $f_m(d) \downarrow$ for all $m \geq n$ and $f_\omega(d) = \sqcup\langle f_{i+n}(d) \rangle$. If $f_\omega(d) \downarrow val(d')$, then, it must be $f_m(d) \downarrow val(d_m)$, and hence $[b/\mathtt{x}]e \rightsquigarrow b'$ for some b' such that $(d_m, b') \in \tilde{F}_\tau(R,S)$ for all $m \geq n$. Since $d' = \sqcup\langle d_{i+n} \rangle$, applying the inductive hypothesis to $\tilde{F}_\tau$, it must be $(d', b') \in \tilde{F}_\tau(R, S)$ as required. A similar argument yields the required property when $f_\omega(d) \downarrow \mathit{raise}_\tau(d')$ and this concludes the proof of condition C1. Note that the determinacy of evaluation in TMLE is used in the above argument (e.g. by assuming that there exist one b' which works for all $m \geq n$). As expected, this fails in the case of unbounded nondeterminism.

For C2, let $R_1 \in \mathcal{R}(A_1)$, $R_2 \in \mathcal{R}(A_2)$, $S_1 \in \mathcal{R}(B_1)$ and $S_2 \in \mathcal{R}(B_2)$; let $f : A_2 \rightharpoonup A_1$ be a morphism such that $f : R_2 \subset R_1$ and let $g : B_1 \rightharpoonup B_2$ be one such that $g : S_1 \subset S_2$; we show that $F_\sigma(f, g) : (\tilde{F}_\sigma)_{A_1,B_1}(R_1, S_1) \subset (\tilde{F}_\sigma)_{A_2,B_2}(R_2, S_2)$. Again, the proof is by induction on the structure of σ, where the cases $\sigma = \texttt{nat}$ and $\sigma = \texttt{exc}$ are trivial.

We show $F_{\sigma\to\tau}(f, g) : \tilde{F}_{\sigma\to\tau}(R_1, S_1) \subset \tilde{F}_{\sigma\to\tau}(R_2, S_2)$. Let $h \in F_{\sigma\to\tau}(A_1, B_1)$, that is, $h : F_\sigma(B_1, A_1) \rightharpoonup (F_\tau(A_1, B_1) + B_1)$, we are required to prove that, if $(h, \texttt{fn}\,\texttt{x}\!:\!\sigma \Rightarrow e)$ is in $\tilde{F}_{\sigma\to\tau}(R_1, S_1)$ and $F_{\sigma\to\tau}(f, g)(h) \downarrow k$, then $(k, \texttt{fn}\,\texttt{x}\!:\!\sigma \Rightarrow e)$ is in $\tilde{F}_{\sigma\to\tau}(R_2, S_2)$. Note that $k = F_{\sigma\to\tau}(f, g)(h) = (F_\tau(f, g) + g) \circ h \circ F_\sigma(g, f)$.

Let $(d, b) \in \tilde{F}_\sigma(S_2, R_2)$. Assume that $k(d) \downarrow \mathit{val}(d_1)$. There must be a d_2 such that $h(F_\sigma(g, f)(d)) \downarrow \mathit{val}(d_2)$ and $F_\tau(f, g)(d_2) \downarrow d_1$. Since $h(F_\sigma(g, f)(d)) \downarrow \mathit{val}(d_2)$, it must be $F_\sigma(g, f)(d) \downarrow d_3$. Then, by the inductive hypothesis on $F_\sigma(g, f)$, $(d_3, b) \in \tilde{F}_\sigma(S_1, R_1)$ and hence, from the assumption that $(h, \texttt{fn}\,\texttt{x}\!:\!\sigma \Rightarrow e) \in \tilde{F}_{\sigma\to\tau}(R_1, S_1)$, it follows that there exists a b' such that $[b/\texttt{x}]e \rightsquigarrow b'$ and $(d_2, b') \in \tilde{F}_\tau(R_1, S_1)$. Using the inductive hypothesis on $F_\tau(g, f)$, we get $(d_1, b') \in \tilde{F}_\tau(R_2, S_2)$. So, we have shown that, if $k(d) \downarrow \mathit{val}(d_1)$, then $[b/\texttt{x}]e \rightsquigarrow b'$ and $(d_1, b') \in \tilde{F}_\tau(R_2, S_2)$.

On the other hand, if $k(d) \downarrow \mathit{raise}_\tau(d_1)$, it must be $h(F_\sigma(g, f)(d)) \downarrow \mathit{raise}(d_2)$ for some d_2 such that $g(d_2) \downarrow d_1$. Following the same reasoning as above, we obtain $F_\sigma(g, f)(d) \downarrow d_3$ for some d_3 such that $(d_3, b) \in \tilde{F}_\sigma(S_1, R_1)$. From the assumption on h, there must exist a b' such that $[b/\texttt{x}]e \rightsquigarrow \texttt{raise}_\tau(b')$ and $(d_2, b') \in S_1$. Since $g : S_1 \subset S_2$, we obtain $(d_1, b') \in S_2$ as required. So, we have shown that $(k, \texttt{fn}\,\texttt{x}\!:\!\sigma \Rightarrow e)$ is in $\tilde{F}_{\sigma\to\tau}(R_2, S_2)$, and hence $F_{\sigma\to\tau}(f, g) : \tilde{F}_{\sigma\to\tau}(R_1, S_1) \subset \tilde{F}_{\sigma\to\tau}(R_2, S_2)$ as required. $\square$

In view of the above proposition, we shall remove the tilde from $\tilde{F}$. Note that condition C2 is satisfied by the members of F without the admissibility condition on S_2.

Theorem 3.6.5 *There exists a type-indexed family of relations $\leq_\sigma \subseteq [\![\sigma]\!]_{TMLE} \times Can_\sigma$ satisfying the clauses A1-4.*

Proof. The family of operations $(F_{\mathcal{E}})_{A,B} : \mathcal{R}(A) \times \mathcal{R}(B) \to \mathcal{R}(F_{\mathcal{E}}(A,B))$ defined as $F_{\mathcal{E}}(R,S) = \{((\epsilon,d),\epsilon(b) : \mathtt{exc}) \mid (d,b : \zeta(\epsilon)) \in F_{\zeta(\epsilon)}(R,S)\}$ is an admissible action on $\mathcal{R}$ of the functor $F_{\mathcal{E}}$ defined in section 3.5. This can be shown with routine calculations using proposition 3.6.4. Let $\Delta \in \mathcal{R}(E)$ be an invariant relation of $F_{\mathcal{E}}$, which exists by theorem 3.6.2. Note that, since $\alpha^{-1}(\epsilon(d)) = (\epsilon, d)$, the inclusion $\alpha^{-1} : \Delta \subset F_{\mathcal{E}}(\Delta,\Delta)$ implies that any pair in Δ is of the form $(\epsilon(d),\epsilon(b) : \mathtt{exc})$, and hence $\Delta = F_{\mathtt{exc}}(\Delta,\Delta)$. We show that the relations $\leq_\sigma \stackrel{\text{def}}{=} F_\sigma(\Delta,\Delta)$ form a family of approximation relations.

First, note that, for all σ, $F_\sigma(\Delta,\Delta) \subseteq [\![\sigma]\!]_{TMLE} \times Can_\sigma$. Moreover, since Δ is admissible, so is $F_\sigma(\Delta,\Delta)$ and hence clause A1 is satisfied. Clauses A2 and A4 are also satisfied, trivially. As for A3, we have to show that $(\epsilon(d),\epsilon(b) : \mathtt{exc}) \in \Delta$ if and only if $(d,b : \zeta(\epsilon)) \in F_{\zeta(\epsilon)}(\Delta,\Delta)$. Since $\alpha^{-1} : \Delta \subset F_{\mathcal{E}}(\Delta,\Delta)$, if $(\epsilon(d),\epsilon(b) : \mathtt{exc}) \in \Delta$, then $((\epsilon,d),\epsilon(b) : \mathtt{exc}) \in F_{\mathcal{E}}(\Delta,\Delta)$ and hence $(d,b : \zeta(\epsilon)) \in F_{\zeta(\epsilon)}(\Delta,\Delta)$. Conversely, if $(d,b : \zeta(\epsilon)) \in F_{\zeta(\epsilon)}(\Delta,\Delta)$, then $((\epsilon,d),\epsilon(b) : \mathtt{exc}) \in F_{\mathcal{E}}(\Delta,\Delta)$ and $(\epsilon(d),\epsilon(b) : \mathtt{exc}) \in \Delta$ follows from $\alpha : F_{\mathcal{E}}(\Delta,\Delta) \subset \Delta$. □

3.7 Computational adequacy

One of the criteria traditionally considered in the literature to establish how well denotational semantics captures the observable behaviour of programs is *computational adequacy*. Adequacy is concerned with the *data* produced by programs. For example, it requires that, if v is a canonical term of ground type, $[\![MN]\!] = [\![v]\!]$ if and only if $MN \rightsquigarrow v$. However, it does not say how $[\![M]\!]$ should behave when applied to possible elements of its domain which are not denoted by any N in the language: adequate models may fail to equate the denotations of two terms which agree on all operational observations. In [Blo88] and [Mey88], a denotational semantics is said to be computationally adequate with respect to a set $\mathcal{O}$ of observations when, for all *closed* terms M and M' of *ground* type, $[\![M]\!] = [\![M']\!]$ if and only if $M \cong_{\mathcal{O}} M'$, that is if and only if M and M' agree on all observations in $\mathcal{O}$.

Here we use operational semantics to observe TMLE programs. Therefore, since we interpret the programming language via a translation into the metalanguage, it makes sense to state computational adequacy as the requirement that the theory of the metalanguage and the operational semantics agree: let e be a closed TMLE program of type `nat`,

$$\vdash (\![e]\!) = (\![\mathtt{n}]\!) \text{ if and only if } e \leadsto \mathtt{n}.$$

Remark. A stronger version of the *if* direction of this statement holds, as shown by theorem 3.7.1, in which no restriction is put on the type of e. However, the *only if* direction does not apply to arrow types or exceptions. Since the latter may include parameters of arbitrary type, equations such as $(\![\epsilon(\mathtt{fn}\ \mathtt{x}\!:\!\sigma \Rightarrow x)]\!) = (\![\epsilon(\mathtt{fn}\ \mathtt{x}\!:\!\sigma \Rightarrow (\mathtt{fn}\ \mathtt{x}\!:\!\sigma \Rightarrow x)x)]\!)$ are clearly derivable in $ML_T(\Sigma)$ but neither of the two TMLE terms involved evaluates to the other.

Remark. If the programming language allowed nondeterministic computations, the above statement of adequacy would not do: equality must be replaced by a membership predicate, relating programs with the values they may possibly produce. Note that membership is an example of evaluation relation $\Leftarrow_X \subseteq X \times TX$, which we met already at the end of section 3.3. Using such predicates, adequacy can be rephrased as follows:

$$\vdash \underline{n} \Leftarrow (\![e]\!) \text{ if and only if } e \leadsto \mathtt{n}.$$

Theorem 3.7.1 (Adequacy: if) *Let e and c be closed TMLE programs, with c canonical; if $e \leadsto c$, then $\vdash (\![e]\!) = (\![c]\!)$.*

Proof. The proof is a rather lengthy induction on the derivations of operational judgements. As an example we consider derivations of e_1 `handle` $\epsilon(\mathtt{x}) \Rightarrow e_2 \leadsto c$ from (hnd3).

It must be that $e_1 \rightsquigarrow \texttt{raise}_\sigma\ \epsilon(b)$ and $[b/x]e_2 \rightsquigarrow c$ with shorter derivations and hence, by inductive hypothesis, $\llparenthesis e_1 \rrparenthesis = \llparenthesis \texttt{raise}_\sigma\ \epsilon(b) \rrparenthesis$ and $\llparenthesis [b/x]e_2 \rrparenthesis = \llparenthesis c \rrparenthesis$ are derivable. Then, omitting type indices:

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{\llparenthesis e_1 \rrparenthesis = \llparenthesis \texttt{raise}\ \epsilon(b) \rrparenthesis}
{\llparenthesis e_1 \rrparenthesis = \mathit{let}\ x \Leftarrow \mathit{val}(\underline{\epsilon}|b|)\ \mathit{in}\ \mathit{raise}(x)}
}{\llparenthesis e_1 \rrparenthesis = \mathit{raise}(\underline{\epsilon}|b|)}
}{\mathit{handle}_\epsilon(\llparenthesis e_1 \rrparenthesis, \lambda x.\, \llparenthesis e_2 \rrparenthesis) = \mathit{handle}_\epsilon(\mathit{raise}(\underline{\epsilon}|b|), \lambda x.\, \llparenthesis e_2 \rrparenthesis)}
}{\mathit{handle}_\epsilon(\llparenthesis e_1 \rrparenthesis, \lambda x.\, \llparenthesis e_2 \rrparenthesis) = (\lambda x.\, \llparenthesis e_2 \rrparenthesis)|b|}
\qquad
\dfrac{\llparenthesis [b/x]e_2 \rrparenthesis = \llparenthesis c \rrparenthesis}{[|b|/x]\llparenthesis e_2 \rrparenthesis = \llparenthesis c \rrparenthesis}
}{
\dfrac{\mathit{handle}_\epsilon(\llparenthesis e_1 \rrparenthesis, \lambda x.\, \llparenthesis e_2 \rrparenthesis) = \llparenthesis c \rrparenthesis}
{\llparenthesis e_1\ \texttt{handle}\ \epsilon(\texttt{x}) \Rightarrow e_2 \rrparenthesis = \llparenthesis c \rrparenthesis}
}
$$

□

An immediate consequence of the above and theorem 3.5.1 is the following:

Corollary 3.7.2 *Let e and c be as above; if $e \rightsquigarrow c$, then $\vdash \llbracket e \rrbracket_{TMLE} = \llbracket c \rrbracket_{TMLE}$.*

In order to prove the "only if" direction of the adequacy theorem, we use the fact that all TMLE programs are semantically aproximated by their interpretation. This is established by the following lemma, where $\preceq$ and $\leq$ are the families of approximation relations described in the previous section.

Lemma 3.7.3 *Let $\Gamma \vdash e : \sigma$ be a TMLE program, where $\Gamma = (\boldsymbol{x}_1 : \sigma_1, \ldots \boldsymbol{x}_n : \sigma_n)$, and let $d_i \leq_{\sigma_i} b_i$, for $1 \leq i \leq n$; then, $\llbracket \llparenthesis e \rrparenthesis \rrbracket_\Gamma (d_1 \ldots, d_n) \preceq_\sigma [b_i/\boldsymbol{x}_i]e$.*

Proof. We write $\llbracket e \rrbracket(\overline{d}\,)$ for $\llbracket \llparenthesis e \rrparenthesis \rrbracket_\Gamma(d_1, \ldots d_n)$ and $[\,\overline{b}\,]e$ for $[b_1/\texttt{x}_1] \ldots [b_n/\texttt{x}_n]e$; for such $\overline{d}$ and $\overline{b}$, we shall understand that $d_i \leq_{\sigma_i} b_i$, for $1 \leq i \leq n$. The proof is by induction in the derivation of $\Gamma \vdash e : \sigma$. The cases where the last rule applied is (var) or (n) are trivial.

(Op). $\llbracket \texttt{Op}(e_1, e_2) \rrbracket(\overline{d}\,) = \mathit{let}(\lambda x : \mathfrak{N}.\, \mathit{let}(\lambda y : \mathfrak{N}.\, \mathit{val}(\mathit{op}(x, y)))\, \llbracket e_2 \rrbracket(\overline{d}\,))\, \llbracket e_1 \rrbracket(\overline{d}\,)$; if this expression denotes $\mathit{val}(m)$, then $\llbracket e_1 \rrbracket(\overline{d}\,)$ cannot be $\perp$ since $(\mathit{let}\ f)$ is strict, for any f, and $val(m) = val_\perp(inj_1\, m)$ is different from $\perp$. Similarly, $[\![e_1]\!](\overline{d}\,)$ cannot be $raise\,(z)$. In fact, from

$$\begin{aligned}
&let\ (\ldots)\ raise\,(z) = \\
&let_\perp(case\,((\ldots), raise))\ val_\perp(inj_2\, z) = \\
&case\,((\ldots), raise\,, inj_2(z)) = \\
&raise\,(z)
\end{aligned}$$

it would follow $val_\perp(inj_1\, m) = val_\perp(inj_2\, z)$, which cannot be, since $inj_1(m) \neq inj_2(z)$ and $val_\perp$ is injective. Therefore, $[\![e_1]\!](\overline{d}\,)$ must be $val(n_1)$, for some n_1, and hence $[\![\mathtt{Op}(e_1, e_2)]\!](\overline{d}\,) = let(\lambda\, y : \mathfrak{N}.\, val(op\,(n_1, y)))\, [\![e_2]\!](\overline{d}\,)$. By inductive hypothesis, we can assume $[\,\overline{b}\,]e_1 \rightsquigarrow b_1$, for some b_1 such that $n_1 \leq_{\mathtt{nat}} b_1$, which means that $b_1 \equiv \mathtt{n}_1$. By a similar argument, we obtain $[\,\overline{b}\,]e_2 \rightsquigarrow \mathtt{n}_2$ and $m = op\,(n_1, n_2)$. Then, from (Op1), $[\,\overline{b}\,]\mathtt{Op}(e_1, e_2) \rightsquigarrow \mathtt{m}$.

Alternatively, assume $[\![\mathtt{Op}(e_1, e_2)]\!](\overline{d}\,)$ is $raise\,(d)$. As $[\![e_1]\!](\overline{d}\,)$ cannot be $\perp$, it must either be $val(n)$ or $raise\,(z)$. In the first case, we can assume $[\,\overline{b}\,]e_1 \rightsquigarrow \mathtt{n}$. Moreover, it is easy to see that $[\![e_2]\!](\overline{d}\,)$ can be neither a value nor $\perp$. So we assume $[\![e_2]\!](\overline{d}\,) = raise\,(w)$. Then,

$$\begin{aligned}
&raise\,(d) = let\,(\lambda\, x : \mathfrak{N}.\, let\,(\lambda\, y : \mathfrak{N}.\, val(op\,(x, y)))\ raise\,(w))\ val(n) = \\
&let\,(\lambda\, y : \mathfrak{N}.\, val(op\,(n, y)))raise\,(w) = \\
&let_\perp(case\,(\lambda\, y : \mathfrak{N}.\, val(op\,(n, y))),\ raise))val_\perp(inj_2\, w) \\
&case\,(\lambda\, y : \mathfrak{N}.\, val(op\,(n, y))),\ raise\ ,\ inj_2(w)) = raise\,(w)
\end{aligned}$$

forces $w = d$. The inductive hypothesis on $[\![e_2]\!](\overline{d}\,)$ yields $[\,\overline{b}\,]e_2 \rightsquigarrow \mathtt{raise}_{\mathtt{nat}} b$ for some b such that $d \leq_{\mathtt{exc}} b$. Then, using (Op3), we obtain $[\,\overline{b}\,]\mathtt{Op}(e_1, e_2) \rightsquigarrow \mathtt{raise}_{\mathtt{nat}} b$ as required. A similar argument yields the result in the case of $[\![e_1]\!](\overline{d}\,) = raise\,(z)$.

(if). $[\![\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2]\!](\overline{d}\,) = let[\lambda\, x : 1.\, [\![e_1]\!](\overline{d}\,)\,,\ \lambda\, x : \sigma.\, [\![e_2]\!](\overline{d}\,)]\ [\![e]\!](\overline{d}\,)$; assume this expression denotes $val(d)$. Since it must be $[\![e]\!](\overline{d}\,) = val(n)$, we can assume $[\,\overline{b}\,]e \rightsquigarrow \mathtt{n}$. If $n = 0$, from the definition of $[\_, \_]$, we obtain $[\![\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2]\!](\overline{d}\,) = [\![e_1]\!](\overline{d}\,)$.

Then, assuming $[\,\overline{b}\,]e_1 \rightsquigarrow b$ for a b such that $d \leq_\sigma b$, the result follows from (if1). Similarly, the result follows from (if2) when $n \neq 0$. The case where $[\![\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2]\!](\overline{d}\,)$ is $\mathit{raise}(d)$ follows the same pattern as for (Op).

(fn). Since $[\![\mathtt{fn}\ \mathtt{x}{:}\sigma \Rightarrow e]\!](\overline{d}\,) = \mathit{val}(\lambda\, x : [\![\sigma]\!].\, [\![e]\!](\overline{d}\,))$, we are required to prove that $\lambda\, x : [\![\sigma]\!].\, [\![e]\!](\overline{d}\,) \leq_{\sigma\to\tau} \mathtt{fn}\ \mathtt{x}{:}\sigma \Rightarrow [\,\overline{b}\,]e$. This amounts to showing that, for d and b such that $d \leq_\sigma b$, $[\![e]\!](\overline{d}, d) \preceq_\tau [\,\overline{b}, b]e$, which is immediate by inductive hypothesis.

(appl). $[\![e_1 e_2]\!](\overline{d}\,) = \mathit{let}(\lambda\, f : [\![\sigma]\!] \to T[\![\tau]\!].\, (\mathit{let}\ f\ [\![e_2]\!](\overline{d}\,)))\ [\![e_1]\!](\overline{d}\,)$. Assume this expression denotes $\mathit{val}(d)$. As above, $[\![e_1]\!](\overline{d}\,)$ must be $\mathit{val}(f)$, for some f, and so we have $[\![e_1 e_2]\!](\overline{d}\,) = \mathit{let}\ f\ [\![e_2]\!](\overline{d}\,)$. By the inductive hypothesis on e_1, $[\,\overline{b}\,]e_1 \rightsquigarrow \mathtt{fn}\ \mathtt{x}{:}\sigma \Rightarrow e$, with $f \leq_{\sigma\to\tau} \mathtt{fn}\ \mathtt{x}{:}\sigma \Rightarrow e$. By a similar argument, $[\![e_2]\!](\overline{d}\,) = \mathit{val}(d_1)$, $[\![e_1 e_2]\!](\overline{d}\,) = f(d_1)$, $[\,\overline{b}\,]e_2 \rightsquigarrow b_1$ and $d_1 \leq_\sigma b_1$. Then, from the assumptions, $\mathit{val}(d) = f(d_1) \preceq_\tau [b_1/\mathtt{x}]e \rightsquigarrow b_2$ and $d \leq_\tau b_2$. Applying (app1), we obtain $[\,\overline{b}\,](e_1 e_2) \rightsquigarrow b_2$ as required.

Alternatively, if $[\![e_1 e_2]\!](\overline{d}\,) = \mathit{raise}(d)$, $[\![e_1]\!](\overline{d}\,)$ must be either $\mathit{val}(f)$ or $\mathit{raise}(z)$. In the first case, we can assume $[\,\overline{b}\,]e_1 \rightsquigarrow \mathtt{fn}\ \mathtt{x}{:}\sigma \Rightarrow e$ and $f \leq_{\sigma\to\tau} \mathtt{fn}\ \mathtt{x}{:}\sigma \Rightarrow e$. From $\mathit{let}\ f\ [\![e_2]\!](\overline{d}\,) = \mathit{raise}(d)$, $[\![e_2]\!](\overline{d}\,)$ can be either $\mathit{val}(d_1)$ or $\mathit{raise}(z)$. In the first case, we can assume $[\,\overline{b}\,]e_2 \rightsquigarrow b_1$, for some b_1 such that $d_1 \leq_\sigma b_1$. From the assumptions, we obtain $\mathit{raise}(d) = f(d_1) \preceq_\tau [b_1]e$ and hence $[b_1]e \rightsquigarrow \mathtt{raise}_\tau\, b$ for some b such that $d \leq_{\mathtt{exc}} b$. Then, from (app1), $[\,\overline{b}\,](e_1, e_2) \rightsquigarrow \mathtt{raise}_\tau\, b$ as required. The cases where e_1 or e_2 raise exceptions follow the same routine.

(ϵ). $[\![\epsilon(e)]\!](\overline{d}\,) = \mathit{let}(\lambda\, x : [\![\zeta(\epsilon)]\!].\, \mathit{val}(\epsilon(x)))[\![e]\!](\overline{d}\,)$. Assume this expression denotes $\mathit{val}(d)$. It cannot be that $[\![e]\!](\overline{d}\,) = \mathit{raise}(d_1)$, because this would imply the identity $\mathit{val}(d) = \mathit{let}_\perp(\mathit{case}(\lambda\, x : [\![\zeta(\epsilon)]\!].\, \mathit{val}(\epsilon(x))\,,\, \mathit{raise}))\, \mathit{val}_\perp(\mathit{inj}_2\, d_1) = \mathit{raise}(d_1)$. Hence, it must be $[\![e]\!](\overline{d}\,) = \mathit{val}(d_1)$ and we can assume $[\,\overline{b}\,]e \rightsquigarrow b$, for some b such that $d_1 \leq_{\zeta(\epsilon)} b$. From ($\epsilon$1), $[\,\overline{b}\,]\epsilon(e) \rightsquigarrow \epsilon(b)$. From $\mathit{val}(d) = \mathit{let}(\lambda\, x : [\![\zeta(\epsilon)]\!].\, \mathit{val}(\epsilon(x)))\, \mathit{val}(d_1) = \mathit{val}(\epsilon(d_1))$, we have $d = \epsilon(d_1) \leq_{\mathtt{exc}} \epsilon(b)$ as required.

Otherwise, if $[\![\epsilon(e)]\!](\overline{d}\,) = \mathit{raise}(d)$, it is easy to verify that there must be a d_1 such that $[\![e]\!](\overline{d}\,) = \mathit{raise}(d_1)$ and therefore we can assume $[\,\overline{b}\,]e \rightsquigarrow \mathtt{raise}\, b$, for some b such that $d_1 \leq_{\mathtt{exc}} b$. Hence, applying the rule (ϵ2), we obtain $[\,\overline{b}\,]\epsilon(e) \rightsquigarrow \mathtt{raise}_{\mathtt{exc}}\, b$. Moreover, from $raise(d) = let(\lambda x : [\![\zeta(\epsilon)]\!].\, val(\epsilon(x)))\, raise(d_1) = raise(d_1)$, it is $d = d_1 \leq_{\texttt{exc}} b$ as required.

(raise). The expression $[\![\texttt{raise}_\sigma e]\!](\overline{d}) = (let\ raise)\, [\![e]\!](\overline{d})$ cannot denote a value. Assume it denotes $raise(d)$. Since $[\![e]\!](\overline{d})$ cannot be $\perp$, it must be either $val(d_1)$ or $raise(d_1)$. In the first case, we can assume $[\,\overline{b}\,]e \rightsquigarrow b$, for some b such that $d_1 \leq_{\texttt{exc}} b$. Moreover, from $raise(d) = (let\ raise)(val\ d_1) = raise(d_1)$, it follows that $d = d_1$. Then, by (rs1), $[\,\overline{b}\,]\texttt{raise}_\sigma e \rightsquigarrow raise_\sigma b$ as required.

Otherwise, if $[\![e]\!](\overline{d}) = raise(d_1)$, we can assume $[\,\overline{b}\,]e \rightsquigarrow \texttt{raise}_{\texttt{exc}}\, b$, for some b such that $d_1 \leq_{\texttt{exc}} b$, and hence, by (rs2), $[\,\overline{b}\,]\texttt{raise}_\sigma\, e \rightsquigarrow \texttt{raise}_\sigma\, b$. Moreover, from $raise(d) = (let\ raise)\,(raise\ d_1) = let_\perp(case(raise, raise))\, val_\perp(inj_2\, d_1) = raise(d_1)$, it follows that $d = d_1$ and hence $d \leq_{\texttt{exc}} b$ as required.

(handle). $[\![e_1\ \texttt{handle}\ \epsilon(\texttt{x}) \Rightarrow e_2]\!](\overline{d}) = let_\perp(case(val, \overline{e}_2))\, [\![e_1]\!](\overline{d})$, where $\overline{e}_2$ stands for $\lambda x : E.\, case(\lambda x : [\![\zeta(\epsilon)]\!].\, [\![e_2]\!](\overline{d})\,,\ raise\ ,\ \overline{\epsilon}\, x)$. Assume this expression denotes $val(d)$; then, $[\![e_1]\!](\overline{d})$ denotes either $val(d_1)$ or $raise(d_1)$. In the first case, we can assume that $[\,\overline{b}\,]e_1 \rightsquigarrow b$, for a b such that $d_1 \leq_\sigma b$. Then, by (hnd1), $[\,\overline{b}\,](e_1\ \texttt{handle}\ \epsilon(\texttt{x}) \Rightarrow e_2) \rightsquigarrow b$ and, from $val(d) = let_\perp(case(val, \overline{e}_2))\, val(d_1) = val(d_1)$, we have $d = d_1 \leq_\sigma b$ as required.

If $[\![e_1]\!](\overline{d}) = raise(d_1)$, we can assume $[\,\overline{b}\,]e_1 \rightsquigarrow \texttt{raise}_\sigma\, b_1$ and $d_1 \leq_{\texttt{exc}} b_1$. Moreover, $val(d) = let_\perp(case(val, \overline{e}_2))\, raise(d_1) = \overline{e}_2(d_1) = case(\lambda x : [\![\zeta(\epsilon)]\!].\, [\![e_2]\!](\overline{d})\,,\ raise\ ,\ \overline{\epsilon}\, d_1)$. From the definition of $\leq_{\texttt{exc}}$, it must be $d_1 = \epsilon_1(d_2)$ and $b_1 = \epsilon_1(b_2)$, for some ϵ_1 and $d_2 \leq_{\zeta(\epsilon_1)} b_2$. Moreover, it must be $\epsilon = \epsilon_1$ because, otherwise,

$$
\begin{aligned}
& val(d) = \\
& case(\lambda x : [\![\zeta(\epsilon)]\!].\, [\![e_2]\!](\overline{d})\,,\ raise\ ,\ \overline{\epsilon}(\epsilon_1\, d_2)) = \\
& case(\lambda x : [\![\zeta(\epsilon)]\!].\, [\![e_2]\!](\overline{d})\,,\ raise\ ,\ inj_2(\epsilon_1\, d_2)) = \\
& raise(\epsilon_1(d_2)).
\end{aligned}
$$

Hence, from $\overline{\epsilon}(\epsilon\, d_1) = inj_1(d_1)$, we obtain $val(d) = [\![e_2]\!](\overline{d}, d_2)$ and so we can assume $[\,\overline{b}, b_2]e_2 \rightsquigarrow b$, for some b such that $d \leq_\sigma b$. Then, $[\,\overline{b}\,](e_1\ \texttt{handle}\ \epsilon(\texttt{x}) \Rightarrow e_2) \rightsquigarrow b$ follows from (hnd3) as required.

Finally, if $[\![e_1\ \mathtt{handle}\ \epsilon(\mathtt{x}) \Rightarrow e_2]\!](\overline{d}\,) = let_{\perp}(case\,(val, \overline{e}_2))\,[\![e_1]\!](\overline{d}\,) = raise\,(d)$, then $[\![e_1]\!](\overline{d}\,)$ can be neither a value nor $\perp$. So, it must be $[\![e_1]\!](\overline{d}\,) = raise\,(d_1)$ and we can assume $[\,\overline{b}\,]e_1 \leadsto \mathtt{raise}_{\sigma}\, b_1$, for some b_1 such that $d_1 \leq_{\mathtt{exc}} b_1$. As above, it must be $d_1 = \epsilon_1(d_2)$ and $b_1 = \epsilon_1(b_2)$, with $d_2 \leq_{\zeta(\epsilon_1)} b_2$. From the assumptions, we have:

$$\begin{aligned}
&[\![e_1\ \mathtt{handle}\ \epsilon(\mathtt{x}) \Rightarrow e_2]\!](\overline{d}\,) = \\
&let_{\perp}(case\,(val, \overline{e}_2))\, raise\,(\epsilon_1\, d_2) = \overline{e}_2(\epsilon_1\, d_2) = \\
&case\,(\lambda\, x : [\![\zeta(\epsilon)]\!].\, [\![e_2]\!](\overline{d}\,)\,,\ raise\ ,\ \overline{\epsilon}(\epsilon_1\, d_2)).
\end{aligned}$$

If $\epsilon = \epsilon_1$, we obtain $raise\,(d) = [\![e_2]\!](\overline{d}, d_2)$, and we can assume $[\,\overline{b}, b_2]e_2 \leadsto \mathtt{raise}_{\sigma}\, b$, for b such that $d \leq_{\zeta(\epsilon)} b$. Then, applying (hnd3), $[\,\overline{b}\,](e_1\ \mathtt{handle}\ \epsilon(\mathtt{x}) \Rightarrow e_2) \leadsto \mathtt{raise}_{\sigma}\, \epsilon(b)$ as required. Otherwise, if $\epsilon \neq \epsilon_1$, we obtain $raise\,(d) = raise\,(d_2)$, which implies $d = d_2$ and hence $d \leq_{\zeta(\epsilon_1)} b_2$, Moreover, applying (hnd2), $[\,\overline{b}\,](e_1\ \mathtt{handle}\ \epsilon(\mathtt{x}) \Rightarrow e_2) \leadsto \mathtt{raise}_{\sigma}\, \epsilon_1(b_2)$ as required. $\square$

Theorem 3.7.4 (Adequacy: only if) *Let e be a closed TMLE program of type $\mathtt{nat}$; if $\vdash (\![e]\!) = (\![n]\!)$ then $e \leadsto \mathtt{n}$.*

Proof. By soundness of $ML_T(\Sigma)$, it is required to prove that, if $[\![e]\!]_{TMLE} = val\,(n)$, then $e \leadsto \mathtt{n}$. Assuming $[\![e]\!]_{TMLE} = val\,(n)$, lemma 3.7.3 ensures that there exists a b such that $e \leadsto b$ and $n \leq_{\mathtt{nat}} b$. By definition of $\leq_{\mathtt{nat}}$, b must be $\mathtt{n}$ and hence $e \leadsto \mathtt{n}$. $\square$

4 Evaluation Logic

In Moggi's categorical semantics of computations, a strong monad T provides the abstract structure for interpreting computations: it provides computational types TX, for modelling programs computing values of type X, and a canonical lifting of computations with a parameter of type X to computations with a parameter of type TX. In this framework, composition of programs is captured with no commitment to any specific form of computation.

Evaluation Logic (EL) applies the same idea to first order predicate calculus: predicates ϕ over values of type X lift to predicates over programs of type TX, expressing the property of producing values satisfying ϕ, independently of the notion of evaluation. Since the term language of the logic, that is the computational lambda calculus, stems from an attempt to explain computation abstractly, EL's semantics calls for an abstract understanding of the interaction between logic and computation.

Typically, interpretation of programs require domain-like objects. Categories of such objects can hardly accommodate anything more than a geometric logic. However, there are nonobservable properties, such as "f is total," which can be expressed in EL, viz. $\forall\, n.\, \langle x \Leftarrow f n\rangle\, \mathit{true}$, but cannot be *classified* in a category of predomains. Even if not classifiable, such predicates could still be interpreted in such a category, without resorting to topos-teoretic structure. Nevertheless, as shown in section 4.4, having a subobject classifier allows a "standard" interpretation of the logic from the categorical structure of a strong monad T. Since standard interpretation is desirable for modularizing the construction of categorical models, as we see in chapter 5, we regard models with a classifying object as desirable.

An expressive program logic with a standard semantics can be obtained in the setting of *synthetic domain theory*. Based on the slogan that domains are sets, synthetic domain theory seeks a category $\mathcal{C}$ with all good properties for interpreting computations and which is fully reflective in an ambient topos $\mathcal{E}$, with all good properties for interpreting logics. There are many significant examples of this situation among subcategories of PERs in the effective topos. Since standard models require a strong monad T to "act" on predicates, which live in $\mathcal{E}$, we must understand under what circumstances T, whose natural domain is the category $\mathcal{C}$, where computations run, can be assumed to live in $\mathcal{E}$, where the logic is interpreted.

In section 4.1 we introduce the calculus. Section 4.2 presents Pitts' original semantics [Pit91] based on hyperdoctrines. The following three sections describe a standard interpretation of EL based on the simplifying assumptions that a strong monad T is available in the category $\mathcal{E}$ where the logic is interpreted and that it has good interaction with the logical structure. Section 4.6 shows that the first of these assumptions can be made with no loss of generality because any monad T on a category $\mathcal{C}$ can be extended to a monad $\tilde{T}$ on an ambient category $\mathcal{E}$ in which $\mathcal{C}$ is fully reflective by composing T with the reflection. While the construction of $\tilde{T}$ is well known, we claim the observation that the category of T-algebras is equivalent to the category of $\tilde{T}$-algebras (theorem 4.6.3). This equivalence shows that $\tilde{T}$ is in a sense "minimal" among the monads on $\mathcal{E}$ which extend T. We also find necessary and sufficient conditions to make $\tilde{T}$ strong when T is strong.

Section 4.7 describes a more general standard semantics for Evaluation Logic, introduced in [Moga] to overcome the failure of the second of the above assumptions.

In the last section we propose a new standard semantics based on *evaluation relations*. We already came across such relations on several earlier occasions: in section 3.3 we gave an example of their use for expressing general axioms of a computational metalanguage, while in section 3.7 we suggested the use of evaluation relations to state metatheoretical properties such as computational adequacy. Moreover, in section 4.4 we show the pragmatical advantages of using such predicates in a program logic: in the context of

Evaluation Logic, they provide left and right rules for the box modality (see next section), thus making the proof system more manageable.

Evaluation relations can be defined from an endofunctor and first order quantification and hence they may provide a standard interpretation of Evaluation Logic when the structure required in [Moga] is not available. We also show how our definition relates with the one in [Mogb] where higher order structure is required. This is our main contribution in this chapter.

Some of the propositions stated below are left unproven. When this happens, the result is either well known or it follows from routine calculation or the proof is available in the cited literature.

4.1 The calculus

Evaluation Logic is a typed predicate calculus with equality and "evaluation" modalities based on the computational lambda calculus presented in chapter 2.

The types and terms of the logic are the same as in section 2.3. The well-formed formulae are given by formation judgements $\Gamma \vdash \phi\ prop$ which are derived from the following rules:

$$\frac{}{\vdash \mathit{false}\ prop}$$

$$\frac{\Gamma \vdash M : \tau \quad \Gamma \vdash N : \tau}{\Gamma \vdash M = N\ prop}$$

$$\frac{\Gamma \vdash \phi\ prop \quad \Gamma \vdash \psi\ prop}{\Gamma \vdash \phi \supset \psi\ prop}$$

$$\frac{\Gamma \vdash \phi \ \mathit{prop} \quad \Gamma \vdash \psi \ \mathit{prop}}{\Gamma \vdash \phi \wedge \psi \ \mathit{prop}}$$

$$\frac{\Gamma \vdash \phi \ \mathit{prop} \quad \Gamma \vdash \psi \ \mathit{prop}}{\Gamma \vdash \phi \vee \psi \ \mathit{prop}}$$

$$\frac{\Gamma, x : \tau \vdash \phi \ \mathit{prop}}{\Gamma \vdash \forall x : \tau.\ \phi \ \mathit{prop}}$$

$$\frac{\Gamma, x : \tau \vdash \phi \ \mathit{prop}}{\Gamma \vdash \exists x : \tau.\ \phi \ \mathit{prop}}$$

$$\frac{\Gamma, x : \tau \vdash \phi \ \mathit{prop} \quad \Gamma \vdash M : T\tau}{\Gamma \vdash [x \Leftarrow M]\ \phi \ \mathit{prop}}$$

$$\frac{\Gamma, x : \tau \vdash \phi \ \mathit{prop} \quad \Gamma \vdash M : T\tau}{\Gamma \vdash \langle x \Leftarrow M \rangle\ \phi \ \mathit{prop}}$$

We write "*true*" for $* = *$ and "$\neg\phi$" for $\phi \supset \mathit{false}$. The modal operators above are called *necessity* and *possibility* and will sometimes be written "$\Box$" and "$\Diamond$." We assume that such operators have the same precedence of $\forall$ and $\exists$, which we assume higher than that of all other logical connectives.

The intended meaning of $[x \Leftarrow M]\ \phi$ is that for all values x to which $M : TX$ evaluates, $\phi(x)$ holds, while $\langle x \Leftarrow M \rangle\ \phi$ says that there exists a value x to which M evaluates such that $\phi(x)$. Indeed, $\Box$ and $\Diamond$ are more *quantifiers* over values produced by computations than they are *modalities* in the traditional sense. It may help the intuition to consider the following set theoretic interpretation for computations with side effects. Let the computational type TX be defined as $(X \times S)_\perp^S$.

$$[\![w : TX \vdash [x \Leftarrow w]\ \phi\ prop]\!] \;=\; \{w \mid \forall\, s, x, s'.\, ws = (x, s') \supset \phi(x)\} \tag{4.1}$$

$$[\![w : TX \vdash \langle x \Leftarrow w\rangle\ \phi\ prop]\!] \;=\; \{w \mid \exists\, s, x, s'.\, ws = (x, s') \wedge \phi(x)\} \tag{4.2}$$

The basic *truth* judgements of the logic are intuitionistic sequents in context, written $\Gamma; \Delta \vdash \phi$, where Γ is a type context, Δ a finite set of formulae and ϕ is a formula, expressing the fact that ϕ is logically entailed by the assumptions in Δ. We write $\Gamma \vdash \phi$ for $\Gamma; \emptyset \vdash \phi$. In [CP92] it is argued that, an *intuitionistic* framework is more suitable than a classical one for capturing the behavioural properties of computation.

The rules for deriving truth judgements include the axioms of the computational lambda calculus introduced in section 2.3, the standard rules for intuitionistic predicate calculus with equality, and the following rules concerning the evaluation modalities. We divide them into *general* and *special* depending on whether or not they describe inferences that can be made for any notion of computation. Furthermore, we distinguish between rules that involve only the structure of a strong endofunctor and rules involving the operation of a monad.

General axioms for strong endofunctors:

$$\Box\_true \quad \frac{\Gamma \vdash M : T\tau}{\Gamma \vdash [x \Leftarrow M]\ true} \qquad\qquad \Diamond\_false \quad \frac{\Gamma \vdash M : T\tau}{\Gamma \vdash \neg\langle x \Leftarrow M\rangle\ false}$$

$$\Box\_intro \quad \frac{\Gamma, x : \tau; \Delta, \phi \vdash \psi}{\Gamma, z : T\tau; \Delta, [x \Leftarrow z]\ \phi \vdash [x \Leftarrow z]\ \psi} \quad x \notin FV(\Delta)$$

$$\Diamond\_intro \quad \frac{\Gamma, x : \tau; \Delta, \phi \vdash \psi}{\Gamma, z : T\tau; \Delta, \langle x \Leftarrow z\rangle\ \phi \vdash \langle x \Leftarrow z\rangle\ \psi} \quad x \notin FV(\Delta)$$

$$\Box\_\wedge \quad \frac{\Gamma \vdash M : T\tau \quad \Gamma, x : \tau \vdash \phi\ prop \quad \Gamma, x : \tau \vdash \psi\ prop}{\Gamma; [x \Leftarrow M]\ \phi, [x \Leftarrow M]\ \psi \vdash [x \Leftarrow M]\ (\phi \wedge \psi)}$$

$$\Diamond\_\vee \quad \frac{\Gamma \vdash M : T\tau \quad \Gamma, x : \tau \vdash \phi\ prop \quad \Gamma, x : \tau \vdash \psi\ prop}{\Gamma; \langle x \Leftarrow M\rangle\ (\phi \vee \psi) \vdash \langle x \Leftarrow M\rangle\ \phi \vee \langle x \Leftarrow M\rangle\ \psi}$$

$$\Box\_\Diamond \quad \frac{\Gamma \vdash M : T\tau \quad \Gamma, x : \tau \vdash \phi\ prop \quad \Gamma, x : \tau \vdash \psi\ prop}{\Gamma; [x \Leftarrow M]\ \phi, \langle x \Leftarrow M\rangle\ \psi \vdash \langle x \Leftarrow M\rangle\ (\phi \wedge \psi)}$$

General axioms for strong monads:

$$\Box\_val \quad \frac{\Gamma \vdash M : \tau \quad \Gamma, x : \tau \vdash \phi\ prop}{\Gamma; \phi(M) \vdash [x \Leftarrow val(M)]\ \phi} \qquad \Diamond\_val \quad \frac{\Gamma \vdash M : \tau \quad \Gamma, x : \tau \vdash \phi\ prop}{\Gamma; \langle x \Leftarrow val(M)\rangle\ \phi \vdash \phi(M)}$$

$$\Box\_let \quad \frac{\Gamma \vdash M : T\tau \quad \Gamma, x : \tau \vdash N : T\sigma \quad \Gamma, y : \sigma \vdash \phi\ prop}{\Gamma; [x \Leftarrow M]\ [y \Leftarrow N]\ \phi \vdash [y \Leftarrow (let\ x \Leftarrow M\ in\ N)]\ \phi}$$

$$\Diamond\_let \quad \frac{\Gamma \vdash M : T\tau \quad \Gamma, x : \tau \vdash N : T\sigma \quad \Gamma, y : \sigma \vdash \phi\ prop}{\Gamma; \langle y \Leftarrow (let\ x \Leftarrow M\ in\ N)\rangle\ \phi \vdash \langle x \Leftarrow M\rangle\ \langle y \Leftarrow N\rangle\ \phi}$$

Special axioms for strong endofunctors:

$$\Box\_\supset \quad \frac{\Gamma \vdash \phi\ prop \quad \Gamma, x : \tau \vdash \psi\ prop}{\Gamma; \phi \supset [x \Leftarrow M]\ \psi \vdash [x \Leftarrow M]\ (\phi \supset \psi)}$$

$$\Box\_\forall \quad \frac{\Gamma \vdash M : T\tau \quad \Gamma, x : \tau, y : \sigma \vdash \phi\ prop}{\Gamma; \forall\, y : \sigma.\ [x \Leftarrow M]\ \phi \vdash [x \Leftarrow M]\ \forall\, y : \sigma.\ \phi}$$

$$\Diamond\_\exists \quad \frac{\Gamma \vdash M : T\tau \quad \Gamma, x : \tau, y : \sigma \vdash \phi\ prop}{\Gamma; \langle x \Leftarrow M\rangle\ \exists\, y : \sigma.\ \phi \vdash \exists\, y : \sigma.\ \langle x \Leftarrow M\rangle\ \phi}$$

$$\Diamond\_= \quad \frac{\Gamma \vdash M : \tau \quad \Gamma \vdash N : \tau \quad \Gamma \vdash L : T\sigma \quad \Gamma, x : \sigma \vdash \phi\ prop}{\Gamma; M = N \wedge \langle x \Leftarrow L\rangle\ \phi \dashv\vdash \langle x \Leftarrow L\rangle\ (M = N \wedge \phi)}$$

Special axioms for strong monads:

$$\Box\_T^* \quad \frac{\Gamma \vdash M : T\tau \quad \Gamma, x : \tau \vdash N : \sigma \quad \Gamma, y : \sigma \vdash \phi \ prop}{\Gamma; [y \Leftarrow (let\ x \Leftarrow M\ in\ val(N))]\ \phi(y) \vdash [x \Leftarrow M]\ \phi(N)} \quad (\dashv \text{ for } \Diamond\_T^*)$$

$$\Box\_val^* \quad \frac{\Gamma \vdash M : \tau \quad \Gamma, x : \tau \vdash \phi \ prop}{\Gamma; [x \Leftarrow val(M)]\ \phi \vdash \phi(M)} \quad (\dashv \text{ for } \Diamond\_val^*)$$

$$\Box\_let^* \quad \frac{\Gamma \vdash M : T\tau \quad \Gamma, x : \tau \vdash N : T\sigma \quad \Gamma, y : \sigma \vdash \phi \ prop}{\Gamma; [y \Leftarrow (let\ x \Leftarrow M\ in\ N)]\ \phi \vdash [x \Leftarrow M]\ [y \Leftarrow N]\ \phi} \quad (\dashv \text{ for } \Diamond\_let^*)$$

Note that the opposites of $\Box\_\wedge$, $\Diamond\_\vee$, $\Box\_\supset$, $\Box\_\forall$ and $\Diamond\_\exists$ follow from $\Box\_intro$, $\Diamond\_intro$ and the usual axioms for $\wedge$, $\vee$, $\supset$, $\forall$ and $\exists$. Except for quantification ($\forall$ and $\exists$) and implication ($\supset$), which are not considered in [Pit91], Pitts' axioms are the same as the above general and special axioms, provided Δ is empty in $\Box\_intro$ and $\Diamond\_intro$. The reason for such a restriction is discussed in section 5.3.

Remark. The above axioms are presented so as to match the $\langle T, val, let\rangle$ view of a strong monad. In [Mogb], a different set of axioms is given, matching the $\langle T, \eta, \mu, t\rangle$ view. Moggi's axioms are equivalent to our general rules for necessity plus $\Box\_T^*$. In particular, Moggi's $\Box\_\mu$ is an instance of $\Box\_let$, $\Box\_t^*$ follows from $\Box\_\wedge$ and $\Box\_T^*$, while $\Box\_T$ follows from $\Box\_val$ and $\Box\_let$. Note also that $\Box\_T^*$ follows from $\Box\_val^*$ and $\Box\_let$.

Remark. $\Box\_val^*$ implies the *mono requirement*, that is $val(M) = val(N) \vdash M = N$. In fact, taking $\phi(x) \equiv (x = N)$, we have:

$$\phi(N) \vdash [x \Leftarrow val(N)]\ \phi(x) \vdash [x \Leftarrow val(M)]\ \phi(x) \vdash \phi(M) \equiv (M = N).$$

Remark. As pointed out in [Pit91], the following evaluation properties of a program $M : T\tau$ can be expressed in the logic:

"M can evaluate to v" (written $v \Leftarrow M$): $\langle x \Leftarrow M\rangle\,(x = v)$

"M can produce a value" ($M \Downarrow$): $\langle x \Leftarrow M\rangle$ *true*

"M cannot produce a value" ($M \Uparrow$): $[x \Leftarrow M]$ *false*

Others, such as "M must evaluate to v" cannot be expressed.

4.2 Hyperdoctrine semantics

Pitts' interpretation of Evaluation Logic is based on Lawvere's categorical notion of *hyperdoctrine* ([Law69,Law70]). Such a notion can be variously specialized in order to suit the particular features of the logic being considered. We first focus on the connectives of the calculus presented in section 4.1 and later on the modalities.

Definition 4.2.1 *A* first order hyperdoctrine *over a category $\mathcal{C}$ with products is a pseudo-functor $\mathcal{H} : \mathcal{C}^{op} \to$ Preord such that:*

- *each fibre $\mathcal{H}(X)$ is a Heyting prealgebra, that is it has finite meets, joins and Heyting implication; moreover, the functions $\mathcal{H}(f)$ preserve this structure;*

- *for all projections $\pi_X : Y \times X \to Y$, the functions $\mathcal{H}(\pi_X)$ have left and right adjoints $\exists_X \dashv \mathcal{H}(\pi_X) \dashv \forall_X$ satisfying the Beck-Chevalley condition: for all $f : A \to B$,*

$$\begin{tikzcd}
\mathcal{H}(B) \arrow[r, "\mathcal{H}(f)"] \arrow[dr, phantom, "\cong"] & \mathcal{H}(A) \\
\mathcal{H}(B \times X) \arrow[u, "{\exists_X, \forall_X}"] \arrow[r, "\mathcal{H}(f \times X)"'] & \mathcal{H}(A \times X) \arrow[u, "{\exists_X, \forall_X}"']
\end{tikzcd}$$

In the above definition $\mathcal{C}$ is called the *base* category; meets and joins are written $\wedge$ and $\vee$, with $\top$ and $\perp$ the empty ones; Heyting implication, written $\Rightarrow$, is such that

$(\_) \wedge \phi \dashv \phi \Rightarrow (\_)$; the functions $\mathcal{H}(f)$ are called *reindexing* and written f^*. If posets are used instead of preorders, $\mathcal{H}$ is a functor and the diagram above commutes strictly.

Note that, when infinite joins are available, and reindexing preserves them, an adjunction $f^* \dashv \forall_f$ is obtained by defining $\forall_f \phi \stackrel{\text{def}}{=} \bigvee \{\xi \mid f^*\xi \leq \phi\}$. Similarly, $\exists_f \dashv f^*$ is obtained from infinite meets.

Remark. To model a calculus including a formal treatment of proofs, a more general notion of hyperdoctrine would be required, where fibres are general categories. On the other hand, we do not require the fibres to be *posets* so as to have hyperdoctrines of classes of admissible monos, as in section 4.3.

Definition 4.2.2 *A first order hyperdoctrine $\mathcal{H}$ is said to* have equalities *if there exist left adjoints $\exists_{\Delta \times id}$ to reindexing along morphisms $\Delta \times id : A \times B \to (A \times A) \times B$ satisfying the Beck-Chevalley condition:*

$$\begin{tikzcd}
\mathcal{H}(A \times A) \arrow[r, "\pi^*"] & \mathcal{H}((A \times A) \times B) \\
\mathcal{H}A \arrow[u, "\exists_\Delta"] \arrow[r, "\pi^*"'] \arrow[ur, phantom, "\cong"] & \mathcal{H}(A \times B) \arrow[u, "\exists_{\Delta \times id}"']
\end{tikzcd}$$

Writing "=" for $\exists\top$, the above adjunction states that $x, y : X; x = y \vdash \phi(x, y)$ if and only if $x : X; true \vdash \phi(x, x)$.

Digression on Frobenious reciprocity. In [Pit91], where the logic features no connective $\supset$ and the fibres need not have exponentials, the functions $\exists_{\Delta \times id}$ are required to satisfy *Frobenious reciprocity*:

Definition 4.2.3 *Let $f : A \to B$ be a morphism in a category $\mathcal{C}$ and let $\exists_f \dashv f^*$ for a hyperdoctrine $\mathcal{H}$ on $\mathcal{C}$. We say that $\exists_f$ satisfies* Frobenious reciprocity *when, for all*

$\phi \in \mathcal{H}A$ and $\psi \in \mathcal{H}B$,

$$(\exists_f \phi) \wedge \psi \cong \exists_f(\phi \wedge f^* \psi).$$

The logical intuition behind Frobenious reciprocity is that $\exists\, y.\, (\phi(x,y) \wedge \psi(x))$ if and only if $(\exists\, y.\, \phi(x,y)) \wedge \psi(x)$. The following proposition shows that, in a first order hyperdoctrine, all functions $\exists_f$ satisfy this property.

Proposition 4.2.4 *Let f be an arbitrary morphism in a category $\mathcal{C}$, and let $\exists_f \dashv f^*$ for a first order hyperdoctrine over $\mathcal{C}$. The function $\exists_f$ satisfies Frobenious reciprocity.*

Proof. There is an obvious morphism $\exists_f(\phi \wedge f^*\psi) \to (\exists_f \phi) \wedge \psi$. The inverse to this is obtained by exploiting the fact that f^* preserves exponentials in the fibres:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\exists_f(\phi \wedge f^*\psi) \to \exists_f(\phi \wedge f^*\psi)}{\phi \wedge f^*\psi \to f^*\exists_f(\phi \wedge f^*\psi)}}{\phi \to (f^*\psi) \supset f^*\exists_f(\phi \wedge f^*\psi)}}{\phi \to f^*(\psi \supset \exists_f(\phi \wedge f^*\psi))}}{\exists_f \phi \to \psi \supset \exists_f(\phi \wedge f^*\psi)}}{(\exists_f \phi) \wedge \psi \to \exists_f(\phi \wedge f^*\psi)}$$

□

The definition of hyperdoctrine is often given (e.g. [Pav90]) by requiring left and right adjoints to reindexing along *arbitrary* morphisms. However such adjoints can be obtained from the adjoints along projections when the hyperdoctrine has equality:

Proposition 4.2.5 *For a first order hyperdoctrine with equality on a category $\mathcal{C}$ with products, there exist left and right adjoints $\exists_f \dashv f^* \vdash \forall_f$ for any morphism f in $\mathcal{C}$.*

Proof. Let $\Gamma, x : A \vdash M(x) : B$; logically, the proposition $\Gamma, y : B \vdash \exists_M(\phi(x))$ *prop* is defined as $\exists\, x.\, (y = M \wedge \phi)$ and $\Gamma, y : B \vdash \forall_M(\phi(x))$ *prop* as $\forall\, x.\, (y = M \supset \phi)$. The bijection defining the adjunction for $\forall_M$ is shown below while that for $\exists_M$ is derived similarly.

$$\cfrac{\cfrac{\cfrac{\Gamma, x : A; \psi(M) \vdash \phi(x)}{\Gamma, y, B, x : A; \psi(y) \wedge y = M \vdash \phi(x)}}{\Gamma, y, B, x : A; \psi(y) \vdash y = M \supset \phi(x)}}{\Gamma, y, B; \psi(y) \vdash \forall x. (y = M \supset \phi(x))}$$

$\square$

First order predicate calculus with equality is interpreted in a first order hyperdoctrine with equality $\mathcal{H}$ as follows. A formation judgement $\Gamma \vdash \phi$ *prop* is interpreted as an object of $\mathcal{H}[\![\Gamma]\!]$, which we write "$[\![\phi]\!]$" or even "$\phi$" when no confusion arises. A truth judgement $\Gamma; \Delta \vdash \phi$ is true when $\bigwedge \Delta \leq \phi$ in the poset $[\![\Gamma]\!]$. The interpretation $[\![\cdot]\!]$ of a formula is recursively defined as follows:

$$\begin{aligned}
[\![\mathit{false}]\!] &\stackrel{\text{def}}{=} \bot \\
[\![M = N]\!] &\stackrel{\text{def}}{=} = (M, N) \\
[\![\phi \supset \psi]\!] &\stackrel{\text{def}}{=} \phi \Rightarrow \psi \\
[\![\phi \wedge \psi]\!] &\stackrel{\text{def}}{=} \phi \wedge \psi \\
[\![\phi \vee \psi]\!] &\stackrel{\text{def}}{=} \phi \vee \psi \\
[\![\forall x : X.\, \phi]\!] &\stackrel{\text{def}}{=} \forall_X \phi \\
[\![\exists x : X.\, \phi]\!] &\stackrel{\text{def}}{=} \exists_X \phi
\end{aligned}$$

We now turn to the evaluation modalities.

Definition 4.2.6 (Pitts) *Let T be a strong monad on a category $\mathcal{C}$ with products; a T-*modality *on a (first order) hyperdoctrine $\mathcal{H}$ over $\mathcal{C}$ is specified by a family of order-preserving functions*

$$\square_{A,B} : \mathcal{H}(A \times B) \to \mathcal{H}(A \times TB),$$

one for each pair of objects A and B in $\mathcal{C}$, satisfying the following conditions:

- *(naturality) for all $f : A \to B$ in $\mathcal{C}$, $\square_{A,C} \circ (f \times id_C)^* = (f \times id_{TC})^* \circ \square_{B,C}$;*
- *(unit) for all A and B in $\mathcal{C}$, $(id_A \times val_B)^* \circ \square_{A,B} = id$;*

- *(lifting) for all* $f : A \times B \to TC$ *in* $\mathcal{C}$, $\Box_{A,B} \circ \langle \pi, f \rangle^* \circ \Box_{A,C} = \langle \pi, (\mathit{let}\ f) \rangle^* \circ \Box_{A,C}$.

T-modalities $\Box$ and $\Diamond$ are used in [Pit91] to interpret the evaluation modalities as follows:

$$[\![\Gamma, z : T\tau \vdash [x \Leftarrow z]\ \phi]\!] \stackrel{\text{def}}{=} \Box_{\Gamma,\tau}\phi$$
$$[\![\Gamma, z : T\tau \vdash \langle x \Leftarrow z \rangle\ \phi]\!] \stackrel{\text{def}}{=} \Diamond_{\Gamma,\tau}\phi$$

The general and special axioms of section 4.1 are sound with respect to the above interpretation, provided $\Box$ and $\Diamond$ have the following properties:

- $\Box$ preserves finite meets;
- $\Diamond$ preserves finite joins;
- $\Diamond\phi \wedge \Box\psi \leq \Diamond(\phi \wedge \psi)$;
- $\exists_{\Delta \times id}(\Diamond\phi) = \Diamond\exists_{\Delta \times id}\phi$;
- $\Box$ commutes with $\forall_X$;
- $\Diamond$ commutes with $\exists_X$.

In [Pit91], several examples of operators in concrete models are given. Among them, the modalities for computations with side effect are based on a hyperdoctrine $\mathcal{H}$ over the category of cpos such that $\mathcal{H}(A)$, that is the semilattice of propositions on A, contains subsets of $A \times S$, where S is the cpo of states.

The interpretation of Evaluation Logic just described requires the additional structure of a hyperdoctrine and T-modalities. However, a particular hyperdoctrine may be available on the category $\mathcal{C}$, for which an appropriate modality for interpreting $\Box$ (possibly not T-modal, but still validating the *general* axioms) can be defined automatically from the monad T. Moreover, if such a hyperdoctrine has higher order structure, an appropriate modal operator $\Diamond$ can be defined from $\Box$. Moggi calls such models "standard." We present standard interpretations in sections 4.4 through 4.7 after few mathematical preliminaries.

4.3 The logic of a class of admissible monos

In "standard" models, the structure required to interpret the logic is to be found in the same (base) category where types are interpreted. The idea is to define modal operators by making the categorical gear for interpreting computational types act upon that for interpreting the logic.

Logical structure can be found in topoi. They provide models for higher order logics, where predicates on a type A are interpreted in the category $Sub(A)$ of *subobjects* of A. Such an interpretation relies on several features of topoi: for example, for every A, $Sub(A)$ is a Heyting algebra; for any $f : A \to B$ there is a (logical) structure-preserving reindexing functor $f^* : Sub(B) \to Sub(A)$ with left and right adjoints; Ω is an internal Heyting algebra, and so on.

Various fragments of logical systems can be interpreted in arbitrary categories in which only restricted classes $\mathcal{M}$ of subobjects are considered. For example, *geometric logics* such as that described in [Vic89], with operators $\bigvee$ and $\wedge$, can be modelled by the algebra of "observable" properties of cpos, where elements of $\mathcal{M}$ are domains of partial continuous functions or Scott-opens.

First we show what kind of structure $\mathcal{M}$ in a category $\mathcal{C}$ allows interpretation of predicate calculus. Then, in order to find suitable interpretations for the evaluation modalities, we study the interaction between structures $\mathcal{M}$ and strong monads for interpreting computation. Here, M-functors and M-natural transformations are introduced. We use fibrations to show how such objects arise naturally as 1 and 2-cells in a 2-category of categories with structure.

Definition 4.3.1 *A* class of display maps $\mathcal{D}$ *in a category* $\mathcal{C}$ *is a collection of morphisms of* $\mathcal{C}$ *such that:*

- $\mathcal{D}$ *contains all the identities;*

- *$\mathcal{D}$ is closed under composition;*

- *if $d \in \mathcal{D}$ and f is an arbitrary morphism in $\mathcal{C}$ with the same target, a pullback of d along f exists and, for any such pullback, $f^{-1}d \in \mathcal{D}$.*

In [Tay92], Taylor adopts a more liberal approach in defining the notion of *display structure*, which is a class of morphisms closed under a *functorial choice* of pullbacks.

Display maps arise in categorical models of dependent types, where they act as projections of dependent contexts: for every type τ depending on a context Γ, a context $\Gamma, x : \tau$ is defined together with a projection $\Gamma, x : \tau \to \Gamma$ for modelling weakening. The "comprehension" of τ in $\Gamma, x : \tau$ can be abstractly modelled as follows. Let $\mathcal{C}^{\to}$ be the category whose objects are arrows of $\mathcal{C}$ and morphisms are commuting squares and let $cod : \mathcal{C}^{\to} \to \mathcal{C}$ be the codomain fibration.

Definition 4.3.2 (Jacobs) *A* comprehension category *is a functor $\mathcal{P} : \mathcal{E} \to \mathcal{B}^{\to}$ such that:*

- *$cod \circ \mathcal{P} : \mathcal{E} \to \mathcal{B}$ is a fibration;*

- *if f is cartesian in $\mathcal{E}$, then $\mathcal{P}f$ is a pullback in $\mathcal{B}$.*

The idea is that, if σ and τ are objects in the fibre over $[\![\Gamma]\!]$ interpreting types $\Gamma \vdash \sigma$ *type* and $\Gamma \vdash \tau$ *type*, the (weakened) judgement $\Gamma, x : \tau \vdash \sigma$ *type* is interpreted by reindexing σ along $\mathcal{P}\tau : [\![\Gamma, x : \tau]\!] \to [\![\Gamma]\!]$. In [Jac91], Jacobs defines display maps to be morphisms of the form $\mathcal{P}\tau$. This is slightly more general than definition 4.3.1, as we argue below.

Proposition 4.3.3 *Let $\mathcal{D}$ be a class of display maps; let $\mathcal{D}$ also indicate the category whose objects are elements of the class $\mathcal{D}$ and morphisms are commuting squares. The inclusion $\mathcal{D} \hookrightarrow \mathcal{C}^{\to}$ is a comprehension category.*

This proposition justifies the use of terms such as "fibres," "reindexing" etc. in the context of a category with a class of display maps.

It is easy to verify that $\mathcal{D} \hookrightarrow \mathcal{C}^{\to} \to \mathcal{C}$ is a fibration whose cartesian morphisms are pullback squares. Morphisms in the fibre $\mathcal{D}_A$ over A are commuting triangles, and the reindexing functor f^* is pulling back along f.

On the other hand, if an inclusion $\mathcal{E} \hookrightarrow \mathcal{C}^{\to}$ is a comprehension category, the objects of $\mathcal{E}$ need not yield a class of display maps. In fact, such a class must be closed under *any* choice of pullbacks, and hence all isomorphisms must be included in it. Hence, for example, the class of all identities is not a class of display maps although it forms a comprehension category $\mathcal{E} = \mathcal{C} \hookrightarrow \mathcal{C}^{\to}$.

Definition 4.3.4 *Let $\mathcal{D}$ be a class of display maps in $\mathcal{C}$; a fibration $p : \mathcal{E} \to \mathcal{C}$ is said to have $\mathcal{D}$-products ($\mathcal{D}$-sums) when:*

- *for all $d \in \mathcal{D}$, d^* has right (left) adjoint, written $d^* \dashv \forall_d$ ($\exists_d \dashv d^*$);*

- *the* Beck-Chevalley *condition holds, that is, for all pullbacks of elements $d \in \mathcal{D}$ as below, the canonical natural transformation $g^*\forall_d \dot{\to} \forall_b f^*$ ($\exists_b f^* \dot{\to} g^*\exists_d$) is an isomorphism.*

$$\begin{tikzcd} \cdot \arrow[r, "f"] \arrow[d, "b"'] \arrow[dr, phantom, very near start, "\lrcorner"] & \cdot \arrow[d, "d"] \\ \cdot \arrow[r, "g"'] & \cdot \end{tikzcd}$$

When $p : \mathcal{E} \to \mathcal{C}$ is understood, we say that $\mathcal{E}$ has $\mathcal{D}$-products (sums). We understand that a class $\mathcal{D}$ of display maps is the source of the fibration $\mathcal{D} \hookrightarrow \mathcal{C}^{\to} \to \mathcal{C}$.

Let $\mathcal{C}$ be a category with binary products. The cartesian projections of $\mathcal{C}$, that is morphisms that are part of a product diagram, form a class $\mathcal{D}_\pi$ of display maps. We say that a fibration has *indexed products* (*sums*) if it has $\mathcal{D}_\pi$-products (sums). Similarly, if

S is an object of $\mathcal{C}$, the class of morphisms $\cdot \to A$ such that $S \leftarrow \cdot \to A$ is a product diagram forms a class $\mathcal{D}_S$ of display maps and a fibration with $\mathcal{D}_S$-products (sums) is said to have S-indexed products (sums). If $\pi : A \times S \to A$ is in $\mathcal{D}_S$ and A is understood, we write $\forall_S$ for $\forall_\pi$.

Proposition 4.3.5 *Any class $\mathcal{D}$ of display maps has $\mathcal{D}$-sums, where $\exists_d b = d \circ b$.*

Proposition 4.3.6 *Let $\mathcal{D}$ be a class of display maps. All fibres $\mathcal{D}_A$ have terminal object 1_A and binary products $a \times_A b \stackrel{\text{def}}{=} \exists_b(b^* a)$ and reindexing preserves them.*

It is well known that, if $\mathcal{C}$ is a category with finite limits, $\mathcal{C}^{\to}$ has $\mathcal{C}^{\to}$-products if and only if all slice categories $\mathcal{C}/A$ are cartesian closed and reindexing preserves such structure (that is: $cod : \mathcal{C}^{\to} \to \mathcal{C}$ is a fibred CCC). This result can be generalized to arbitrary classes of display maps. Here we show one direction:

Proposition 4.3.7 *In a class of display maps $\mathcal{D}$ with $\mathcal{D}$-products, all fibres have exponentials and reindexing preserves them.*

Proof. Let $b : B \to A$ in $\mathcal{D}$; the exponential $(\_)^b$ in the fibre $\mathcal{D}_A$ is defined by the equation $d^b \stackrel{\text{def}}{=} \forall_b b^* d$, as shown by the following figure:

$$\begin{tikzcd}
\mathcal{M}_A \arrow[r, shift left, "b^*"] \arrow[rr, bend left=40, "(_) \times_A b"] \arrow[r, phantom, "\perp"] & \mathcal{M}_B \arrow[l, shift left, "\forall_b"] \arrow[r, shift left, "\exists_b"] \arrow[r, phantom, "\perp"] & \mathcal{M}_A \arrow[l, shift left, "b^*"] \arrow[ll, bend left=40, "(_)^b_A"]
\end{tikzcd}$$

In order to show preservation of exponentials, let $b, d \in \mathcal{D}_A$ and let f have codomain A in $\mathcal{C}$. Applying the Beck-Chevalley condition to the square

$$\begin{tikzcd} \cdot \arrow[d, "b"'] & \cdot \arrow[l, "g"'] \arrow[d, "f^*b"] \arrow[dl, phantom, "\lrcorner", very near start] \\ \cdot & \cdot \arrow[l, "f"] \end{tikzcd}$$

we obtain: $f^*(d^b) \stackrel{\text{def}}{=} f^*\forall_b b^* d \cong \forall_{f^*b} g^* b^* d \cong \forall_{f^*b} (f^*b)^* f^* d \stackrel{\text{def}}{=} (f^*d)^{f^*b}$. $\square$

Definition 4.3.8 (Rosolini) *A* class of admissible monos $\mathcal{M}$ *is a class of display maps whose elements are all monos.*

Elements of a class of admissible monos can be used for interpreting formulae in different fragments of predicate calculus, each fragment requiring different properties of $\mathcal{M}$. A trivial example of class of admissible monos in $\mathcal{C}$ is that of all isos in $\mathcal{C}$, modelling the predicates *true*. This class is reflective in $\mathcal{C}^{\to}$, with the unit of the reflection mapping $f : A \to B$ to id_B. If $\mathcal{C}$ has *strict* initial object 0, that is one such that any map into it is an isomorphism, *false* can be interpreted by adding to the above class all morphisms $0 \to A$. The collection of all monos in a category $\mathcal{C}$ with pullbacks forms a class of admissible monos $\mathcal{M}$. If $\mathcal{C}$ is a topos, there is a reflection $\mathcal{C}^{\to} \to \mathcal{M}$, whose unit η_f is the image of f.

With the structure described so far, we can interpret the *negative* fragment of predicate calculus with equality: let $\mathcal{M}$ be a class of admissible monos such that:

- $\mathcal{M}$ has $\mathcal{M}$-products;

- $\Delta \in \mathcal{M}$;

- $\mathcal{M}$ has indexed products.

From proposition 4.3.6, each fibre has a terminal object to interpret *true* and binary products to interpret $\wedge$. The first condition above allows the interpretation of implication, as shown by proposition 4.3.7. The second implies that $\Delta \times id \in \mathcal{M}$. The third condition yields $\forall_X$ for all types X (and, since $\mathcal{M}$ has equalities, also $\forall_f$ for arbitrary f).

The interpretation of disjunction requires additional closure properties of $\mathcal{M}$. One way to proceed is to assume sums in $\mathcal{C}$ and a factorization system (see section 4.7), so that $A \bigcup B \rightarrowtail C$ is the image of the mediating morphisms $A + B \to C$. The expected properties of $\bigcup$, such as idempotence, hold in this setting. Another way is to use the same construction as in *Sets*, that is pulling back A along B and pushing out the result. However this construction yields an idempotent opertator only when $\mathcal{C}$ is a pre-topos. A third possibility is to resort to higher order structure:

Definition 4.3.9 *Let $\mathcal{C}$ be a category with terminal object* 1. *A monomorphism $\top : 1 \to \Sigma$ is called a* classifier *if, for every f in $\mathcal{C}$ with codomain Σ, $f^{-1}\top$ exists and, moreover, $f^{-1}\top = g^{-1}\top$ implies $f = g$. A mono of the form $f^{-1}\top$ is called a Σ-object and f is called its* classifying map. *A classifier is called a* dominance *if the class of Σ-objects is closed under composition.*

Clearly, the Σ-objects of a dominance form a class of admissible monos. Alternatively, one can waive the condition that a classifier has domain 1 (then, for example, id_0 would be one) and define a dominance as a classifier whose Σ-objects form a class of admissible monos. Then, the domain of a dominance is forced to be a terminal object in $\mathcal{C}$ because the identities are classified.

There can be several (not isomorphic) dominances in a category. For example, in the category *Cpo* of cpos and continuous functions, id_1 is a dominance classifying the class of all isos, while $\top : 1 \to \{\bot \leq \top\}$ classifies the Scott-opens. The two-element poset $\{\bot \leq \top\}$ is called *Sierpiński space.*

If the class $\mathcal{M}$ of Σ-objects of a dominance satisfies some of the closure properties described earlier in this section, one can interpret first order predicate calculus in $\mathcal{M}$.

Proposition 4.3.10 *Let $\top$ be a dominance in a category $\mathcal{C}$ and let the class $\mathcal{M}$ of its Σ-objects have indexed and $\mathcal{M}$-products. The fibres of $\mathcal{M}$ are Heyting prealgebras, rein-*

dexing preserves the structure and has left and right adjoints satisfying the Beck-Chevalley condition.

The idea is to view Σ as an "object of propositions" and to define first order logical operators using implication and (higher order) universal quantification. In particular, let $\xi \stackrel{\text{def}}{=} \pi_\Gamma^* \top$ be the mono interpreting the generic predicate $\Gamma, \xi : \Sigma \vdash \xi$ *prop* in $\mathcal{M}_{\Gamma \times \Sigma}$ and let $\forall_\xi$ stand for $\forall_\Sigma$; from $\Rightarrow$ and $\forall_X$, which we get from the above structure as shown earlier, we define $\bot$, $\vee$ and $\exists_X$ in $\mathcal{M}_\Gamma$ as follows:

$$\begin{aligned} \bot &\stackrel{\text{def}}{=} \forall_\xi \xi \\ \phi \vee \psi &\stackrel{\text{def}}{=} \forall_\xi ((\phi \Rightarrow \xi) \Rightarrow (\psi \Rightarrow \xi) \Rightarrow \xi) \\ \exists_X \phi &\stackrel{\text{def}}{=} \forall_\xi (\forall_X (\phi \Rightarrow \xi) \Rightarrow \xi). \end{aligned}$$

Remark. The structure described in proposition 4.3.10 defines a *tripos* ([HJP80]) in $\mathcal{C}$. In fact, tripoi are more general structures, which only require a *generic predicate* ($\top$ for the fibration $\mathcal{M}$) where no uniqueness is required of the classifying maps. However, uniqueness makes it easy to find "horizontal" (that is internal) structure corresponding to that just described. For example, $\forall_B$ yields a map $\mathcal{C}(A \times B, \Sigma) \to \mathcal{C}(A, \Sigma)$ natural in A; hence, applying Yoneda to $\mathcal{C}(\_, \Sigma^B) \cong \mathcal{C}(\_ \times B, \Sigma) \dot{\to} \mathcal{C}(\_, \Sigma)$, we obtain an internal $\forall_B : \Sigma^B \to \Sigma$.

4.4 Endofunctors and classes of admissible monos

What is still missing of a standard model of EL is the modalities. In this section we describe a simple interpretation of $\Box$ and $\Diamond$ in a category with a class of admissible monos and validate the axioms for strong endofunctors, while in the next section we deal with the ones for monads.

Let $\mathcal{C}$ be a category with a class of admissible monos $\mathcal{M}$ as in proposition 4.3.10, and let $\langle T, t \rangle$ be a strong endofunctor mapping $\mathcal{M}$ to $\mathcal{M}$; the following semantics for the evaluation modalities, proposed in [Mogb], is called *standard* in the sense that it is completely determined by the strong endofunctor $\langle T, t\rangle$:

$$\frac{\Gamma, x : X \vdash \phi \ prop}{\Gamma, w : TX \vdash [x \Leftarrow w]\ \phi \ prop} \qquad \begin{array}{l}\phi \\ \Box_{\Gamma,X}\phi \stackrel{\text{def}}{=} t^*_{\Gamma,X} T\phi\end{array} \tag{4.3}$$

$$\frac{\Gamma, x : X \vdash \phi \ prop}{\Gamma, w : TX \vdash \langle x \Leftarrow w\rangle\ \phi \ prop} \qquad \begin{array}{l}\phi \\ \Diamond_{\Gamma,X}\phi \stackrel{\text{def}}{=} \forall_\xi(\Box_{\Gamma\times\Sigma,X}(\phi \Rightarrow \xi) \Rightarrow \xi)\end{array} \tag{4.4}$$

The interpretation of $\Diamond$ suggests that in a higher order Evaluation Logic, possibility can be expressed in terms of necessity:

$$\langle x \Leftarrow z\rangle\ \phi \ \stackrel{\text{def}}{=}\ \forall\xi : \Sigma.\,([x \Leftarrow z]\ (\phi \supset \xi) \supset \xi).$$

Note that (4.2) of section 4.1 is obtained by interpreting this formula in the category of sets.

Lemma 4.4.1 *The axiom $\Box\_\wedge$ is sound with respect to* (4.3) *in a category with a class of admissible monos $\mathcal{M}$ when T preserves meets. All other $\Box$-ed* general *axioms for strong endofunctors are sound with respect to interpretation in a category with the structure described in proposition 4.3.10. In particular, $\Box\_\Diamond$ is sound when $\Diamond$ is interpreted as in* (4.4).

Proof. All diagrams involved in the proof of proposition 4.4.1 are quite simple. To give an example, we show that $\Box\_\Diamond$ is a consequence of the axioms for $\forall$ and $\Box$:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\phi,\ \psi,\ (\phi\wedge\psi)\supset\xi\ \vdash\ \xi}{\phi,\ (\phi\wedge\psi)\supset\xi\ \vdash\ \psi\supset\xi}}{[x \Leftarrow w]\ \phi,\ [x \Leftarrow w]\ ((\phi\wedge\psi)\supset\xi)\ \vdash\ [x \Leftarrow w]\ (\psi\supset\xi)}}{[x \Leftarrow w]\ \phi,\ [x \Leftarrow w]\ ((\phi\wedge\psi)\supset\xi),\ [x \Leftarrow w]\ (\psi\supset\xi)\supset\xi\ \vdash\ \xi}}{[x \Leftarrow w]\ \phi,\ [x \Leftarrow w]\ (\psi\supset\xi)\supset\xi\ \vdash\ [x \Leftarrow w]\ ((\phi\wedge\psi)\supset\xi)\supset\xi}}{[x \Leftarrow w]\ \phi,\ \forall\xi.\,([x \Leftarrow w]\ (\psi\supset\xi)\supset\xi)\ \vdash\ \forall\xi.\,([x \Leftarrow w]\ ((\phi\wedge\psi)\supset\xi)\supset\xi)}}{[x \Leftarrow w]\ \phi,\ \langle x \Leftarrow w\rangle\,\psi\ \vdash\ \langle x \Leftarrow w\rangle\,(\phi\wedge\psi)}$$

$\square$

In order to validate the axioms for $\Diamond$, additional properties are required of $\square$. Such properties are expressed in the logic by $\square\_\forall$ and $\square\_\supset$.

Definition 4.4.2 (Moggi) *Let $\mathcal{C}$, $\langle T, t\rangle$, $\mathcal{M}$ and $\square$ be as above and let $\mathcal{M}$ have $\mathcal{D}$-products, for $\mathcal{D}$ a class of display maps. We say that $\square$* commutes with $\mathcal{D}$-products *if diagrams of the following form commute: let $d : A \to B$ in $\mathcal{D}$,*

$$\begin{tikzcd}
\mathcal{M}_{A\times C} \arrow[r, "\square_{A,C}"] \arrow[d, "\forall_{d\times id}"'] & \mathcal{M}_{A\times TC} \arrow[d, "\forall_{d\times id}"] \\
\mathcal{M}_{B\times C} \arrow[r, "\square_{B,C}"'] & \mathcal{M}_{B\times TC}
\end{tikzcd}$$

$\square\_\forall$, which is known in the context of modal logics as *Barcan formula*, is just the EL spelling of the diagram above, with $\mathcal{D} \stackrel{\text{def}}{=} \mathcal{D}_\pi$ the class of cartesian projections. It is easy to verify that $\square\_\supset$ is sound when $\mathcal{M}$ has $\mathcal{M}$-products and $\square$ commutes with them. Note that, in any fibred cartesian closed $\mathcal{M}$, for $m : B \to A \in \mathcal{M}$, m^* always has a right adjoint $\forall_m$. In fact, defining $\forall_m n \stackrel{\text{def}}{=} (m \circ n)^m$, we have:

$$\mathcal{M}_A(l, (m\circ n)^m) \cong \mathcal{M}_A(l \times_A m, m\circ n) = \mathcal{M}_A(m\circ m^* l, m\circ n) \cong \mathcal{M}_B(m^* l, n).$$

Theorem 4.4.3 *The general and special axioms for strong endofunctors are sound with respect to interpretation* (4.3,4.4) *in a category as in proposition 4.3.10 when $\square$ commutes with indexed and $\mathcal{M}$-products.*

To prove this theorem, we introduce *evaluation relations* ($a \Leftarrow_A w$), to be read "program w evaluates to value a," over $A \times TA$, and use them to obtain "left" and "right" rules for $\square$. This makes the proof theory of the logic more manageable. Following the same idea of (4.4), define:

$$a \Leftarrow_A w \quad \stackrel{\text{def}}{=} \quad \forall\, \phi : \Sigma^A.\ ([x \Leftarrow w]\ \phi(x) \supset \phi(a)) \tag{4.5}$$

as in [Mogb]. It is quite easy to verify that $[x \Leftarrow w]\ (x \Leftarrow w)$ holds in all models satisfying $\Box\_\forall$ and $\Box\_\supset$ [Mogb, Theorem 2.13]. We call the following derived rules, $\Box\_\mathit{right}$ and $\Box\_\mathit{left}$ respectively:

$$\dfrac{\dfrac{\Delta, x \Leftarrow w \vdash \phi(x)}{\Delta, [x \Leftarrow w]\ (x \Leftarrow w) \vdash [x \Leftarrow w]\ \phi(x)}\ x \notin FV(\Delta)}{\Delta \vdash [x \Leftarrow w]\ \phi(x)}$$

$$\dfrac{\Delta, \phi(M) \vdash \psi \qquad \dfrac{\dfrac{\Delta \vdash M \Leftarrow w}{\Delta \vdash \forall P.\,([x \Leftarrow w]\ P(x) \supset P(M))}}{\Delta \vdash [x \Leftarrow w]\ \phi(x) \supset \phi(M)}}{\Delta, [x \Leftarrow w]\ \phi(x) \vdash \psi}$$

Note that, using $\Box\_\mathit{left}$, one can derive the converse of $\Box\_\mathit{right}$. Theorem 4.4.3 follows from Lemma 4.4.1 by showing that all general axioms for $\Diamond$ and special axioms for strong endofunctors are derivable from the axioms for $\Box$, $\Box\_\mathit{right}$ and $\Box\_\mathit{left}$. As an example, we show the derivation of $\Diamond\_\vee$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\phi, \phi \supset \xi \vdash \xi \qquad x \Leftarrow w \vdash x \Leftarrow w}{x \Leftarrow w, \phi, [x \Leftarrow w]\ (\phi \supset \xi) \vdash \xi}\ \Box\_\mathit{left}}{x \Leftarrow w, \phi \vdash \forall \xi.\,([x \Leftarrow w]\ (\phi \supset \xi) \supset \xi)}}{x \Leftarrow w, \phi \vdash \langle x \Leftarrow w\rangle\ \phi \vee \langle x \Leftarrow w\rangle\ \psi} \qquad \dfrac{\ldots\text{same as before}}{x \Leftarrow w, \psi \vdash \langle x \Leftarrow w\rangle\ \phi \vee \langle x \Leftarrow w\rangle\ \psi}}{x \Leftarrow w, \phi \vee \psi \vdash \langle x \Leftarrow w\rangle\ \phi \vee \langle x \Leftarrow w\rangle\ \psi}}{x \Leftarrow w \vdash (\phi \vee \psi) \supset (\langle x \Leftarrow w\rangle\ \phi \vee \langle x \Leftarrow w\rangle\ \psi)}}{\vdash [x \Leftarrow w]\ ((\phi \vee \psi) \supset (\langle x \Leftarrow w\rangle\ \phi \vee \langle x \Leftarrow w\rangle\ \psi))}\ \Box\_\mathit{right}}{\forall \xi.\,([x \Leftarrow w]\ ((\phi \vee \psi) \supset \xi) \supset \xi) \vdash \langle x \Leftarrow w\rangle\ \phi \vee \langle x \Leftarrow w\rangle\ \psi}}{\langle x \Leftarrow w\rangle\ (\phi \vee \psi) \vdash \langle x \Leftarrow w\rangle\ \phi \vee \langle x \Leftarrow w\rangle\ \psi}$$

Note that the condition of Lemma 4.4.1 that T preserves meets in the $\mathcal{M}$-fibres disappears in the above theorem since $\Box\_\wedge$ is a consequence of $\Box\_\supset$ and $\Box\_\forall$ as shown below:

$$\cfrac{\cfrac{[x \Leftarrow w]\ \phi \vdash [x \Leftarrow w]\ \phi}{[x \Leftarrow w]\ \phi, x \Leftarrow w \vdash \phi} \qquad \cfrac{[x \Leftarrow w]\ \psi \vdash [x \Leftarrow w]\ \psi}{[x \Leftarrow w]\ \psi, x \Leftarrow w \vdash \psi}}{\cfrac{[x \Leftarrow w]\ \phi, [x \Leftarrow w]\ \psi, x \Leftarrow w \vdash \phi \wedge \psi}{[x \Leftarrow w]\ \phi, [x \Leftarrow w]\ \psi, \vdash [x \Leftarrow w]\ (\phi \wedge \psi)}}$$

4.5 Monads and classes of admissible monos

Proposition 4.5.1 *The* general *axioms for strong monads are sound with respect to interpretation in a category with the structure described in proposition 4.3.10.*

Next, we study the interaction between a class of admissible monos $\mathcal{M}$ and a strong monad T in order to find suitable conditions for validating the *special* axioms for strong monads.

In models satisfying $\Box\_T^*$, substitution in modal formulae corresponds to pullbacks. A minimal condition for the latter is that T preserves pullbacks of elements of $\mathcal{M}$ along morphisms of the form $f \times id$. Moggi calls such functors *strongly mono preserving* and notices that they are not closed under composition. This may be unfortunate, since one cannot verify this property for complex functors by verifying it holds for their components, as we do in the next chapter. Definition 4.5.2 below imposes a stronger requirement on functors T that is preserved under composition and guarantees a good behaviour of the modalities under substitution.

Comprehension categories are the objects of a (two-)category *Comp* whose morphisms $\mathcal{P} \to \mathcal{Q}$ are fibration morphisms $(F, G) : \mathit{cod} \circ \mathcal{P} \to \mathit{cod} \circ \mathcal{Q}$ making the following diagram commute:

$$\begin{tikzcd}
\mathcal{E} \arrow[r, "G"] \arrow[d, "\mathcal{P}"'] & \mathcal{H} \arrow[d, "\mathcal{Q}"] \\
\mathcal{C}^{\to} \arrow[r, "F^{\to}"] \arrow[d, "cod"'] & \mathcal{B}^{\to} \arrow[d, "cod"] \\
\mathcal{C} \arrow[r, "F"] & \mathcal{B}
\end{tikzcd}$$

From this diagram it is clear that, when $\mathcal{E}$ and $\mathcal{H}$ are classes of admissible monos (or, for that matter, classes of display maps), and $\mathcal{P}$ and $\mathcal{Q}$ the inclusions, G is completely determined by F. In this case, we write $F^{\rightarrow}\colon \mathcal{E} \to \mathcal{H}$ the restriction of the homonymous functor $\mathcal{C}^{\rightarrow} \to \mathcal{B}^{\rightarrow}$. We call Cat_M the full subcategory of *Comp* whose objects are categories with a class of admissible monos.

Definition 4.5.2 *Morphisms in* Cat_M *are called* M-functors.

Unfolding this definition, an M-functor $(\mathcal{C}, \mathcal{M}) \to (\mathcal{B}, \mathcal{N})$ is a functor $F : \mathcal{C} \to \mathcal{B}$ such that $m \in \mathcal{M}$ implies $Fm \in \mathcal{N}$ and F preserves pullbacks of elements of $\mathcal{M}$ (the cartesian morphisms of $cod \circ \mathcal{P}$). Obviously, M-functors are strongly mono preseving, thus making $\Box$ commute with substitution.

Remark. An M-functor $F : \mathcal{C} \to \mathcal{D}$ is a functor that lifts to the categories of partial maps, that is to a functor $pF : p\mathcal{C} \to p\mathcal{D}$ (see 2.5).

$\Box$

When T is an M-functor, $\Box\_T^*$ is sound. In fact, note that, for $N : \tau \to \sigma$, the sequent $w : T\tau \vdash let\ x \Leftarrow w\ in\ val(N) : T\sigma$ is interpreted as TN and hence $\Box\_T^*$ corresponds to the implication $(TN)^*T\phi \supset T(N^*\phi)$, while the opposite holds because $(TN)^*T\phi$ is a pullback.

Digression on *enriched* M-functors. Let $\mathcal{M}$ be a class of admissible monos in a category $\mathcal{C}$ enriched over itself and let $\mathcal{C}$ have pullbacks of morphisms of the form $(m \circ \_) : [X, dom(m)] \to [X, cod(m)]$, for $m \in \mathcal{M}$. Then, $\mathcal{M}$ is a $\mathcal{C}$-category. In particular, for $m : A \to B$ and $n : C \to D$ in $\mathcal{M}$, the object $[m, n]$ is the vertex of the vertex of the (inner) pullback square in the diagram below.

$$\begin{tikzcd}
{[m,n]} \arrow[r, "\pi_0"] \arrow[d, "\pi_1"'] \arrow[dr, phantom, "\lrcorner", very near start] & {[A,C]} \arrow[r, "t"] \arrow[d, "n\circ_"] & {[TA,TC]} \arrow[dd, "Tn\circ_"] \\
{[B,D]} \arrow[r, "_\circ m"'] \arrow[d, "t"'] & {[A,D]} \arrow[dr, "t"] & \\
{[TB,TD]} \arrow[rr, "_\circ Tm"'] & & {[TA,TD]}
\end{tikzcd}$$

The morphism mediating the pair $id_A : [A,A] \leftarrow 1 \rightarrow [B,B] : id_B$ is the identity map $id_m : 1 \rightarrow [m,m]$ and composition is similarly defined. It is easy to verify that $cod : \mathcal{M} \rightarrow \mathcal{C}$ is a $\mathcal{C}$-functor with the projection $\pi_1 : [m,n] \rightarrow [B,D]$ as strength. Moreover, a strong endofunctor $\langle T, t\rangle : \mathcal{C} \rightarrow \mathcal{C}$ lifts to a strong endofunctor $\langle T^{\rightarrow}, t^{\rightarrow}\rangle$ on $\mathcal{M}$ with $t^{\rightarrow}$: $[m,n] \rightarrow [Tm, Tn]$ the morphism mediating $[TA,TC] \stackrel{t\circ\pi_0}{\leftarrow} [m,n] \stackrel{t\circ\pi_1}{\rightarrow} [TB,TD]$ in the diagram above.

A *strong M-endofunctor* is a strong endofunctor T such that the morphism of $\mathcal{C}$-endofunctors $T \circ cod = cod \circ T^{\rightarrow}$: $\mathcal{M} \rightarrow \mathcal{C}$ is a morphism of fibrations.

□

In order to validate the val^* and let^* axioms, we introduce the two-categorical notion of *M-cartesian* natural transformation. Cat_M is a two-category whose 2-cells $F \Rightarrow G$ are natural transformations $\sigma : F \dot{\rightarrow} G$ making the following diagram commute: writing $\sigma^{\rightarrow}_m = (\sigma_A, \sigma_B)$ for $m : A \rightarrow B \in \mathcal{M}$,

$$\begin{tikzcd}
\mathcal{M} \arrow[r, bend left, "F^{\rightarrow}"] \arrow[r, bend right, "G^{\rightarrow}"'] \arrow[r, phantom, "\Downarrow \sigma^{\rightarrow}"] \arrow[d, "cod"'] & \mathcal{N} \arrow[d, "cod"] \\
\mathcal{C} \arrow[r, bend left, "F"] \arrow[r, bend right, "G"'] \arrow[r, phantom, "\Downarrow \sigma"] & \mathcal{B}
\end{tikzcd}$$

Definition 4.5.3 *Given M-functors F and G, a natural transformation $\sigma : F \Rightarrow G$ is called* M-cartesian *when the components of $\sigma^{\rightarrow}$ are catresian.*

Unfolding this definition, $\sigma : F \dot{\to} G$ is called M-cartesian when the naturality diagrams are pullbacks:

$$\begin{tikzcd} {} \arrow[r, "\sigma"] \arrow[d, "Fm"'] \arrow[dr, phantom, very near start, "\lrcorner"] & {} \arrow[d, "Gm"] \\ {} \arrow[r, "\sigma"'] & {} \end{tikzcd}$$

Examples of M-cartesian natural transformations are the unit of the lifting monad described in 2.5 and that of the *state readers* monad described in 5.1. A nonexample is the unit of $TX = 1$.

Proposition 4.5.4 *M-cartesian natural transformations are closed under horizontal and vertical composition.*

In particular, if $\langle T, t^T \rangle$ and $\langle S, t^S \rangle$ are strong M-endofunctors with t^T and t^S M-cartesian, then t^{TS} is also M-cartesian, just by straight composition of pullbacks. The condition that the tensorial strength t of an M-functor T is M-cartesian can be expressed in Evaluation Logic with the following equivalence:

$$\Box\_t^* \qquad \frac{\Gamma, x : \tau \vdash \phi \ prop \quad \Gamma, y : \sigma \vdash \psi \ prop}{\Gamma, x : \tau, z : T\sigma; [y \Leftarrow z] \ (\phi \wedge \psi) \dashv\vdash \phi \wedge [y \Leftarrow z] \ \psi}$$

While the left-to-right entailment holds in any model of the general axioms, $\Box\_t^*$ does not hold, for example, when computations may fail to terminate, since any nonterminating z would make the formula to the left vacuously true. An immediate consequence of $\Box\_t^*$ is:

$$\Gamma, z : T\tau; [x \Leftarrow z] \ \phi \vdash \phi \qquad x \notin FV(\phi).$$

When η is M-cartesian, $\Box\_val^*$ is sound. Moreover, $\Diamond\_val^*$ can be derived from $\Box\_val^*$ even in models where $\Box\_\supset$ and $\Box\_\forall$ do not hold. In section 4.1 we remarked that the mono requirement follows from $\Box\_\mathit{val}^*$. Now we show that, in models satisfying this rule, the components of η are *strong* monos.

We write $f \downarrow g$, for two arbitrary morphisms f and g in a category $\mathcal{C}$, when they satisfy the following *diagonal fill-in* property: for all u and v such that $v \circ f = g \circ u$, there exists a unique h such that

$$\begin{tikzcd} {} \arrow[r, "f"] \arrow[d, "u"'] & {} \arrow[d, "v"] \arrow[dl, dashed, "h"'] \\ {} \arrow[r, "g"'] & {} \end{tikzcd}$$

The *strong* monos of $\mathcal{C}$ are the monos m such that $e \downarrow m$ for all *epis* e in $\mathcal{C}$. Any equalizer is a strong mono. When η is M-cartesian, its components are equalizers, as shown by the following diagram:

$$\begin{tikzcd} {} \arrow[r, "\eta"] \arrow[d, "\eta"'] \arrow[dr, phantom, "\lrcorner", very near start] & {} \arrow[d, "T\eta"] \\ {} \arrow[r, "\eta T"'] & {} \end{tikzcd}$$

When T is an M-functor and μ is M-cartesian, $\Box\_\mathit{let}^*$ is sound. From $\Box\_\mathit{let}^*$, $\Box\_\supset$ and $\Box\_\forall$, one can derive $\Diamond\_\mathit{let}^*$. The property that μ is M-cartesian is expressed in [Mogb] by the axiom $\Box\_\mu^*$, which is an instance of $\Box\_\mathit{let}^*$. As we shall see in section 5.3, such a property gives to $\Box$ a stronger modal flavour: in the formula $[x \Leftarrow M]\ \phi$, whatever evaluation goes on in ϕ, happens "after" the evaluation of M.

4.6 Monads in the ambient category

Let $\mathcal{C}$ be a full reflective subcategory of $\mathcal{E}$, let $R \dashv I$ be the reflection, with unit ρ and counit ϵ, and let $T = \langle T, \eta, \mu \rangle$ be a monad on $\mathcal{C}$. The adjunction $F^T R \dashv IU^T$ shown in the diagram below defines a monad $\tilde{T}$ on $\mathcal{E}$.

$$
\begin{tikzcd}
\mathcal{E} \arrow[r, shift left, "R"] \arrow[dr, shift left, "F^{\tilde{T}}"] & \mathcal{C} \arrow[l, hook', shift left, "I"] \arrow[r, shift left, "F^T"] & \mathcal{C}^T \arrow[l, shift left, "U^T"] \arrow[dl, dashed, "K"] \\
& \mathcal{E}^{\tilde{T}} \arrow[ul, shift left, "U^{\tilde{T}}"] &
\end{tikzcd}
$$

Note that, since I is full, ϵ is invertible, and $\epsilon_X^{-1} = \rho_X$. The unit of $\tilde{T}$ is $\tilde{\eta} = \eta R \circ \rho$, the multiplication is $\tilde{\mu} = \mu R \circ T(\rho T R)^{-1}$, while the counit of the adjunction $F^T R \dashv IU^T$ is $\tilde{\epsilon}_{\alpha : TX \to X} = \alpha \circ T \rho_X^{-1} : \mu_{RX} \to \alpha$. The functor $K : \mathcal{C}^T \to \mathcal{E}^{\tilde{T}}$ of comparison with algebras is: $K(\alpha : TX \to X) = IU^T \tilde{\epsilon}_\alpha = \alpha \circ T \rho_X^{-1}$. Below we prove that such a functor is half of an adjoint equivalence of categories. This result shows that the extension $\tilde{T}$ of T to the ambient category, which is needed for a standard semantics of Evaluation Logic, is in a sense "minimal."

Lemma 4.6.1 *Any full reflective subcategory $\mathcal{C}$ of a category $\mathcal{E}$ is closed under retractions. In particular, if A is a retract of some object in $\mathcal{C}$, then $\rho_A : A \cong RA$.*

Proof. Let $A \xrightarrow{a} B \xrightarrow{b} A = id_A$ with B in $\mathcal{C}$ and let $k = b \circ \rho_B^{-1} \circ Ra : RA \to A$. Then:

$$
\begin{aligned}
&k \circ \rho_A = b \circ \rho_B^{-1} \circ Ra \circ \rho_A = b \circ \rho_B^{-1} \circ \rho_B \circ a = id_A \text{ and} \\
&\rho_A \circ k = \rho_A \circ b \circ \rho_B^{-1} \circ Ra = Rb \circ Ra = id_{RA}.
\end{aligned}
$$

□

Corollary 4.6.2 *Let $H : \mathcal{J} \to \mathcal{C}$ and let $\nu : l \dot{\to} IH$ be limiting in $\mathcal{E}$; then ρ_l is an iso and $\nu \circ \rho_l^{-1} : IRl \dot{\to} IH$ is also a limit in $\mathcal{E}$ with vertex in $\mathcal{C}$.*

Proof. Let $h : IR\,l \to l$ be mediating for the cone $(\rho IH)^{-1} \circ IR\nu : IR\,l \dot{\to} IH$ as in the diagram.

$$\begin{tikzcd} l \arrow[r, "\nu"] \arrow[d, shift right, "\rho_l"'] & IH \arrow[d, shift right, "\rho IH"'] \\ IR\,l \arrow[r, "IR\,\nu"] \arrow[u, shift right, "h"'] & IRIH \arrow[u, shift right, "(\rho IH)^{-1}"'] \end{tikzcd}$$

From $\nu \circ h \circ \rho_l = (\rho IH)^{-1} \circ IR\,\nu \circ \rho_l = \nu$ it follows that $h \circ \rho_l = id_l$. Then Lemma 4.6.1 applies and $IR\,l \in \mathcal{C}$ is a limit for IH. $\square$

Note that lemma 4.6.1 is also a direct consequence of its corollary since sections are limits. In particular, if $A \xrightarrow{a} B \xrightarrow{b} A = id_A$, a is a split equalizer of $a \circ b$ and id_B.

Theorem 4.6.3 *Let T be a monad in a category $\mathcal{C}$ which is fully reflective in a category $\mathcal{E}$ and let $\tilde{T}$ be the extension of T to $\mathcal{E}$ obtained as above. The category of $\tilde{T}$-algebras is equivalent to the category of T-algebras and the comparison functor K is the equivalence.*

Proof. Let $H : \mathcal{E}^{\tilde{T}} \to \mathcal{C}^T$ be the functor mapping $\alpha : TRA \to A$ to $\rho_A \circ \alpha$ and $h : \alpha \to \beta$ to $R(h)$. Note that $H(h) = (\rho \circ h)^{\dagger}$, where $(\_)^{\dagger}$ is the natural isomorphism given by $R \dashv I$ and inverse to $(\_ \circ \rho)$. We prove the theorem by showing that H is full (i) and faithful (ii) and that every T-algebra α is isomorphic to the H image of the $\tilde{T}$-algebra $K\alpha$ (iii).

(i) Let $\alpha : TRA \to A$ and $\beta : TRB \to B$ be $\tilde{T}$-algebras, and let $f : H(\alpha) \to H(\beta)$ be a morphism of T-algebras. Calling $h : A \to B$ the morphism $\beta \circ T(f \circ \rho_A) \circ \eta_A$, a simple calculation shows that $\rho_B \circ h = f \circ \rho_A$ and hence $H(h) = (\rho_B \circ h)^{\dagger} = (f \circ \rho_A)^{\dagger} = f$.

(ii) Let $h, k : A \to B$ be a morphisms between $\tilde{T}$-algebras α and β as above and let $H(h) = H(k)$. Since β is a retraction, by lemma 4.6.1 ρ_B is an iso. Then, $\rho_B \circ h = H(h) \circ \rho_A = H(k) \circ \rho_A = \rho_B \circ k$ implies $h = k$.

(iii) Let $\alpha : TA \to A$; the unit $\rho_A : A \to RA$, which is an iso because A is in $\mathcal{C}$, lifts to an isomorphism $\alpha \to HK\alpha$ of T-algebras. Moreover, it is routine to verify that H is left (and right) adjoint to K, where, for $\alpha : TRA \to A$ and $\beta : TB \to B$, the transpose of $f : \alpha \to K\beta$ is $f^\dagger : RA \to B$ viewed as an algebra morphism $H\alpha \to \beta$. $\square$

Next theorem establishes a necessary and sufficient condition for the reflection R of $\mathcal{E}$ in $\mathcal{C}$ (not necessarily full), and hence for the monad $\tilde{T}$ obtained as described above, to have tensorial strength.

Theorem 4.6.4 *Let $\mathcal{C}$ be a reflective subcategory of a category $\mathcal{E}$ with products, let R be the reflection, with unit ρ, and let $T = \langle T, \eta, \mu \rangle$ be a monad on $\mathcal{C}$. The monad $\tilde{T}$ defined as above has a tensorial strength if and only if R preserves products.*

Proof. (If) This is an immediate consequence of A. Kock's result in [Koc72], which establishes a one to one correspondence between strengths and monoidal structures for monads over symmetric monoidal closed categories. In particular, if the the natural transformation $\xi_{A,B} : RA \times RB \to R(A \times B)$ is the monoidal structure of R, then $r_{A,B} \stackrel{\text{def}}{=} \xi_{A,B} \circ (\rho_A \times 1)$ is a tensorial strength. However, monoidal functors are not required to *preserve* the monoidal structure, i.e. ξ need not be an iso, so we have to work in the *only if* direction.

(Only if) We show that $\xi_{A,B} \stackrel{\text{def}}{=} \mu_{A\times B} \circ Rr_{B,A} \circ r_{RA,B} : RA \times RB \to R(A \times B)$ is inverse to the transpose $(\rho_A \times \rho_B)^\dagger$ of $\rho_A \times \rho_B$ (since I preserves limits, we won't distinguish between products in $\mathcal{C}$ and in $\mathcal{E}$). To show $\xi_{A,B}$ is a right inverse we prove that $\xi_{A,B} \circ (\rho_A \times \rho_B)^\dagger \circ \rho_{A\times B} = \rho_{A\times B}$. In fact,

$$
\begin{aligned}
&\xi_{A,B} \circ (\rho_A \times \rho_B)^\dagger \circ \rho_{A\times B} = \\
&\xi_{A,B} \circ (\rho_A \times \rho_B) = \\
&\mu_{A\times B} \circ Rr_{B,A} \circ r_{RA,B} \circ (1 \times \rho_B) \circ (\rho_A \times 1) = \\
&\mu_{A\times B} \circ Rr_{B,A} \circ \rho_{RA\times B} \circ (\rho_A \times 1) = \\
&\mu_{A\times B} \circ \rho_{R(A\times B)} \circ r_{A,B} \circ (\rho_A \times 1) = \\
&r_{A,B} \circ (\rho_A \times 1) = \rho_{A\times B}.
\end{aligned}
$$

By a similar chase, $\xi_{A,B}$ is also shown to be a left inverse. Here the fact that $(\rho_A \times \rho_B)^\dagger = \langle R\pi, R\pi \rangle$ is also used. $\square$

Note that the property of preserving products is satisfied by interesting known reflections of the effective topos $\mathcal{E}\!f\!f$. For example, the category of strictly effective objects in $\mathcal{E}\!f\!f$ is an exponential ideal (that is X^Y is strictly effective for X strictly effective and arbitrary Y) and this is necessary and sufficient condition for a full reflection to preserve products.

Proposition 4.6.5 *Let T, $\tilde{T}$, R and ρ be as above. If T has tensorial strength t and R preserves products, $\tilde{T}$ has tensorial strength $\tilde{t}_{A,B} \stackrel{def}{=} T\xi_{A,B} \circ t_{RA,RB} \circ (\rho_A \times 1)$, where ξ is the tensorial structure of the reflection.*

The extension of T to the ambient category $\mathcal{E}$ presented above has the questionable feature of mapping everything inside $\mathcal{C}$. For example, if T is the identity on $\mathcal{C}$, $\tilde{T}$ is the reflection R, while the identity on $\mathcal{E}$ would be a more natural choice. Moreover, when the interpretation described in 4.4 is adopted, $\tilde{T}$ may not have nice interaction with $\mathcal{M}$ even if T does. In section 5.2 we adopt a different solution to extend the monad $(\_ \times S)^S_\perp$ for side effects.

4.7 Standard semantics

There are significant cases in denotational semantics where the interpretation (4.3) of $\Box$ described in section 4.4 goes wrong. For example, let T be Plotkin's powerdomain functor $TX = F(X_\perp)$, where FA is the free semilattice over the cpo A. In [Moga], Moggi noticed that this functor fails to map monos to monos and proposed a new semantical framework for interpreting EL, where extra structure is required in the category hosting the logic while less is expected from the monad T over the category hosting programs. We only give a sketchy account of this setting (see [Moga] for details) since the simpler interpretation described above is enough to present our next application. It is assumed that:

- a stable factorization system $(E, \mathcal{M})$ is available in a category $\mathcal{E}$ with finite limits;
- $cod : \mathcal{E}^{\rightarrow} \rightarrow \mathcal{E}$ has a full reflective subfibration $q : \mathcal{C} \rightarrow \mathcal{E}$;
- $\begin{tikzcd}[column sep=small] \mathcal{C} \arrow[rr, "T"] \arrow[dr, "q"'] & & \mathcal{C} \arrow[dl, "q"] \\ & \mathcal{E} & \end{tikzcd}$ is a monad fibred over $\mathcal{E}$.

Via the reflection $\mathcal{E}^{\rightarrow} \rightarrow \mathcal{C}$, T is extended to a monad $\tilde{T} : \mathcal{E}^{\rightarrow} \rightarrow \mathcal{E}^{\rightarrow}$ fibred over $\mathcal{E}$ as done in section 4.6, thus allowing interpretation of computational types in $\mathcal{E}$. However, the above setting is more general, since *dependent types* are also supported. While describing our notation, we recall the standard interpretation of dependent types in a fibration $\mathcal{E}^{\rightarrow} \rightarrow \mathcal{E}$.

Contexts are interpreted as objects of $\mathcal{E}$ and we write them as sequences $(A, B \ldots)$ of types. Types $\Gamma \vdash A\ type$ are morphisms $(\Gamma, A) \rightarrow \Gamma$ of $\mathcal{E}$. Terms $\Gamma \vdash M : A$ are morphisms $M : \Gamma \rightarrow (\Gamma, A)$ such that $A \circ M = id$. Propositions $\Gamma \vdash \phi\ prop$ are monos $\rightarrowtail \Gamma$ in $\mathcal{M}$. For the interpretation of *val* and *let*, note first that, from a monad T fibred over $\mathcal{E}$, one obtains a *strong* monad T_I in each fibre $\mathcal{E}/I$, and hence a strong monad T_1 in $\mathcal{E} \cong \mathcal{E}/1$. While the interpretation of *val* is quite straightforward, some attention must be paid to the binding discipline of *let*:

$$\frac{\Gamma \vdash \sigma, \tau\ types \quad \Gamma \vdash N : T\sigma \quad \Gamma, x : \sigma \vdash M : T\tau}{\Gamma \vdash let\ x \Leftarrow N\ in\ M : T\tau}$$

The interpretation of *let* is $[\![let\ x \Leftarrow N\ in\ M]\!] \stackrel{\text{def}}{=} \mu \circ T_\Gamma M' \circ N$, as in the diagram below:

$$\begin{tikzcd}
(\Gamma, \sigma, T\tau) \arrow[r] \arrow[dr, "T\tau"] \arrow[dr, phantom, very near start, "\lrcorner"] & (\Gamma, T\tau) \arrow[dr, "T\tau"] & (\Gamma, T^2\tau) \arrow[l, "\mu"'] & (\Gamma, T\sigma) \arrow[l, "TM'"'] \arrow[dl, "T\sigma"'] \\
& (\Gamma, \sigma) \arrow[r, "\sigma"'] \arrow[u, "M'"'] \arrow[ul, bend left, "M"] & \Gamma \arrow[ur, bend right, "N"'] &
\end{tikzcd}$$

Remark. Note that, while M may "depend" on x, not so its type. This situation can be generalized by assuming an evaluation predicate $\sigma \xleftarrow{v} (\Leftarrow_\sigma) \xrightarrow{p} T\sigma$ as in section 4.8. Then, from a type $\Gamma, x : \sigma \vdash \tau$ *type*, a new type $\Gamma, w : T\sigma \vdash \Pi x \Leftarrow w.\, \tau$ *type* is obtained by pulling back τ along v and then applying $\forall_p$.

$\Box$

The factorization system comes in for interpreting $\Box$. Different factorization systems may be available in a category $\mathcal{C}$ and each choice may allow interpretation of different logical features. For example, the regular monos in the category of ω-sets have a classifier $1 \to \nabla 2$, which allows interpretation of higher order quantification. Below we provide the relevant definitions and sketch the interpretation of $\Box$.

Definition 4.7.1 *A* factorization system *in a category $\mathcal{C}$ is a pair (E, M) of classes of morphisms of $\mathcal{C}$ such that: (**A**) both contain all isos and are closed under composition and (**B**) all morphisms in $\mathcal{C}$ are uniquely (up to isomorphism) of the form $m \circ e$ for $m \in M$ and $e \in E$. A factorization system is called* stable *if pullbacks of $m \circ e$, for $m \in M$ and $e \in E$, factorize into $m' \circ e'$ as below:*

$$
\begin{tikzcd}
\cdot \arrow[r, "e'"] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \cdot \arrow[r, "m'"] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \cdot \arrow[d] \\
\cdot \arrow[r, "e"'] & \cdot \arrow[r, "m"'] & \cdot
\end{tikzcd}
$$

On factorization systems see [FK72] and the more recent [CJKP94]. As in [FK72], we write $H^{\uparrow} \stackrel{\text{def}}{=} \{e \text{ in } \mathcal{C} \mid \forall\, h \in H.\, e \downarrow h\}$ and $H^{\downarrow} \stackrel{\text{def}}{=} \{m \text{ in } \mathcal{C} \mid \forall\, h \in H.\, h \downarrow m\}$, for H a class of morphisms in $\mathcal{C}$ and $f \downarrow g$ as in section 4.5. Proposition 4.7.4 gives an alternative view of factorization systems.

Definition 4.7.2 *A* prefactorization system *in a category $\mathcal{C}$ is a pair (E, M) of classes of morphisms of $\mathcal{C}$ such that $E = M^{\uparrow}$ and $M = E^{\downarrow}$.*

Lemma 4.7.3 *A factorization system is a prefactorization system.*

Proof. First we show that, for all $e \in E$ and $m \in M$, $e \downarrow m$. Let $v \circ e = m \circ u$ and let $u = A \xrightarrow{e'} U \xrightarrow{m'} C$ and $v = B \xrightarrow{e''} V \xrightarrow{m''} D$ be their factorizations. Then $m \circ m' \circ e'$ and $m'' \circ e'' \circ e$ are both factorizations of the same morphism and hence there must be an isomorphism $\phi : V \to U$ making the appropriate diagrams commute. It is routine to verify that $m'' \circ \phi \circ e' : B \to C$ is an appropriate diagonal fill-in morphism. The above shows that $E \subseteq M^{\uparrow}$ and $M \subseteq E^{\downarrow}$. To see that $M^{\uparrow} \subseteq E$, take $f \in M^{\uparrow}$ and write it $f = m \circ e$; m has an inverse to fill in the square $m \circ e = id \circ f$. Hence $f \in E$ because of 4.7.1.(A). Similarly for $E^{\downarrow} \subseteq M$. □

Proposition 4.7.4 *(E, M) is a factorization system in $\mathcal{C}$ if and only if it is a prefactorization system and all morphisms in $\mathcal{C}$ are of the form $m \circ e$ for* some *$m \in M$ and $e \in E$.*

Proof. The "only if" half of this follows immediately from the lemma. In the other direction, it is easy to see that E and M in a prefactorization system satisfy 4.7.1.(A). Moreover, if a factorization $f = m \circ e$ exists for any f in $\mathcal{C}$, it must be essentially unique. In fact, from a square $m \circ e = m' \circ e'$, one obtains two opposite fill-in diagonals which are easily shown to be each other's inverse. □

Remember that a monomorphism (epimorphism) is called *regular* when it is an equalizer (coequalizer). A category is called *regular* when it has finite limits, coequalizers and pullbacks of regular epis are regular epis. If m is a regular mono, $e \downarrow m$ for any epi e. In general, the converse does not hold, that is, elements of (regular monos)$^{\uparrow}$ need not be epis. Same thing for monos and regular epis. However, it is the case that:

Proposition 4.7.5 *Regular epis and monos form a stable factorization system (E_R, M) in a regular category.*

Proof. Let $m \circ u = v \circ e$ for $m : A \to B$ a mono and $e : C \to D$ the coequalizer of f and g. The fill-in diagonal is the mediating morphism $D \to A$ obtained from $u \circ f = u \circ g$.

Then we only have to show that any f has a factorization $f = m \circ e$ for some $m \in M$ and $e \in E_R$, as the uniqueness follows from the fill-in property. Let (h, k) be the kernel pair of f and let $e : B \to D$ be their coequalizer. Since $f \circ h = f \circ k$, we obtain a mediating m such that $f = m \circ e$. Let (u, v) be the kernel pair of m. It is easy to show that, since e is epi, so is the mediating $g : A \to C$; then, from $u \circ g = v \circ g$ we get $u = v$ and hence m is a mono. $\square$

Note that the class $\mathcal{M}$ of a (proper) factorization system in $\mathcal{E}$ is a class of admissible monos and it extends to one fibred over $\mathcal{E}$. A mono $m \in \mathcal{M}_{\Gamma,X}$ is a morphism in the fibre $\mathcal{C}/\Gamma$ and so is $\xrightarrow{Tm} TX$. Define $\Box_{\Gamma,X} m \in \mathcal{M}_{\Gamma,TX}$ to be the image of Tm.

Remark. It would be interesting to rephrase the above semantics in terms of an arbitrary fibration, using comprehension categories or, alternatively, locally small fibrations.

4.8 Evaluation relations

In the previous section we saw that a general standard semantics of the evaluation modality $\Box$ requires a category with a stable factorization system. We also saw that the proposed standard interpretation of $\Diamond$ requires structure to support higher order quantification. When such structure is available, an evaluation relation $(a \Leftarrow_A w)$ over $A \times TA$ can be defined from $\Box$ as in 4.5. However, given evaluation relations, only first order structure is needed to define $\Box$ and $\Diamond$:

$$[x \Leftarrow w]\ \phi(x) \stackrel{\text{def}}{=} \forall x : A.\ x \Leftarrow w \supset \phi(x) \tag{4.6}$$

$$\langle x \Leftarrow w\rangle\ \phi(x) \stackrel{\text{def}}{=} \exists x : A.\ x \Leftarrow w \wedge \phi(x) \tag{4.7}$$

Categorically, this corresponds to having a monic pair $A \xleftarrow{p_1} (\Leftarrow_A) \xrightarrow{p_2} TA$ and defining $\Box\phi$ as $\forall_{p_2}(p_1^*\phi)$ and $\Diamond\phi$ as $\exists_{p_2}(p_1^*\phi)$.

When $\Box$ and $\Diamond$ are defined as above, all the axioms for strong endofunctors of section 4.1 are derivable, from the rules of first order predicate calculus. Then, we observe that there are cases, for example when working in a logos [FS90, 1.7], in which $\Box$ and $\Diamond$ can be defined from $\Leftarrow$ but not vice versa.

Remark. Definitions (4.6) and (4.7) sometimes fail to yield the "expected" modalities. For example, let T be the monad for dynamic allocation of section 2.5, where $(a \Leftarrow w)$ is read "w represents the value a after a few allocations," and $new : T(Loc)$ is an operation for creating new locations. While there is no value $l : Loc$ such that $l \Leftarrow new$, $\langle x \Leftarrow new \rangle\, true$ holds for a $\Diamond$ defined as in [Mogb, example 4.11].

$\Box$

In section 4.4, we also observed that, given an evaluation relation and rules for the modalities in left and right form, such as $\Box$_*left* and $\Box$_*right*, inferences about modal formulae are easier than by using the rules in section 4.1, where each modality is related separately with each logical operator. The example was the derivation of $\Diamond$_$\vee$. Note that $\Box$_*left* and $\Box$_*right* are derivable for $\Box$ defined as in (4.6).

The above observations suggest that evaluation relations may yield both semantic and proof-theoretic benefits: a more general standard semantics and a more manageable proof system. A denotational account of evaluation could also allow a systematic approach to the study of computational adequacy.

In most cases there is an obvious evaluation relation associated with each notion of computation. Here are some examples (for the monad of dynamic allocations, described in 2.5, we write $\sigma_k^n : k \to F^n k$ for the morphisms $\sigma_k^0 = id_k$ and $\sigma_k^{n+1} = \sigma_{F^n k} \circ \sigma_k^n$):

Notion of computation	$\boldsymbol{TA}$	$\boldsymbol{a \Leftarrow z}$
exceptions	$A + E$	$inl(a) = z$
side effects	$(A \times S)^S$	$\exists\, s_1, s_2.\, z(s_1) = \langle a, s_2 \rangle$
nondeterminism	$\mathcal{P}(A)$	$a \in z$
dynamic allocation	$(TA)k = \sum_{n:N} A(F^n k)$	$z_k = (n, A\sigma_k^n(a_k))$

Given an endofunctor $T : \mathcal{C} \to \mathcal{C}$, evaluation relations can be obtained from first order structure as follows. Assuming a lifting monad $\langle (\_)_\perp, \theta \rangle$ (see 2.5) in $\mathcal{C}$,

$$a \Leftarrow_A z \stackrel{\text{def}}{=} \forall f : A \to A_\perp.\ (Tf)z = (T\theta)z \supset \exists x : A.\ fx = \theta a. \tag{4.8}$$

For all monads mentioned above, this formula produces the expected evaluation relation. The purpose of the lifting is to have "enough" morphisms f over which to quantify, even in degenerate cases such as $A = 1$.

Remark. If the mono requirement is satisfied (see 2.3), the above $\Leftarrow$ satisfies the axiom $a \Leftarrow \mathit{val}(a)$. If $\Box$ is defined by (4.6), formulae such as $\Box\_\mathit{let}$ (4.1) can be derived from $a \Leftarrow \mathit{let}(f, z) \supset \exists x.\, x \Leftarrow z \wedge a \Leftarrow fx$.

$\Box$

Below we show the conditions under which (4.8) is equivalent to (4.5). Let $\phi : X \rightarrowtail A$; we adopt the interpretation $[x \Leftarrow w]\ \phi \stackrel{\text{def}}{=} \exists z : TX.\ (T\phi)z = w$.

Theorem 4.8.1 *If T preserves pullbacks of the unit θ of the lifting monad, then (4.5) implies (4.8).*

Proof. Let $f : A \to A_\perp$, and let ϕ be the domain of f:

$$\begin{tikzcd} X \arrow[r] \arrow[d, tail, "\phi"'] \arrow[dr, phantom, "\lrcorner", very near start] & A \arrow[d, tail, "\theta"] \\ A \arrow[r, "f"'] & A_\perp \end{tikzcd}$$

Assuming $(Tf)w = (T\theta)w$, there exists $z : TX$ such that $(T\phi)z = w$:

$$
\begin{tikzcd}
1 \arrow[dr, dashed, "z"] \arrow[drr, bend left, "w"] \arrow[ddr, bend right, "w"'] & & \\
& TX \arrow[r] \arrow[d, tail, "T\phi"'] \arrow[dr, phantom, "\lrcorner", very near start] & TA \arrow[d, tail, "T\theta"] \\
& TA \arrow[r, "Tf"'] & TA_\perp
\end{tikzcd}
$$

Then, from (4.5), $\phi(a)$ holds and, therefore, there exists an $x : A$ such that $fa = \theta x$:

$$
\begin{tikzcd}
1 \arrow[dr, dashed] \arrow[drr, bend left, "x"] \arrow[ddr, bend right, "a"'] & & \\
& X \arrow[r] \arrow[d, tail, "\phi"'] \arrow[dr, phantom, "\lrcorner", very near start] & A \arrow[d, tail, "\theta"] \\
& A \arrow[r, "f"'] & A_\perp
\end{tikzcd}
$$

$\square$

Theorem 4.8.2 *If the unit θ of the lifting monad classifies all predicates of the logic, (4.8) implies (4.5).*

Proof. Let ϕ be a predicate on A and let f be the transpose of the partial map $[\phi, \phi] : A \rightharpoonup A$, that is:

$$
\begin{tikzcd}
X \arrow[r, "\phi"] \arrow[d, tail, "\phi"'] \arrow[dr, phantom, "\lrcorner", very near start] & A \arrow[d, tail, "\theta"] \\
A \arrow[r, "f"'] & A_\perp
\end{tikzcd}
$$

Assuming $[x \Leftarrow w]\ \phi$, that is a z such that $(T\phi)z = w$, one has:

$$(Tf)w = (Tf)(T\phi)z = (T\theta)(T\phi)z = (T\theta)w$$

and hence, from (4.8), there exists $x : A$ such that $fx = \theta a$. Then, a factorizes through ϕ because:

$$\begin{tikzcd}
1 \arrow[drr, bend left, "a"] \arrow[dr, dashed] \arrow[ddr, bend right, "x"'] & & \\
& X \arrow[r, "\phi"] \arrow[d, tail, "\phi"'] \arrow[dr, phantom, "\lrcorner", very near start] & A \arrow[d, tail, "\theta"] \\
& A \arrow[r, "f"'] & A_\perp
\end{tikzcd}$$

$\square$

5 Application: partial correctness

Partial correctness specifications are statements involving programs and assertions about their states. A well known example of a formal calculus based on such specifications is Hoare Logic (HL), which gives an intuitive grasp of simple `while`-programs.

To deal with more elaborate forms of computation than expressed by `while`-programs, HL requires some alteration. For instance, expressions of the programming language can be admitted inside assertions in so far as they are free from computational phenomena such as nontermination, side effects or exceptions. This "mathematical purity" is violated by many common programming devices such as function procedures, or block expressions.

In the block expression $\mathtt{begin}\, C\, \mathtt{result}\, E$, for example, the command C may fail to terminate, in which case the value of the whole expression is undefined, or it may modify variables which are not local to it. As for E, it may try to divide a number by zero, which may raise some exception. All such cases invalidate Reynolds' approach to program specification and are in fact proscribed in [Rey81]. Tennent [Ten91] admits possibly undefined expressions in the assertion language by introducing atomic formulae, such as Kleene equalities, "localizing" the effects of nontermination. Still, it is not clear how meaningful a reasoning can be carried out in the style of Hoare triples with partial expressions.

As for expressions with side effects, noninterference specifications must be introduced for reasoning about assignments. The problem, here, is that assertions may change the state in which expressions are evaluated, as in the following instance of the assignment axiom: $\{\mathtt{begin}\; b := \mathit{false}\; \mathtt{result}\; \neg b\}\; b' := \neg b\; \{\mathtt{begin}\; b := \mathit{false}\; \mathtt{result}\; b'\}$, which is false in all states where b is true. To prevent this, assertions should not interfere with the value of the expression being assigned, that is, in the example above, $(b := \mathit{false})\#\neg b$. Sophisticated semantic tools must be used for languages with block structure to correctly validate equivalences of programs involving nonlocal variables: *possible world semantics* ([OT92,OT93]) provides a safe barrier to affecting nonlocal states.

For more than two decades Hoare Logic has been adapted to cope with a variety of computational features, including procedures, local variables, gotos and parallelism (see [Apt81]). Most of the proposed variations have the flavour of ad hoc solutions: every time a new set of proof rules and axioms are proposed to give an account of a programming construct, new semantics and new proofs of soundness must be found, most of the time from scratch. Such a painstaking process can be simplified and made more systematic by basing the logic on an abstract notion of computation.

In chapter 1 we argued that Moggi's approach to denotational semantics based on monads supports a modular view of computation ([Mog90b,Wad92,CM93,Cen94]). The problem of finding *robust* theories of computation, that is to find sets of proof rules and axioms capable of surviving modifications to the semantics, can be addressed in the formal framework of Evaluation Logic, which is based on Moggi's monadic approach. Still, we must first understand if, in this framework, simple things can be done in a simple fashion. With this goal in mind, we set EL to work on *partial correctness*.

In section 5.1 we study the interaction between the monad $TX = (X \times S)_\perp^S$ to interpret possibly nonterminating computations with side effects and a class of admissible monos $\mathcal{M}$ to interpret formulae. We give appropriate closure conditions on $\mathcal{M}$ to guarantee a natural behaviour of formulae with respect to substitution and we validate axioms to be used later on. In section 5.2 we adopt the semantic setting of section 4.6, where a strong monad T is defined on a category $\mathcal{C}$ "of predomains," which is fully reflective in an ambient category $\mathcal{E}$ "of sets." We look for the abstract properties such models should have in order to validate the axioms introduced in section 5.1. In particular, we establish sufficient conditions for the monad $TX = (X \times S)_\perp^S$, which we assume defined in $\mathcal{C}$, to extend directly to $\mathcal{E}$, without using the reflection, as proposed in 4.6. It would be interesting to see whether other variations of the side-effects monad, e.g. the ones involving Phoa and Taylor's synthetic version of Plotkin powerdomains, would support a similar extension.

Then we consider a different set of axioms, proposed by Pitts in [Pit91], where EL's evaluation modalities have a stronger modal flavour. In section 5.3 we present a translation of this calculus, which we call ELse, into EL making implicit state variables visible. Using the results of 5.1 we show that the constants employed in this translation are defined for a large class of models and that theorems are translated into theorems.

In section 5.4 we show that Hoare Logic translates very naturally in ELse without committing to any of HL's assumptions as to the computational behaviour of expressions. Again, all theorems in HL are shown to be derivable in ELse. Moreover, the translation extends to *annotated* programs, that is programs which syntactically include assertions such as loop invariants, thus supporting Floyd-Hoare's method of *inductive assertions* (see [Hes92]) which is of practical importance in proofs of correctness. In 5.5 we develop a full proof of correctness for a textbook example of a `while`-program for integer division using, besides the axioms previously introduced, Scott induction. Section 5.6 discusses the admissibility of this principle.

5.1 Nontermination and side effects

State readers. First we consider simple computations of the form $QA \stackrel{\text{def}}{=} A^S$, which we call *state readers*. Note that, if $\mathcal{M}$ is a class of admissible monos, $m \in \mathcal{M}$ need not imply $m^X \in \mathcal{M}$. For example, taking as $\mathcal{M}$ the class of Scott-opens in the category of cpos and $\eta : N \to N_\perp$ the property of being defined, η^N is the nonobservable predicate "f is total" on partial continuous functions.

Lemma 5.1.1 *Let S be an object in a category $\mathcal{C}$ such that $(\_) \times S \dashv (\_)^S$. For all objects A in $\mathcal{C}$, there is an adjunction $\pi^* \dashv \forall_s : \mathcal{C}/A \rightharpoonup \mathcal{C}/A \times S$. In particular, $\forall_s f \cong \eta^* f^S$ for η the unit $A \to (A \times S)^S$, and vice versa $f^S \cong \forall_s \epsilon^* f$, where ϵ is the evaluation map.*

Lemma 5.1.2 *Let $\mathcal{M} \subseteq \mathcal{M}'$ be classes of admissible monos both with S-products, and let $\forall_s$ and $\forall'_s$ be the respective families of quantifiers. If $m \in \mathcal{M}$ then $\forall_s m \cong \forall'_s m$.*

The above is an instance of a well known more general result, the *absoluteness of indexed products* (see [HP89, proposition 2.8]).

Theorem 5.1.3 *Let a class of admissible monos $\mathcal{M}$ have S-products $\forall_S$. For $m \in \mathcal{M}$, $m^S \cong \forall_s \epsilon^* m \in \mathcal{M}$ and $Q \stackrel{def}{=} (\_)^S$ is an M-functor.*

Proof. $(\_)^S$ preserves limits because it is a right adjoint, so it is enough to show that $m^S \in \mathcal{M}$ if $m \in \mathcal{M}$. By lemma 5.1.1, there is an adjunction $\pi^* \dashv \forall'_s : \mathcal{M}'_A \rightharpoonup \mathcal{M}'_{A\times S}$, where $\mathcal{M}'$ is the class of all monos in $\mathcal{C}$, and $m^S \cong \forall'_s \epsilon^* m$. Therefore, since $\mathcal{M} \subseteq \mathcal{M}'$, by lemma 5.1.2, $m^S \cong \forall_s \epsilon^* m \in \mathcal{M}$. $\square$

Breaking into the structure of state readers, we can express $\Box$ and computational lifting for Q-computations in terms of $\forall$ and λ. Later we employ such axioms to validate properties of more complex computations built from Q.

$$\Box\_Q \quad \frac{\Gamma, x : X \vdash \phi(x)\ prop}{\Gamma, r : QX; \forall\, s.\, \phi(r(s)) \dashv\vdash [x \Leftarrow r]_Q\, \phi(x)}$$

$$let\_Q \quad \frac{x : A \vdash e(x) : B}{r : QA \vdash let\ x \Leftarrow r\ in\ [e(x)] = \lambda\, s.\, e(r(s))}$$

Lifting. Let $(\_)_\perp$ be a lifting monad defined from a class of admissible monos $\mathcal{N}$ as described in section 2.5. Like Q above, $(\_)_\perp$ preserves monos, but $m \in \mathcal{N}$ need not imply $m_\perp \in \mathcal{N}$. A counterexample is $m : 0 \to 1$ when $\mathcal{N}$ is the class of Scott-opens. However, the class of admissible monos $\mathcal{M}$ that we consider for interpreting the logic will contain, in general, more than domains of admissible partial functions and should not be confused with the class $\mathcal{N}$ of such domains from which $(\_)_\perp$ is defined. We will assume $\mathcal{N} \subseteq \mathcal{M}$. Theorem 5.1.8 establishes a sufficient condition on a class of admissible monos $\mathcal{M}$ to make $(\_)_\perp$ an M-functor.

Lemma 5.1.4 *Let $f : B \to A$ be a map in $\mathcal{C}$. If $f^* \dashv \forall_f : \mathcal{C}/A \rightharpoonup \mathcal{C}/B$, then the exponential y^f is defined for all y in $\mathcal{C}/A$ and $y^f \cong \forall_f f^* y$.*

Proof.

$$\cfrac{\cfrac{x \longrightarrow \forall_f f^* y}{f^* x \longrightarrow f^* y}}{x \times_A f = f \circ f^* x \longrightarrow y}$$

□

Lemma 5.1.5 *Let $m : B \to A$ be a mono. If the exponential y^m is defined for all y in $\mathcal{C}/A$, then $m^* \dashv \forall_m : \mathcal{C}/A \rightharpoonup \mathcal{C}/B$ and $\forall_m f \cong (m \circ f)^m$.*

Proof.

$$\cfrac{\cfrac{x \longrightarrow (m \circ f)^m}{x \times_A m = m \circ m^* x \longrightarrow m \circ f}}{m^* x \longrightarrow f}$$

□

Lemma 5.1.6 *If a lifting monad $\langle (\_)_\perp, \eta \rangle$ is defined in a category $\mathcal{C}$ with pullbacks, then $\eta_A^* \dashv \forall_{\eta_A} : \mathcal{C}/A_\perp \rightharpoonup \mathcal{C}/A$. In particular, for $f : B \to A$, $\forall_{\eta_A} f = \chi(\eta_B, f) = f_\perp$.*

Lemma 5.1.7 *Lifting preserves pullbacks.*

Proof. Let $Z = X \times_A Y$ be the vertex of the pullback of f along g, and let $h : W \to X_\perp$ and $k : W \to Y_\perp$ be such that $f_\perp \circ h = g_\perp \circ k$. The mediating $W \to Z_\perp$ is obtained from the obvious mediating $W' \to Z$, where $W' = (h^* \eta_X) \times_W (k^* \eta_Y)$. □

Theorem 5.1.8 *Let a class of admissible monos $\mathcal{M}$ and a lifting monad $\langle (\_)_\perp, \eta \rangle$ be defined in $\mathcal{C}$, and let the components of η be in $\mathcal{M}$. If $\mathcal{M}$ has $\mathcal{M}$-products, $(\_)_\perp$ is an M-functor.*

Proof. Let $m : B \rightarrowtail A$ in $\mathcal{M}$. By lemma 5.1.6, $\forall_{\eta_A} : \mathcal{C}/A \rightharpoonup \mathcal{C}/A_\perp$ exists and $m_\perp = \forall_{\eta_A} m$. By lemma 5.1.4, η_A-powers exist in $\mathcal{C}/A_\perp$, and, by lemma 5.1.5, we have $(\eta_A \circ m)^{\eta_A} \cong \forall_{\eta_A} m = m_\perp$. But the exponential $(\eta_A \circ m)^{\eta_A}$ in $\mathcal{C}/A_\perp$ must also be exponential in $\mathcal{M}_{A_\perp}$, hence $m_\perp \in \mathcal{M}$ by hypothesis. Lemma 5.1.7 finishes the job. $\square$

Theorem 5.1.9 *Let $\mathcal{M}$ be a class of admissible monos in $\mathcal{C}$ with $\mathcal{M}$ and S-products. The strong endofunctor $(\_ \times S)^S_\perp$ is an M-functor and hence $\Box_{A,B} : \mathcal{M}_{A\times B} \to \mathcal{M}_{A\times TB}$ commutes with substitution.*

Proof. Since $(\_ \times S)$ is an M-functor for any $\mathcal{M}$, the result follows from Theorem 5.1.3, Theorem 5.1.8 and from the fact that M-functors compose. $\square$

Cartesian units. Now we consider a *family* of monads suitable for interpreting computations with side effects. If T is a strong monad, one can model T-computations with side effects using the monad $\mathcal{F}T = T(\_ \times S)^S$. For example, we just showed that $(\_ \times S)^S_\perp$ is an M-functor by showing that $\mathcal{F}$ preserves M-functors and that $(\_)_\perp$ is one. We shall procede similarly to show that $(\_ \times S)^S_\perp$ validates the axiom $\Box\_val^*$ (see section 4.1). We believe that one obtains greater insight into a notion of computation from an analysis of *constructors* such as $\mathcal{F}$ above than from the study of the corresponding monad in isolation.

Below we use EL to address the elements of a class of admissible monos $\mathcal{M}$ defined in a category with a strong endofunctor T. If $\phi \in \mathcal{M}$, $T\phi$ is written $[x \Leftarrow z]\ \phi(x)$. Similarly, a restricted form of the computational metalanguage can be used when only a strong endofunctor (rather than a monad) is available on $\mathcal{C}$; in this case, $T(f)$ is written *let* $x \Leftarrow z$ *in* *val*$(f(x))$. When several such endofunctors are involved, we decorate the above operations with indices. For example:

$$\begin{array}{rcl} [x \Leftarrow z]_{TQ}\ \phi(x) & \cong & [y \Leftarrow z]_T\ [x \Leftarrow y]_Q\ \phi(x) \\ let_{TQ}\ x \Leftarrow z\ in\ val_{TQ}(e) & = & let_T\ y \Leftarrow z\ in\ val_T(let_Q\ x \Leftarrow y\ in\ val_Q(e)). \end{array}$$

The condition that the unit of a monad T is M-cartesian is expressed in EL by the equivalence $[x \Leftarrow val_T(x)]_T\ \phi(x) \cong \phi(x)$. For example, let T be $(\_)^S$, from $\Box\_Q$ we have: $[x \Leftarrow val(x)]\ \phi(x) \cong \forall\, s.\, \phi((\lambda\, s.\, x)s) \cong \forall\, s.\, \phi(x) \cong \phi(x)$. Here we assumed that S is not empty. Lemma 5.1.6 shows that the unit of $(\_)_\perp$ is also cartesian and hence, to get the result for $(\_ \times S)^S_\perp$, we only have to show:

Theorem 5.1.10 *$\mathcal{F}$ preserves monads with cartesian units.*

Proof. Splitting $\mathcal{F}T$ into its components and spelling out the $\Box$ modalities for $Q \stackrel{\text{def}}{=} (\_)^S$ and $R \stackrel{\text{def}}{=} (\_ \times S)$ as shown above:

$$[x \Leftarrow z]_{\mathcal{F}T}\ \phi(x) \cong [w \Leftarrow z]_Q\ [y \Leftarrow w]_T\ [x \Leftarrow y]_R\ \phi(x) \cong \forall\, s.\, [x, s' \Leftarrow z(s)]_T\ \phi(x)$$

Then, writing $val_{\mathcal{F}T}(x)$ as $\lambda\, s.\, val_T(x, s)$ and assuming T satisfies $\Box\_val^*$, we have:

$$\begin{aligned}
[x \Leftarrow val(x)]\ \phi(x) &\cong \forall\, s.\, [x, s' \Leftarrow (\lambda\, s.\, val_T(x, s))s]_T\ \phi(x) \\
&\cong \forall\, s.\, [x, s' \Leftarrow val_T(x, s)]_T\ \phi(x) \cong \forall\, s.\, \phi(x) \cong \phi(x)
\end{aligned}$$

$\Box$

Next we show that lifting preserves cartesian closed structure in the fibres. This is more precisely stated below, in the corollary to the following more general:

Theorem 5.1.11 *Let $\langle (\_)_\perp, \eta \rangle$ be a lifting monad in $\mathcal{C}$ and let the exponential x^y be defined in $\mathcal{C}/A$; then $(x_\perp)^{y_\perp}$ is defined in $\mathcal{C}/A_\perp$ and $(x^y)_\perp \cong (x_\perp)^{y_\perp}$.*

Proof. We show that there exists an isomorphism $\mathcal{C}(z, (x^y)_\perp) \cong \mathcal{C}(z, (x_\perp)^{y_\perp})$, natural in z:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{z \longrightarrow (x^y)_\perp \cong \forall_{\eta_A}(x^y)}{\eta_A^* z \longrightarrow x^y}}{\eta_A^* z \times y \longrightarrow x}}{\eta_A \circ (\eta_A^* z \times y) \longrightarrow \eta_A \circ x}}{(\eta_A \circ \eta_A^* z) \times y_\perp \longrightarrow \eta_A \circ x}}{\eta_A \times (z \times y_\perp) \cong (\eta_A \times z) \times y_\perp \longrightarrow \eta_A \circ x}}{z \times y_\perp \longrightarrow (\eta_A \circ x)^{\eta_A} \cong x_\perp}}{z \longrightarrow (x_\perp)^{y_\perp}}$$

□

Corollary 5.1.12 *Let a class of admissible monos $\mathcal{M}$ and a lifting monad $\langle (\_)_\perp, \eta\rangle$ be defined in $\mathcal{C}$, and let the components of η be in $\mathcal{M}$. If $\mathcal{M}$ has $\mathcal{M}$-products, then $(m^n)_\perp \cong (m_\perp)^{n_\perp}$, for m and n in the same M-fibre.*

Note that the equivalence established in the above corollary can be expressed in EL as follows (a similar axiom also holds for $(\_ \times S)$):

$$\Gamma, w : X_\perp; [x \Leftarrow w]_\perp\ (\phi \supset \psi) \dashv\vdash [x \Leftarrow w]_\perp\ \phi \supset [x \Leftarrow w]_\perp\ \psi$$

Part "$\vdash$" of the above axiom holds for $(\_)_\perp$-computations but not all interpretations of T validate it; for example, it does not hold for nondeterministic computations.

5.2 Side effects in the ambient category

Combining the synthetic domain theory view of domains as sets with Moggi's view of monads as notions of computation, one gets a scenario for denotational semantics where a strong monad T is defined on a category $\mathcal{C}$ of predomains which is fully reflective in a category $\mathcal{E}$ of sets. In section 4.6 we showed that T can always be extended to a monad

$\tilde{T} = TR : \mathcal{E} \to \mathcal{E}$, where R is the reflection, so that the respective categories of algebras are equivalent. Moreover, under mild conditions $\tilde{T}$ also has a tensorial strength.

However, assuming a class $\mathcal{M}$ of monos to interpret predicates in $\mathcal{E}$, the interpretation of EL described in 4.1 requires $\tilde{T}$ to interact nicely with $\mathcal{M}$. For example, $\tilde{T}$ should preserve monos in $\mathcal{M}$; when T is the identity functor, this amounts to R preserving monos and this need not happen in general.

Example. Given an element q of a poset Q, let $[q] \stackrel{\text{def}}{=} \{x \mid x \leq q \text{ or } q \leq x\}$. Consider the functor $R : \mathit{Poset} \to \mathit{Set}$ mapping a poset Q into the set $\{[q] \mid q \in Q\}$ of maximal subsets of connected elements. It is immediate to verify that R is a full reflection. Let $\mathbf{2}$ be the vertical two-element poset and 2 the horizontal one. There is a mono $m : 2 \to \mathbf{2}$ in Poset but no monos $R2 = 2 \to 1 = R\mathbf{2}$ in Set. So R cannot preserve monos.

$\square$

We conclude that, to make the interpretation 4.3 work in the general case, additional conditions are required to make $\tilde{T}$ behave properly, such as R preserving finite limits (although something milder would be enough). Here we consider a simpler solution where $\tilde{T}$ is essentially the same functor as T, thus making the results established in section 5.1 directly applicable to $\tilde{T}$. Note, however, that similar constructions may not be available for all T. There is a natural $\tilde{T} \stackrel{\text{def}}{=} (\_ \times S)_{\perp}^{S}$ in $\mathcal{E}$ which restricts to T in $\mathcal{C}$. We show that each piece of the structure of T can be found in $\mathcal{E}$ extending the corresponding structure in $\mathcal{C}$.

First, the inclusion $\mathcal{C} \hookrightarrow \mathcal{E}$ preserves binary products because it is a right adjoint; assuming that R preserves binary products too, the inclusion also preserves exponentials; in fact, let A and B be in $\mathcal{C}$ and let B^A be their $\mathcal{C}$-exponential; then:

$$\begin{aligned}\mathcal{E}(X, B^A) \cong \mathcal{C}(RX, B^A) \cong \mathcal{C}(RX \times A, B) \cong \mathcal{C}(RX \times RA, B) \cong \\ \mathcal{C}(R(X \times A), B) \cong \mathcal{E}(X \times A, B).\end{aligned}$$

So, the only problem in taking $\tilde{T} = (\_ \times S)_{\perp}^{S}$ is to find a suitable lifting monad to extend $\langle (\_)_{\perp}, \eta \rangle$.

Lemma 5.2.1 *Let $\langle R, I, \rho\rangle : \mathcal{E} \rightharpoonup \mathcal{C}$ be a reflection of a category $\mathcal{E}$ into its full subcategory $\mathcal{C}$ and let $t : 1 \to \Sigma$ be a dominance in $\mathcal{C}$. If R preserves pullbacks of t, then t is a dominance in $\mathcal{E}$.*

Proof. We have to show that t is *generic* in a class of Σ-subobjects in $\mathcal{E}$ closed under composition. Without loss of generality we assume $t = Rt$. Let $m = h^{-1}t = k^{-1}t$. Since R preserves pullbacks of t, both Rh and Rk classify Rm, so $Rh = Rk$ and hence $h = k$. This shows that t is generic. Let $m = f^{-1}t : C \to B$, $n = g^{-1}t : D \to C$ and let h be the unique morphism whose transpose $h^\dagger$ classifies $Rn \circ Rm$. Since the squares describing naturality of ρ on n and m are pullbacks, it must be $n \circ m = h^{-1}t$, which makes $n \circ m$ a Σ-subobject. $\square$

Let t be a dominance in $\mathcal{C}$; $\langle (\_)_\perp, \eta\rangle$ is defined in $\mathcal{C}$ iff $t^* \dashv \forall_t : \mathcal{C}/\Sigma \rightharpoonup \mathcal{C}$. However, if, for example, $\mathcal{E}$ is locally cartesian closed, such an adjunction is also defined in $\mathcal{E}$, as follows from lemma 5.1.5.

Theorem 5.2.2 *Let $\langle R, I, \rho\rangle : \mathcal{E} \rightharpoonup \mathcal{C}$ be a reflection of a category $\mathcal{E}$ in its full subcategory $\mathcal{C}$ and let $\langle (\_)_\perp, \eta\rangle$ be a lifting monad in $\mathcal{C}$. If the reflection preserves pullbacks of $t = \eta_1$ and the reindexing functor along t has right adjoint $\forall_t : \mathcal{E} \to \mathcal{E}/\Sigma$, then $\langle (\_)_\perp, \eta\rangle$ extends to a lifting monad on $\mathcal{E}$.*

Proof. By lemma 5.2.1, t is a dominance in $\mathcal{E}$. Then $\forall_t$ provides each object of $\mathcal{E}$ with classifiers of partial maps whose domains are Σ-subobjects. So we have to check that, for every object A of $\mathcal{C}$, η_A is a Σ-partial map classifier in $\mathcal{E}$. Let $(m, f) : B \rightharpoonup A$ be a Σ-partial map, and let $f^\dagger$ be the transpose of f as in the diagram.

$$\begin{tikzcd}
C \arrow[r, "\rho_C"] \arrow[d, tail, "m"'] \arrow[dr, phantom, "\lrcorner", very near start] & RC \arrow[r, "f^\dagger"] \arrow[d, tail, "Rm"'] \arrow[dr, phantom, "\lrcorner", very near start] & A \arrow[d, tail, "\eta_A"] \\
B \arrow[r, "\rho_B"'] & RB \arrow[r, "\chi(Rm, f^\dagger)"'] & A_\perp
\end{tikzcd}$$

Since R preserves pullbacks of t, Rm is a Σ-subobject in $\mathcal{C}$. Let $\chi(Rm, f^\dagger) : RB \to A_\perp$ classify $(Rm, f^\dagger)$ in $\mathcal{C}$; it is easy to verify that, since the square describing naturality of ρ on m is a pullback, $\chi(Rm, f^\dagger) \circ \rho_B$ classifies (m, f) in $\mathcal{E}$. $\square$

5.3 Assertions

We now have a semantic setting in which to interpret a theory of the Evaluation Logic for possibly nonterminating programs with side effects and we know that substitution and $\Box\_val^*$ hold in this setting. However, there is something "modal" to reasoning about `while`-programs with Hoare-triples that makes Hoare-like logics particularly intuitive and which is not captured by the above semantics.

If we want formulae to be interpreted "in a state," like *assertions* in Hoare Logic, we must adopt a nonstandard interpretation of EL, as in [Pit91], where formulae with a free variable of type X are interpreted as predicates over $X \times S$, for S the interpretation of states. Then, modelling computations on X as partial maps $S \rightharpoonup (X \times S)$, we can give the following set theoretic interpretation of the evaluation modalities ([Pit91]), where side effects are "passed" across the brackets (compare it with the corresponding equations in section 4.1):

$$\begin{aligned} [\![w : TX \vdash [x \Leftarrow w]\, \varphi \ \mathit{assert}]\!] &= \{w, s \mid \forall\, x, s'.\, ws = (x, s') \supset \varphi(x, s')\} \\ [\![w : TX \vdash \langle x \Leftarrow w \rangle\, \varphi \ \mathit{assert}]\!] &= \{w, s \mid \exists\, x, s'.\, ws = (x, s') \wedge \varphi(x, s')\} \end{aligned}$$

Here, we adopt as primitive calculus Moggi's version of the Evaluation Logic, EL ([Mogb]), which is based on a standard semantics, and derive inference rules for a calculus of assertions to be used in reasoning about programs without explicit reference to the states of computation. In the derived calculus of assertions, which we call ELse, the evaluation modalities have the above intuitive meaning. ELse is essentially the Evaluation Logic proposed by Pitts in [Pit91] with a few special constants to get hold of simple imperative programming constructs. We shall call "assertions" the formulae of ELse and "propositions" those of EL.

EL*se*. ELse has the same syntax as EL. The term language is the computational lambda calculus over a signature Σ which includes: the type Int of integers with the usual arithmetical operations; the type $Bool$ of booleans with the usual boolean operations and the conditional $cond_X : Bool \to X \to X \to X$; the operations $up_l : Int \to T1$ and $ct_l : T(Int)$ respectively of memory update and lookup; a fixed point operator $rec : (T1 \to T1) \to T1$, with the fixed point property $rec(M) = M(rec(M))$. Intuitively, the program $up_l(n)$ assigns value n to the memory location l, while the computation ct_l returns the value in l. For these operators we adopt the axioms given in [Pit91]. Simple `while`-programs can be translated into the computational metalanguage over Σ as shown in section 5.4. The inference rules of ELse include all general and special axioms of section 4.1, with empty Δ in $\Box\_intro$ and $\Diamond\_intro$. Hence, ELse extends the calculus in [Pit91] to include all operators of first order predicate calculus.

Notational conventions. We let $e_1; e_2$ stand for $let\ x \Leftarrow e_1\ in\ e_2$ when x is not free in e_2. Similarly, when $p : T1$, we drop the dummy z in $[z \Leftarrow p]\ \theta$ and write $[p]\ \theta$ instead. When no confusion arises, we shall informally use logical variables as names for locations. So, if $\vec{x} = x_1, \ldots x_n$, we write $[\vec{x} \Leftarrow ct]\ \theta$ for the formula $[x_1 \Leftarrow ct_{x_1}]\ \ldots [x_n \Leftarrow ct_{x_n}]\ \theta$ and $let\ \vec{x} \Leftarrow ct\ in\ e$ for $let\ x_1 \Leftarrow ct_{x_1}\ in\ (\ldots(let\ x_n \Leftarrow ct_{x_n}\ in\ e)\ldots)$. Finally, we simply write $up(\vec{x})$ for $up_{x_1}(x_1); \ldots up_{x_n}(x_n)$.

Examples. An example to illustrate the different meaning of necessity in ELse and EL is the judgement $[u \Leftarrow ct_x]\ u \geq 0 \vdash [up_x(-1)]\ [u \Leftarrow ct_x]\ u \geq 0$, which is derivable from $\Box\_intro$ in EL, but false in ELse, where $\Box\_intro$ is subject to the restriction mentioned above. Conversely, $\vdash [up_x(1)]\ [u \Leftarrow ct_x]\ u \geq 0$ is derivable in ELse but false in EL, since $\Box\_let^*$, which is an axiom in ELse, is not sound under the interpretation (4.3) of EL when the μ of the monad T is not M-cartesian. Note that already the monad $(\_)^S$ fails to satisfy this axiom.

Translating assertions into propositions. Assertions $\Gamma \vdash \varphi$ *assert* translate into propositions $\Gamma, s : S \vdash (\![\varphi]\!)_s$ *prop*, where $(\![\varphi]\!)_s$ may contain a free variable s. All theorems in EL*se* are shown to be derivable in EL once the invisible state variables have been made explicit.

As suggested in [Mogb], there is an obvious way of "simulating" EL*se*'s modalities in EL in the case of computations with side effects: the idea is to implement propagation of states across formulae manually, via two constants $h : S \to T1$ and $k : TS$. Intuitively, the program $h(s)$ updates the current state with s, while k is the computation that returns the current state as value. Such gadgets are defined, for example, for all monads in the family $\mathcal{F}$ of section 5.1, which includes models for partial and nondeterministic computations with side effects. Let H be an arbitrary monad and let $T \stackrel{\text{def}}{=} \mathcal{F}H = H(\_ \times S)^S$. Then, $h(s) \stackrel{\text{def}}{=} val_{QH}(*, s) = \lambda\, s'.\, val_H(*, s)$ and $k \stackrel{\text{def}}{=} \lambda\, s.\, val_H(s, s)$. For the two constants h and k we assume the following special axioms: let *skip* be the trivial program $val(*) : T1$,

$$
\begin{array}{ll}
\text{Ax\_1} & h(s); k = h(s); val(s) \\
\text{Ax\_2} & let\ \ s \Leftarrow k\ \ in\ \ h(s) = skip \\
\Box\_h & [x \Leftarrow (let\ \ y \Leftarrow w\ \ in\ \ (h(s); z))]\ \phi \vdash [y \Leftarrow w]\ [x \Leftarrow h(s); z]\ \phi \\
\Box\_k & \dfrac{\Gamma \vdash [s \Leftarrow k]\ \phi}{\Gamma, s : S \vdash \phi}
\end{array}
$$

Proposition 5.3.1 Ax_1/2 $\Box$_h *and* $\Box$_k *are valid for all monads of the family* $\mathcal{F}$.

Proof. Again we use EL as the internal logic of a category where the functor $R \stackrel{\text{def}}{=} (\_ \times S)$ and the monads H and $Q \stackrel{\text{def}}{=} (\_)^S$ are defined. We only show the argument for Ax_1, as for the others a similar routine is repeated.

$$h(s); val(s) \;=\; let_{QHR}\; x \Leftarrow h(s) \; in \; val_{QHR}(s) \;=$$
$$let_{QH}\; y \Leftarrow val_{QH}(*, s) \; in \; val_{QH}(let_R\; x \Leftarrow y \; in \; val_R(s)) \;=$$
$$val_{QH}(let_R\; x \Leftarrow \langle *, s\rangle \; in \; val_R(s)) \;=$$
$$val_{QT}\langle s, s\rangle \;=\; val_Q((\lambda\, s'.\, val_T(s', s'))s) \;=$$
$$let_{QHR}\; x \Leftarrow val_{QH}(*, s) \; in \; \lambda\, s'.\, val_T(s', s') \;=\; h(s); k.$$

□

Let $K : TX \to T(X \times S)$ be the operator

$$K(w) \stackrel{\text{def}}{=} (let \;\; x \Leftarrow w \;\; in \;\; (let \;\; s \Leftarrow k \;\; in \;\; val(x, s))).$$

Intuitively, K runs the computation w and returns the final state as part of the result. An assertion $\Gamma \vdash \varphi$ *assert* in EL*se* is translated into a proposition $\Gamma, s : S \vdash (\![\varphi]\!)_s$ *prop* in EL as follows:

$$\begin{aligned}
(\![e_1 = e_2]\!)_s &\stackrel{\text{def}}{=} e_1 = e_2 \\
(\![\varphi \wedge \vartheta]\!)_s &\stackrel{\text{def}}{=} (\![\varphi]\!)_s \wedge (\![\vartheta]\!)_s \qquad \text{(and similarly for all other first order connectives)} \\
(\![\, [x \Leftarrow w] \; \varphi]\!)_s &\stackrel{\text{def}}{=} [x, s' \Leftarrow h(s); K(w)] \; (\![\varphi]\!)_{s'}
\end{aligned}$$

Proposition 5.3.2 *If* $\Gamma; \Delta \vdash \phi$ *is a theorem in ELse,* $\Gamma, s : S; (\![\Delta]\!)_s \vdash (\![\phi]\!)_s$ *is a theorem in EL with* $\Box$*_val***,* AX*_1/2,* $\Box$*_h and* $\Box$*_k.*

Proof. All axioms in EL*se* can be translated and derived in El. Here we show a derivation of the axiom ($\Box$*_let**), which best illustrates the differences between the two logics of assertions and of propositions. From AX_2 and $\Box$_h the following sequent is derivable:

$$\text{LM\_K} \qquad [x \Leftarrow (let \;\; y \Leftarrow w \;\; in \;\; z)] \; \phi \vdash [y, s \Leftarrow K(w)] \; [x \Leftarrow h(s); z] \; \phi.$$

Then:

$$
\begin{array}{c}
\llparenthesis\, [y \Leftarrow (let\ x \Leftarrow z\ in\ w)]\ \varphi \rrparenthesis_s \\
\hline
[y, s_1 \Leftarrow h(s); K(let\ x \Leftarrow z\ in\ w)]\ \llparenthesis \varphi \rrparenthesis_{s_1} \\
\hline
[y, s_1 \Leftarrow h(s); let\ y_1 \Leftarrow (let\ x \Leftarrow z\ in\ w)\ in\ let\ s_2 \Leftarrow k\ in\ [y_1, s_2]]\ \llparenthesis \varphi \rrparenthesis_{s_1} \\
\hline
[y, s_1 \Leftarrow (let\ x \Leftarrow (h(s); z)\ in\ K(w))]\ \llparenthesis \varphi \rrparenthesis_{s_1} \\
\hline
[x, s_2 \Leftarrow K(h(s); z)]\ [y, s_1 \Leftarrow h(s_2); K(w)]\ \llparenthesis \varphi \rrparenthesis_{s_1} \quad (\textrm{L}\scriptstyle{\textrm{M}}\textrm{\_K}) \\
\hline
[x, s_2 \Leftarrow h(s); K(z)]\ [y, s_1 \Leftarrow h(s_2); K(w)]\ \llparenthesis \varphi \rrparenthesis_{s_1} \\
\hline
\llparenthesis\, [x \Leftarrow z]\ [y \Leftarrow w]\ \varphi \rrparenthesis_s
\end{array}
$$

$\square$

Note that a slightly stronger version of the axioms in [Pit91] can be derived in EL. Consider the axiom:

$$
\Box\_intro \qquad \frac{\Gamma, x : X; \Delta, \varphi \vdash \psi}{\Gamma, w : TX; \Delta, [x \Leftarrow w]\ \varphi \vdash [x \Leftarrow w]\ \psi} \qquad (x \notin FV(\Delta), \Delta \text{ state-free})
$$

In [Pit91] Δ must be empty, as usually happens in modal logics. This condition can be explained in our setting by noticing that one thing the modality does, as is clear from the above definition of $\llparenthesis\, [x \Leftarrow w]\ \varphi \rrparenthesis_s$, is to bind a hidden state variable implicitly free in φ and ψ. Hence such a variable should not be free in Δ, that is, Δ should be independent of the state. One easy way to meet such condition is by requiring that Δ contains no modalities. That's how we shall read the side condition of $\Box\_intro$ and how we shall use it in section 5.5.

The following proposition is an immediate consequence of 5.1.10, 5.2.2, 5.3.1 and 5.3.2.

Proposition 5.3.3 *Models of the form $\mathcal{F}(H)$, where the unit of H is cartesian, support a standard interpretation of ELse.*

5.4 Partial correctness

Partial correctness can be expressed by using an operation tr mapping *annotated programs* to assertions. An elementary annotated program is a pair (p, θ), where the assertion θ is understood as a postcondition of $p : T1$. Intuitively, $tr(p, \theta)$ is the *weakest liberal precondition* of p satisfying θ, which is expressed in ELse by the formula $[p]\ \theta$. This can be extended to a simple form of annotated programs Q obtained by alternating programs and assertions:

$$Q \ ::= \ nil \mid (p, \theta); Q$$

In particular, $(skip, \theta); Q$ is an annotated program that starts with an assertion and $Q; (p, true)$ is one that ends with a program. The function tr is defined as follows:

$$\begin{aligned} tr(nil) &= true \\ tr((p, \theta); Q) &= [p]\ (\theta \wedge tr(Q)). \end{aligned}$$

Intuitively, $tr(Q)$ is satisfied by all states s such that all assertions in Q are true when they are reached during the execution of Q in s.

Annotation may provide valuable information for proving correctness. For example, the following derivable rule infers $\zeta \vdash tr((p, \theta); Q)$ from $\zeta \vdash tr(p, \theta)$ and $\theta \vdash tr(Q)$. Such a rule is used in 5.5 and referred to as TR_TRANS:

$$\dfrac{\dfrac{\dfrac{\zeta \vdash tr(p, \theta)}{\zeta \vdash [p]\ \theta} \qquad \dfrac{\dfrac{\theta \vdash tr(Q)}{\vdash \theta \supset tr(Q)}}{\vdash [p]\ (\theta \supset tr(Q))}}{\zeta \vdash [p]\ (\theta \wedge tr(Q))}}{\zeta \vdash tr((p, \theta); Q)}$$

Given an input condition θ and an annotated program Q, we say *partial correctness* specifications are statements of the form $\theta \supset tr(Q)$. This echoes the usual view of Hoare triples in modal frameworks, where $\{\vartheta\}P\{\zeta\}$ is interpreted as $\vartheta \supset [P]\,\zeta$ (see, for example, the account of Hoare Logic in the modal mu-calculus given in [BS92]). Note also that, as one would expect in such contexts, the above formula can be converted into a statement of *total* correctness by changing $[p]$ into $\langle p \rangle$ in the definition of tr.

Next we translate Hoare triples involving programs of a `while`-language into statements of partial correctness in ELse and show that all theorems of Hoare Logic translate into theorems of ELse.

Hoare Logic's *assertion language* is a first order language $\mathcal{L}$ with equality (see [Apt81]). In the following we take $\mathcal{L}$ to be Peano arithmetic with minus. It will be convenient to assume that variables of $\mathcal{L}$ are variables of ELse, so that $\vartheta \in \mathcal{L}$ is also an assertion in ELse.

Terms of type X in $\mathcal{L}$ are translated into terms of type TX of ELse by a family of functions $(\!|\_|\!)_\Gamma$, indexed by (integer) contexts $\Gamma = x_1 : Int, \ldots x_n : Int$.

$$
\begin{array}{rcll}
(\!|x|\!)_\Gamma & = & ct_x & \text{if } x \notin dom(\Gamma) \\
(\!|x|\!)_\Gamma & = & [x] & \text{if } x \in dom(\Gamma) \\
(\!|n|\!)_\Gamma & = & [n] & \\
(\!|t_1 + t_2|\!)_\Gamma & = & let\ x \Leftarrow (\!|t_1|\!)_\Gamma\ in\ let\ y \Leftarrow (\!|t_2|\!)_\Gamma\ in\ [x+y] & \\
 & & & \\
(\!|absurd|\!)_\Gamma & = & absurd & \\
(\!|t_1 = t_2|\!)_\Gamma & = & [x \Leftarrow (\!|t_1|\!)_\Gamma]\ [y \Leftarrow (\!|t_2|\!)_\Gamma]\ x = y & \text{and similarly for } \leq, \geq, >, \text{ and } < \\
(\!|\phi \wedge \psi|\!)_\Gamma & = & (\!|\phi|\!)_\Gamma \wedge (\!|\psi|\!)_\Gamma & \text{and similarly for } \vee \text{ and } \supset \\
(\!|\forall x.\phi|\!)_\Gamma & = & \forall x.(\!|\phi|\!)_{\Gamma,x} & \text{and similarly for } \exists.
\end{array}
$$

The translation enforces EL's disciplined interaction between logic and computation on HL. Logical and program variables, which are distinct in EL but not in HL, are separated, thus making explicit that bit of "computation" which is going on in HL's assertions: reading the memory. A possibly open $\vartheta \in \mathcal{L}$ is translated into a closed assertion $(\!|\vartheta|\!)$ in which free variables x are replaced by constants state readers ct_x whereas bound variables are treated as genuine logical variables.

Example. The assertion $x = y$ holds of all states where locations x and y contain the same value. However, $\forall\, x.\, x = y$ requires the content of y to be equal to any *integer* x. Our translation distinguishes between logical and program variables as follows:

$$\begin{array}{rcl} (\![x = y]\!) & = & [u \Leftarrow ct_x]\ [v \Leftarrow ct_y]\ u = v \\ (\![\forall\, x.\, x = y]\!) & = & \forall\, u.\, [v \Leftarrow ct_y]\ u = v \end{array}$$

$\square$

The language $\mathcal{W}$ of `while`-programs, is the least class of programs such that:

- for every variable $x \in \mathit{Ide}$ and $\mathcal{L}$-term t, $x := t \in \mathcal{W}$;
- if C, C_1 and $C_2 \in \mathcal{W}$, then $C_1; C_2 \in \mathcal{W}$ and, for every quantifier-free formula $B \in \mathcal{L}$, $\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2$ and $\texttt{while}\ B\ \texttt{do}\ C\ \texttt{od} \in \mathcal{W}$.

A `while`-program P is translated into a closed ELse-term $(\![P]\!)$ of computational type $T1$ as follows (we omit the subscripts Γ):

$$\begin{array}{rcl} (\![x := t]\!) & = & \mathit{let}\ \ x \Leftarrow (\![t]\!)\ \ \mathit{in}\ \ \mathit{up}_x(x) \\ (\![C_1; C_2]\!) & = & (\![C_1]\!); (\![C_2]\!) \\ (\![\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!) & = & \mathit{let}\ \ b \Leftarrow (\![B]\!)\ \ \mathit{in}\ \ \mathit{cond}\,(b, (\![C_1]\!), (\![C_2]\!)) \\ (\![\texttt{while}\ B\ \texttt{do}\ C\ \texttt{od}]\!) & = & \mathit{rec}\,(\lambda\, p : T1.\, (\mathit{let}\ \ b \Leftarrow B\ \ \mathit{in}\ \ \mathit{cond}\,(b, (\![C]\!); p\,, \mathit{skip}))). \end{array}$$

Note that here B is parsed as a boolean expression.

In the next section we prove the correctness of a program for integer division. This program is annotated with a loop invariant, which provides a guideline for the proof. Simple annotated `while`-programs of the form $(C_1\{\vartheta_1\}; C_2\{\vartheta_2\}\ldots)$, obtained by alternating commands with assertions, translate straightforwardly into annotated programs $(((\![C_1]\!), (\![\vartheta_1]\!)); ((\![C_2]\!), (\![\vartheta_2]\!))\ldots)$. This is not yet satisfactory since annotations should also be allowed, for example, in the body of while-loops. To handle this situation, the logic should be extended with recursive definitions of predicates, but we shall forgo this pleasure for the moment.

Hoare triples translate straightforwardly into partial correctness specifications in ELse:

$$\{\vartheta\}P\{\zeta\} \stackrel{\text{def}}{=} (\!|\vartheta|\!) \supset tr\,(\!|P\{\zeta\}|\!)$$

Lemma 5.4.1 *For all $\mathcal{L}$-terms t with free variables $\vec{y}, \vec{x}$, $(\!|t|\!)_{\vec{y}} = let\ \ \vec{x} \Leftarrow ct\ \ in\ \ [t]$. For all formulae $\vartheta(\vec{y}, \vec{x}) \in \mathcal{L}$, $(\!|\vartheta|\!)_{\vec{y}} \dashv\vdash [\vec{x} \Leftarrow ct]\ \vartheta$.*

Lemma 5.4.2 (Substitution) *Let $\vartheta(x)$ and t be in $\mathcal{L}$. $(\!|\vartheta(t)|\!)_\Gamma \dashv\vdash [x \Leftarrow (\!|t|\!)_\Gamma]\ (\!|\vartheta(x)|\!)_{\Gamma,x}$.*

Proposition 5.4.3 *If $\{\vartheta\}P\{\zeta\}$ is a theorem in Hoare Logic, $(\!|\vartheta|\!) \vdash tr\,(\!|P\{\zeta\}|\!)$ is derivable in ELse.*

Proof. As an example, we show the derivation of the axiom for assignment:

$$\{\phi(t)\}x := t\{\phi(x)\}.$$

For simplicity we assume that x is the only free variable in ϕ. The proof is presented in a natural deduction style and it makes use of the derivable judgement LM_3: $\Gamma; \phi \vdash [up_y(x)]\ [x \Leftarrow ct_y]\ \phi$ (ϕ state-free) from section 5.5.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(\!|\phi(t)|\!)}{[x \Leftarrow (\!|t|\!)]\ (\!|\phi(x)|\!)_x}\ \text{by lemma 5.4.2}}{[x \Leftarrow (\!|t|\!)]\ \phi(x)}\ \text{by lemma 5.4.1}}{[x \Leftarrow (\!|t|\!)]\ [up_x(x)]\ [x \Leftarrow ct]\ \phi(x)}\ \text{by LM\_3}}{\dfrac{[let\ \ x \Leftarrow (\!|t|\!)\ \ in\ \ up_x(x)]\ [x \Leftarrow ct]\ \phi(x)}{[(\!|x := t|\!)]\ (\!|\phi(x)|\!)}\ \text{by lemma 5.4.1}}}{tr\,(\!|x := t\,\{\phi(x)\}|\!)}$$

□

5.5 Integer division

Here we consider a classic textbook example, a partial correctness specification $\{\vartheta\}P\{\zeta\}$ of a `while`-program for integer division. We translate such specification in ELse as shown in section 5.4 and derive it in the calculus of assertions. In fact, we derive a stronger version $(\![\{\vartheta\}P'\{\zeta\}]\!)$, where P' is obtained by annotating P with a loop invariant.

Besides TR_TRANS introduced above, we use the following special axioms for $cond$, up and ct:

$$\frac{\Gamma; \Delta, b = \mathit{true} \vdash \phi(M) \quad \Gamma; \Delta, b = \mathit{false} \vdash \phi(N)}{\Gamma; \Delta \vdash \phi(\mathit{cond}(b, M, N))} \quad \textsc{Cond}$$

$$\frac{\Gamma \vdash [x \Leftarrow ct_y]\ \phi}{\Gamma, x : X \vdash [up_y(x)]\ \phi} \quad \textsc{Ct\_Up}$$

Moreover, in reasoning about `while`-loops, we use Scott-induction:

$$\textsc{Ind} \quad \frac{\Gamma; \Delta \vdash \phi(\uparrow) \quad \Gamma, p : T1; \Delta, \phi(p) \vdash \phi(M(p))}{\Gamma; \Delta \vdash \phi(\mathit{rec}\,(M))} \quad (\phi \text{ inductive in } p)$$

where $\uparrow$ stands for the always diverging program $\mathit{rec}\,(\lambda\, p.\, p)$. The side condition of IND asks for fixed-point admissibility: since our predicates (and assertions) are arbitrary subsets of domains, not all formulae $\Gamma, x : X \vdash \phi\ \mathit{prop}$ are suitable for fixed-point induction on x. In section 5.6 we give a criterion for establishing the side condition; we show that admissibility of a proposition ϕ can often be checked via syntactic analysis of ϕ (similarly for assertions) and prove the soundness of the proposed algorithm.

The following derivable sequents are used in the proof below:

LM_1 $\quad \Gamma; \Delta, [x \Leftarrow w]\ (\phi \supset \psi) \vdash [x \Leftarrow w]\ \phi \supset [x \Leftarrow w]\ \psi$

LM_2 $\quad \Gamma, w : TX; \phi \vdash [x \Leftarrow w]\ \phi \qquad (x \notin FV(\phi), \phi \text{ state-indep.})$

LM_3 $\quad \Gamma; \phi \vdash [up_y(x)]\ [x \Leftarrow ct_y]\ \phi \qquad (\phi \text{ state-indep.})$

LM_4 $\quad \Gamma; \Delta, [up_y(x)]\ \phi \dashv\vdash [up_y(x)]\ [x \Leftarrow ct_y]\ \phi$

These are the definitions of P, P', θ, ζ and their translation in ELse:

$$
\begin{aligned}
\{\vartheta\}P\{\zeta\} &\stackrel{\text{def}}{=} \{x \geq 0 \land y \geq 0\}P\{a \cdot y + b = x \land b \geq 0 \land b < y\} \\
P &\stackrel{\text{def}}{=} b := x\,;\, a := 0\,;\, \texttt{while}\ b \geq y\ \texttt{do}\ b := b - y\,;\, a := a + 1\ \texttt{od} \\
\xi &\stackrel{\text{def}}{=} a \cdot y + b = x \land b \geq 0 \\
P' &\stackrel{\text{def}}{=} b := x\,;\, a := 0\,;\, \{\xi\}\,;\, \texttt{while}\ b \geq y\ \texttt{do}\ b := b - y\,;\, a := a + 1\ \texttt{od} \\
(\!|\{\vartheta\}P\{\zeta\}|\!) &\equiv ([x, y \Leftarrow ct]\ \vartheta) \supset \\
&\quad [let\ x \Leftarrow ct_x\ in\ up_b(x)\,;\, up_a(0)\,;\, rec\,(\lambda p : T1.\ let\ b, y \Leftarrow ct\ in \\
&\qquad cond\,(b \geq y, \\
&\qquad\quad let\ b, y \Leftarrow ct\ in\ up_b(b - y)\,;\, let\ a \Leftarrow ct_a\ in\ up_a(a + 1)\,;\, p, \\
&\qquad\quad skip\,))]\ [x, y, a, b \Leftarrow ct]\ \xi \land b < y \\
(\!|\{\vartheta\}P'\{\zeta\}|\!) &\equiv ([x, y \Leftarrow ct]\ \vartheta) \supset \\
&\quad tr\,((let\ x \Leftarrow ct_x\ in\ up_b(x)\,;\, up_a(0)\,,\, [x, y, a, b \Leftarrow ct]\ \xi)\,; \\
&\qquad (rec(\ldots)\,,\, [x, y, a, b \Leftarrow ct]\ \xi \land b < y)) \\
&\equiv ([x, y \Leftarrow ct]\ \vartheta) \supset \\
&\quad [z \Leftarrow (let\ x \Leftarrow ct_x\ in\ up_b(x)\,;\, up_a(0))]\ (([x, y, a, b \Leftarrow ct]\ \xi) \land \\
&\qquad [w \Leftarrow rec(\ldots)]\ [x, y, a, b \Leftarrow ct]\ \xi \land b < y)
\end{aligned}
$$

We derive the above formula in natural deduction style, which allows greater compactness. Some obvious steps are omitted.

1. since $\vartheta \supset (0 \cdot y + x = x \land x \geq 0)$ is a tautology:
   $[up_b(x); up_a(0)]\ (\vartheta \supset (0 \cdot y + x = x \land x \geq 0))$

2. from 1:
   $([up_b(x); up_a(0)]\ \vartheta) \supset [up_b(x); up_a(0)]\ 0 \cdot y + x = x \land x \geq 0$

3. since θ is state-independent, from 2:
   $\vartheta \supset [up_b(x); up_a(0)]\ 0 \cdot y + x = x \land x \geq 0$

4. from 3:

   $[x, y \Leftarrow ct]\ (\vartheta \supset [up_b(x); up_a(0)]\ 0 \cdot y + x = x \wedge x \geq 0)$

5. from 4:

   $([x, y \Leftarrow ct]\ \vartheta) \supset [x, y \Leftarrow ct]\ [up_b(x); up_a(0)]\ 0 \cdot y + x = x \wedge x \geq 0$

6. from 5:

   $([x, y \Leftarrow ct]\ \vartheta) \supset [x, y \Leftarrow ct]\ [up_b(x); up_a(0)]\ [a, b \Leftarrow ct]\ \xi$

7. from 6:

   $([x, y \Leftarrow ct]\ \vartheta) \supset [let\ \ x \Leftarrow ct_x\ \ in\ \ up_b(x); up_a(0)]\ [x, y, a, b \Leftarrow ct]\ \xi$

8. since $[\uparrow]$ *absurd* is a tautology, obtain $\Phi(\uparrow) \equiv$

   $([x, y, a, b \Leftarrow ct]\ \xi) \supset [\uparrow]\ [x, y, a, b \Leftarrow ct]\ \xi \wedge b < y$

9. since $b < y \supset (\xi \supset (\xi \wedge b < y))$ is a tautology:

   $[up\,(x, y, a, b)]\ (b < y \supset (\xi \supset (\xi \wedge b < y)))$

10. from 9:

    $[up\,(x, y, a, b)]\ (b < y \supset (\xi \supset [skip]\ [x, y, a, b \Leftarrow ct]\ \xi \wedge b < y))$

11. from 10:

    $([up\,(x, y, a, b)]\ b < y) \supset [up\,(x, y, a, b)]\ (\xi \supset [skip]\ [x, y, a, b \Leftarrow ct]\ \xi \wedge b < y)$

12. since $b < y$ is state-independent, from 11 obtain $b < y \supset \Psi(skip) \equiv$

    $b < y \supset [up\,(x, y, a, b)]\ (\xi \supset [skip]\ [x, y, a, b \Leftarrow ct]\ \xi \wedge b < y)$

13. assume $\Phi(p) \equiv$

    $([x, y, a, b \Leftarrow ct]\ \xi) \supset [p]\ [x, y, a, b \Leftarrow ct]\ \xi \wedge b < y$

14. from 13:

    $[up_b(b - y); up_a(a + 1)]\ (([x, y, a, b \Leftarrow ct]\ \xi) \supset [p]\ [x, y, a, b \Leftarrow ct]\ \xi \wedge b < y)$

15. since $(b \geq y \wedge \xi(x, y, a, b)) \supset \xi(x, y, a + 1, b - y)$, by Lm_3:

    $(b \geq y \wedge \xi) \supset [up\,(x, y); up_b(b - y); up_a(a + 1)]\ [x, y, a, b \Leftarrow ct]\ \xi$

16. from 14 and 15:

$$
\begin{aligned}
&(b \geq y \wedge \xi) \supset \\
&\quad ([up\,(x,y); up_b(b-y); up_a(a+1)]\,(([x,y,a,b \Leftarrow ct]\ \xi) \supset \\
&\qquad [p]\ [x,y,a,b \Leftarrow ct]\ \xi \wedge b < y) \wedge \\
&\quad [up\,(x,y); up_b(b-y); up_a(a+1)]\ [x,y,a,b \Leftarrow ct]\ \xi)
\end{aligned}
$$

17. from 16:

$$(b \geq y \wedge \xi) \supset [up\,(x,y); up_b(b-y); up_a(a+1); p]\ [x,y,a,b \Leftarrow ct]\ \xi \wedge b < y$$

18. from 17:

$$
\begin{aligned}
&[up\,(x,y,a,b)] \\
&\quad (b \geq y \supset (\xi \supset [up_b(b-y); up_a(a+1); p]\ [x,y,a,b \Leftarrow ct]\ \xi \wedge b < y))
\end{aligned}
$$

19. since $b \geq y$ is state-independent, from 18:

$$
\begin{aligned}
&b \geq y \supset \\
&\quad [up\,(x,y,a,b)]\ (\xi \supset [up_b(b-y); up_a(a+1); p]\ [x,y,a,b \Leftarrow ct]\ \xi \wedge b < y)
\end{aligned}
$$

20. from 19 obtain $b \geq y \supset \Psi(let\ \ldots) \equiv$

$$
\begin{aligned}
&b \geq y \supset [up\,(x,y,a,b)]\ (\xi \supset \\
&\qquad\qquad [let\ \ b,y \Leftarrow ct\ \ in\ \ up_b(b-y); let\ \ a \Leftarrow ct_a\ \ in\ \ up_a(a+1); p] \\
&\qquad\qquad\quad [x,y,a,b \Leftarrow ct]\ \xi \wedge b < y)
\end{aligned}
$$

21. from 12 and 20, by COND obtain $\Psi(cond\,(\ldots)) \equiv$

$$
\begin{aligned}
&[up\,(x,y,a,b)]\ (\xi \supset \\
&\quad [cond\,(b \geq y, \\
&\qquad\qquad let\ \ b,y \Leftarrow ct\ \ in\ \ up_b(b-y); let\ \ a \Leftarrow ct_a\ \ in\ \ up_a(a+1)\ ;\ p, \\
&\qquad\qquad skip\,))]\ [x,y,a,b \Leftarrow ct]\ \xi \wedge b < y
\end{aligned}
$$

22. from 21, by CT_UP

$$
\begin{aligned}
&[x,y,a,b \Leftarrow ct]\ (\xi \supset \\
&\quad [cond\,(b \geq y, \\
&\qquad\qquad let\ \ b,y \Leftarrow ct\ \ in\ \ up_b(b-y); let\ \ a \Leftarrow ct_a\ \ in\ \ up_a(a+1)\ ;\ p, \\
&\qquad\qquad skip\,))]\ [x,y,a,b \Leftarrow ct]\ \xi \wedge b < y
\end{aligned}
$$

23. from 22:

$$([x,y,a,b \Leftarrow ct]\ \xi) \supset$$
$$[x,y,a,b \Leftarrow ct]$$
$$[\mathit{cond}\,(b \geq y,$$
$$\mathit{let}\ \ b,y \Leftarrow ct\ \ \mathit{in}\ \ up_b(b-y); \mathit{let}\ \ a \Leftarrow ct_a\ \ \mathit{in}\ \ up_a(a+1)\,;\,p,$$
$$\mathit{skip}\,))]\ [x,y,a,b \Leftarrow ct]\ \xi \wedge b < y$$

24. from 23 obtain $\Phi(M(p)) \equiv$

$$([x,y,a,b \Leftarrow ct]\ \xi) \supset$$
$$[\mathit{let}\ \ b,y \Leftarrow ct\ \ \mathit{in}$$
$$\mathit{cond}\,(b \geq y,$$
$$\mathit{let}\ \ b,y \Leftarrow ct\ \ \mathit{in}\ \ up_b(b-y); \mathit{let}\ \ a \Leftarrow ct_a\ \ \mathit{in}\ \ up_a(a+1)\,;\,p,$$
$$\mathit{skip}\,)]\ [x,y,a,b \Leftarrow ct]\ \xi \wedge b < y$$

25. by Ind, from 8 and 13. . . 24 obtain $\Phi(\mathit{rec}\,(M)) \equiv$

$$([x,y,a,b \Leftarrow ct]\ \xi) \supset$$
$$[\mathit{rec}\,(\lambda p : T1.\ \mathit{let}\ \ b,y \Leftarrow ct\ \ \mathit{in}$$
$$\mathit{cond}\,(b \geq y,$$
$$\mathit{let}\ \ b,y \Leftarrow ct\ \ \mathit{in}\ \ up_b(b-y); \mathit{let}\ \ a \Leftarrow ct_a\ \ \mathit{in}\ \ up_a(a+1)\,;\,p,$$
$$\mathit{skip}\,))]\ [x,y,a,b \Leftarrow ct]\ \xi \wedge b < y$$

26. from 7 and 25, by TR_Trans:

$$([x,y \Leftarrow ct]\ \vartheta) \supset \mathit{tr}\,((\mathit{let}\ \ x \Leftarrow ct_x\ \ \mathit{in}\ \ up_b(x)\,;\,up_a(0)\,,\,[x,y,a,b \Leftarrow ct]\ \xi)\,;$$
$$(\mathit{rec}(\ldots)\,,\,[x,y,a,b \Leftarrow ct]\ \xi \wedge b < y)).$$

$\square$

5.6 Inductive formulae

Scott-induction was used in section 5.5 to prove properties of while loops. Here we establish a sufficient syntactic condition on formulae $\phi(x)$ to ensure their admissibility for fixed-point induction on x. In view of theorem 5.6.1, this condition can be expressed as $x \notin \mathit{free}_{-}(\phi)$, where the set free_{-} is defined, together with its dual free_{+}, as follows:

$$
\begin{array}{rcll}
\mathit{free}_{-}(\mathit{absurd}) & = & \emptyset & \\
\mathit{free}_{-}(t_1 \leq t_2) & = & \emptyset & \text{and similarly for } \geq, >, <, \text{ and } = \\
\mathit{free}_{-}(\phi \wedge \psi) & = & \mathit{free}_{-}(\phi) \cup \mathit{free}_{-}(\psi) & \text{and similarly for } \vee \\
\mathit{free}_{-}(\phi \supset \psi) & = & \mathit{free}_{+}(\phi) \cup \mathit{free}_{-}(\psi) & \\
\mathit{free}_{-}(\forall x.\, \phi) & = & \mathit{free}_{-}(\phi) - \{x\} & \\
\mathit{free}_{-}(\exists x.\, \phi) & = & \mathit{free}(\phi) - \{x\} & \\
\mathit{free}_{-}([x \Leftarrow t]\ \phi) & = & \mathit{free}_{-}(\phi) & \text{if } x \notin \mathit{free}_{-}(\phi) \\
 & = & \mathit{free}_{-}(\phi) \cup \mathit{free}(t) - \{x\} & \text{otherwise}
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{free}_{+}(\mathit{absurd}) & = & \emptyset & \\
\mathit{free}_{+}(t_1 \leq t_2) & = & \emptyset & \text{and similarly for } \geq, >, <, \text{ and } = \\
\mathit{free}_{+}(\phi \wedge \psi) & = & \mathit{free}_{+}(\phi) \cup \mathit{free}_{+}(\psi) & \text{and similarly for } \vee \\
\mathit{free}_{+}(\phi \supset \psi) & = & \mathit{free}_{-}(\phi) \cup \mathit{free}_{+}(\psi) & \\
\mathit{free}_{+}(\forall x.\, \phi) & = & \mathit{free}(\phi) - \{x\} & \\
\mathit{free}_{+}(\exists x.\, \phi) & = & \mathit{free}_{+}(\phi) - \{x\} & \\
\mathit{free}_{+}([x \Leftarrow t]\ \phi) & = & \mathit{free}(\phi) \cup \mathit{free}(t) - \{x\} &
\end{array}
$$

In the following theorem we take cpos as our domains. However, the proof also goes through in the context of models of synthetic domain theory. e.g. with extensional PERs, adopting the appropriate notion of ω-chain.

Theorem 5.6.1 *Let $\Gamma, x : X \vdash \phi$ be a formula in EL.*

A. If $x \notin free_{-}(\phi)$, then, for all $u \in [\![\Gamma]\!]$ and ω-chains $\langle d_n \rangle$, if $[\![\phi]\!](u, d_i)$ for all d_i in $\langle d_n \rangle$, then $[\![\phi]\!](u, \sqcup\langle d_n \rangle)$.

B. If $x \notin free_{+}(\phi)$, then, for all $u \in [\![\Gamma]\!]$ and ω-chains $\langle d_n \rangle$, if $[\![\phi]\!](u, \sqcup\langle d_n \rangle)$, then $[\![\phi]\!](u, d_i)$ for some d_i in $\langle d_n \rangle$.

Proof. We just discuss the interesting cases: implication and necessity. Assume $x \notin free_{-}(\phi \supset \psi)$, that is, $x \notin free_{+}(\phi)$ and $x \notin free_{-}(\psi)$. Let $\langle d_n \rangle$ be an ω-chain such that $[\![\phi \supset \psi]\!](d_i)$ for all $d_i \in \langle d_n \rangle$. Assume $[\![\phi]\!](\sqcup\langle d_n \rangle)$. Since $x \notin free_{+}(\phi)$, it must be $[\![\phi]\!](d_i)$ for infinitely many d_i in $\langle d_n \rangle$, otherwise, pruning such elements from $\langle d_n \rangle$, one would get a subchain with the same l.u.b. and with no elements satisfying $[\![\phi]\!]$. So, there is a subchain $\langle c_n \rangle$ of $\langle d_n \rangle$ all elements of which satisfy $[\![\phi]\!]$. Then, by the assumption, $[\![\psi]\!](d_i)$ for all d_i in $\langle c_n \rangle$, and hence, since $x \notin free_{-}(\psi)$, $\sqcup\langle c_n \rangle = \sqcup\langle d_n \rangle$ satisfy $[\![\psi]\!]$. This shows that $[\![\phi \supset \psi]\!](\sqcup\langle d_n \rangle)$.

The definition of $free_{-}([x \Leftarrow z]\ \phi)$ says that, if ϕ is ω-inductive in x, then $[x \Leftarrow z]\ \phi$ is ω-inductive in z; also, if $\psi(x, y)$ is ω-inductive in x, then $[y \Leftarrow z]\ \psi(x, y)$ is ω-inductive in x. We show the first of these statements. Assume $\forall s.\, \phi(z_i(s))$ holds for all z_i in the ω-chain $\langle z_n \rangle$, show that $\forall s.\, \phi(\sqcup\langle z_n \rangle s)$ holds. For an arbitrary $\bar{s}$, all elements of $\langle z_n(\bar{s}) \rangle$ satisfy ϕ by hypothesis; then, since ϕ is ω-inductive in x, $\phi(\sqcup\langle z_n(\bar{s}) \rangle) = \phi(\sqcup\langle z_n \rangle \bar{s})$ must hold. $\square$

Remark. Theorem 5.6.1 improves a similar one in [Ten91]. In particular, it allows formulae such as $\neg(\neg(x \sqsubseteq 7))$, rejected in [Ten91], to be recognized as admissible.

Remark. In the above proof we used the interpretation of necessity as in EL, rather than as in EL*se*. This is in fact the worse case, since it involves a quantification over states and, hence, $x \notin free_{+}(\phi)$ implies neither $x \notin free_{+}([y \Leftarrow z]\ \phi)$ nor $z \notin free_{+}([y \Leftarrow z]\ \phi)$. However, such quantification disappears in EL*se* and hence the admissibility conditions for fixed-point induction are more liberal for assertions than for formulae.

Remark. In view of the last remark, we conclude that theorem 5.6.1 validates the use of Scott induction in section 5.5, since $p \notin \mathit{free}_{\_}(\Phi(p))$.

6 Semantic constructors

In this chapter we present and develop the theory of semantic constructors (see chapter 1), which were introduced by E. Moggi in [Mog90a,Mog91c] with the intention of achieving a modular approach to the study and development of denotational semantics.

To what extent can different notions of computation be studied in isolation? Which properties do models produced by a constructor retain of the models from which they are constructed and upon which properties are they conservative? We address these questions in the first two sections of this chapter. In the rest, we discuss the *syntactic* presentation of constructors, a technique introduced in [CM93], and relate the power of constructors with that of the metalanguages that are used to present them syntactically.

A direct method for presenting a constructor is to produce structure for interpreting one metalanguage from structure for interpreting another. In the next chapter, we develop applications where constructors for exceptions and resumptions are presented in this fashion by using the Extended Calculus of Constructions as metalanguage.

Here, we focus on an indirect method, where constructors are presented in a purely syntactical fashion as translations of metalanguages. In general, a translation $\mathcal{L}_2 \to \mathcal{L}_1$ of formal languages yields, contravariantly, a *relative interpretation* $Mod(\mathcal{L}_1) \to Mod(\mathcal{L}_2)$ mapping models of one language into models of the other [KR77]. This situation extends to logical theories, where it is understood that theorems of one theory are translated into theorems of the other. For example, given a strong monad in a category $\mathcal{C}$, a translation $ML_T \to ML_T$ gives a recipe to construct a new strong monad on $\mathcal{C}$.

For some logics, a dual construction is also available. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be theories expressed in *universal Horn clauses* ([Kea75]). Such theories can be viewed as categories

$th(\mathcal{L}_i)$ with finite limits (lex categories) in the same way as categories with finite products provide views of algebraic theories where no distinction is made between primitive and derived operations. In fact, this is the basic idea of categorical logic:

> "a logical theory has an intrinsic existence independent of its presentation, and this existence is best represented by a category" ([See82]).

In this framework, one speaks of "functorial semantics" because interpretation of a theory $\mathcal{L}$ in a category $\mathcal{C}$ with appropriate structure is viewed as a structure-preserving functor $th(\mathcal{L}) \to \mathcal{C}$. A well known result of P. Gabriel and F. Ulmer ([GU75]) states that any filtered colimit-preserving functor $Mod(\mathcal{L}_1) \to Mod(\mathcal{L}_2)$ with a left adjoint arises as relative interpretation from a lex functor $th(\mathcal{L}_2) \to th(\mathcal{L}_1)$ and hence from a translation $\mathcal{L}_2 \to \mathcal{L}_1$. The natural question is whether a similar duality can be established for theories of the computational metalanguage and whether a sufficiently rich class of constructors $Mod(\Sigma_1) \to Mod(\Sigma_2)$ can be obtained from translations $ML_T(\Sigma_2) \to ML_T(\Sigma_1)$.

In the setting of Gabriel-Ulmer duality, translations and interpretations live in the same category (of lex categories) and relative interpretation is studied as composition. To mimic this situation, we need a presentation-independent view of a theory $ML_T(\Sigma)$ as an object of $Mod(\Sigma)$. A natural candidate would be the syntactic category $\mathcal{T}(\Sigma)$ described in section 2.4. This category has a canonical strong monad and canonical Σ structure. Moreover, interpretation of $ML_T(\Sigma)$ in a Σ-model $(\mathcal{C}, T, \mathcal{A})$ corresponds to a morphism $(\llbracket\_\rrbracket, id) = \mathcal{T}(\Sigma) \to (\mathcal{C}, T, \mathcal{A})$ in $Mod(\Sigma)$. However, interpretations and structure preserving functors do not coincide: since interpretation requires $\llbracket T\tau\rrbracket = T\llbracket\tau\rrbracket$, morphism $(\llbracket\_\rrbracket, \sigma) : \mathcal{T}(\Sigma) \to (\mathcal{C}, T, \mathcal{A})$ correspond to interpretations only when $\sigma = id$.

This suggests that, in order for translations to capture interesting constructors, a more powerful syntax must be used than that of the computational metalanguage. For example, the constructor for resumptions $(\mathcal{F}T)X \stackrel{\mathrm{def}}{=} \mu Y.T(X+Y)$ that we present in the next chapter needs inductive types, which cannot be axiomatized in $ML_T(\Sigma)$. The examples that we present in this chapter are written in HML, an expressive type theory, similar to the one in [Mog91a], in which the computational metalanguages can be easily axiomatized. We give an intrinsic characterization of the constructors that can be presented in HML. This characterization generalizes to presentations in any formal language whose models are described by finite limit theories; examples are the simply typed and the higher order polymorphic lambda calculi.

Synopsis. In section 6.1 we define the category $Mod(\Sigma)$ of models of $ML_T(\Sigma)$ introducing the notion of Σ-homomorphism (not in Moggi's presentation). In 6.2 we introduce semantic constructors and we investigate the properties of a particular class of them, called "pointed." Examples of such constructors are the ones for exceptions, resumptions, interactive input and output. In particular, we determine a class of equations *preserved* by pointed constructors (theorem 6.2.5) and one whose equations are *reflected* (that is, a sublanguage of $ML_T(\Sigma)$ over which pointed constructors are conservative; theorem 6.2.7). Preservation of equations is also addressed in [Mog90b]. Our treatment differs from Moggi's in that the latter considers a more restricted class of constructors and puts more constraints on the metalanguage (viz. lambda abstraction only on constant types). In section 6.3 we present HML and, in 6.4, we use it to present a rather powerful constructor that can be specialized to capture models of several forms of computation. Most of the material presented 6.3 and 6.4 is from [CM93] and we claim no originality there. After introducing the basic ideas of functorial semantics in section 6.5, we investigate the properties of categories of models of HML theories in 6.6 and 6.7. The class of syntactic constructors corresponding to HML translations is characterized in 6.8.

6.1 Categories of Σ-models

In section 2.4 we defined a suitable notion of signature for a typed lambda calculus $\mathcal{L}(\Sigma)$ and a notion of Σ-model, that is, a category in which the relevant structure is singled out. Some of the structure may be defined by universal properties and we called "additional" all the rest. It is reasonable to expect that models form a 2-category $Mod(\Sigma)$ whose

1-cells are functors preserving the universal structure, while additional requirements (see example below) may be appropriate to relate the additional structure.

Example. Strong monad morphisms $(\mathcal{C}, T) \to (\mathcal{D}, S)$ were defined in section 2.2 as pairs $(U : \mathcal{C} \to \mathcal{D}, \sigma : UT \dot{\to} SU)$ where the functor U preserves the universal structure of $\mathcal{C}$ in $\mathcal{D}$, while the natural transformation σ relates the monads T and S with suitable diagrams.

$\square$

Here we describe categories of models of computational metalanguages $ML_T(\Sigma)$, whose objects we shall henceforth call Σ-models and whose morphisms are strong monad morphisms with extra conditions relating Σ-structures. In the next section the *semantic constructors* of the modular approach are introduced as *natural* maps between categories of models.

Defining morphisms between Σ-models is not straightforward because the usual mixed variance problem gets in the way of relating the Σ structure of the two models. In fact we must restrict signatures as shown below. (The notions of *type scheme*, *polytype* etc. were defined in section 2.4.)

A polytype is called *computationally positive* (*negative*) when it contains no negative (positive) occurrences of the type constructor T (we shall usually drop the word "computationally". More formally: 1, $K \in \Sigma_\tau$ and $X \in \chi$ are both positive and negative polytypes; if τ_1 and τ_2 are both positive (negative), so is $\tau_1 \times \tau_2$; if τ_1 is negative (positive) and τ_2 is positive (negative), $\tau_1 \to \tau_2$ is positive (negative); if τ is positive, so is $T\tau$. For example, $(T(K \to TX) \to X) \to TX$ is positive, while $TX \to TX$ is neither positive nor negative. It is easy to verify that a polytype is both positive and negative if and only if it contains no occurrences of T.

Positive (negative) polytypes are not closed under "indiscriminate" substitution. As in section 6.2 we need a closure result, we characterize well behaved substitution in Lemma 6.1.1.

A type variable is said to “occur positively (negatively)” in a polytype according to the following rules: X occurs neither positively nor negatively in τ if it does not occur in τ; X occurs positively in X; X occurs positively (negatively) in $\tau_1 \times \tau_2$ if it occurs positively (negatively) in τ_1 or in τ_2 or in both; X occurs positively in $\tau_1 \to \tau_2$ if it occurs positively in τ_2 or negatively in τ_2 or both; X occurs negatively in $\tau_1 \to \tau_2$ if it occurs negatively in τ_2 or positively in τ_2 or both; X occurs positively (negatively) in $T\tau$ if it occurs positively (negatively) in τ.

Let σ and τ be polytypes. We say that the substitution $[\sigma/X]\tau$ is *sign-preserving* when σ is positive if X occurs positively in τ and σ is negative if X occurs negatively (hence σ must be both when X occurs both ways). We call *sign-reversing* the obvious converse.

Lemma 6.1.1 *Sign-preserving substitution preserves positive polytypes; sign-reversing substitution preserves negative polytypes.*

Proof. Let α, β and γ range over the symbols $+$ and $-$, to be read as “positive” and “negative,” and let $\bar{\alpha}$ be the complement of α. We write $(\alpha\beta\gamma)$ for the statement “if τ is α, X occurs β-ly in τ and σ is γ, then $[\sigma/X]\tau$ is α.” We show the cases $(+++)$, $(--+)$, $(+--)$ and $(-+-)$ by induction on τ.

For $\tau = 1$ and $\tau = K \in \Sigma_\tau$ all four cases hold trivially. For $\tau = X$, $(+++)$ holds because $[\sigma/X]X$ is positive for positive σ, while the other cases hold trivially because their hypotheses are not satisfied. For $\tau = \tau_1 \times \tau_2$, each statement holds because, using the inductive hypothesis, the same statement must hold for τ_1 and τ_2. For $\tau = \tau_1 \to \tau_2$, $(\alpha\beta\gamma)$ holds because, using the inductive hypothesis, $(\alpha\beta\gamma)$ must hold for τ_2 and $(\bar{\alpha}\bar{\beta}\gamma)$ must hold for τ_1. For $\tau = T\tau_1$, $(+++)$ and $(+--)$ hold because, by inductive hypothesis, identical statements must hold for τ_1, while the other cases hold trivially because their hypotheses are not satisfied. □

Let Σ_τ be a collection of constant type symbols. Let $\mathcal{C}$ and $\mathcal{D}$ be cartesian closed categories and let both have a Σ_τ-structure and an endofunctor, respectively $T : \mathcal{C} \to \mathcal{C}$

and $S : \mathcal{D} \to \mathcal{D}$. Σ_τ-polytypes are interpreted in $\mathcal{C}$ and $\mathcal{D}$ from the above data. Let $U : \mathcal{C} \to \mathcal{D}$ be a functor preserving the universal and Σ_τ structure. We make the simplifying assumption that U does so *on the nose*, although this is not crucial for the development of the theory. In particular, this means that $U[\![K]\!] = [\![K]\!]$ for all K in Σ_τ (all type constants but T are nullary in $ML_T(\Sigma)$). Given a natural transformation $\sigma : UT \dot{\to} SU$, we associate with every computationally positive (negative) polytype $\tau(X_1, \ldots X_n)$ and type assignments $A_1, \ldots A_n$ in $\mathcal{C}$ a morphism

$$\sigma^+(\tau) : U[\![\tau]\!]_{A_1,\ldots A_n} \to [\![\tau]\!]_{UA_1,\ldots UA_n}$$

(σ^-, with opposite direction, for negative τ) as follows: writing $U\tau$ for $U[\![\tau]\!]_{A_1,\ldots A_n}$, τU for $[\![\tau]\!]_{UA_1,\ldots UA_n}$, σ_τ for $\sigma_{[\![\tau]\!]_{A_1,\ldots A_n}}$ and $(\_)^{(\_)} : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}$ for the functor mapping (A, B) to B^A,

$$\begin{array}{rcl}
\sigma^{\pm}1 & = & id_1 \\
\sigma^{\pm}X_i & = & id_{UA_i} \\
\sigma^{\pm}K & = & id_{[\![K]\!]} \\
\sigma^{\pm}(\tau_1 \times \tau_2) & = & \sigma^{\pm}(\tau_1) \times \sigma^{\pm}(\tau_2) \\
\sigma^{+}(\tau_1 \to \tau_2) & = & \sigma^{+}(\tau_2)^{\sigma^-(\tau_1)} = \lambda f : U\tau_1 \to U\tau_2.\, \sigma^+(\tau_2) \circ f \circ \sigma^-(\tau_1) \\
\sigma^{-}(\tau_1 \to \tau_2) & = & \sigma^{-}(\tau_2)^{\sigma^+(\tau_1)} = \lambda f : \tau_1 U \to \tau_2 U.\, \sigma^-(\tau_2) \circ f \circ \sigma^+(\tau_1) \\
\sigma^{+}(T\tau) & = & S(\sigma^+(\tau)) \circ \sigma_\tau
\end{array}$$

Proposition 6.1.2 *Let U and σ be as above. If a polytype τ is both positive and negative, $U[\![\tau]\!] = [\![\tau]\!]U$ and $\sigma^+(\tau) = \sigma^-(\tau) = id$.*

Proof. By induction on τ, For example: if $\tau_1 \to \tau_2$ is both positive and negative, so must be, by a simple argument, also τ_1 and τ_2; hence $\sigma^+(\tau_1 \to \tau_2) = \sigma^-(\tau_1 \to \tau_2) = id^{id} = id$. □

Σ-structures can be related in a simple diagrammatical way if they are restricted to positive type schemes, that is, type schemes involving only positive polytypes. Since this is the case in most examples of operations associated with notions of computation, we will henceforth restrict our signatures to include only operations with positive schemes. At the end of this section we discuss some nonexamples.

Definition 6.1.3 *A* Σ-homomorphism $(\mathcal{C}, T, \mathcal{A}) \to (\mathcal{D}, S, \mathcal{B})$ *between* Σ*-models is a strong monad morphism* $(U, \sigma) : (\mathcal{C}, T) \to (\mathcal{D}, S)$ *such that* U *preserves the* Σ_τ *structure on the nose and, for all* $op : \forall X_1, \ldots X_n.\, \tau_1, \ldots \tau_m \longrightarrow \tau$ *in* Σ_ϵ *and type assignments* $A_1, \ldots A_n$*, the following diagram commutes:*

$$\begin{tikzcd}
U[\![\tau_1]\!]_{A_1,\ldots A_n} \times \ldots U[\![\tau_m]\!]_{A_1,\ldots A_n} \arrow[r, "U[\![op]\!]_{A_1,\ldots A_n}"] \arrow[d, "\sigma^+(\tau_1) \times \ldots \sigma^+(\tau_m)"] & U[\![\tau]\!]_{A_1,\ldots A_n} \arrow[d, "\sigma^+(\tau)"] \\
[\![\tau_1]\!]_{UA_1,\ldots UA_n} \times \ldots [\![\tau_m]\!]_{UA_1,\ldots UA_n} \arrow[r, "[\![op]\!]_{UA_1,\ldots UA_n}"'] & [\![\tau]\!]_{UA_1,\ldots UA_n}
\end{tikzcd} \tag{6.1}$$

If op is an operation for which the above diagram commutes, we say that (U, σ) is natural with respect to op.

Example. Let $(U, \sigma) : (\mathcal{C}, T) \to (\mathcal{D}, S)$ be a morphism of strong endofunctors and let operations $val : \forall X.\, X \longrightarrow TX$ and $let : \forall X, Y.\, (X \to TY), TX \longrightarrow TY$, satisfying the equations of section 2.2, be defined in $\mathcal{C}$ and $\mathcal{D}$. The above diagram applied to such operations yields equations defining (U, σ) as a strong monad morphism. Conversely, let (U, σ) be a morphism between strong monads over $\mathcal{C}$ and $\mathcal{D}$. The operations $st : \forall X, Y.\, (X \to Y), TX \longrightarrow TY$ derived as in section 2.2 in $\mathcal{C}$ and $\mathcal{D}$ make the above diagram commute: $\sigma_B(U st_{A,B}(f, w)) = st_{UA,UB}(f, \sigma_A w)$, and this corresponds to saying that (U, σ) is a morphism of strong endofunctors.

Proposition 6.1.4 Σ*-models and* Σ*-homomorphisms form a category* $Mod(\Sigma)$*.*

Proof. (Strong) monad morphisms $(U, \sigma) : (\mathcal{C}, T) \to (\mathcal{D}, S)$ and $(V, \rho) : (\mathcal{D}, S) \to (\mathcal{E}, R)$ compose as follows: $(V, \rho) \circ (U, \sigma) = (V \circ U, \rho U \circ V\sigma)$. We show that $(V, \rho) \circ (U, \sigma)$ is a

Σ_ϵ-homomorphism by proving the following equations:

$$(\rho U \circ V\sigma)^{+}\tau = \rho^{+}(\tau)U \circ V\sigma^{+}(\tau) \tag{6.2}$$

$$(\rho U \circ V\sigma)^{-}\tau = V\sigma^{-}(\tau) \circ \rho^{-}(\tau)U \tag{6.3}$$

Then, the result follows from the commutativity of the diagram below:

$$\begin{tikzcd}
VU[\![\tau_1]\!]_{A_1,\ldots A_n} \times \ldots VU[\![\tau_m]\!]_{A_1,\ldots A_n} \arrow[r, "VU[\![op]\!]_{A_1,\ldots A_n}"] \arrow[d, "V\sigma^{+}(\tau_1) \times \ldots V\sigma^{+}(\tau_m)"] & VU[\![\tau]\!]_{A_1,\ldots A_n} \arrow[d, "V\sigma^{+}(\tau)"] \\
V[\![\tau_1]\!]_{UA_1,\ldots UA_n} \times \ldots V[\![\tau_m]\!]_{UA_1,\ldots UA_n} \arrow[r, "V[\![op]\!]_{UA_1,\ldots UA_n}"] \arrow[d, "\rho^{+}(\tau_1)U \times \ldots \rho^{+}(\tau_m)U"] & V[\![\tau]\!]_{UA_1,\ldots UA_n} \arrow[d, "\rho^{+}(\tau)U"] \\
[\![\tau_1]\!]_{VUA_1,\ldots VUA_n} \times \ldots [\![\tau_m]\!]_{VUA_1,\ldots VUA_n} \arrow[r, "[\![op]\!]_{VUA_1,\ldots VUA_n}"] & [\![\tau]\!]_{VUA_1,\ldots VUA_n}
\end{tikzcd}$$

Equations 6.2 and 6.3 are proven by induction on τ, of which we show only a few cases as the others are similar. For a positive $T\tau$:

$$\begin{array}{ll}
(\rho U \circ V\sigma)^{+}(T\tau) = & \text{(by definition of } (_)^{+}(T\tau)) \\
R(\rho U \circ V\sigma)^{+}(\tau) \circ (\rho U \circ V\sigma)_\tau = & \text{(by inductive hypothesis)} \\
R(\rho^{+}(\tau)U \circ V\sigma^{+}(\tau)) \circ \rho_{U\tau} \circ V\sigma_\tau = & \text{(since } R \text{ is a functor)} \\
R\rho^{+}(\tau)U \circ RV\sigma^{+}(\tau) \circ \rho_{U\tau} \circ V\sigma_\tau = & \text{(since } \rho \text{ is a natural transformation)} \\
R\rho^{+}(\tau)U \circ \rho_{\tau U} \circ VS\sigma^{+}(\tau) \circ V\sigma_\tau = & \text{(since } V \text{ is a functor)} \\
(R\rho^{+}(\tau)U \circ \rho_{\tau U}) \circ V(S\sigma^{+}(\tau) \circ \sigma_\tau) = & \text{(by definition of } (_)^{+}(T\tau)) \\
\rho^{+}(T\tau)U \circ V\sigma^{+}(T\tau). &
\end{array}$$

Similarly, for positive $\tau_1 \to \tau_2$:

$$
\begin{aligned}
&(\rho U \circ V\sigma)^{+}(\tau_1 \to \tau_2) = \\
&\lambda f : VU\tau_1 \to VU\tau_2.\, (\rho U \circ V\sigma)^{+}(\tau_2) \circ f \circ (\rho U \circ V\sigma)^{-}(\tau_1) = \\
&\lambda f : VU\tau_1 \to VU\tau_2.\, \rho^{+}(\tau_2)U \circ V\sigma^{+}(\tau_2) \circ f \circ V\sigma^{-}(\tau_1) \circ \rho^{-}(\tau_1)U = \\
&(\lambda g : V\tau_1 U \to V\tau_2 U.\, \rho^{+}(\tau_2)U \circ g \circ \rho^{-}(\tau_1)U) \circ \\
&\qquad (\lambda f : VU\tau_1 \to VU\tau_2.\, V\sigma^{+}(\tau_2) \circ f \circ V\sigma^{-}(\tau_1)) = \\
&(\lambda g : V\tau_1 U \to V\tau_2 U.\, \rho^{+}(\tau_2)U \circ g \circ \rho^{-}(\tau_1)U) \circ V(\lambda f : U\tau_1 \to U\tau_2.\, \sigma^{+}(\tau_2) \circ f \circ \sigma^{-}(\tau_1)) = \\
&\rho^{+}(\tau_1 \to \tau_2)U \circ V\sigma^{+}(\tau_1 \to \tau_2)
\end{aligned}
$$

□

As to 2-cells in $Mod(\Sigma)$, no natural condition can be stated for an arbitrary $\nu : U \dot{\to} V$ to coherently relate U and V occurrences of operations in Σ. However, coherence is guaranteed when 2-cells are restricted to isomorphisms. [Pow94] studies (2- and tri-) categories with structure arising from modelling logical frameworks for computer science.

Remark. Some operations commonly associated with computational features of programming languages do not have positive type scheme. One example is the operator $callcc_{A,B} : ((A \to TB) \to TA) \to TA$ for computations with continuations; another example, introduced in section 6.4, is $C_{A,B} : (A \to TB) \times (HTA \to TB) \times TA \to TB$, which performs case analysis on generalized resumptions.

When a signature Σ contains such operators, the general notion of Σ-homomorphism introduced above cannot be adopted. In that case, one can always reduce the amount of structure upon which morphisms in $Mod(\Sigma)$ are required to be natural. For example, if naturality were only required on *val* and *let*, a morphism in $Mod(\Sigma)$ would be just any strong monad morphism. However, as we shall see in the next section (theorems 6.2.5 and 6.2.7), the less we require of morphisms in $Mod(\Sigma)$, the less we can say of semantic constructors in terms of preserving equations.

6.2 Semantic constructors

A *semantic constructor* $\mathcal{F}$ is a family of functions $\mathcal{F}_\Sigma$ mapping the objects of $Mod(\Sigma)$ to the objects of $Mod(\Sigma + \Sigma_\mathcal{F})$. When clear from the context, we shall understand that such maps act only on objects and write:

$$\mathcal{F}_\Sigma : Mod(\Sigma) \to Mod(\Sigma + \Sigma_\mathcal{F}).$$

$\mathcal{F}_\Sigma$ is indexed by signatures of the metalanguage; each $\mathcal{F}$ is defined over a restricted range $Sig_\mathcal{F}$ of signatures. For example, *uniform redefinitions*, to be introduced below, can be defined only over operations with certain type schemes. Besides cutting off "unmanageable" signatures, $Sig_\mathcal{F}$ may also require a minimal supply of constant types and operations. This minimal supply may be thought of as the *parameters* of the constructor. An example of parameter is the type E of exceptions in the translation presented in section 7.2.

Example. Consider a constructor $\mathcal{F}$ mapping a monoid M into the group obtained by freely adjoining a unary operation "-" to M and quotienting by suitable equations: $a + (-a) = 0$, $-(a + b) = (-a) + (-b)$, etc. This constructor is defined on any category of Σ-algebras, provided Σ includes $+$.

□

In Moggi's definition [Mog91c], semantic constructors must satisfy a *naturality* condition that we shall discuss below. To state this condition a notion of *signature morphism* is required, which we provide in definition 6.2.1 for the general form of lambda calculus $\mathcal{L}(\Sigma)$ described in chapter 2.

Let $\rho_i(Y_1, \dots Y_l)$, $i = 1 \dots k$, $\sigma_j(Y_1, \dots Y_l)$, $j = 1 \dots n$, and $\tau(X_1, \dots X_n)$ be Σ-polytypes; a *derived* Σ-*operation* of arity $\forall Y_1, \dots Y_l.\, \rho_1, \dots \rho_k \longrightarrow \tau(\sigma_1, \dots \sigma_n)$ is either an operation in Σ (of same arity) or a sequence of $m+1$ derived Σ-operations op_i of arity

$\forall Y_1, \ldots Y_l.\, \rho_1, \ldots \rho_k \longrightarrow \tau_i(\sigma_1, \ldots \sigma_n)$, $i = 1 \ldots m$, and $op : \forall X_1, \ldots X_n.\, \tau_1, \ldots \tau_m \longrightarrow \tau$, which we write $op\,(op_1, \ldots op_m)$.

Definition 6.2.1 *A* signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ *maps constant type symbols K of arity n in Σ_1 to Σ_2-polytypes $\tau(X_1, \ldots X_n)$, and constant operation symbols op of arity $\forall X_1, \ldots X_n.\, \tau_1, \ldots \tau_m \longrightarrow \tau$ in Σ_1 to derived Σ_2-operations $\sigma(op)$ of arity $\forall X_1, \ldots X_n.\, \sigma(\tau_1), \ldots \sigma(\tau_m) \longrightarrow \sigma(\tau)$.*

Remark. For computational metalanguages $ML_T(\Sigma)$ signature morphisms only deal with constants in Σ, that is, they do not affect T, *val* or *let*. Other possibilities may be considered, e.g. mapping symbols to terms. We do not further discuss such notions since signature morphisms play a marginal role in the rest of this thesis. However, note that, since the naturality condition on constructors in definition 6.2.2 quantifies over such morphisms, the more general the notion, the harder it is for $\mathcal{F}_\Sigma$ to qualify as natural.

□

A signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ induces Σ_1 structure in the underlying category of any Σ_2-model $\mathcal{M}$: the structure for interpreting a symbol $\xi \in \Sigma_1$ is obtained by interpreting $\sigma(\xi)$ in $\mathcal{M}$. If this Σ_1-structure satisfies the axioms of $ML_T(\Sigma_1)$, we write $Mod\,(\sigma) : Mod\,(\Sigma_2) \to Mod\,(\Sigma_1)$.

Definition 6.2.2 (Moggi) *A* semantic constructor $\mathcal{F}$ *is a $Sig_{\mathcal{F}}$-indexed family of functions $\mathcal{F}_\Sigma$ from the objects of $Mod\,(\Sigma)$ to the objects of $Mod\,(\Sigma + \Sigma_{\mathcal{F}})$ satisfying the following naturality condition: for all signature morphisms $\sigma : \Sigma_1 \to \Sigma_2$ in $Sig_{\mathcal{F}}$ such that $Mod\,(\sigma) : Mod\,(\Sigma_2) \to Mod\,(\Sigma_1)$, the following diagram commutes:*

$$\begin{tikzcd}
Mod\,(\Sigma_2) \arrow[r, "\mathcal{F}_{\Sigma_2}"] \arrow[d, "Mod\,(\sigma)"'] & Mod\,(\Sigma_2 + \Sigma_{\mathcal{F}}) \arrow[d, "Mod\,(\sigma + \Sigma_{\mathcal{F}})"] \\
Mod\,(\Sigma_1) \arrow[r, "\mathcal{F}_{\Sigma_1}"'] & Mod\,(\Sigma_1 + \Sigma_{\mathcal{F}})
\end{tikzcd}$$

Naturality ensures that all components of $\mathcal{F}$ agree on the definition of $\Sigma_{\mathcal{F}}$ structure by restricting to the same $\mathcal{F}_{\Sigma_0} : Mod(\Sigma_0) \to Mod(\Sigma_0 + \Sigma_{\mathcal{F}})$, where Σ_0 is the signature of parameters, and that reinterpretation of any $\Sigma \in Sig_{\mathcal{F}}$ can be described by considering each operation separately.

It is reasonable to expect that theories of $(\Sigma + \Sigma_{\mathcal{F}})$-computation extend theories of Σ-computation. Hence, in order to show that the construction of a categorical structure $\mathcal{FM}$ out of a Σ-model $\mathcal{M}$ actually defines a map $Mod(\Sigma) \to Mod(\Sigma + \Sigma_{\mathcal{F}})$, it is useful to know whether $\mathcal{F}$ preserves the truth of certain formulae. In chapter 5, for example, we showed, in the context of Evaluation Logic, that the constructor $(\mathcal{F}T)X \stackrel{\text{def}}{=} T(X + S)^S$ preserves the validity of $\Box\_val^*$.

Example. Any $\mathcal{F}$ mapping Σ-algebras into $(\Sigma + \Sigma_{\mathcal{F}})$-algebras endowed with a family of Σ-homomorphisms $f_{\mathcal{A}} : \mathcal{A} \to \mathcal{FA}$, preserves the truth of closed equations over Σ. Similarly, let $\mathcal{F} : Mod(\Sigma) \to Mod(\Sigma + \Sigma_{\mathcal{F}})$; a morphism $(U, \sigma) : \mathcal{M} \to \mathcal{FM}\rceil_{\Sigma}$ in $Mod(\Sigma)$ relates interpretations of (positively typed) operations $op \in \Sigma_\epsilon$ in the two models. Let $\Gamma \vdash \omega : \tau$, with positive Γ and τ, be a derived operation obtained by composing the primitive operations in Σ_ϵ, *val* and *let*; it is easy to see that $[\![\omega]\!]$ makes diagram 6.1 commute. Any equation $\omega_1 = \omega_2$ between *closed* terms of the above kind which holds in $\mathcal{M}$ also holds in $\mathcal{FM}$, since $[\![\omega_1]\!]^{\mathcal{FM}} = \sigma^+ U[\![\omega_1]\!]^{\mathcal{M}} = \sigma^+ U[\![\omega_2]\!]^{\mathcal{M}} = [\![\omega_2]\!]^{\mathcal{FM}}$.

$\square$

In the rest of this section we investigate the properties of semantic constructors $\mathcal{F}$, which we call "pointed," endowed with a family of morphisms $(U, \sigma) : \mathcal{M} \to \mathcal{FM}$, where $\mathcal{M}$ and $\mathcal{FM}$ have the same underlying category and U is the identity. Note that theorem 6.2.4 holds for any (U, σ) in $Mod(\Sigma)$. We shall ask two questions about pointed constructors: which equations do they *preserve* and which do they *reflect*? The first of these questions was first addressed in [Mog90b], where a solution is sought for a special class of pointed constructors (called *parametric extensions*) and for a fragment of the metalanguage in which lambda abstraction is only allowed on constant types.

We call "light" the terms of $ML_T(\Sigma)$ freely generated by rules B1, B2, B4, B5 and B6 of section 2.3, together with the following:

B3.bis. Let $op : \forall X_1, \ldots X_n.\, \tau_1, \ldots \tau_m \longrightarrow \tau$ be an operation in Σ_ϵ, let $\sigma_1, \ldots \sigma_n$ be types such that σ_i is positive (negative) if X_i occurs positively (negatively) in τ, and let $M_j : \tau_i(\sigma_1, \ldots \sigma_n)$ for $j = 1, \ldots m$, then $op_{\sigma_1,\ldots\sigma_n}(M_1, \ldots M_m)$ is a term of type $\tau(\sigma_1, \ldots \sigma_n)$;

B7.bis. if $x \in \chi_\sigma$, T does not occur in σ and $M : \tau$, then $\lambda x : \sigma.\, M$ is a term of type $\sigma \to \tau$.

Lemma 6.2.3 *Light terms in positive contexts have positive type.*

Proof. By induction on the structure of terms. The term $*$ has positive type 1. For pairs and projections the result follows immediately from the inductive hypotheses. A light term $op(M_1, \ldots M_m)$, where $M_j : \tau_j(\sigma_1, \ldots \sigma_n)$ and op has positive type scheme $\forall X_1, \ldots X_n.\, \tau_1, \ldots \tau_m \longrightarrow \tau$, must satisfy the conditions of B3.bis, which yield a positive $\tau(\sigma_1, \ldots \sigma_n)$ by lemma 6.1.1. Let Γ be a context in which all types are positive. Let $\Gamma \vdash MN : \tau_2$; since M is light, by inductive hypothesis it must be $\Gamma \vdash M : \tau_1 \to \tau_2$ with positive $\tau_1 \to \tau_2$. Hence τ_2 must be positive. Let $\Gamma \vdash \lambda x : \tau_1.\, M : \tau_1 \to \tau_2$; since τ_1 must contain no occurrences of T, by inductive hypothesis, it must be $\Gamma, x : \tau_1 \vdash M : \tau_2$ with positive τ_2; hence $\tau_1 \to \tau_2$ is positive. $\square$

Theorem 6.2.4 *Let $(id, \sigma) : \langle \mathcal{C}, T, \mathcal{A} \rangle \to \langle \mathcal{C}, S, \mathcal{B} \rangle$ be a morphism of Σ-models. Any light term M and positive Γ such that $\Gamma \vdash M : \tau$ in $ML_T(\Sigma)$ satisfies the following diagram: writing $[\![\_]\!]^T$ for interpretation in $\langle \mathcal{C}, T, \mathcal{A} \rangle$ and $[\![\_]\!]^S$ for interpretation in $\langle \mathcal{C}, S, \mathcal{B} \rangle$,*

$$\begin{tikzcd}
\llbracket\Gamma\rrbracket^T \arrow[r, "\llbracket M\rrbracket^T_\Gamma"] \arrow[d, "\sigma^+(\Gamma)"'] & \llbracket\tau\rrbracket^T \arrow[d, "\sigma^+(\tau)"] \\
\llbracket\Gamma\rrbracket^S \arrow[r, "\llbracket M\rrbracket^S_\Gamma"'] & \llbracket\tau\rrbracket^S
\end{tikzcd}$$

Proof. The interesting cases are lambda abstraction and application, since for $* : 1$ the result follows from the universality of 1, while for variables, application of constants, pairs and projections it follows from the universality of products. Let $\Gamma \vdash MN : \tau_2$. Assume $\Gamma \vdash M : \tau_1 \to \tau_2$ and $\Gamma \vdash N : \tau_1$ make the diagram commute. The soundness of such assumptions depends on lemma 6.2.3, which ensures that both $\tau_1 \to \tau_2$ and τ_1 are positive. So τ_1 must contain no occurrences of T and, by lemma 6.1.2, $\sigma^+(\tau_1) = \sigma^-(\tau_1) = id$; we write τ_1 for $\llbracket\tau_1\rrbracket^T = \llbracket\tau_1\rrbracket^S$. Since $\sigma^+(\tau_1 \to \tau_2) = \lambda f : \tau_1 \to \llbracket\tau_2\rrbracket^T.\, (\lambda x : \tau_1.\, \sigma^+(\tau_2)(fx))$ and $\llbracket N\rrbracket^T_\Gamma = \llbracket N\rrbracket^S_{\sigma^+(\Gamma)}$, we have:

$$\begin{array}{ll}
\sigma^+(\tau_2)\llbracket MN\rrbracket^T_\Gamma = & \text{(by lambda abstraction)} \\
(\lambda f : \tau_1 \to \llbracket\tau_2\rrbracket^T.\, (\lambda x : \tau_1.\, \sigma^+(\tau_2)(fx)))\ (\llbracket M\rrbracket^T_\Gamma, \llbracket N\rrbracket^T_\Gamma) = & \\
 & \text{(by definition of } \sigma^+(\tau_1 \to \tau_2)) \\
(\sigma^+(\tau_1 \to \tau_2)\ \llbracket M\rrbracket^T_\Gamma)\ \llbracket N\rrbracket^T_\Gamma = & \text{(by inductive hypothesis on } M) \\
\llbracket M\rrbracket^S_{\sigma^+(\Gamma)}\llbracket N\rrbracket^T_\Gamma = & \text{(by inductive hypothesis on } N) \\
\llbracket M\rrbracket^S_{\sigma^+(\Gamma)}\llbracket N\rrbracket^S_{\sigma^+(\Gamma)} = & \\
\llbracket MN\rrbracket^S_{\sigma^+(\Gamma)} &
\end{array}$$

Similarly, let $\Gamma \vdash \lambda x : \tau_1.\, M : \tau_1 \to \tau_2$ with no T in τ_1. Then $(\Gamma, x : \tau_1)$ is a positive context and we can assume the property for $\Gamma, x : \tau_1 \vdash M : \tau_2$. Therefore:

$$\begin{aligned}
&\sigma^+(\tau_1 \to \tau_2)\ [\![\lambda\, x : \tau_1 .\, M]\!]^T_\Gamma = \\
&(\lambda\, f : \tau_1 \to [\![\tau_2]\!]^T .\, (\lambda\, x : \tau_1 .\, \sigma^+(\tau_2)(f\, x)))\ (\lambda\, x : \tau_1 .\, [\![M]\!]^T_{\Gamma,x}) = \\
&\lambda\, x : \tau_1 .\, \sigma^+(\tau_2)\ [\![M]\!]^T_{\Gamma,x} = \qquad\qquad \text{(by inductive hypothesis)} \\
&\lambda\, x : \tau_1 .\, [\![M]\!]^S_{\sigma^+(\Gamma),x} = \\
&[\![\lambda\, x : \tau_1 .\, M]\!]^S_{\sigma^+(\Gamma)}.
\end{aligned}$$

□

Theorem 6.2.5 *Let $(id, \sigma) : \langle \mathcal{C}, T, \mathcal{A} \rangle \to \langle \mathcal{C}, S, \mathcal{B} \rangle$ be a morphism of Σ-models. If M and N are light terms of $ML_T(\Sigma)$ with free variables in context Γ and T does not occur in Γ, then $[\![M]\!]^T_\Gamma = [\![N]\!]^T_\Gamma$ implies $[\![M]\!]^S_\Gamma = [\![N]\!]^S_\Gamma$.*

Proof. Theorem 6.2.4 applies to M and N because Γ is positive. Moreover $\sigma^+(\Gamma)$ is the identity and hence $[\![M]\!]^S_\Gamma = \sigma^+(\tau)\ [\![M]\!]^T_\Gamma = \sigma^+(\tau)\ [\![N]\!]^T_\Gamma = [\![N]\!]^S_\Gamma$. □

An immediate corollary to this theorem is that all equations between closed light terms are preserved by pointed constructors. Note that, although the axiom schemes of most theories of computation involve only light terms, the above theorem ensures only the preservation of light instances of such schemes.

Remark. One may wonder whether, in case the category $\mathcal{C}$ is *well pointed*, preserving closed equations is enough to preserve also the open ones (over positive contexts). A category $\mathcal{C}$ with terminal object 1 is called well pointed when the homset functor $\mathcal{C}(1, \_)$ is faithful, that is, when two morphisms f and g are equal whenever $fx = gx$ for all global elements x. In general, an axiom scheme $\emptyset \vdash F(M) = G(M)$ does not enforce $[\![F]\!] = [\![G]\!]$ in all models $\mathcal{M}$. However, assume $\mathcal{M}$ is well pointed. One can extend the language with one constant $\underline{x} : \tau$, where τ is the type of the metavariable M, for each global element $x : 1 \to [\![\tau]\!]$ of $\mathcal{M}$, and extend the interpretation with $[\![\underline{x}]\!] = x$. Then, for all $x : 1 \to [\![\tau]\!]$, $[\![F]\!] \circ x = [\![F(\underline{x})]\!] = [\![G(\underline{x})]\!] = [\![G]\!] \circ x$ and hence $[\![F]\!] = [\![G]\!]$. In order to make a similar argument work when two interpretations $[\![\_]\!]^T$ and $[\![\_]\!]^S$ are at stake, it is necessary that each global element $1 \to SX$ factorizes as $1 \to TX \xrightarrow{\sigma_X} SX$ for some global element of TX.

□

Our second question about pointed semantic constructors $\mathcal{F}$ is whether they are *conservative* on some class of equations, that is, for what M and N it is the case that $[\![M]\!]^{\mathcal{FM}} = [\![N]\!]^{\mathcal{FM}}$ implies $[\![M]\!]^{\mathcal{M}} = [\![N]\!]^{\mathcal{M}}$.

We shall call "light" the polytypes of $ML_T(\Sigma)$ freely generated by the rules A1-A5 of section 2.3, with the extra condition, in A5, that σ contains no occurrences of T in a light $\sigma \to \tau$. Lightly typed terms need not be light, nor light terms lightly typed.

Lemma 6.2.6 *Let $(\sigma, U) : \langle \mathcal{C}, T, \mathcal{A} \rangle \to \langle \mathcal{D}, S, \mathcal{B} \rangle$ be a morphism of Σ-models. If the components of σ are monos and S is mono preserving, then $\sigma^+(\tau)$ is a mono for any light polytype τ.*

Proof. By induction on the structure of τ. For exponentials it uses the fact that $\sigma^+(\tau_1 \to \tau_2) = (\sigma^+(\tau_2))^{\tau_1}$ and that the functor $(\_)^{\tau_1}$ preserves monos because it is a right adjoint. For $T\tau$ it uses the assumptions that the components of σ are monos and S is mono preserving. □

Theorem 6.2.7 *Let M and N be light terms of light type τ in a positive context Γ, and let $(id, \sigma) : \langle \mathcal{C}, T, \mathcal{A} \rangle \to \langle \mathcal{C}, S, \mathcal{B} \rangle$ be a morphism of Σ-models. If the components of σ are monos and S is mono preserving, then $[\![M]\!]^S_\Gamma = [\![N]\!]^S_\Gamma$ implies $[\![M]\!]^T_\Gamma = [\![N]\!]^T_\Gamma$.*

Proof. M and N satisfy the diagram of theorem 6.2.4. Hence,

$$\sigma^+(\tau) \circ [\![M]\!]^T_\Gamma = [\![M]\!]^S_\Gamma \circ \sigma^+(\Gamma) = [\![N]\!]^S_\Gamma \circ \sigma^+(\Gamma) = \sigma^+(\tau) \circ [\![N]\!]^T_\Gamma.$$

Since, by lemma 6.2.6, $\sigma^+(\tau)$ is a mono, $[\![M]\!]^T_\Gamma = [\![N]\!]^T_\Gamma$. □

In [Mog90b], Moggi investigates similar properties for a variety of constructions of the modular approach, such as *uniform redefinitions.* Let T be a strong monad and let

$G(T)$ be a map $obj(\mathcal{C}) \to obj(\mathcal{C})$ on the objects of the underlying category $\mathcal{C}$ of T. A constructor $\mathcal{F}$ such that $(\mathcal{F}T)X = T(GTX)$ can uniformly redefine operations of the form $op_X : \zeta(TX)$, where $\zeta(\_)$ is a type scheme:

$$(\mathcal{F}op)_X \stackrel{\text{def}}{=} op_{GTX} : \zeta(T(GTX)) \equiv \zeta((\mathcal{F}T)X)$$

Examples of constructors of the form $(\mathcal{F}T)X = T(GTX)$ are the one for complexity, $\mathcal{F}TA = T(A \times M)$ for M a monoid, and all instances of the constructor of generalized resumptions described in section 6.4, among which the one for exceptions. An example of uniformly redefinable operation is $or_X : TX \times TX \to TX$ of section 2.5, where $\zeta(X)$ is $X \times X \to X$.

Equations involving instances of redefinable operations are not always preserved by uniform redefinitions. For example, let $op_X : TX$; the identity monad satisfies $op_1 =_{T1} x$ but, for $\mathcal{F}TX = T(A+E)$, $\mathcal{F}op_1 =_{\mathcal{F}T1} x$ does not hold, that is $op_{1+E} \stackrel{?!}{=}_{1+E} x$. However, equation schemes only involving uninstantiated occurrences of polymorphic operations are preserved. For example, the commutativity law $or_X(x, y) =_{TX} or_X(y, x)$, is preserved by uniform redefinitions of or.

Remark. Preserving equations can be also studied syntactically, by reasoning about theories and translations rather than models and constructors. For example, a constructor $Mod(t) : Mod(\Sigma) \to Mod(\Sigma + \Sigma_{\mathcal{F}})$ deriving from a translation t of $ML_T(\Sigma + \Sigma_{\mathcal{F}})$ into $ML_T(\Sigma)$ would preserve the truth of equations $M = N$ when $t(M) = t(N)$ is a consequence of $M = N$ in $ML_T(\Sigma)$.

6.3 HML

As mentioned earlier, interesting semantic constructors can be presented as translations of metalanguages, provided a powerful enough syntax is adopted. In [CM93], a metalanguage HML is used to that effect. HML is not biased towards monads, but it is expressive enough to easily axiomatize desirable metalanguage features, including monads, and to describe concisely uniform redefinitions.

HML is defined in appendix A. There are rules for deriving well formed *kinds*, *operators*, *type schemes*, *terms* and *formulae*. There are also rules for deriving compound objects such as *contexts*, that is sequences of variable declarations, *signatures*, that is sequences of constant declarations, and *theories*, that is sequences of formulae.

In a pure type system [Bar92], *kind* and *scheme* would be called "variable universes," because kinds and type schemes can be used in contexts as ranges of variables. An object u of kind k is called an *operator*, while an object e of type scheme σ is called a *term*. Besides judgements of well formedness, HML features equality judgements on operators and schemes, and truth judgements involving formulae of first order logic with equality.

Polymorphic operations are expressed in HML as follows. There is a constant kind Ω whose operators, called *types*, lift to type schemes. Variables of kind Ω can be "quantified" upon to form type schemes. For example, the signature of a theory of monads is:

$$\begin{aligned} T &: \Omega \Rightarrow \Omega, \\ val &: \Pi v : \Omega.\, v \Rightarrow Tv, \\ let &: \Pi v_1, v_2 : \Omega.\, (v_1 \Rightarrow Tv_2) \times Tv_1 \Rightarrow Tv_2 \end{aligned}$$

while the axioms 2.3, 2.4, and 2.5 of section 2.2 provide the related theory.

The above type schemes are also types in the polymorphic lambda calculus, PLC [Gir72,Rey74], of which we shall consider the version described in [See87]. However, there are differences between HML and PLC in the treatment of polymorphism. For example, the *orders* of PLC, corresponding to kinds in HML, only admit exponentials of the form Ω^A. PLC has both product (Π) and sum (Σ) types, while HML only has products. Particularly significant is also that, unlike PLC, HML distinguishes between types and type schemes: there may be schemes, for example the ones involving Π, which may correspond to no operator of kind Ω. Motivations for such an approach can be found in [Mog91a], while the semantical implications of it are discussed in section 6.7.

Since types and type schemes are distinguished in HML, operators of the form $k \Rightarrow \Omega$ would not suffice to express uniform redefinitions in full generality. For example, the type scheme $S(X) \stackrel{\text{def}}{=} \Pi Y : \Omega.\, X \times Y$ cannot be expressed using an operator $S : \Omega \Rightarrow \Omega$. Therefore HML features *parametric schemes* of the form $S : k \Rightarrow \mathit{scheme}$.

In HML, schemes have products and exponentials. These operations must be reflected in the world of kinds if they are to be used together with operators to form types. For example, the scheme $X : \Omega \vdash T(X \times X) : \mathit{scheme}$ requires an operator $\times : \Omega \times \Omega \rightarrow \Omega$. Below we present signatures and theories for reflecting products and sums. Exponentials can be treated similarly.

The following signatures and theories can be combined to describe computational models. Of course, the analysis of models must support this process as some theories may be inconsistent with each other.

Notation. Sometimes we leave type arguments implicit in polymorphic terms. For example, if $F : (\Pi X : \Omega.\, X \Rightarrow X)$ and $M : A : \Omega$, we may write FM for $F[A]M$. We assume that $\times$ binds more tightly than $\Rightarrow$ and that application is left associative, reading $(f\ g\ x)$ as $(f\ g)x$. If $F : X \times Y \Rightarrow Z$ and $M : X$, we may write FM for $(\lambda\, y : Y.\, FMy) : Y \Rightarrow Z$.

Product reflection (signature).

$unit : \Omega,$

$In_1 : 1 \Rightarrow unit,$

$Out_1 : unit \Rightarrow 1,$

$prod : \Omega \times \Omega \Rightarrow \Omega,$

$In_\times : \Pi v_1, v_2 : \Omega.\, (v_1 \times v_2) \Rightarrow prod\langle v_1, v_2\rangle,$

$Out_\times : \Pi v_1, v_2 : \Omega.\, prod\langle v_1, v_2\rangle \Rightarrow (v_1 \times v_2)$

Product reflection (theory).

$\forall\, x : 1.\ Out_1(In_1 x) = x,$

$\forall\, x : unit.\ In_1(Out_1 x) = x,$

$\forall v_1, v_2 : \Omega, x : v_1 \times v_2.\ Out_\times(In_\times x) = x,$

$\forall v_1, v_2 : \Omega, x : prod\langle v_1, v_2\rangle.\ In_\times(Out_\times x) = x$

Sum reflection (signature).

$0 : \Omega,$

$+ : \Omega \times \Omega \Rightarrow \Omega,$

$Z : \Pi v : \Omega.\, 0 \Rightarrow v,$

$inj_1 : \Pi v_1, v_2 : \Omega. v_1 \Rightarrow v_1 + v_2,$

$inj_2 : \Pi v_1, v_2 : \Omega. v_2 \Rightarrow v_1 + v_2,$

$case : \Pi v_1, v_2, v : \Omega.\, (v_1 \Rightarrow v) \times (v_2 \Rightarrow v) \times (v_1 + v_2) \Rightarrow v$

Sum reflection (theory).

$\forall v : \Omega, f : 0 \Rightarrow v.\ f = Z(v),$

$\forall v_1, v_2, v : \Omega, f_1 : v_1 \Rightarrow v, f_2 : v_2 \Rightarrow v.\ (\lambda\, x.\ case\, f_1\ f_2\ (inj_1 x)) = f_1,$

$\forall v_1, v_2, v : \Omega, f_1 : v_1 \Rightarrow v, f_2 : v_2 \Rightarrow v.\ (\lambda\, y.\ case\, f_1\ f_2\ (inj_2 y)) = f_2,$

$\forall v_1, v_2, v : \Omega, f : (v_1 + v_2) \Rightarrow v.\ case(\lambda\, x.\ f(inj_1 x), \lambda\, y.\ f(inj_2 y)) = f$

As shown in chapter 3 in the case of a programming language with exceptions, programs of type B with a parameter of type A translate into terms of type $A \to TB$ in the computational metalanguage. Hence, interpretation of recursively defined programs requires fixed point operators over types of the form $A \to TB$. The following is an axiomatization of such operators in HML. It requires a type constructor T which would typically be provided by the signature of monads.

Fixed points (signature).
$Y : \Pi v_1, v_2 : \Omega. \, ((v_1 \Rightarrow Tv_2) \Rightarrow (v_1 \Rightarrow Tv_2)) \times v_1 \Rightarrow Tv_2$

Fixed points (theory).
$\forall v_1, v_2 : \Omega, f : (v_1 \Rightarrow Tv_2) \Rightarrow (v_1 \Rightarrow Tv_2). \, Y(f) = f(Yf)$

Algebraically complete categories [Fre91] provide interpretations for data types which can be mathematically described as "minimal" solutions to recursive equations $D \cong FD$, for F a covariant functor. More precisely, a category is called algebraically complete if every covariant endofunctor (in a suitable 2-categorical sense) has initial algebra. We write $\alpha_F : F(\mu F) \to \mu F$ the initial F-algebra and $It_F(\beta) : \mu F \to B$ the unique F-homomorphism from α_F to $\beta : FB \to B$. By a well known result due to Lambek [Lam68], α_F is an isomorphism with inverse $\gamma_F \stackrel{\mathrm{def}}{=} It_F(F\alpha_F)$. In computer science, datatypes arising as initial algebras of covariant endofunctors are called *inductive types.*

Remark. Similarly, when $\mathcal{C}$ is algebraically complete and $\mathcal{A}$ is any category, a *type constructor* $\tilde{F}$ can be defined from a bifunctor $F : \mathcal{A} \times \mathcal{C} \to \mathcal{C}$:

$$\tilde{F}A \stackrel{\mathrm{def}}{=} \mu X. \, F(A, X).$$

$\tilde{F}$ is in fact a functor $\mathcal{A} \to \mathcal{C}$ mapping arrows $f : A \to B$ to

$$\tilde{F}f \stackrel{\mathrm{def}}{=} It(H_f)$$

where $H_f = F(A, \tilde{F}B) \overset{F(f,\tilde{F}B)}{\longrightarrow} F(B, \tilde{F}B) \overset{\alpha_{F(B,\_)}}{\longrightarrow} \tilde{F}B$. It is easy to verify that the usual `map` function of LISP is the morphisms part of the list constructor $\mathit{List}(A) \stackrel{\text{def}}{=} \mu X.\,(1+(A\times X))$.

$\square$

In [CM93], inductive types are axiomatized in HML as follows:

Inductive types (signature).

$\mu : (\Omega \Rightarrow \Omega) \Rightarrow \Omega,$

$\alpha : \Pi F : \Omega \Rightarrow \Omega . F(\mu F) \Rightarrow \mu F,$

$\mathit{It} : \Pi F : \Omega \Rightarrow \Omega, v : \Omega . (\Pi v_1, v_2 : \Omega . (v_1 \Rightarrow v_2) \times F v_1 \Rightarrow F v_2) \times (F v \Rightarrow v) \times \mu F \Rightarrow v$

Inductive types (theory).

$\forall F : \Omega \Rightarrow \Omega, v : \Omega, f : (\Pi v_1, v_2 : \Omega .\, (v_1 \Rightarrow v_2) \times F v_1 \Rightarrow F v_2), \beta : F v \Rightarrow v.$

$\quad \mathit{strength}(f) \supset \forall x : F(\mu F).\ \mathit{It}\ f\ \beta\ (\alpha x) = \beta(f(\mathit{It}\ f\ \beta)x),$

$\forall F : \Omega \Rightarrow \Omega, v : \Omega, f : (\Pi v_1, v_2 : \Omega .\, (v_1 \Rightarrow v_2) \times F v_1 \Rightarrow F v_2), \beta : F v \Rightarrow v, g : \mu F \Rightarrow v.$

$\quad \mathit{strength}(f) \wedge (\forall x : F(\mu F).\ g(\alpha x) = \beta(f\ g\ x)) \supset g = \mathit{It}\ f\ \beta,$

where $\mathit{strength}(f)$ is a formula asserting that f preserves identities and composition. This is not completely satisfactory as the operations work on separate pieces of information, viz. $F : \Omega \Rightarrow \Omega$ and $f : \Pi v_1, v_2 : \Omega .\, (v_1 \Rightarrow v_2) \times F v_1 \Rightarrow F v_2$, rather than on some representation of a functor. For example, α is asked to deliver an initial algebra for a functor of which it is only given the map on objects. In chapter 7, we axiomatize inductive types in the Extended Calculus of Constructions, where functors can be represented by assembling in a Σ-type a map on objects, one on arrows and a proof that the latter is a strength for the former.

Remark on inductive types and fixed points. The axiomatization of fixed points given above is rather weak. However, it is enough to show the following. Let (F, f) be a strong endofunctor. A fixed point operator Y on arrow types yields an algebra morphism $A \to B$ for all F-algebras $\beta : FB \to B$ and isomrphisms $\alpha : FA \to A$. Writing $G \stackrel{\text{def}}{=} \lambda h : A \to B .\, \lambda x : A .\, \beta\,(f\ h\ (\alpha^{-1} x))$, we have:

$$\beta \circ f(YG) = \lambda\, z : FA.\, \beta\, (f\, (YG)\, z)$$
$$= \lambda\, z : FA.\, (\lambda\, x : A.\, \beta\, (f\, (YG)\, (\alpha^{-1} x)))\, (\alpha z)$$
$$= \lambda\, z : FA.\, G(YG)\, (\alpha z)$$
$$= \lambda\, z : FA.\, YG\, (\alpha z) = YG \circ \alpha.$$

Since, in the theory of inductive types, α_F is an isomorphism, in the combined theory of inductive types and fixed points, $(It_F\; f\; \beta) = YG$ for the uniqueness of $(It\; f\; \beta)$.

Vice versa, let $\langle T, val, let \rangle$ be a strong monad; a fixed point operator Y of the form $Y_{A,B} : (A \to TB) \to (A \to TB) \to A \to TB$ satisfying $F(YF) = YF$ can be defined as in [CP92] from an initial T-algebra and a *fixed point element* $\omega : 1 \to T(\mu T)$ equalizing $id_{\mu T}$ and $val \circ \alpha_T$.

6.4 Relative interpretation of HML

In this section we introduce the notion of relative interpretation of HML and show an application where a relative interpretation is used to present a semantic constructor syntactically.

The sets of *raw kinds* ($\mathcal{K}$), *operators* ($\mathcal{U}$), *type schemes* ($\mathcal{S}$), *terms* ($\mathcal{E}$) and *formulae* ($\mathcal{P}$), which are part of the raw syntax of HML, are defined in appendix A. We collectively refer to the elements of these sets as *raw expressions*. Below we use the metavariables K, C, S, c and P to range respectively over the sets $Const_K$, $Const_O$, $Const_S$, $Const_E$ and $Const_P$ of constants. The sets Var_O and Var_E of variables are ranged over by v and x.

Let $Const = Const_K \cup Const_O \cup Const_S \cup Const_E \cup Const_P$ and let $Const_1$ be a subset of $Const$; we call HML($Const_1$) the set of raw expressions over $Const_1$, that is the ones generated by the grammar in appendix A from the constants in $Const_1$. Given subsets $Const_1$ and $Const_2$ of $Const$, a *raw translation* of HML($Const_2$) into HML($Const_1$) is a map $(\!|\_|\!)$ from $Const_2$ to the raw expressions over $Const_1$ such that

$(\![K]\!) \in \mathcal{K}$,

$(\![C]\!) \in \mathcal{U}$ and it has no free variables,

$(\![S]\!) \in \mathcal{S}$ and it has at most one free variable $v \in \mathit{Var}_O$,

$(\![c]\!) \in \mathcal{E}$ and it has no free variables,

$(\![P]\!) \in \mathcal{P}$ and it has at most two free variables $v \in \mathit{Var}_O$ and $x \in \mathit{Var}_E$.

This map has an obvious extension, which we also write $(\![\_]\!)$, to the raw expressions over Const_2, where $(\![S(u)]\!) = [(\![u]\!)/v](\![S]\!)$ and $(\![P(u,e)]\!) = [(\![u]\!)/v]\,[(\![e]\!)/x](\![P]\!)$.

Let Σ be a well formed signature in HML; we call HML(Σ) the "pure" HML over Σ, that is the set of judgements which are derivable from the axiom $\vdash \Sigma\ \mathit{sig}$ and all the inference rules except for the ones for signatures and (add-prop). Let Σ_1 and Σ_2 be well formed signatures and let Const_1 and Const_2 be the sets of constants in Σ_1 and Σ_2 respectively; a *translation* $\theta : \mathrm{HML}(\Sigma_2) \to \mathrm{HML}(\Sigma_1)$ is a raw translation of HML(Const_2) into HML(Const_1) such that, for all kinds $K : \mathit{kind}$, $C : k$,$S : k \Rightarrow \mathit{scheme}$, $c : \sigma$ and $P : (v : k)\sigma \Rightarrow \mathit{prop}$ in Σ_2, the following judgements are derivable:

$\vdash_{\Sigma_1} (\![K]\!)\ \mathit{kind}$,

$\vdash_{\Sigma_1} (\![C]\!) : (\![k]\!)$,

$v : (\![k]\!) \vdash_{\Sigma_1} (\![S]\!)\ \mathit{scheme}$,

$\vdash_{\Sigma_1} (\![c]\!) : (\![\sigma]\!)$,

$v : (\![k]\!), x : (\![\sigma]\!) \vdash_{\Sigma_1} (\![P]\!)\ \mathit{prop}$.

The map induced on the raw syntax by a translation preserves derivability, that is, if $\Gamma \vdash_{\Sigma_2} u_1 = u_2 : k$ is in HML(Σ_2) then $(\![\Gamma]\!) \vdash_{\Sigma_1} (\![u_1]\!) = (\![u_2]\!) : (\![k]\!)$ is in HML(Σ_1), if $\Gamma \vdash_{\Sigma_2} \sigma_1 = \sigma_2$ is in HML(Σ_2) then $(\![\Gamma]\!) \vdash_{\Sigma_1} (\![\sigma_1]\!) = (\![\sigma_2]\!)$ is in HML(Σ_1) and, if $\Gamma; \Delta \vdash_{\Sigma_2,\emptyset} \phi$ is in HML(Σ_2) then $(\![\Gamma]\!), (\![\Delta]\!) \vdash_{\Sigma_1,\emptyset} (\![\phi]\!)$ is in HML(Σ_1). This can be shown by induction on the length of derivation, with the aid of a substitution lemma.

Let $\vdash_\Sigma \mathcal{T}\ \mathit{theory}$ be derivable in HML; we call HML($\Sigma, \mathcal{T}$) the set of judgements that are derivable from the axioms $\vdash \Sigma\ \mathit{sig}$ and $\vdash_\Sigma \mathcal{T}\ \mathit{theory}$ and all the rules but the ones for signatures and theories. We also write $\mathcal{T}$ for $\mathrm{HML}(\Sigma, \mathcal{T})$ and call it a theory over Σ. When Σ and $\mathcal{T}$ are understood, we drop the indices in $\vdash_\Sigma$ and $\vdash_{\Sigma,\mathcal{T}}$.

Given theories $\mathcal{T}_1$ and $\mathcal{T}_2$ over Σ_1 and Σ_2 respectively, a *relative interpretation* $\theta : \mathrm{HML}(\Sigma_2, \mathcal{T}_2) \to \mathrm{HML}(\Sigma_1, \mathcal{T}_1)$ is a translation as above such that, if $\Gamma; \Delta \vdash_{\Sigma_2,\mathcal{T}_2} \phi$ is derivable, then so is $(\![\Gamma]\!); (\![\Delta]\!) \vdash_{\Sigma_1,\mathcal{T}_1} (\![\phi]\!)$.

A theory $ML_T(\Sigma)$ of the computational metalanguage can be expressed in HML by giving a signature $\tilde{\Sigma}$ and a theory $\mathcal{T}$ over $\tilde{\Sigma}$ such that $\mathrm{HML}(\tilde{\Sigma}, \mathcal{T})$ is a conservative extension of $ML_T(\Sigma)$, provided a suitable translation of $ML_T(\Sigma)$ into $\mathrm{HML}(\tilde{\Sigma})$. The advantage of representing theories of the computational metalanguage in HML is that type constructors become legal expressions of the language and so do polymorphic operations; we shall benefit from this approach in section 6.8 where translations of HML theories are viewed as functors, with no need of introducing naturality conditions on the operations.

In the rest of this section we show how a relative interpretation can be used to present a semantic constructor for resumptions. In section 2.5, we described resumptions of very primitive sorts of programs. Now we consider more realistic cases. For example, interleaved executions of possibly nonterminating programs acting on a shared store can be described by interpreting a program P of type A in a domain

$$D \cong S \to ((A + D) \times S)_\perp,$$

that is, as a partial function from stores into pairs (q, s), where the store s records the effects of one atomic step of evaluation and q is either a value of type A, in which case P has done, or what is left of P.

The computational structure described by the above equation can be split into three separate modules, one for resumptions, $\mathcal{F}$, one for states, $\mathcal{S}$, and one for nontermination, $(\_)_\perp$:

$$\begin{aligned}
(\mathcal{F}T)A &\stackrel{\mathrm{def}}{=} \mu X \,.\, T(A + X) \\
(\mathcal{S}T)A &\stackrel{\mathrm{def}}{=} S \to T(A \times S) \\
D &\stackrel{\mathrm{def}}{=} (\mathcal{F}(\mathcal{S}(_)_\perp))A.
\end{aligned}$$

Below we describe a more general version of $\mathcal{F}$, called the constructor of *generalized resumptions*. This constructor is defined by a relative interpretation $(\!|\_|\!) : \mathrm{HML}(\Sigma_0 + \Sigma_{\mathcal{F}}, \mathcal{T}_0 + \mathcal{T}_{\mathcal{F}}) \to \mathrm{HML}(\Sigma_0, \mathcal{T}_0)$ and uniform redefinitions. Σ_0 and $\mathcal{T}_0$ contain respectively the signatures and the theories of monads, sums and inductive types. Moreover, Σ_0 contains the signature (and $\mathcal{T}_0$ the obvious axioms) of a strong endofunctor:

$$H : \Omega \Rightarrow \Omega$$

$$st : \Pi X, Y : \Omega.(X \Rightarrow Y) \times HX \Rightarrow HY.$$

$\Sigma_{\mathcal{F}}$ and $\mathcal{T}_{\mathcal{F}}$ describe the mathematics of resumptions in terms of the operations:

$$\tau : \Pi X : \Omega.\, TX \to TX$$

$$C : \Pi X, Y : \Omega.\, (X \Rightarrow TY) \times (HTX \Rightarrow TY) \times TX \Rightarrow TY$$

the first of which performs one folding step with no T-computation involved, while the second is essentially a case analysis. The translation $(\!|\_|\!)$ is as follows: writing $F_A(X)$ for $T(A + HX)$ and $\gamma_{F_A} : \mu X.F_A(X) \to F_A(\mu X.F_A(X))$ for $It_{F_A}(F_A(\alpha_{F_A}))$,

$$\begin{array}{rcl}
(\!|TA|\!) & \stackrel{\mathrm{def}}{=} & \mu X.\, T((\!|A|\!) + HX) \\
(\!|val_A(a)|\!) & \stackrel{\mathrm{def}}{=} & \alpha_{F_A}(val\,(inj_1(\!|a|\!))) \\
(\!|\tau_A(z)|\!) & \stackrel{\mathrm{def}}{=} & \alpha_{F_A}(val\,(inj_2(\!|z|\!))) \\
(\!|let_{A,B}\ f|\!) & \stackrel{\mathrm{def}}{=} & It_{F_A}(\lambda w : T(A + H(\!|TB|\!)). \\
 & & \quad \alpha_{F_B}(let\ (\lambda c : A + H(\!|TB|\!).\, \gamma_{F_B}(case\ (\!|f|\!)\ (\!|\tau_B|\!)\ c))\, w) \\
(\!|C_{A,B}\ f\ g\ z|\!) & \stackrel{\mathrm{def}}{=} & \alpha_{F_B}(let\ (\lambda c : A + H(\!|TA|\!).\, \gamma_{F_B}(case\ (\!|f|\!)\ (\!|g|\!)\ c))\ (\gamma_{F_A}(\!|z|\!))).
\end{array}$$

Remark. $(\!|let|\!)$ is sequential composition. $\square$

By suitably instantiating H, one obtains the constructors $T(A + E)$ of exceptions, $\mu X.\,T(A{+}X)$ of resumptions, $\mu X.\,T(A{+}(O{\times}X))$ of interactive output and $\mu X.\,T(A{+}X^I)$ of interactive input. All such constructors are pointed, in the sense that there is a strong monad morphism $(id, T(inj_1)) : (\mathcal{C}, T) \to (\mathcal{C}, \mathcal{F}T)$. All that is required to make theorems 6.2.5 and 6.2.7 work is that such morphisms are natural with respect to the operations associated with T-computations. Note that, in view of definition 6.1.3 one should exclude from $\Sigma_{\mathcal{F}}$ operations with nonpositive type scheme such as $C_{A,B}$ in order to make $\mathcal{F}$ pointed.

Let Σ_0 be the following HML signature:

$G : (\Omega \Rightarrow \Omega) \times \Omega \Rightarrow \Omega$,

$S : \Omega \Rightarrow \mathit{scheme}$,

$T : \Omega \Rightarrow \Omega$

Uniform redefinitions of operations op of type scheme $\Pi X : \Omega. S(TX)$ can be described in HML by the following translation $(\!|\_|\!) : \mathrm{HML}(\Sigma_0 + \{op\}) \to \mathrm{HML}(\Sigma_0 + \{op\})$:

$$
\begin{aligned}
(\!|TA|\!) &\stackrel{\mathrm{def}}{=} T(GT(\!|A|\!)) \\
(\!|op_A|\!) &\stackrel{\mathrm{def}}{=} op_{GT(\!|A|\!)}
\end{aligned}
$$

Suitable instantiations of G and S yield appropriate uniform redefinitions for specific constructors and operations. In the next chapter we apply a uniform redefinition for the constructor of resumptions, where $GTA = A + \mu X. T(A + X)$, to an operator $or : TX \times TX \Rightarrow TX$ of nondeterministic choice, where $SX = X \times X \Rightarrow X$. The redefined or is then used in the context of computations with resumption to define an operator of parallel composition.

Remark There are interesting constructors in denotational semantics that cannot be described as HML translations. One is that for *local variables*, which involves a change of category.

6.5 Functorial semantics

In section 6.7, we introduce models of HML theories; moreover, we show that such theories can be viewed as categories with structure and their models as structure-preserving functors. This is often done in categorical logics and often referred to as *functorial semantics* [KR77]. In this section, we present the main ideas of functorial semantics and introduce locally finitely presentable categories, that are used in section 6.8 to characterize the categories of functorial models of HML theories.

The idea of functorial semantics is that certain kinds of logical theories correspond to certain choices of categorical properties $\mathcal{P}$, sometimes called *doctrines* [KR77], so that theories of kind $\mathcal{P}$ are identified with categories with $\mathcal{P}$ structure. Such an approach was first adopted by Lawvere [Law75] to obtain a "presentation invariant" view of algebraic theories, where two theories are identified if the primitive operations of each one are derivable in the other. An historical overview of this subject can be found in [BW85, 4.5].

In the setting of functorial semantics, models of a theory T in a category $\mathcal{E}$ with $\mathcal{P}$ structure correspond to $\mathcal{P}$-preserving functors from T to $\mathcal{E}$; we write $[T, \mathcal{E}]_{\mathcal{P}}$ the category of such functors. Translations $\phi : T_2 \to T_1$ of one theory into another also correspond to $\mathcal{P}$-preserving functors; the map of models $\phi^* : [T_1, \mathcal{E}]_{\mathcal{P}} \to [T_2, \mathcal{E}]_{\mathcal{P}}$ obtained by precomposing with ϕ is often called a *relative interpretation.*

Example: algebras as functors. In [KR77], a presentation invariant view of an algebraic theory is given as a category T whose objects are the natural numbers $0, 1 \ldots$, with n the n-fold categorical product of 1. Such categories are often called *Lawvere theories.* A morphism of Lawvere theories is a finite product-preserving functor F such that $F1 = 1$. Of course, only *finitary* theories, that is theories on operations of finite arity, are thus represented.

A T-algebra is a functor $T \to \mathit{Sets}$ preserving finite products, and we write $[T, \mathit{Sets}]_{alg}$ the full subcategory of such objects in $[T, \mathit{Sets}]$. Let $\phi : T_2 \to T_1$ preserve finite products; the relative interpretation $\phi^* : [T_1, \mathit{Sets}]_{alg} \to [T_2, \mathit{Sets}]_{alg}$ has left adjoint [GU75]. Let T_0 be the Lawvere theory whose only morphisms are the projections; this is the initial object in the category of Lawvere theories and its category of algebras is Sets itself: $[T_0, \mathit{Sets}]_{alg} \cong \mathit{Sets}$. The relative interpretation induced by the unique $T_0 \to T$ is the forgetful functor $U\alpha = \alpha(1)$, whose left adjoint is the free T-algebra construction $F : \mathit{Sets} \to [T, \mathit{Sets}]_{alg}$. We say that the monad induced on Sets by this adjunction *classifies* the theory T. A consequence of the Yoneda lemma is that T is equivalent to the dual of the full subcategory of $[T, \mathit{Sets}]_{alg}$ whose objects are free algebras of the form $F(n)$, for n a natural number [KR77]. As Theorem 6.5.4 shows, the same phenomenon arises in the doctrine of finite limit theories. We use this property in section 6.8 to give a categorical characterization of HML relative interpretation.

Digression on algebras and monads. Any monad on Sets cassifies some algebraic theory, provided operations of arbitrarily large arity are allowed, in which case the theory is called *infinitary*. Vice versa, any (possibly infinitary) algebraic theory is classified by some monad on Sets. This is Linton's classical result [Lin66]. Kelly and Power [KP93] extend this result to any locally finitely presentable category (see below), but only for finitary monads, that is monads whose underlying functor preserves filtered colimits. In [Rob95], E. Robinson points out that the correspondence between monads and algebraic theories in categories enriched over symmetric monoidal closed categories is rather general and does not depend on finiteness.

Let T be a strong monad (we are now describing a special case of the above situation, where $\mathcal{C}$ is enriched over itself and T is a $\mathcal{C}$-monad). The correspondence between T-algebras and algebras of the theory $\hat{T}$ classified by T is as follows: for every T-algebra $\alpha : TA \to A$ there is a family of morphisms $\hat{\alpha}_B = \alpha \circ \mathit{eval} \circ (st \times id) : A^B \times TB \to A$, satisfying the axioms specified in [Rob95]. Using lambda notation, $\hat{\alpha}_B(f, z) = \alpha(Tf(z))$. The morphisms $\hat{\alpha}_B$ represent all the operations of arity B of a $\hat{T}$-algebra on A; these operations are indexed by TB. Conversely, any family $F_B : A^B \times TB \to A$ satisfying the axioms, yields a T-algebra $F_A(id_A)$. The two constructions are mutually inverse. Note that $let_{B,A} = (\hat{\mu}_A)_B$ provides the operations of arity B of the free algebra on TA.

□

A class of algebras closed under subalgebras, images and products is called a *variety*. Varieties can be characterized as classes of models of equational theories; this is known as Birkhoff's variety theorem (see [Wec92]). Similarly, *locally finitely presentable* (*lfp*) categories (see below) can be characterized as categories of models of finite limit theories. Here, we state a classic result about such theories and lfp categories that we use in section 6.8. Proofs can be found in [Pit82,AR94], together with the pointers to the standard literature.

A *finite limit theory*, or *lex* (from *l*eft *ex*act) *theory* is a category with all finite limits. We call *Lex* the category of small lex categories and finite limits-preserving functors (lex functors). The category $[T, Sets]_{lex}$ of set valued models of a lex theory T is cocomplete. In fact one can say more. The following definitions are from [Pit82, 4.19]:

Definition 6.5.1 *Let $\mathcal{A}$ be a locally small category with small filtered colimits. We say that an object A of $\mathcal{A}$ is* finitely presentable *when the representable functor $\mathcal{A}(A, \_)$ preserves small filtered colimits.*

Definition 6.5.2 *A class X of objects of a category $\mathcal{A}$ is called* conservative *when, given $f : B \to C$ in $\mathcal{A}$, if $\mathcal{A}(A, f) : \mathcal{A}(A, B) \to \mathcal{A}(A, C)$ is a bijection for each $A \in X$, then f is an isomorphism.*

Definition 6.5.3 *$\mathcal{A}$ is* locally finitely presentable *(lfp) if it is locally small, cocomplete and it contains a conservative* set *X of objects, each of which is finitely presentable and such that any finitely presentable object in $\mathcal{A}$ is isomorphic to some object in X.*

Theorem 6.5.4 ([GU75]) *If T is a small lex category, then $[T, Sets]_{lex}$ is lfp and its full subcategory of finitely presentable objects is equivalent to T^{op}. Conversely, if $\mathcal{C}$ is lfp,* *the dual of its full subcategory of finitely presentable objects is equivalent to a small lex category T and $[T, Sets]_{lex} \cong \mathcal{C}$.*

In [GU75], Gabriel and Ulmer go further and establish a duality between the category of finite limit theories and the one of lfp categories: the function mapping lex categories T to functor categories $[T, Sets]_{lex}$ extends to a functor $[\_, Sets]_{lex} : Lex^{op} \to LFP$, where LFP is the category of lfp categories and filtered colimit-preserving functors with a left adjoint. Conversely, let $\mathcal{C}$ be an lfp category; we write $\mathcal{C}_{fp}$ the full subcategory of finitely presentable objects of $\mathcal{C}$. This category is *essentially* small, i.e. equivalent to a small category and, in what follows, we shall leave the equivalence implicit. Let $\Psi : \mathcal{C} \to \mathcal{D}$ be a morphism in LFP; the functor Φ which is left adjoint to Ψ restricts to a cocartesian functor $\Phi_{fp} : \mathcal{D}_{fp} \to \mathcal{C}_{fp}$ and hence it yields a morphism $\Phi^{op}_{fp} : \mathcal{D}^{op}_{fp} \to \mathcal{C}^{op}_{fp}$ in Lex. In this way, the map $\mathcal{C} \mapsto \mathcal{C}^{op}_{fp}$ extends to a functor $(\_)^{op}_{fp} : LFP \to Lex^{op}$ which forms an adjoint equivalence with $[\_, Sets]_{lex}$ (see [Pit82] for details).

In the next section we give examples of categorical notions that can be described by finite limit theories. Categories themselves are such an example. The existence of a finite limit theory of categories implies that the category Cat of small categories, is lfp. Our goal is to show that all the categorical structure of which models of HML are made is described by a finite limit theory, and hence that the categoy of such models is lfp. This allows us to give an intrinsic characterization of relative interpretation of HML theories by using the Gabriel-Ulmer duality.

6.6 Finite limit presentation of category theory

The categorical structure required for interpreting HML is now described formally in *finite limit formulae*, that is formulae of the form:

$$\forall x.\, \phi(x) \supset \exists! y.\, \psi(x, y)$$

where x and y are tuples of variables, and ϕ and ψ conjunctions of atomic formulae. Such formulae are equivalent to conjunctions of Horn clauses and are called *universal Horn sentences* in [KR77]. We usually drop the universal quantifier but understand that it is there to bind the free variables.

In [Kea75], Keane establishes a correspondence between finite limit formulae and lex categories: models of a first order theory whose axioms are finite limit formulae can be regarded as finite limit-preserving functors from a suitable lex category (an "abstract theory") into *Sets* and homomorhisms between models as natural transformations. In this way, we obtain a representation of models of HML. The same result could be obtained by working with finite limit sketches instead (see [BW85] for details), but these are harder to read than the corresponding theories when complicated categorical structure is involved.

Categories. The finite limit theory of categories has signature consisting of two sorts $\mathcal{C}_A$ and $\mathcal{C}_O$ representing respectively the collections of arrows and that of objects, function symbols *src* and *tgt* of arity $\mathcal{C}_A \to \mathcal{C}_O$ representing source and target, a function symbol e of arity $\mathcal{C}_O \to \mathcal{C}_A$ for the identities and a predicate symbol m of arity $\mathcal{C}_A \times \mathcal{C}_A \times \mathcal{C}_A$ for composition. The axioms of the theory are:

$$tgt(f) = src(g) \supset \exists! h.\, m(f,g,h)$$
$$m(f,g,h) \supset tgt(f) = src(g) \wedge src(f) = src(h) \wedge tgt(g) = tgt(h)$$
$$m(f,g,u) \wedge m(g,h,v) \wedge m(f,v,w) \supset m(u,h,w)$$

$$src(e(a)) = a \wedge tgt(e(a)) = a$$
$$m(e(a),f,g) \supset f = g$$
$$m(f,e(a),g) \supset f = g$$

The first two axioms state that composition is a partial function defined on composable arrows. The third axiom, expressing the commutativity of m, and its dual $m(f,g,u) \wedge m(g,h,v) \wedge m(u,h,w) \supset m(f,v,w)$ are interderivable.

It is easy to verify that categories are set theoretic models of the finite limit theory of categories and functors are homomorphisms between such models. Hence, applying

Keane's result, there exists a lex category T_{Cat} such that:

$$Cat \cong [T_{Cat}, Sets]_{lex}. \tag{6.4}$$

Remark. In order to describe categories one needs source and target functions of type $\mathcal{C}_A \to \mathcal{C}_A$ and composition as above, where the part of objects is played by the identities. This theory is *essentially algebraic* [Fre72], that is, an equational theory involving *partial* operations, e.g. composition, ordered in a list such that the domain of each operation is defined by equations in the previous.

Slices of *Cat*. Let $\mathcal{C}$ be a category. We extend the above finite limit theory of categories as follows: there is a constant $\ulcorner A \urcorner : \mathcal{C}_O$ for each object A of $\mathcal{C}$ and a constant $\ulcorner f \urcorner : \mathcal{C}_A$ for each arrow $f : A \to B$ with axioms $src \ulcorner f \urcorner = \ulcorner A \urcorner$ and $tgt \ulcorner f \urcorner = \ulcorner B \urcorner$. There are axioms $e \ulcorner A \urcorner = \ulcorner id_A \urcorner$ for the identities and similarly for composition. We call this extension the *finite limit theory of* $\mathcal{C}$.

A model m of such a theory consists of a small category $\mathcal{D}$ (the internal category in *Sets* whose object of objects is $m(\mathcal{C}_O)$, whose object of arrows is $m(\mathcal{C}_A)$ and so on) together with a map of objects $A \mapsto m\ulcorner A \urcorner$ from $\mathcal{C}$ to $\mathcal{D}$ and a map of arrows $f \mapsto m\ulcorner f \urcorner$, all satisfying suitable diagrams, making of m a functor $\mathcal{C} \to \mathcal{D}$. It is routine to verify that homomorphisms $h : m_1 \to m_2$ amount to commuting triangles

$$\begin{tikzcd} & \mathcal{C} \arrow[dl, "m_1"'] \arrow[dr, "m_2"] & \\ \mathcal{D}_1 \arrow[rr, "h"'] & & \mathcal{D}_2. \end{tikzcd}$$

Applying Keane's result to the theory of $\mathcal{C}$, there exists a lex category $th(\mathcal{C})$ such that

$$\mathcal{C}/Cat \cong [th(\mathcal{C}), Sets]_{lex}.$$

Note that, since the equivalence $[T_1, Sets]_{lex} \cong [T_2, Sets]_{lex}$ implies $T_1 \cong T_2$, we conclude that $th(\emptyset) \cong T_{Cat}$, where $\emptyset$ is the empty category.

Functors. The finite limit theory of functors, which we write $F : \mathcal{A} \to \mathcal{B}$, consists of two copies $\mathcal{A}$ and $\mathcal{B}$ of the above theory of categories, a function symbol $F : \mathcal{A}_A \to \mathcal{B}_A$ and the following axioms:

$$src\,(f) = src\,(g) \supset src\,(Ff) = src\,(Fg)$$
$$tgt\,(f) = tgt\,(g) \supset tgt\,(Ff) = tgt\,(Fg)$$
$$F(ex) = e(src\,(F(ex)))$$
$$m(g, f, h) \wedge m(Ff, Fg, k) \supset Fh = k$$

Note that the derived operation $F_O : \mathcal{A}_O \to \mathcal{B}_O$ defined as $F_O x = src\,(F(ex))$ satisfies:

$$src\,(Ff) = F_O(src\,(f))$$
$$tgt\,(Ff) = F_O(tgt\,(f))$$
$$F(ex) = e(F_O x).$$

We shall drop the subscript in F_O if no ambiguity arises. It is easy to verify that functors are set theoretic models of $F : \mathcal{A} \to \mathcal{B}$.

Natural transformations. They are similarly defined by a theory $\nu : F \dot{\to} G : \mathcal{A} \to \mathcal{B}$ with a function symbol $\nu : \mathcal{A}_O \to \mathcal{B}_A$ and axioms:

$$src\,(\nu(x)) = Fx$$
$$tgt\,(\nu(x)) = Gx$$
$$m(\nu(src\,(f)), Gf, h) \wedge m(Ff, \nu(tgt\,(f)), k) \supset h = k.$$

Adjunctions. We write $F \dashv G : \mathcal{A} \rightharpoonup \mathcal{B}$ for the finite limit theory of adjunctions, which consists of two natural transformations $\eta : I \dot{\to} GF : \mathcal{A} \to \mathcal{A}$ and $\epsilon : FG \dot{\to} I : \mathcal{B} \to \mathcal{B}$ satisfying the triangular laws $\epsilon F \circ F\eta = id$ and $G\epsilon \circ \eta G = id$, that is:

$$m(F(\eta(x)),\, \epsilon(Fx),\, e(Fx))$$
$$m(\eta(Gy),\, G(\epsilon(y)),\, e(Gy)).$$

Cloven and split fibrations. A cloven fibration is a functor $p : \mathcal{E} \to \mathcal{B}$ with a choice of cartesian morphisms. The finite limit theory of such objects features a predicate *cart* on $\mathcal{E}_A$, to capture cartesian morphisms, and a predicate $\mathit{cleav}(f, a, h) \subseteq \mathcal{B}_A \times \mathcal{E}_O \times \mathcal{E}_A$ assigning to each morphism f of $\mathcal{B}$ and object a in the fibre over the target of f a cartesian morphism h over f with target a:

$$\mathit{cart}(h) \wedge \mathit{tgt}(h) = \mathit{tgt}(k) \wedge m(v, p(h), p(k)) \supset \exists! w.\, p(w) = v \wedge m(w, h, k)$$
$$\mathit{tgt}(f) = p(a) \supset \exists! h.\, \mathit{cleav}(f, a, h)$$
$$\mathit{cleav}(f, a, h) \supset \mathit{cart}(h) \wedge p(h) = f \wedge \mathit{tgt}(h) = a.$$

If $p : \mathcal{E} \to \mathcal{B}$ is cloven and $f : A \to B$ is a morphism in $\mathcal{B}$, $f^* : \mathcal{E}_B \to \mathcal{E}_A$ mapping objects b over B to the source of the cartesian h such that $\mathit{cleav}(f, b, h)$ is called *reindexing along* f. A cloven fibration is said to *split* when reindexing is consistent with composition, that is: $(g \circ f)^* = f^* \circ g^*$. In finite limit clauses:

$$m(f, g, j) \wedge m(h, k, l) \wedge \mathit{cleav}(g, a, k) \wedge \mathit{cleav}(f, \mathit{src}(k), h) \supset \mathit{cleav}(j, a, l).$$

Fibred adjunctions. Let $p : \mathcal{E} \to \mathcal{B}$ and $q : \mathcal{D} \to \mathcal{B}$ be fibrations; a functor $F : \mathcal{E} \to \mathcal{D}$ such that $q \circ F = p$ is said to be *over* $\mathcal{B}$, and written $F : p \to q$, when it preserves cartesian maps:

$$\mathit{cart}(h) \supset \mathit{cart}(F(h)).$$

When p and q are understood, we shall still write $F : \mathcal{E} \to \mathcal{D}$ to mean F over $\mathcal{B}$. In particular, if A is an object of $\mathcal{B}$, we understand $\mathcal{B}/A$ as the source of the domain fibration $\mathit{dom}_A : \mathcal{B}/A \to \mathcal{B}$.

A morphism in the total category of a fibration p is called *vertical* if it is mapped by p to an identity. A *fibred adjunction* between functors F and G over $\mathcal{B}$ is an adjunction $F \dashv G$ with vertical unit and counit. It is easy to see that the unit is vertical if and only if so is the counit. Hence, to turn a pair of adjoint functors over $\mathcal{B}$ into a fibred adjunction, it is enough to postulate:

$$p(\eta(a)) = e(p(a)).$$

Fibred cartesian closed structure. Fibred terminal objects, fibred products, and fibred exponentials are defined "globally" by the fibred adjunctions shown in the picture and discussed below.

$$\begin{tikzcd} \mathcal{E} \arrow[rr, shift left, "p"] \arrow[rr, phantom, "\perp"] \arrow[dr, "p"'] & & \mathcal{B} \arrow[ll, shift left, "\mathbf{1}"] \arrow[dl, "id"] \\ & \mathcal{B} & \end{tikzcd} \qquad \begin{tikzcd} \mathcal{E} \arrow[rr, shift left, "\Delta"] \arrow[rr, phantom, "\perp"] \arrow[dr, "p"'] & & \mathcal{E} \times_{\mathcal{B}} \mathcal{E} \arrow[ll, shift left, "prod"] \arrow[dl, "p \times_{\mathcal{B}} p"] \\ & \mathcal{B} & \end{tikzcd} \qquad \begin{tikzcd} Cart(\mathcal{E}) \times_{\mathcal{B}} \mathcal{E} \arrow[rr, shift left, "\widetilde{prod}"] \arrow[rr, phantom, "\perp"] \arrow[dr, "|p| \times_{\mathcal{B}} p"'] & & Cart(\mathcal{E}) \times_{\mathcal{B}} \mathcal{E} \arrow[ll, shift left, "\widetilde{exp}"] \arrow[dl, "|p| \times_{\mathcal{B}} p"] \\ & \mathcal{B} & \end{tikzcd}$$

A fibred terminal object for p is given by a fibred right adjoint to $p : p \to id_{\mathcal{B}}$ over $\mathcal{B}$. Let $p : \mathcal{E} \to \mathcal{B}$ be a fibration and $F : \mathcal{C} \to \mathcal{B}$ a functor; the pullback F^*p of p along F in *Cat* is a fibration, which is cloven or split when p is cloven or split [Jac91, 1.1.3]. Note that, given two copies $F : \mathcal{A} \to \mathcal{C}$ and $G : \mathcal{B} \to \mathcal{C}$ of the theory of functors, there is a straightforward axiomatization in finite limit formulae of the pullback F^*G. When $q : \mathcal{D} \to \mathcal{B}$ is a fibration, we write $q \times_{\mathcal{B}} p$ for the fibration $(q^*p) \circ q : \mathcal{D} \times_{\mathcal{B}} \mathcal{E} \to \mathcal{B}$. Fibred products are given by a fibred right adjoint *prod* to the obvious diagonal functor over $\mathcal{B}$. Similarly, let $|p| : Cart(\mathcal{E}) \to \mathcal{B}$ be restriction of p to the category $Cart(\mathcal{E})$ whose objects are the same as $\mathcal{E}$ and whose morphisms are the cartesian, and let $\widetilde{prod} : Cart(\mathcal{E}) \times_{\mathcal{B}} \mathcal{E} \to Cart(\mathcal{E}) \times_{\mathcal{B}} \mathcal{E}$ map (A, B) to $(A, prod(A, B))$; fibred exponentials are given by a fibred right adjoint $\widetilde{exp}$ to $\widetilde{prod}$ [Jac91].

Generic objects. A generic object for a fibration $p : \mathcal{E} \to \mathcal{B}$ is an object $\top$ of $\mathcal{E}$ such that there exists a cartesian morphism $X \to \top$ for each X in $\mathcal{E}$. If p is split, a generic object is given by an object Ω of $\mathcal{B}$ and an equivalence $\ulcorner\_\urcorner : dom_{\Omega} \to |p|$, where $|p| : Split(\mathcal{E}) \to \mathcal{B}$ is the restriction of p to the category $Split(\mathcal{E})$ whose objects are the same as $\mathcal{E}$ and whose morphisms are the ones of the splitting [Jac91]. In this case, $\top = \ulcorner id_{\Omega} \urcorner$ is generic.

Fibrations with a generic object are used to interpret type theories with type variables. A minimal example is simply typed λ-calculus with judgements of the form $A : type \vdash A \to A : type$.

$\mathcal{D}$-products. The following construction endows a fibration $p : \mathcal{E} \to \mathcal{B}$ with right adjoints to reindexing functors along cartesian projections satisfying the Beck-Chevalley condition. In 4.3.4, we called such adjoints "$\mathcal{D}$-products," where $\mathcal{D}$ is the comprehension category of cartesian projections. In [See87], Seely calls *weak completeness* the corresponding notion in the world of indexed categories.

Let $\mathcal{B}$ have binary products. Let $\mathcal{D}$ be the category whose objects are projections $a \times b \to b$ and whose arrows are pullbacks:

$$\begin{tikzcd} a \times b \arrow[r, "id \times f"] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & a \times c \arrow[d] \\ b \arrow[r, "f"'] & c \end{tikzcd}$$

that we write as pairs (a, f). Hence, $\mathcal{D}_O = \mathcal{B}_O \times \mathcal{B}_O$ and $\mathcal{D}_A = \mathcal{B}_O \times \mathcal{B}_A$. Let $D : \mathcal{D} \to \mathcal{B}$ be the domain functor $(a, f) \mapsto id_a \times f$ and $C : \mathcal{D} \to \mathcal{B}$ the codomain functor $(a, f) \mapsto f$. Let $\mathcal{D}_0$ and $\mathcal{D}_1$ be the vertices of the pullbacks of p respectively along D and C. The functor $P : \mathcal{D}_1 \to \mathcal{D}_0$ mapping (f, x), with $f : a \times px \to px$, to (f, f^*x) is a fibred functor $C^*p \to D^*p$ over $\mathcal{B}$. Then, p has $\mathcal{D}$-products if and only if P has fibred right adjoint [Jac91].

6.7 Models of HML

To simplify the treatment of semantics, we consider only the equational fragment of HML, as the full type theory can be handled similarly. Therefore, we do not include predicates in the signatures, we allow only equations as formulae and we consider only truth judgements $\Gamma; \Delta \vdash_{\Sigma,\mathcal{T}} \phi$ with empty Δ. Moreover, since universal quantification is not available anymore, we allow theories to contain open equations ϕ as axioms and require that the free variables of ϕ are in $dom(\Gamma)$ as side condition to the rule (axiom).

Below, when talking about an HML theory, we shall always understand an equational one.

This fragment of HML is similar to the polymorphic lambda calculus PLC, mentioned in section 6.3. There is a correspondence between theories of PLC and categorical structures called PL-categories [See87]. A PL category $(F, \mathcal{B})$ consists of a category $\mathcal{B}$ with finite products and a distinguished object Ω such that $(\_ \times \Omega) \dashv (\_)^{\Omega}$, and an indexed category $F : \mathcal{B}^{op} \to \mathit{Cat}$ satisfying certain conditions [See87, Def. 2.1]. A type τ of PLC has a double interpretation in $(F, \mathcal{B})$: it is interpreted as a morphism $A \to \Omega$ of $\mathcal{B}$ in a judgement $\tau : \Omega$ and as an object of FA in a judgement $M : \tau$. To make this consistent, PL categories require an isomorphism:

$$hom(A, \Omega) \xrightarrow{\ulcorner \_ \urcorner} obj(FA) \tag{6.5}$$

such that $Ff\ulcorner g \urcorner = \ulcorner g \circ f \urcorner$. This condition forces F to be a functor (rather than a pseudofunctor) and therefore it corresponds to $\ulcorner \_ \urcorner$ being *natural* in A.

Remark. PL-categories can otherwise be described as split fibrations $p : \mathcal{E} \to \mathcal{B}$ with extra structure. In particular, the natural isomorphism (6.5) corresponds to p having a generic object. Note that, if $p : \mathcal{E} \to \mathcal{B}$ is obtained by applying the Grothendieck construction [Jac91] to the indexed category of a PL-category, $\top$ is the object $(\ulcorner id \urcorner, \Omega)$ in the fibre over Ω. In fact, for any object (X, A) of the total category $\mathcal{E}$, there is a cartesian morphism $(f, id) : (X, A) \to (\ulcorner id \urcorner, \Omega)$, where $X = \ulcorner f \urcorner$.

$\square$

In the previous section, we saw that a generic object for a split fibration amounts to an equivalence $\mathcal{B}/\Omega \cong \mathit{Split}(\mathcal{E})$ over $\mathcal{B}$. To model the lifting of HML types to type schemes, where no such equivalence is required, we only ask for a functor $F : \mathcal{B}/\Omega \to \mathcal{E}$ over $\mathcal{B}$. Proposition 6.7.1 below, ensures that such a functor behaves well with respect to reindexing, so that interpretation may commute with substitution.

Let p be a fibration and let A be an object of $\mathcal{B}$. A fibred version of the Yoneda lemma [Jac91, 1.1.9] asserts that there is an equivalence of categories $\mathcal{E}_A \cong \mathit{Fib}(\mathcal{B})(dom_A, p)$, natural in a suitable sense, where $\mathit{Fib}(\mathcal{B})$ is the category of fibrations with base $\mathcal{B}$ and functors over $\mathcal{B}$. In particular, the equivalence maps an object X over A to the functor $(\_)^*X : \mathcal{B}/A \to \mathcal{E}$ and, conversely, a functor $F : \mathcal{B}/A \to \mathcal{E}$ to $F(Id_A)$.

Proposition 6.7.1 *Let $p : \mathcal{E} \to \mathcal{B}$ be a fibration and let $\xrightarrow{f}\xrightarrow{g} A$ be morphisms in $\mathcal{B}$; if $F : \mathcal{B}/A \to \mathcal{E}$ is a functor over $\mathcal{B}$, then $f^*F(g) \cong F(g \circ f)$.*

Proof. $f^*F(g) \cong f^*g^*F(Id_A) \cong (g \circ f)^*F(Id_A) \cong F(g \circ f)$. □

In the case of split fibrations, fibred Yoneda yields an isomorphisms between $\mathcal{E}_A$ and $\mathit{Fib}_{split}(\mathcal{B})(dom_A, p)$, and the equivalence of proposition 6.7.1 turns into an equality. This isomorphism says that types can be coherently lifted to type schemes by just distinguishing an object in the total category.

Definition 6.7.2 *A* $\lambda\overline{\omega}$-category *is a split fibration with fibred cartesian closed structure over a cartesian closed base, a distinguished object in the total category and right adjoints to reindexing along cartesian projections satisfying the Beck-Chevalley condition.*

We use $\lambda\overline{\omega}$-categories to model HML theories. Note that, besides the decay of the generic object to a mere "distinguished," $\lambda\overline{\omega}$-categories differ from PL-categories in that they support no Σ-types. On the other hand, they have all exponentials in the base to model polymorphism of higher order with full generality.

An *interpretation* of an HML theory $\mathcal{T}$ over a signature Σ in a $\lambda\overline{\omega}$-category $p : \mathcal{E} \to \mathcal{B}$ is a map defined on the derivable judgements of $\mathcal{T}$ as follows (we write only the relevant part of a judgement inside the semantic brackets when the rest is understood):

Kinds. A Σ_K-*structure* in p is an assignment of an object $[\![K]\!]$ of $\mathcal{B}$ to each constant K : *kind* in Σ. Given such a structure, the kinds of $\mathcal{T}$ are interpreted as objects of $\mathcal{B}$, where 1, $\times$ and $\Rightarrow$ have the obvious meaning. Moreover, the *kind* interpretation $[\![\Gamma]\!]^K$ of a context Γ in $\mathcal{B}$ is: $[\![\emptyset]\!]^K = 1$, $[\![\Gamma, v : k]\!]^K = [\![\Gamma]\!]^K \times [\![k]\!]$ and $[\![\Gamma, x : \sigma]\!]^K = [\![\Gamma]\!]^K$.

If $(v_1 : k_1, \ldots v_n : k_n)$ is the context containing all the variables over kinds in Γ, we write $\Gamma\downarrow$ for the kind $k_1 \times \ldots \times k_n$; then, $[\![\Gamma]\!]^K = [\![\Gamma\downarrow]\!]$.

Operators. A Σ_{KO}-*structure* is a Σ_K-structure together with an assignment of a global elemet $[\![C]\!] : 1 \to [\![k]\!]$ in $\mathcal{B}$ to each $C : k$ in Σ. Given a Σ_{KO}-structure, operators $\Gamma \vdash u : k$ are interpreted as morphisms $[\![u]\!] : [\![\Gamma]\!]^K \to [\![k]\!]$ in $\mathcal{B}$. Ignoring variables ranging over schemes, interpretation of operators in $\mathcal{B}$ is just as usual for simply typed lambda calculi in cartesian closed categories.

All the rules for deriving operator equality are sound with respect to this interpretation; that is: if $\Gamma \vdash u_1 = u_2 : k$ is derivable, then $[\![\Gamma \vdash u_1 : k]\!] = [\![\Gamma \vdash u_2 : k]\!]$.

Type schemes. A Σ_{KOS}-*structure* is a Σ_{KO}-structure together with an assignment of an object $[\![S]\!]$ in the fibre over $[\![k]\!]$ to each $S : k \Rightarrow \mathit{scheme}$ in Σ. We use the same name for the functor

$$\begin{tikzcd} \mathcal{B}/[\![k]\!] \arrow[rr, "{[\![S]\!]}"] \arrow[dr, "dom"'] & & \mathcal{E} \arrow[dl, "p"] \\ & \mathcal{B} & \end{tikzcd}$$

over $\mathcal{B}$ which corresponds to the object $[\![S]\!]$ by the isomorphism introduced after proposition 6.7.1.

Given a Σ_{KOS}-structure, a type scheme $\Gamma \vdash \sigma\ \mathit{scheme}$ is interpreted as an object $[\![\sigma]\!]$ in the fibre over $[\![\Gamma]\!]^K$. If $\Gamma \vdash u : k$ and $S : k \Rightarrow \mathit{scheme}$, $[\![S(u)]\!] = [\![S]\!][\![u]\!]$. If $\Gamma \vdash u : \Omega$, then $[\![\Gamma \vdash u\ \mathit{scheme}]\!] = [\![u]\!]^*\top$, where $\top$ is the distinguished object of p. Schemes of the form 1, $\sigma_1 \times \sigma_2$ and $\sigma_1 \Rightarrow \sigma_2$ are interpreted in the obvious way using the fibred cartesian closed structure of p. If $\Gamma, v : k \vdash \sigma\ \mathit{scheme}$, $[\![\Pi v : k.\,\sigma]\!] = \Pi_{\Gamma,k}[\![\sigma]\!]$, where $\Pi_{\Gamma,k}$ is right adjoint to the reindexing functor along the projection $\pi_{\Gamma,k} : [\![\Gamma]\!]^K \times [\![k]\!] \to [\![\Gamma]\!]^K$. As for operators, if $\Gamma \vdash \sigma_1 = \sigma_2$ is derivable, then $[\![\sigma_1]\!] = [\![\sigma_2]\!]$.

The *type scheme* interpretation of a context Γ is an object $[\![\Gamma]\!]^S$ of $\mathcal{E}$ over $[\![\Gamma]\!]^K$ defined as follows: $[\![\emptyset]\!]^S = 1$, $[\![\Gamma, v : k]\!]^S = \pi^*_{\Gamma,k}[\![\Gamma]\!]^S$, $[\![\Gamma, x : \sigma]\!]^S = [\![\Gamma]\!]^S \times [\![\sigma]\!]$. We write $\Gamma \uparrow$ for the product $\sigma_1 \times \ldots \times \sigma_n$ of all schemes σ_i such that $x_i : \sigma_i$ is in Γ. Since the reindexing functors preserve products, we have that $[\![\Gamma]\!]^S = [\![\Gamma \uparrow]\!]$.

Terms. A Σ-*structure* is a Σ_{KOS}-structure together with an assignment of a morphism $[\![c]\!] : 1 \to [\![\sigma]\!]$ to each $c : \sigma$ in Σ. Given a Σ-structure, a term $\Gamma \vdash e : \sigma$ is interpreted as a morphism $[\![e]\!] : [\![\Gamma]\!]^S \to [\![\sigma]\!]$ in the fibre over $[\![\Gamma]\!]^K$. Variables, constants, $* : 1$, pairing, projection, λ-abstraction and λ-application are interpreted as usual using the cartesian closed structure in the fibres. If $\Gamma, v : k \vdash e : \sigma$, then $[\![\Lambda v : k.\, e]\!] = [\![e]\!]^\dagger$, where $f^\dagger$ is the transpose of f for the adjunction $\pi^*_{\Gamma,k} \dashv \Pi_{\Gamma,k}$. If $\Gamma \vdash e : (\Pi v : k.\, \sigma)$ and $\Gamma \vdash u : k$, then $[\![e[u]]\!] = \langle id, [\![u]\!]\rangle^*[\![e]\!]^{\dagger\dagger}$, where $(\_)^{\dagger\dagger}$ is inverse to $(\_)^\dagger$.

Equations. An equation $\Gamma \vdash_{\Sigma,\mathcal{T}} e_1 =_\sigma e_2$ *holds* in (or is satisfied by) a $\lambda\varpi$-category with a Σ-structure if $[\![e_1]\!] = [\![e_2]\!]$.

$\square$

Since the interpretation of a theory $\mathcal{T}$ over Σ in a $\lambda\varpi$-category p is completely determined by a Σ-structure $\mathcal{S}$ in p, we can view interpretations as pairs $(p, \mathcal{S})$. A *model* of $\mathcal{T}$ is an interpretation satisfying the axioms. If $\Gamma \vdash_{\Sigma,\mathcal{T}} e_1 =_\sigma e_2$ holds in all models of $\mathcal{T}$, we write:

$$\Gamma \models_{\Sigma,\mathcal{T}} e_1 =_\sigma e_2.$$

Theorem 6.7.3 (Soundness) $\Gamma \vdash_{\Sigma,\mathcal{T}} e_1 =_\sigma e_2$ *implies* $\Gamma \models_{\Sigma,\mathcal{T}} e_1 =_\sigma e_2$.

Proof. As usual, we have to show that the inference rules preserve truth. All this is quite standard in categorical models of type theory, where each pair of β and η-rules correspond to suitable adjunctions. We only show $\Pi.\beta$ as example:

$$\begin{aligned}
&[\![\Gamma \vdash [u/v]e : [u/v]\sigma]\!] \;= \\
&\langle id, [\![u]\!]\rangle^*[\![\Gamma, v : k \vdash e : \sigma]\!] \;= \\
&\langle id, [\![u]\!]\rangle^*([\![\Gamma, v : k \vdash e : \sigma]\!]^{\dagger})^{\dagger\dagger} \;= \\
&\langle id, [\![u]\!]\rangle^*[\![\Gamma \vdash \Lambda v : k.\, e : (\Pi v : k.\, \sigma)]\!]^{\dagger\dagger} \;= \\
&[\![\Gamma \vdash (\Lambda v : k.\, e)[u] : [u/v]\sigma]\!].
\end{aligned}$$

$\square$

Any HML theory $\mathcal{T}$ has a syntactic model $(p_{\mathcal{T}} : \mathcal{E}_{\mathcal{T}} \to \mathcal{B}_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}})$. Objects of $\mathcal{B}_{\mathcal{T}}$ are kinds. Morphisms $k_1 \to k_2$ of $\mathcal{B}_{\mathcal{T}}$ are sequents $v : k_1 \vdash u : k_2$ modulo the equations of $\mathcal{T}$. Since the operators used to construct $\mathcal{B}_{\mathcal{T}}$ have at most one free variable v, we can identify them up to renaming of v. Identities are $v : k \vdash v : k$ and composition is given by substitution. $\mathcal{B}_{\mathcal{T}}$ has cartesian closed structure given by the obvious operations on kinds and operators.

Objects of $\mathcal{E}_{\mathcal{T}}$ are sequents $v : k \vdash \sigma\ \mathit{scheme}$ modulo the equations of $\mathcal{T}$. Morphisms $(v_1 : k_1 \vdash \sigma\ \mathit{scheme}) \to (v_2 : k_2 \vdash \tau\ \mathit{scheme})$ are pairs of sequents

$$((v_1 : k_1 \vdash u : k_2), (v_1 : k_1, x : \sigma \vdash e : [u/v_2]\tau))$$

modulo the equations of $\mathcal{T}$. As for operators, schemes and terms are identified up to renaming of free variables. When the rest is understood, we write (k, σ) for the objects of $\mathcal{E}_{\mathcal{T}}$ and (u, e) for the arrows. Identities and composition in $\mathcal{E}_{\mathcal{T}}$ are obvious.

It is easy to see that $p_{\mathcal{T}}$ mapping (k, σ) to k and (u, e) to u is a fibration whose cartesian morphisms are the ones where e is an iso. Moreover, let $v_1 : k_1 \vdash u : k_2$ and let $v_2 : k_2 \vdash \sigma\ \mathit{scheme}$; the cleavage obtained by taking $(u, id_{[u/v_2]\sigma})$ as chosen cartesian morphism over u with target (k_2, σ) makes $p_{\mathcal{T}}$ split.

This fibration has a distinguished object $v : \Omega \vdash v : \mathit{scheme}$ and fibred cartesian closed structure given by the obvious operations on schemes and terms. As for $\mathcal{D}$-products, $\Pi_{k_1,k_2}(w : k_1 \times k_2, \sigma) \stackrel{\text{def}}{=} (v_1 : k_1, \Pi v_2 : k_2.\, [\langle v_1, v_2\rangle/w]\sigma)$ is right adjoint to reindexing $\pi^*_{k_1,k_2}(v_1 : k_1, \tau) = (w : k_1 \times k_2, [\pi_1(w)/v_1]\tau)$. In fact, the derivable rule scheme

$$(\Pi\mathrm{I}\_\mathcal{T}) \qquad \frac{w : k_1 \times k_2, x : [\pi_1(w)/v_1]\tau \vdash e : \sigma}{v_1 : k_1, x : \tau \vdash \Lambda v_2 : k_2.\, [\langle v_1, v_2\rangle/w]e : (\Pi v_2 : k_2.\, [\langle v_1, v_2\rangle/w]\sigma)}$$

yields a bijection $(\mathcal{E}_\mathcal{T})_{k_1\times k_2}(\pi^*_{k_1,k_2}\tau, \sigma) \to (\mathcal{E}_\mathcal{T})_{k_1}(\tau, \Pi_{k_1,k_2}\sigma)$, natural in σ and τ, with inverse:

$$(\Pi\mathrm{E}\_\mathcal{T}) \qquad \frac{v_1 : k_1, x : \tau \vdash e : (\Pi v_2 : k_2.\, [\langle v_1, v_2\rangle/w]\sigma)}{w : k_1 \times k_2, x : [\pi_1(w)/v_1]\tau \vdash [\pi_1(w)/v_1]e[\pi_2(w)] : \sigma} \;.$$

A canonical Σ-structure $\mathcal{S}_\mathcal{T}$ can be found in $p_\mathcal{T}$ by interpreting each constant as itself. In particular, a constant $S : k \Rightarrow \mathit{scheme}$ is interpreted as $v : k \vdash S(v)\ \mathit{scheme}$.

Proposition 6.7.4 *Interpretation in $(p_\mathcal{T}, \mathcal{S}_\mathcal{T})$ is such that:*

$$\begin{aligned}
&[\![k]\!] = k\\
&[\![\Gamma]\!]^K = \Gamma\downarrow\\
&[\![\Gamma \vdash u : k]\!] \;=\; w : \Gamma\downarrow \vdash [\pi_i(w)/v_i]u : k\\
&[\![\Gamma \vdash \sigma\ \mathit{scheme}]\!] \;=\; w : \Gamma\downarrow \vdash [\pi_i(w)/v_i]\sigma\ \mathit{scheme}\\
&[\![\Gamma]\!]^S = w : \Gamma\downarrow \vdash [\pi_i(w)/v_i]\Gamma\uparrow\\
&[\![\Gamma \vdash e : \sigma]\!] \;=\; w : \Gamma\downarrow, x : [\pi_i(w)/v_i]\Gamma\uparrow \vdash [\pi_i(x)/x_i]\,[\pi_i(w)/v_i]e : [\pi_i(w)/v_i]\sigma.
\end{aligned}$$

Proof. It is a long induction on the derivation of judgements. We show one example for kinds (v) and one for terms $(\Pi\mathrm{I})$. Let $\Gamma\downarrow = k_1 \times \ldots \times k_n$.

(v) $\quad [\![\Gamma \vdash v_j : k_j]\!] =$
$(w : \Gamma\downarrow \vdash \pi_j(w) : k_j) =$
$(w : \Gamma\downarrow \vdash [\pi_i(w)/v_i]v_j : k_j)$.

Let $w : \Gamma\downarrow$ and $u : k$ be operatos; we write $\pi_i\langle w, u\rangle = \pi_i(w)$ and $\pi\langle w, u\rangle = u$ the projections of $\Gamma\downarrow \times k$. By inductive hypothesis, $[\![\Gamma, v : k \vdash e : \sigma]\!]$ is equal to $z : \Gamma\downarrow \times k, x : [\pi_i(z)/v_i]\Gamma\uparrow \vdash [\pi_i(x)/x_i]\,[\pi_i(z)/v_i]\,[\pi(z)/v]e : [\pi_i(z)/v_i]\,[\pi(z)/v]\sigma$; then, by $(\Pi\mathrm{I}\_\mathcal{T})$ we have:

$$
\begin{aligned}
(\Pi\mathrm{I})\quad & [\![\Gamma \vdash \Lambda\, v : k.\, e : (\Pi\, v : k.\, \sigma)]\!] = \\
& w : \Gamma\downarrow, x : [\pi_i(w)/v_i]\Gamma\uparrow \vdash\ \Lambda\, v : k.\, [\langle w, v\rangle/z]\, [\pi_i(x)/x_i]\, [\pi_i(z)/v_i]\, [\pi(z)/v]e : \\
& \qquad\qquad\qquad\qquad\qquad\qquad \Pi\, v : k.\, [\langle w, v\rangle/z]\, [\pi_i(z)/v_i]\, [\pi(z)/v]\sigma = \\
& w : \Gamma\downarrow, x : [\pi_i(w)/v_i]\Gamma\uparrow \vdash\ \Lambda\, v : k.\, [\pi_i(x)/x_i]\, [\pi_i(w)/v_i]e : (\Pi\, v : k.\, [\pi_i(w)/v_i]\sigma) = \\
& w : \Gamma\downarrow, x : [\pi_i(w)/v_i]\Gamma\uparrow \vdash\ [\pi_i(x)/x_i]\, [\pi_i(w)/v_i]\Lambda\, v : k.\, e : [\pi_i(w)/v_i]\Pi\, v : k.\, \sigma.
\end{aligned}
$$

□

Using the above proposition, it is trivial to verify that the axioms of $\mathcal{T}$ are satisfied by $(p_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}})$ because of their immediate derivability; hence $(p_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}})$ is a model.

Theorem 6.7.5 (Completeness) $\Gamma \models_{\Sigma,\mathcal{T}} e_1 =_\sigma e_2$ *implies* $\Gamma \vdash_{\Sigma,\mathcal{T}} e_1 =_\sigma e_2$.

Proof. If $\Gamma \models_{\Sigma,\mathcal{T}} e_1 =_\sigma e_2$, then $\Gamma \vdash_{\Sigma,\mathcal{T}} e_1 =_\sigma e_2$ must hold in $(p_{\mathcal{T}}, \mathcal{S}_{\mathcal{T}})$. Since semantic identity in the $\lambda\overline{\omega}$-category $p_{\mathcal{T}}$ corresponds to provable equality, the judgement $w : \Gamma\downarrow, x : [\pi_i(w)/v_i]\Gamma \uparrow\vdash [\pi_i(x)/x_i]\, [\pi_i(w)/v_i]e_1 =_{[\pi_i(w)/v_i]\sigma} [\pi_i(x)/x_i]\, [\pi_i(w)/v_i]e_2$ must be derivable. Then, the result is obtained by substituting $\Gamma \vdash \langle v_1, \ldots v_n\rangle : \Gamma\downarrow$ for w and $\Gamma \vdash \langle x_1, \ldots x_m\rangle : \Gamma\uparrow$ for x. □

Remark. In [See87], a relative interpretation of PLC theories is defined semantically to be an interpretation of one theory in the term model of another. Similarly, the syntactic and semantic notions of relative interpretation of HML coincide: any interpretation of an HML theory $\mathcal{T}_1$ in a syntactic $\lambda\overline{\omega}$-category $p_{\mathcal{T}_2}$ factorizes through the canonical interpretation of $\mathcal{T}_2$ in $p_{\mathcal{T}_2}$ via a unique relative interpretation $\mathcal{T}_1 \to \mathcal{T}_2$.

6.8 Syntactic presentation of constructors

In this section we give a characterization of the semantic constructors which can be presented as relative interpretations of HML theories, as in the example of section 6.4.

Outline of the section. Our first step is to get rid of the syntax by adopting an abstract, presentation invariant view of HML theories as $\lambda\varpi$-categories. To justify this step, we establish an equivalence between the category of HML theories and that of $\lambda\varpi$-categories (theorem 6.8.4), like is done in [See87] for PLC. Then, the models of a theory $\mathcal{T}$ are defined functorially as morphisms of $\lambda\varpi$-categories; this is justified by proposition 6.8.5. The category $Mod(\mathcal{T})$ of such models is shown to be equivalent to the category $[T, Sets]_{lex}$ of set valued models of a finite limit theory T, which is locally finitely presentable. Moreover, we show that the functors $Mod(\phi) : Mod(\mathcal{T}_1) \to Mod(\mathcal{T}_2)$ induced by a relative interpretation $\phi : \mathcal{T}_2 \to \mathcal{T}_1$ have left adjoints and that they preserve filtered colimits (theorem 6.8.10). This allows us to use the Gabriel-Ulmer duality to give an intrinsic characterization of functors of the form $Mod(\phi)$ (theorem 6.8.12).

Definition 6.8.1 *A* $\lambda\varpi$-functor $p_1 \to p_2$ *between* $\lambda\varpi$*-categories* $p_1 : \mathcal{E}_1 \to \mathcal{B}_1$ *and* $p_2 : \mathcal{E}_2 \to \mathcal{B}_2$ *is a morphism* (J, H) *of fibrations such that* $J : \mathcal{B}_1 \to \mathcal{B}_2$ *preserves the cartesian closed structure,* $H : \mathcal{E}_1 \to \mathcal{E}_2$ *preserves the distinguished object, the fibred cartesian closed structure and* $\mathcal{D}$*-products. We call* $\lambda\varpi$*-Cat the category of small* $\lambda\varpi$*-categories and* $\lambda\varpi$*-functors.*

We make the simplifying assumption that all the structure is preserved "on the nose," although this is not crucial for the development of the theory. In particular, preserving $\mathcal{D}$-products means that, for all $\pi_A : B \times A \to B$ in $\mathcal{B}$, $H_B \Pi_A = \Pi_B H_{B\times A}$.

Proposition 6.8.2 *Any model of an HML theory* $\mathcal{T}$ *in a* $\lambda\varpi$*-category* p *factorizes through the canonical interpretation in* $p_{\mathcal{T}}$ *via a unique* $\lambda\varpi$*-functor* $p_{\mathcal{T}} \to p$.

We omit the proof of this proposition, which only involves routine calculations.

In order to establish an equivalence between $\lambda\varpi$-Cat and the category of HML theories, we must make our notion of theory more general. To represent in HML the information contained in a $\lambda\varpi$-category, we must allow signatures to contain infinitely many constants (of finite arity) and theories to contain infinitely many axioms. Moreover, in order to capture the structure of the base category syntactically, we must allow the axioms of a theory to include equations between *kinds* and *operators*. We call *HMLC* the category of such theories and relative interpretations.

Now we can give a construction (dual to that of syntactic models) to produce an HML theory from a $\lambda\varpi$-category $p : \mathcal{E} \to \mathcal{B}$. Let Σ_p be the following signature: the objects of $\mathcal{B}$ are constant kinds; the arrows $C : K_1 \to K_2$ of $\mathcal{B}$ are constant operators $C : K_1 \Rightarrow K_2$; objects S in $\mathcal{E}_K$ are constant schemes $S : K \Rightarrow \mathit{scheme}$; morphisms $c : S_1 \to S_2$ of $\mathcal{E}$ above $C : K_1 \to K_2$ are constant terms $c : (\Pi\, v : K_1.\, S_1(v) \Rightarrow S_2(C(v)))$. We call $\mathcal{T}_p$ the theory over Σ_p whose axioms are equations $C_2(C_1(v)) = C_3(v)$, for all $C_2 \circ C_1 = C_3$ in $\mathcal{B}$, $c_2[C_1(v)](c_1[v](x)) = c_3[v](x)$, for all $c_2 \circ c_1 = c_3$ in $\mathcal{E}$ and $C_1 = p(c_1)$, and all the equations describing the categorical structure of p, e.g. $K_1 \Rightarrow K_2 = K_3$ for all $K_2^{K_1}$ in $\mathcal{B}$, etc. The canonical interpretation mapping each constant into itself makes of p a model of $\mathcal{T}_p$.

Proposition 6.8.3 *Any model of an HML theory $\mathcal{T}$ in a $\lambda\varpi$-category p factorizes through the canonical interpretation of $\mathcal{T}_p$ in p via a unique relative interpretation $\mathcal{T} \to \mathcal{T}_p$.*

Let p be a model of $\mathcal{T}$ and let $[\![\_]\!]$ be the interpretation; the relative interpretation given by the above proposition maps constant kinds K to $[\![K]\!]$, constant operators C to $[\![C]\!](*)$, constant type scheme S to $[\![S]\!](v)$ and constant terms c to $[\![c]\!](*)$.

Theorem 6.8.4 *HMLC* $\cong$ *$\lambda\varpi$-Cat.*

Proof. The function mapping $\lambda\varpi$-categories p to HML theories $\mathcal{T}_p$ can be extended to a functor $\mathcal{T}_{(\_)} : \lambda\varpi\text{-}\mathit{Cat} \to \mathit{HMLC}$ mapping a $\lambda\varpi$-functor $F : p_1 \to p_2$ to the relative interpretation $\mathcal{T}_{p_1} \to \mathcal{T}_{p_2}$ obtained by applying the above proposition to $\mathcal{T}_{p_1} \xrightarrow{[\![\_]\!]} p_1 \xrightarrow{F} p_2$. Similarly, the function mapping HML theories $\mathcal{T}$ to their syntactic models $p_{\mathcal{T}}$ extends to a functor $p_{(\_)} : \mathit{HMLC} \to \lambda\varpi\text{-}\mathit{Cat}$ mapping a relative interpretation $\phi : \mathcal{T}_1 \to \mathcal{T}_2$ to the $\lambda\varpi$-functor $p_{\mathcal{T}_1} \to p_{\mathcal{T}_2}$ obtained by applying proposition 6.8.2 to $\mathcal{T}_1 \xrightarrow{\phi} \mathcal{T}_2 \xrightarrow{[\![\_]\!]} p_{\mathcal{T}_2}$. Moreover, from propositions 6.8.2 and 6.8.3, we obtain a bijection between the homsets $\mathit{HMLC}(\mathcal{T}, \mathcal{T}_p)$ and $\lambda\varpi\text{-}\mathit{Cat}(p_{\mathcal{T}}, p)$, natural in $\mathcal{T}$ and p. Hence $p_{(\_)} \dashv \mathcal{T}_{(\_)}$. It is easy to see that $\eta : p \to p_{\mathcal{T}_p}$ is a natural isomorphism whose inverse is the $\lambda\varpi$-functor associated with the canonical interpretation of $\mathcal{T}_p$ in p by proposition 6.8.2. Similarly, $\epsilon : \mathcal{T}_{p_\mathcal{T}} \to \mathcal{T}$ is a natural isomorphism whose inverse is the relative interpretation associated with the canonical interpretation of $\mathcal{T}$ in $p_\mathcal{T}$ by proposition 6.8.3. This shows that $p_{(\_)} \dashv \mathcal{T}_{(\_)}$ is an adjoint equivalence between *HMLC* and $\lambda\varpi$-*Cat*. $\square$

Before the syntax can abandon the scene, a suitable functorial notion of category of models of a HML theory must be established. Let Σ be an HML signature; a Σ-*homomorphism* $(p, \mathcal{S}) \to (q, \mathcal{R})$ between interpretations of a theory $\mathcal{T}$ over Σ is a $\lambda\varpi$-functor $(J, K) : p \to q$ such that, for all constant kinds K, operators C, parametric schemes S and terms c in Σ, $J[\![K]\!]_\mathcal{S} = [\![K]\!]_\mathcal{R}$, $J[\![C]\!]_\mathcal{S} = [\![C]\!]_\mathcal{R}$, $H[\![S]\!]_\mathcal{S} = [\![S]\!]_\mathcal{R}$ and $H[\![c]\!]_\mathcal{S} = [\![c]\!]_\mathcal{R}$. The category of models of a theory $\mathcal{T}$ over Σ and Σ-homomorphisms form a category $Mod(\mathcal{T})$.

Remark. Using the notion of Σ-homomorphism, proposition 6.8.2 can be rephrased as follows: let $(p, \mathcal{S})$ be a model of a theory $\mathcal{T}$ over Σ; there exists a unique Σ-homomorphism $(p_\mathcal{T}, \mathcal{S}_\mathcal{T}) \to (p, \mathcal{S})$; that is, the syntactic model $(p_\mathcal{T}, \mathcal{S}_\mathcal{T})$ is initial in the category $Mod(\mathcal{T})$.

Proposition 6.8.5 *$Mod(\mathcal{T}) \cong p_\mathcal{T}/\lambda\varpi$-Cat.*

The object map of this equivalence is given by proposition 6.8.2. Let Σ be the signature of $\mathcal{T}$, it is easy to verify that there is a one-to-one correspondence between Σ-homomorphisms $(p, \mathcal{S}) \to (q, \mathcal{R})$ and commuting triangles

$$\begin{tikzcd} & p_\mathcal{T} \arrow[dl, "\mathcal{S}"'] \arrow[dr, "\mathcal{R}"] & \\ p \arrow[rr] & & q. \end{tikzcd}$$

In view of proposition 6.8.4 and 6.8.5, we write $Mod(p)$ for the category $p/\lambda\varpi$-*Cat* of models of the "$\lambda\varpi$-theory" p. From now on, the syntax of *HML* will not concern us anymore.

In section 6.6, we wrote a finite limit theory of categories and invoked Keane's result to conclude that there exists a lex category T_{Cat} such that $Cat \cong [T_{Cat}, Sets]_{lex}$. The same argument applies to $\lambda\overline{\varpi}$-categories, whose structure was also described in finite limit sentences: there exists a lex category $T_{\lambda\overline{\varpi}}$ such that $\lambda\overline{\varpi}\text{-}Cat \cong [T_{\lambda\overline{\varpi}}, Sets]_{lex}$. Hence, from theorem 6.5.4, it follows that the category $\lambda\overline{\varpi}\text{-}Cat$ is locally finitely presentable. Since any slice of a locally λ-presentable category is locally λ-presentable [AR94, 1.57], and hence any slice of an lfp category is lfp, the following proposition holds:

Proposition 6.8.6 *The category $Mod(p)$ is locally finitely presentable.*

Digression on accessible categories. Closure under slicing does not hold for λ-*accessible* categories [AR94], that is if the condition of co-completeness is removed: a slice of a λ-accessible category may not be λ-accessible, although it would be accessible for some $\lambda' > \lambda$.

$\square$

Next step towards our characterization theorem is to extend the map Mod to a functor $\lambda\overline{\varpi}\text{-Cat}^{op} \to LFP$.

Proposition 6.8.7 *Let $h : A \to B$ be a morphism in a category $\mathcal{C}$ with pushouts; the functor $h/\mathcal{C} : B/\mathcal{C} \to A/\mathcal{C}$ has left adjoint.*

Proof. Let $F : A/\mathcal{C} \to B/\mathcal{C}$ be the functor mapping $f : A \to C$ to the pushout of $\xleftarrow{f} \cdot \xrightarrow{h}$ as in the figure.

$$\begin{tikzcd}
A \arrow[r, "h"] \arrow[d, "f"'] & B \arrow[d, "F(f)"'] \arrow[ddr, bend left, "g"] & \\
C \arrow[r] \arrow[drr, bend right] & \cdot \arrow[ul, phantom, "\ulcorner", very near start] \arrow[dr, dashed] & \\
 & & D
\end{tikzcd}$$

Given $g : B \to D$, the bijection $(B/\mathcal{C})(Ff, g) \cong (A/\mathcal{C})(f, g \circ h) = (A/\mathcal{C})(f, (h/\mathcal{C})g)$, natural in f and g, yields an adjunction $F \dashv h/\mathcal{C}$. $\square$

Remember that a functor $F : \mathcal{A} \to \mathcal{B}$ creates filtered colimits when, for every filtered category $\mathcal{J}$ and functor $G : \mathcal{J} \to \mathcal{A}$ such that $F \circ G$ has a colimit $\nu : F \circ G \dot{\to} C$, there exists a unique cone $\sigma : G \dot{\to} A$ such that $F\sigma = \nu$ and moreover σ is a colimit.

Let X be an object in a category $\mathcal{C}$; we write $cod_X : X/\mathcal{C} \to \mathcal{C}$ the functor mapping arrows $X \to Y$ to their codomain Y. We drop the subscript when no confusion arises.

Lemma 6.8.8 *Let X be an object in a category $\mathcal{C}$; the functor cod_X creates filtered colimits.*

Proof. Let J be a filtered category, let $F : J \to X/\mathcal{C}$ be a diagram in $X/\mathcal{C}$ and let $\nu : cod_X \circ F \dot{\to} C$ be a colimit in $\mathcal{C}$. Since J is filtered, the composites $\nu_j \circ F(j)$, for all objects j of J, are equal to the same arrow $g : X \to C$. As an object of $X/\mathcal{C}$, g is the vertex of cone on F with components $\nu_j : F(j) \to g$; moreover, there can be no other cone μ such that $cod_X \circ \mu = \nu$. Using the universality of ν, this cone is shown to be a colimit in $X/\mathcal{C}$. $\square$

Proposition 6.8.9 *Let $f : X \to Y$ be an arrow in a category $\mathcal{C}$ with filtered colimits; the functor $f/\mathcal{C} : Y/\mathcal{C} \to X/\mathcal{C}$ of precomposition with f preserves filtered colimits.*

Proof. Let ν be a colimiting cone over a diagram F in $Y/\mathcal{C}$. Since $\mathcal{C}$ has all filtered colimits and, by the above lemma, $cod_Y : Y/\mathcal{C} \to \mathcal{C}$ creates them, the cone $cod_Y \circ \nu$ must be colimiting in $\mathcal{C}$. Since $cod_Y \circ \nu = cod_X \circ (f/\mathcal{C}) \circ \nu$ and $cod_X : X/\mathcal{C} \to \mathcal{C}$ creates filtered colimits, it follows that $(f/\mathcal{C}) \circ \nu$ is a colimit in $X/\mathcal{C}$ as required. $\square$

Theorem 6.8.10 *Let $\phi : p \to q$ be a $\lambda\overline{\omega}$-functor; the functor $Mod(\phi) \stackrel{def}{=} \phi/\lambda\overline{\omega}\text{-}Cat : q/\lambda\overline{\omega}\text{-}Cat \to p/\lambda\overline{\omega}\text{-}Cat$ of precomposition with p is a morphism in LFP and hence the relative interpretation map Mod extends to a functor $\lambda\overline{\omega}\text{-}Cat^{op} \to LFP$.*

Proof. Since $\lambda\overline{\omega}$-*Cat* is lfp, hence complete, proposition 6.8.7 ensures that the functor $\phi/\lambda\overline{\omega}\text{-}Cat = Mod(\phi)$ has left adjoint, while proposition 6.8.9 ensures that it preserves filtered colimits. □

In view of the above theorem, we call *Mod* the *relative interpretation* functor.

In [BW85, 4.4], the finite limit theory generated by a finite limit sketch is obtained as the full subcategory of the category of models of the sketch whose objects are *representable*. Similarly, we obtain a functor $th : \lambda\overline{\omega}\text{-Cat} \to Lex$ by composing *Mod* with the functor $(\_)^{op}_{fp}$ of section 6.5, which returns the subcategory of finitely presentable objects of an lfp category: $th(p) = (p/\lambda\overline{\omega}\text{-Cat})^{op}_{fp}$.

Let α be the natural transformation $Mod \stackrel{\cdot}{\to} ([\_, Sets]_{lex} \circ th^{op})$ whose components are the equivalences of categories $Mod(p) \cong [(Mod(p))^{op}_{fp}, Sets]_{lex}$; the pair (th, α) forms a morphism of indexed lfp categories:

$$\begin{tikzcd}
Lex^{op} \arrow[rr, "{[_, Sets]_{lex}}"] & & LFP \\
\lambda\overline{\omega}\text{-}Cat^{op} \arrow[u, "th^{op}"] \arrow[urr, "Mod"'] \arrow[ur, phantom, "\Leftarrow\alpha"] & &
\end{tikzcd}$$

Let Mod_{lex} be the category whose objects are lex functors into *Sets* and whose morphisms $F \to G$, for $F : T_1 \to Sets$ and $G : T_2 \to Sets$, are pairs (H, ν), where the functor $H : T_1 \to T_2$ preserves finite limits and ν is a natural transformation $F \stackrel{\cdot}{\to} G \circ H$; let $\hat{\alpha} : \lambda\overline{\omega}\text{-Cat}^{\to} \to Mod_{lex}$ be the functor $\hat{\alpha}(m) = \alpha_{dom(m)}(m)$; noticing that models are fibred over theories by the domain fibration $dom : \lambda\overline{\omega}\text{-Cat}^{\to} \to \lambda\overline{\omega}\text{-Cat}$, the functor *th* can alternatively be viewed as part of a morphism of fibrations:

$$\begin{tikzcd}
\lambda\overline{\omega}\text{-}Cat^{\to} \arrow[r, "\hat{\alpha}"] \arrow[d, "dom"'] & Mod_{lex} \arrow[d, "dom_{lex}"] \\
\lambda\overline{\omega}\text{-}Cat \arrow[r, "th"] & Lex
\end{tikzcd}$$

The functor *Mod* is not full, that is, not all filtered colimit-preserving right adjoint functors $Mod(q) \to Mod(p)$ are syntactically induced by a relative interpretation $p \to q$ in $\lambda\overline{\varpi}$-*Cat*. In fact, we need an extra condition and the following lemma to justify it:

Lemma 6.8.11 *Let $\theta : th(p) \to th(q)$ be a lex functor; if $\theta \circ th(?_p) = th(?_q)$, where $?_x$ is the unique morphism $\emptyset \to x$ from the initial object in $\lambda\overline{\varpi}$-Cat, then θ is of the form $th(\phi)$ for some $\lambda\overline{\varpi}$-functor $\phi : p \to q$.*

We adopt the following informal argument as proof of this lemma. The idea is to use *sketches* to give a more concrete picture of categories of the form $th(p)$. In fact, we consider a simplified scenario, where categories take the place of $\lambda\overline{\varpi}$-categories.

Consider the finite limit sketch of categories C described in [BW90, 9.1.4]; models in *Sets* of this sketch correspond to small categories and arrows between models to functors. The graph of C is:

$$\begin{tikzcd} C_R \arrow[r, dashed, "\text{various}", "\text{arrows}"'] & C_Q \arrow[r, "m"] & C_A \arrow[r, shift left=3, "src"] \arrow[r, "tgt"'] & C_O \arrow[l, shift left=3, "e"] \end{tikzcd}$$

The nodes C_A and C_O represent the set of arrows and of objects; *src* and *tgt* represent the source and target maps while e yields the identities. The sketch C includes a cone over the diagram $C_A \xrightarrow{src} C_O \xleftarrow{tgt} C_A$, whose vertex C_Q represents composable pairs of arrows. The edge m represents composition. Similarly, C_R represents composable triples, with obvious arrows into C_Q.

For any finite limit sketch S, there exists a lex theory T such that the category of models of S in *Sets* is equivalent to $[T, Sets]_{lex}$ (see [BW90, 9.3.1]). Since the equivalence $[T_1, Sets]_{lex} \cong [T_2, Sets]_{lex}$ implies $T_1 \cong T_2$, from (6.4) we conclude that the lex category T_{Cat} introduced in section 6.6 is (equivalent to) the category associated with the above sketch C.

Similarly, given a specific category $\mathcal{C}$, we call $sk(\mathcal{C})$ the finite limit sketch obtained as follows: a node 1 is added to the graph of C together with a cone over the empty diagram with vertex 1. Moreover, for each object A of $\mathcal{C}$, $sk(\mathcal{C})$ has an edge $\ulcorner A \urcorner : 1 \to C_O$ and, for each arrow $f : A \to B$, an edge $\ulcorner f \urcorner : 1 \to C_A$ with diagrams $src \circ \ulcorner f \urcorner = \ulcorner A \urcorner$ and $tgt \circ \ulcorner f \urcorner = \ulcorner B \urcorner$. Identities and composition of $\mathcal{C}$ are represented in $sk(\mathcal{C})$ straightforwardly. Applying to $sk(\mathcal{C})$ the same argument that we applied above to C, we conclude that the finite limit theory associated with $sk(\mathcal{C})$ is the lex category $th(\mathcal{C})$ introduced in section 6.6. There is an obvious inclusion $th(?_{\mathcal{C}}) : T_{Cat} \cong th(\emptyset) \hookrightarrow th(\mathcal{C})$.

It is easy to verify that there is a one-to-one correspondence between lex functors $\theta : th(\mathcal{C}) \to th(\mathcal{D})$ such that $\theta \circ th(?_{\mathcal{C}}) = th(?_{\mathcal{D}})$ and functors $\mathcal{C} \to \mathcal{D}$. In fact, this equation ensures that θ is a C-homomorphism, that is, that it preserves the objects C_A, C_O and so on. Therefore, θ maps objects $1 \to C_O$ of $\mathcal{C}$ to objects $1 \to C_O$ of $\mathcal{D}$ and arrows to arrows in such a way that identities and composition are preserved. A similar correspondence can be established if categories are replaced by $\lambda\overline{\omega}$-categories and functors by $\lambda\overline{\omega}$-functors. This concludes our informal proof of lemma 6.8.11.

Theorem 6.8.12 (Characterization theorem) *Let p and q be $\lambda\overline{\omega}$-categories, let $?_p$ and $?_q$ be as in lemma 6.8.11 and let $\Psi : Mod(q) \to Mod(p)$ be a filtered colimit-preserving functor with a left adjoint; Ψ is of the form $Mod(\phi)$, for some $\lambda\overline{\omega}$-functor $\phi : p \to q$, if and only if $Mod(?_p) \circ \Psi = Mod(?_q)$.*

Proof. (Only if) Noticing that the functors $Mod(?_x) : Mod(x) \to Mod(\emptyset) \cong \lambda\overline{\omega}$-Cat are the codomain projections $(x \to y) \mapsto y$, the result follows immediately from the definition of Mod.

(If) For simplicity we treat the equivalences $Mod(p) \cong [th(p), Sets]_{lex}$ as identities. Let Φ, Φ_p and Φ_q be left adjoints respectively to Ψ, $Mod(?_p)$ and $Mod(?_q)$. From $Mod(?_p) \circ \Psi = Mod(?_q)$, it follows that $\Phi \circ \Phi_p = \Phi_q$ and hence, since $(\Phi_p)^{op}_{fp} = th(?_p)$ (and similarly for q), we have $\Phi^{op}_{fp} \circ th(?_p) = th(?_q)$. By lemma 6.8.11, it follows that

$\Phi^{op}_{fp} = th\,(\phi)$, for some $\phi : p \to q$. Then, $\Psi = [\Phi^{op}_{fp}, Sets]_{lex} = [th\,(\phi), Sets]_{lex} = Mod\,(\phi)$.

$\square$

The intuition behind this theorem is that $th\,(\emptyset) = T_{\lambda\overline{\omega}}$ provides a "signature" for all cartesian theories in *Lex* which describe $\lambda\overline{\omega}$-categories. Functors $\Psi : Mod\,(q) \to Mod\,(p)$ induced by relative interpretations of $\lambda\overline{\omega}$-theories correspond, in the Gabriel-Ulmer sense, to $th\,(\emptyset)$-homomorphisms between finite limit theories: this is expressed by the condition in lemma 6.8.11 which is equivalent to $Mod\,(?_p) \circ \Psi = Mod\,(?_q)$.

This characterization of relative interpretation can be adapted to any formal language whose models are described by a finite limit theory. For example, theories of the simply typed lambda calculus with unit type and products correspond to catresian closed categories [LS86], which are set-valued models of a finite limit theory, as shown in section 6.6. Therefore, functors $\Psi : Mod\,(T_1) \to Mod\,(T_2)$ arising from relative interpretations $T_2 \to T_1$ of simply typed lambda calculi, can be characterized as above. The same argument also applies to theories of the higher order polymorphic lambda calculus, exploiting the correspondence between such theories and PL-categories [See87].

7 Application: constructing with LEGO

In the previous chapter we presented semantic constructors. Here we use a type theory, the Extended Calculus of Constructions (XCC) [Luo82], as a formal framework for developing specific examples. We take advantage of an implementation of XCC, LEGO [LP92], to check the correctness of our implementation. Below we explain how we use LEGO to develop denotational semantics and list the features of XCC that are crucial to the applications that we present.

At the beginning of the '80s, Dana Scott suggested that an intuitionistic universe of sets should be the natural setting in which to develop a theory of domains. Since then, but most notably in recent years, *synthetic domain theory* ([Ros86,Pho90,Hyl90,Tay91, RS]) has sought a good full subcategory of an elementary topos in which to carry out the mathematics of computation.

Our development of denotational semantics is consistent with synthetic domain theory: a category *Dom* to interpret computation is fully embedded in the category $\mathcal{U}$ of XCC types, which provides an intuitionistic universe of sets. We refer to *Dom* as the category "of domains." This is done rather informally, since we do not actually axiomatize domain theoretic structure in *Dom* (for an axiomatic approach to domain theory see [Fio94b,Fio94a,Mog95a]).

The category of domains is implemented in LEGO by assuming a type `Dom:Type` (of names of domains) and a `Dom`-indexed family `E:Dom->Type` of domain-like types. The operation `E`, the likes of which are sometimes called "large eliminations," embeds domains in the universe of types. In this way, the presentation of relevant categorical constructions, such as fixed-point operators or initial algebras of endofunctors, becomes very natural and requires none of the syntactic overhead that a full encoding of category theory in the type theory would call for.

In order to prove that a semantic structure is adequately represented by a term `S` of the type theory, one can show that `S` is interpreted as a structure of the appropriate form in models of the type theory. For example, in section 7.1 we notice that terms of type `Monad` in the implementation define *internal* monads in the category of types. Part of the effort in establishing a precise correspondence between intended mathematical objects and their description in type theory goes into understanding how the notion of equality adopted in the description relates to semantic identy in the models. We use Leibniz equality.

The applications that we present use the following features of the type theory. A *universe* (viz. `Type(0)`) is used for the synthetic domain theory set-up (`Dom` and `E`), as well as for defining uniform redefinitions (section 7.2). *Dependent types* are exploited for representing polymorphic operations, such as the natural transformations *val* and *let* of the computational metalanguage or the operations to raise and handle exceptions. *Sigma types* (dependent sums) provide a useful abstraction mechanism for structuring theories in proof development. We also use *inductive* sigma types in the definition of the internal category of domains (section 7.1). Moreover, *type inference* and *coercion*, specific to the LEGO system, allow agile notation.

Synopsis. In 7.1 we give a formal account of structures and interpretation of the computational metalanguage in LEGO. In 7.2 we implement a semantic constructor for exceptions and one for resumptions. The latter is used in section 7.3, where we present an application in which a model of a computational metalanguage featuring an operator of parallel composition is constructed from models of simpler forms of computation. Some of the material in this section comes from [CM93] and is due to Moggi. Ours is the implementation in LEGO including all the formal proofs. In the last section we discuss a concrete model of the type theory based on ω-sets and show that the structure we assumed in the application can be found in this model.

7.1 Structures and interpretation

In section 2.4 we defined a suitable notion of Σ-structure to interpret lambda calculi of the form $\mathcal{L}(\Sigma)$. Here we represent such structures in the XCC and write a small application in LEGO where they are used in a formal interpretation of the computational metalanguage ML_T. For simplicity, we consider a minimal version of the metalanguage with no unit or product types.

We represent categorical structure in XCC by defining types such as `Functor`, `Monad` etc, which we think of as signatures and whose terms represent structures, such as functors, monads and so forth, on a fixed category *Dom*. Order theoretic structure can be axiomatized in *Dom* as shown in [Fio94a]. Let `Dom` be a type, whose inhabitants we view as names of domains, and let `E` be an externalization map embedding domains in the universe of types:

```
[Dom : Type]
[E : Dom->Type];
```

The category *Dom* arises from these data as a full internal subcategory of the universe of types (see appendix B for the full definition):

```
defn  Dom1 = Sigma [X:Dom](Sigma [Y:Dom](E X)->E Y)] : Type (* arrows *)
defn  Dom2 = ... : Type         (* composable arrows *)
defn  d0 = ... : Dom1->Dom      (* domain *)
defn  d1 = ... : Dom1->Dom      (* codomain *)
defn  i = ... : Dom->Dom1       (* identity *)
defn  o = ... : Dom2->Dom1      (* composition *)
```

As shown by the definition of `Dom1`, the arrows of *Dom* are borrowed from the function spaces of the type theory. *Inductive* sigma types (`Sigma`) are used in the definition of `Dom1` and `Dom2`; they provide the η rule for dependent pairs, which is used for making the diagrams involving the composition map commute. It is immediate to verify that this definition satisfies the appropriate equations. For example, here is the proof in LEGO that the domain of the identity arrow on `X` is `X`:

```
Goal {X:Dom}Eq X (d0 (i X));
Intros;
Immed;
```

In developing the modular approach in LEGO, we use a variety of structures, whose signatures are represented by types such as `Sum`, `Prod`, `Exponential` and `Ind`, all defined in appendix C. The signature `Ind` for *inductive types* includes the higher order operators `mu` and `It` described in section 6.3, taking functors as arguments. Here is the signature of strong monads (in the following, when mentioning a monad, we shall understand a *strong* one, unless otherwise stated):

```
[Monad =
<T:Dom->Dom>
<val:{X|Dom}(E X)->E(T X)>
<let:{X,Y|Dom}((E X)->E(T Y))->(E(T X))->E(T Y)> Unit ];
```

The axioms of a monad should really be part of the type `Monad`, and constructors $\mathcal{F}$:`Monad -> Monad` should include a proof that $\mathcal{F}$ `M` satisfies the appropriate equations. However, to keep the implementation simple, we deal with the axioms separately. Below, we list the axioms of the theory of monads. The functions `T`, `val` and `let` are used to extract from a monad `M` the corresponding data, so that `(T M):Dom->Dom`, and so on.

```
[AX_LUNIT= [M:Monad]{X|Dom}{x:E(T M X)} Eq x (let M (val M|X) x)];
[AX_RUNIT= [M:Monad]{X,Y|Dom}{f:(E X) -> E(T M Y)}
            Eq f ([x:E X]let M f (val M x))];
[AX_ASSOC= [M:Monad]{X,Y,Z|Dom}{c:E(T M X)}{f:(E X) -> E(T M Y)}
            {g:(E Y) -> E(T M Z)}
            Eq (let M ([x:E X]let M g (f x)) c) (let M g (let M f c))];
```

Remark. Interpreting terms of type `Monad` in an extensional model of the type theory, one obtains an internal monad in *Dom*, and therefore a monad fibred over the universe of types. The use of a fibred monad is consistent with the semantics for Evaluation Logic proposed in [Moga].

□

The formal interpretation of the computational metalanguage ML_T requires the structure of products, exponentials and a monad. To write it in LEGO we must first encode ML_T. Assuming the type `Grd` of ground type symbols, the syntax of ML_T is defined by the types `Ty`, `Con`, `Var` and `Exp` representing types, contexts, variables and terms. Contexts are lists of types. De Bruijn's method of variable binding is implemented by including context information inside variables. In particular, the variable x in the context $(C, (x : A), D)$ has the form `(Weak`...`(Weak (Vo C T)`...`)`, where D is represented by the successive applications of `Weak`. The type `(Exp C A)` represents well-formed ML_T terms of type `A` in context `C`. In appendix D, the types `Ty`, `Var` and `Exp` are *inductive* (the inductive types of LEGO, not to be confused with the inductive types in *Dom* provided by the signature `Ind`). This allows interpretation to be defined by induction on the structure of terms.

Remark We do not identify the variables of the object language with the ones of the type theory, as in the LF style, because this would make an inductive definition of interpretation unfeasible. In fact, let terms of type τ of the object language be interpreted by maps $[\![\_]\!]_\tau : \mathit{Exp}(\tau) \to [\![\tau]\!]$ and suppose λ-abstractions are encoded in the type theory as terms ΛF, where $F : \mathit{Exp}(\sigma) \to \mathit{Exp}(\tau)$ represents a term of type τ with a free variable of type σ in the object language. An interpretation $[\![\Lambda F]\!]_{\sigma\Rightarrow\tau} : [\![\sigma \Rightarrow \tau]\!] \equiv [\![\sigma]\!] \to [\![\tau]\!]$ cannot be obtained from F, $[\![\_]\!]_\sigma$ and $[\![\_]\!]_\tau$ unless we know how to map $[\![\sigma]\!]$ back into $\mathit{Exp}(\sigma)$.

□

The encoding ϵ of ML_T in LEGO is defined as follows: Types and contexts are mapped respectively into terms of type `Ty` and `Con`. Terms $\Gamma \vdash M : \sigma$ are mapped into closed terms $\epsilon_{\Gamma,\sigma}(M) : Exp\ \epsilon(\Gamma)\ \epsilon(\sigma)$. We write $\Gamma|\Delta$ for the concatenation of two contexts. The symbols GR, TT and AR are the constructors of ML_T types, while VAR, LMB, APP, LET and VAL are the ones of ML_T terms (see appendix D).

$$
\begin{aligned}
&\epsilon(A) = \mathtt{GR\ A} \quad (\text{for } \mathtt{A{:}Grd})\\
&\epsilon(T\sigma) = \mathtt{TT}\ \epsilon(\sigma)\\
&\epsilon(\sigma \to \tau) = \mathtt{AR}\ \epsilon(\sigma)\ \epsilon(\tau)\\
&\epsilon_{\Gamma|x:\sigma|\Delta,\sigma}(x) = \mathtt{VAR}\,(\mathtt{Weak}\ldots(\mathtt{Vo}\ \epsilon(\Gamma)\ \epsilon(\sigma))\ldots)\\
&\epsilon_{\Gamma,\sigma\to\tau}(\lambda\, x : \sigma.\, M) = \mathtt{LMB}\ \epsilon_{\Gamma|x:\sigma,\tau}(M)\\
&\epsilon_{\Gamma,\tau}(M\, N) = \mathtt{APP}\ \epsilon_{\Gamma,\sigma\to\tau}(M)\ \epsilon_{\Gamma,\sigma}(N)\\
&\epsilon_{\Gamma,T\tau}(let\ \ x \Leftarrow M\ \ in\ \ N) = \mathtt{LET}\ \epsilon_{\Gamma|x:\sigma,T\tau}(N)\ \epsilon_{\Gamma,T\sigma}(M)\\
&\epsilon_{\Gamma,T\sigma}(val(M)) = \mathtt{VAL}\ \epsilon_{\Gamma,\sigma}(M).
\end{aligned}
$$

Proposition 1 *The map ϵ is a bijection between types (contexts) of ML_T and canonical terms of type* Ty *(*Con*) in XCC. For every type σ and context Γ of ML_T, $\epsilon_{\Gamma,\sigma}$ is a bijection between terms $\Gamma \vdash M : \sigma$ and canonical terms of type* Exp *$\epsilon(\Gamma)\ \epsilon(\sigma)$ of XCC.*

A formal interpretation of ML_T in *Dom* is defined in appendix E by recursion on the structure of the syntactic domains. The parameters of the interpretation are structures of type `Prod`, `Exponential` and `Monad`. Terms of type `Grd`, `Ty` and `Con` are interpreted respectively by `I_Grd`, `I_Ty` and `I_Con` as elements of `Dom`, while terms of type `Var` and `Exp` are interpreted by `I_Var` and `I_Exp` in the externalization of the appropriate domain; for example, given terms `P:Prod`, `X:Exponential` and `M:Monad`, a term `e:(Exp c t)` of type `t` in context `c` is interpreted as a map

```
I_Exp P X M (e): (E (I_Con P X M c))->E (I_Ty X M t).
```

One of the benefits of making interpretation parametric in a notion of computaiton is that models can be augmented with new semantic structure with no need of rewriting the interpretation function. Using the above interpretation of the computational metalanguage, let `F:Monad->Monad` implement a monad constructor and let `M` be a structure of type `Monad`, interpretation in `F(M)` is simply:

```
I_Exp P X (F M) (e): (E (I_Con P X (F M) c))->E (I_Ty X (F M) t).
```

7.2 Examples of constructors

In this section we inplement in LEGO two semantic constructors described in [CM93], one for exceptions and one for resumptions. We give a formal proof of the correctness of the first, that is, we show that it maps monads to monads. The second is used in the next section to define and reason about models of parallel computation based on interleaving.

Exceptions. In order to make the LEGO code less cryptical, we first introduce the constructor for exceptions as a translation, as we did for resumptions in section 6.4, using a simpler syntax: the computational metalanguage.

Let $\mathcal{B}$ be a collection of base type symbols, including the symbol E for the base type of exceptions, and let *raise* and *handle* be operations of arities $\forall X.E \longrightarrow TX$ and $\forall X.TX, (E \to TX) \longrightarrow TX$ respectively. Let $\Sigma_1 = (\mathcal{B}, +, inj_1, inj_2, case)$ be a signature for sums and let $\Sigma_2 = (\mathcal{B}, raise, handle)$ be one for exceptions. A constructor $Mod(\Sigma_1) \to Mod(\Sigma_2)$ is defined by the following translation $(\!|\_|\!) : ML_T(\Sigma_2) \to ML_T(\Sigma_1)$:

$$
\begin{array}{rcll}
(\!|B|\!) & \stackrel{\text{def}}{=} & B & \qquad (B \in \mathcal{B}) \\
(\!|\sigma \to \tau|\!) & \stackrel{\text{def}}{=} & (\!|\sigma|\!) \to (\!|\tau|\!) & \\
(\!|T\tau|\!) & \stackrel{\text{def}}{=} & T((\!|\tau|\!) + E) & \\
(\!|val(M)|\!) & \stackrel{\text{def}}{=} & val(inj_1 (\!|M|\!)) & \\
(\!|let(M)|\!) & \stackrel{\text{def}}{=} & let(case\ (\!|M|\!)\ (\lambda x : E.\, val(inj_2\, x))) &
\end{array}
$$

$$\begin{aligned}
(\!|raise(M)|\!) &\stackrel{\text{def}}{=} val(inj_2(\!|M|\!)) \\
(\!|handle(M,N)|\!) &\stackrel{\text{def}}{=} let(case\ \ (\lambda x.\, val(inj_1\, x))\ (\!|N|\!))(\!|M|\!).
\end{aligned}$$

Let $(\mathcal{C}, T, \mathcal{A})$ be a model of $ML_T(\Sigma_1)$; the translation $(\!|\_|\!)$ defines a strong monad $(\!|T|\!)X \stackrel{\text{def}}{=} T(X{+}E)$ and a Σ_2-structure (similarly) in $\mathcal{C}$. There is a strong monad morphism $(id, \sigma) : (\mathcal{C}, T) \to (\mathcal{C}, (\!|T|\!))$, where $\sigma = T(inj_1)$. Such a morphism extends to one in $Mod(\Sigma_2)$, which makes this constructor *pointed* (in the sense of section 6.2).

Let `Sum` be the signature of the theory of sums described in appendix C, providing an operation `+` on domains, injections and an operation of case analysis; the above constructor is implemented in LEGO by a function `Exceptions` which, given `S:Sum`, a domain `Exn:Dom` of exceptions and a monad `M:Monad`, returns a 5-tuple of operations `(E_T, E_val, E_let, raise, handle)`:

```
[Exceptions [S:Sum] [Exn:Dom] [M:Monad] =
        [E_T [X:Dom] : Dom = T M (sum S X Exn)]
        [E_val [X|Dom] [x:E X] : E (E_T X) = val M (in1 S|X|Exn x)]
        [E_let [X,Y|Dom] [f:(E X)->E(E_T Y)] : (E (E_T X)) -> E(E_T Y) =
                let M(case S f ([x:E Exn] val M (in2 S x)))]
        [raise [X|Dom][n:E Exn] : E (T M (sum S X Exn)) =
                (val M (in2 S|X|Exn n))]
        [handle [X|Dom][c:E(T M (sum S X Exn))]
                [h: (E Exn) -> E (T M (sum S X Exn))] : E(T M (sum S X Exn))=
                let M (case S ([x:E X] val M (in1 S x)) h) c]
        (E_T, E_val, E_let, raise, handle)];
```

The first three components of this tuple are extracted and wrapped up as a monad by the function `Exception_Constructor:Sum->Dom->Monad->Monad`. As mentioned earlier, it must be separately verified that, if `M` satisfies the axioms `AX_LUNIT`, `AX_RUNIT` and `AX_ASSOC`, so does the structure `(Exception_Constructor S Exc M)`. Below we formally verify the first axiom, while the others are proven in appendix F. Given a monad `M`, the function `ax_lunit` returns a witness of `(AX_LUNIT M)` (similarly for `ax_runit` and `ax_assoc` in the appendix).

```
Goal {S:Sum}{Exn:Dom}{M:Monad}AX_LUNIT (Exception_Constructor S Exn M);
Normal;
Intros;
Qrepl ax_case S|X|Exn|(T M(sum S X Exn)) (val M|(sum S X Exn));
Qrepl Eq_sym (ax_lunit M|(sum S X Exn)|x);
Immed;
```

Resumptions. Below we present `Resumptions`, which is an implementation in LEGO of the constructor $\mathcal{F}$ described in section 6.4 (for simplicity, $H : \Omega \Rightarrow \Omega$ is the identity).

As with monads, we extract from a structure `F` of type `Functor` an object map `(FT F):Dom->Dom` and a strength `(str F)|X|Y: ((E X)->E Y)->(E(T X))->E(T Y)`. Any structure `M` of type `Monad` has an obvious substructure `(Monad_to_Functor M)` of type `Functor`. Given `S:Sum`, `F:Functor` and `A:Dom`, `(RF S F A)` is the functor mapping `X` to `T(A+X)`, which was written F_A in section 6.4. Similarly, `(Res S Z M)A` represents the domain $\mu(F_A) = \mu$`X.T(A+X)`, where `T` is the underlying endofunctor of the monad `M` and μ comes from the structure `Z:Ind` of inductive types.

```
[RF [S:Sum] [F:Functor] [X:Dom] : Functor =
        ([W:Dom] FT F (sum S X W),
        ([X1,X2|Dom][f:(E X1)->E X2] str F
          (case S (in1 S|X|X2) ([z: E X1] in2 S (f z)))), star:Functor)];

[Res [S:Sum] [Z:Ind] [M:Monad] =
     [X:Dom] mu Z (RF S (Monad_to_Functor M) X)];
```

Given sums, inductive types and a monad, `Resumptions` returns the following operations (see section 6.4 for an informal description):

```
Res:Dom->Dom,
```

`Rval:{X|Dom}(E X)->E (Res X)`,

`Rlet:{X1,X2|Dom}((E X1)->E (Res X2))->(E (Res X1))->E (Res X2)`,

`tau:{X|Dom}(E (Res X))->E (Res X)` and

`CH:{X1,X2|Dom}((E X1)->E (Res X2))->((E (Res X1))->E (Res X2))->`

`(E (Res X1))->E (Res X2)`

```
[Resumptions [S:Sum] [Z:Ind] [M:Monad] =
        [F [X:Dom] = RF S (Monad_to_Functor M) X]
        [Res = Res S Z M]
        [Rval [X|Dom] [x:E X] : E (Res X) =
              alg Z|(F X) (val M (in1 S x))]
        [tau [X|Dom][z:E (Res X)] : E (Res X) =
              alg Z|(F X) (val M (in2 S z))]
        [CH [X1,X2|Dom][f:(E X1)->E (Res X2)]
            [g:(E (Res X1))->E (Res X2)]
            [c:(E (Res X1))] : E (Res X2) =
               alg Z|(F X2) (let M
                ([d:E(sum S X1 (Res X1))] coalg Z|(F X2) (case S f g d))
                (coalg Z|(F X1) c))]
        [Rlet [X1,X2|Dom] [f:(E X1)->E (Res X2)] : (E (Res X1))->E (Res X2) =
                it Z|(F X1) ([c:E (T M (sum S X1 (Res X2)))] alg Z|(F X2)
                            (let M ([z: E (sum S X1 (Res X2))] coalg Z|(F X2)
                                   (case S f (tau|X2) z)) c))]
        (Res, Rval, Rlet, tau, CH, star)];
```

We end this section with the implementation of *uniform redefinitions* (see section 6.2), which are used in the next section by a constructor $\mathcal{F}_\Sigma$ to convert the operations in Σ to a new computational setting. Here, we consider the case of *unary* operations; arbitrary arities are dealt similarly. Uniform redefinitions allow constructors of the form (7.1) below to redefine operations of the form (7.2):

$$(\mathcal{F}\,\mathtt{T})\,\mathtt{X} \stackrel{\text{def}}{=} \mathtt{T(G\ \ T\ \ X)} \tag{7.1}$$

$$op_X : \mathtt{S(T\ X)} \tag{7.2}$$

where `G:Monad->Dom->Dom` and `S:Dom->Type`. For example, let `Exn` be the domain of exceptions; in the exceptions constructor $(\mathcal{F}\,\mathtt{T})\,\mathtt{X} \stackrel{\text{def}}{=} \mathtt{T(X+Exn)}$, we have `(G T X) = X+Exn`, where `G` does not depend on `T`. Similarly, or_X : `(E (T X))->(E (T X))->E (T X)`, the operator of nondeterministic choice introduced in the next section, is of the form (7.2), where `(S X)` is `(E X)->(E X)->E X`. A constructor $\mathcal{F}$ can uniformly redefine such operations as follows:

$$(\mathcal{F} op)_X \stackrel{\text{def}}{=} op_{(GTX)} : \mathtt{S(T(G\ T\ X))} \equiv \mathtt{S}((\mathcal{F}\,\mathtt{T})\,\mathtt{X}).$$

In LEGO:

```
[Uniform_Redefinition1 [G:Monad->Dom->Dom][S:Dom->Type(0)][M:Monad] =
         [c:{X|Dom} S (T M X)] [X|Dom] c|(G M X)];
```

7.3 Parallel composition

In this section we give examples of formal proofs involving the modular constructions introduced earlier. We add a mechanism of resumptions to nondeterministic computations and, from the semantic structures thus obtained, we define an operator of parallel composition and prove in LEGO that this operator is commutative.

Let `T:Dom->Dom` be the object map of a monad `M`. Structures of type `(Fix T)` and `(Nondetm M)` provide operations respectively of fixed point and nondeterministic choice for `M`-computations:

```
[Fix [T:Dom->Dom] =
     <Yop:{X1,X2|Dom}(((E X1)->E(T X2))->(E X1)->E(T X2))->(E X1)->E(T X2)>
      Unit ];
```

```
[Nondetm [M:Monad] =
        <or : {X|Dom} (E (T M X)) -> (E (T M X)) -> E (T M X)> Unit];
```

The theory of nondeterminism states that `or` is commutative, i.e. `or w z = or z w`, and that it is *natural*, i.e. `or (Tf w) (Tf z) = Tf(or w z)`, while the theory of fixed points includes the fixed point property of `Yop` and the axiom `AX_UNIFORMITY` below.

A fixed point operator Y in the category of cpos is said to be *uniform* (see [Gun92, 4.2]) when, for any pair of continuous functions $f : D \to D$ and $G : E \to E$ and strict continuous function $h : D \to E$ such that $h \circ f = g \circ h$, we have $Yg = h(Yf)$. Least elements and strictness can be axiomatized in *Dom* as shown in [Mog95a]. We shall not implement these axioms in LEGO: when required in a formal proof, we verify that a term denotes a strict morphism separately on our blackboard and allow ourself a free supply of witnesses of this condition, which we represent in LEGO with a constant predicate `STRICT`. The uniformity axiom is:

```
[AX_UNIFORMITY = [T|Dom->Dom][F:Fix T]{X1,X2,X3,X4|Dom}
                 {h:((E X1)->E(T X2))->(E X3)->E(T X4)}
                   (STRICT h)->
                     ({f:((E X1)->E(T X2))->(E X1)->E(T X2)}
                      {g:((E X3)->E(T X4))->(E X3)->E(T X4)}
                      ({k:(E X1)->E(T X2)} Eq (h(f k)) (g(h k)))->
                            {x:E X3} Eq (Yop F g x) (h (Yop F f) x))];
```

Starting from a monad `M` and structures `F:(Fix (T M))` and `N:(Nondetm M)`, we add resumptions to `M`-computation; the constructor `Resumptions`, $\mathcal{F}$ for short, is applied to `M` to obtain a monad $\mathcal{F}$`(M)=(Res,Rval,Rlet)` and a $\Sigma_{\mathcal{F}}$structure consisting of the operations `tau` and `CH`. Moreover, applying uniform redefinitions (not expanded here) to `Yop` and `or`, we produce new structures $\mathcal{F}$`F:Fix(T` $\mathcal{F}$`M)` and $\mathcal{F}$`N:(Nondetm` $\mathcal{F}$`M)`, whose operations we call `yrr` and `orr` respectively.

The operation $\texttt{pand}_{A,B} : \texttt{Res}(A) \times \texttt{Res}(B) \to \texttt{Res}(A \times B)$ of parallel composition is defined formally in LEGO in appendix G. Here we give a more digestible version in lambda notation: using infix notation for `orr` and mix-fix notation for `Rlet` (as in the computational metalanguage),

$$\begin{aligned}\texttt{pand}_{A,B} \;=\; \texttt{yrr}\,(\lambda\, h : \texttt{Res}\,(A) \times \texttt{Res}\,(B) \to \texttt{Res}\,(A \times B).\, \lambda w : \texttt{Res}\,(A), z : \texttt{Res}\,(B).\\ \texttt{CH}\,(\lambda\, x : A.\, \texttt{Rlet}\ y \Leftarrow z\ \mathit{in}\ \texttt{Rval}\,\langle x, y\rangle,\ \lambda\, u : \texttt{Res}\,(A).\, h(u, z))w\\ \texttt{orr}\ \texttt{CH}\,(\lambda\, y : B.\, \texttt{Rlet}\ x \Leftarrow w\ \mathit{in}\ \texttt{Rval}\langle x, y\rangle,\ \lambda\, v : \texttt{Res}\,(B).\, h(w, v))z).\end{aligned}$$

Intuitively, $\texttt{pand}_{A,B}(w, z)$ performs one step $w \rightsquigarrow u$ of M-computation on w and then invokes $\texttt{pand}_{A,B}(u, z)$, or it performs one step $z \rightsquigarrow v$ of M-computation on z and then invokes $\texttt{pand}_{A,B}(w, v)$. We look at the first of these branches, $\texttt{CH}_{A,A\times B}(f, g)w$, of which the second is just the dual. $\texttt{CH}_{A,A\times B}$ performs case analysis on $w : \texttt{Res}(A)$; if it is the M-computation of a value x, it returns the pair $\langle x, y\rangle$, where y is the value produced by z; this is done in f. Otherwise, if w is the M-computation of a resumption u, it runs $\texttt{pand}_{A,B}(u, z)$, which is done by g.

Now we focus on an algebraic property of this operation: commutativity. Note that this cannot be expressed as $\texttt{pand}_{A,B}(w, z) = \texttt{pand}_{B,A}(z, w)$, because the types are not quite right. Let `(pswap|P|A|B)` be the isomorphism $(A \times B) \to (B \times A)$, where the structure `P:Prod` provides the $\times$, and let `Rstr` be the morphism map of the functor `Res`; the precise statement of commutativity is:

```
THM_COMMUTATIVITY = {X1,X2|Dom} {w:E(Res X1)} {z:E(Res X2)}
                          Eq (pand z w) (Rstr (pswap P|X1|X2) (pand w z)).
```

Below is a sketch of the proof of `THM_COMMUTATIVITY`, the gory LEGO details of which are given in appendix H.

Let $H_{A,B}$ be the function of which $\texttt{pand}_{A,B}$ is the fixed point, i.e. , $\texttt{pand}_{A,B} = \texttt{yrr}(H_{A,B})$, and let $K : (\texttt{Res}(B) \times \texttt{Res}(A) \to \texttt{Res}(B \times A)) \to (\texttt{Res}(A) \times \texttt{Res}(B) \to \texttt{Res}(A \times B))$ be the function $K(f, w, z) = (\texttt{Rstr pswap})\,(f(z, w))$. `THM_COMMUTATIVITY` is an immediate consequence of:

$$\mathtt{yrr}(H_{A,B}) = K(\mathtt{yrr}(H_{B,A})).$$

To prove this equation we show that $K \circ H_{B,A} = H_{A,B} \circ K$ and that `yrr` is uniform (having proven the strictness of K on the blackboard). The equality $K \circ H_{B,A} = H_{A,B} \circ K$ follows from the *commutativity* of `or` and from the *naturality* of `or` and `CH`. The uniformity of `yrr`, is proven in appendix I. In particular, let the function `Rfix` of type `{S:Sum}{Z:Ind}{M:Monad}(Fix (T M))->Fix (Res S Z M)` apply a uniform redefinition to `Fix`-structures for the resumption constructor; we show:

```
Goal {S:Sum}{Z:Ind}{M:Monad}{F:Fix (T M)}
        (AX_UNIFORMITY F) -> AX_UNIFORMITY (Rfix S Z M F);
```

The proof of this statement uses the naturality of the fixed point operator of the structure `Fix (T M)` and the properties of inductive types, incliding Lambek's lemma on initial algebras, the proof of which is shown in appendix C.

7.4 A concrete model

The interpretation of `pand` requires a certain amount of categorical structure in *Dom*. For example, *Dom* is assumed to support inductive types, whose universal properties are used for proving that `pand` is commutative. We show that all this structure can be found in a nontrivial model of XCC.

It is well known that, if k is an inaccessible cardinal number, all axioms of Zermelo-Frankel set theory are true in V_k (see [End77, theorem 9L]), so that V_k is a miniature (so to speak) version of the class of all sets. Luo's ω-sets semantics for XCC (see [Luo82]) is based on a hierarchy of set universes V_k indexed by inaccessible cardinals. More precisely, as proposed by Tarski in 1938, ZF is augmented with a *large cardinal axiom* postulating for each cardinal number the existence of a larger inaccessible cardinal. In particular, this axiom yields ω inaccessible cardinals $k_0 < k_1 < k_2 \ldots$ for indexing universes V_{k_i} with the required properties for interpreting the type hierarchy of XCC. Such universes satisfy the membership relation $V_{k_i} \in V_{k_{i+1}}$, the inclusion relation $V_{k_i} \subseteq V_{k_{i+1}}$ and support predicative Π and Σ. An inaccessible k, with $k_i < k$ for all i, is also provided to form the set of objects V_k of the all-containing category of sets, $V_k = obj(Sets)$, into which all universes embed with no foundational trouble.

As shown by a well known result of Reynolds concerning System F, there are features of XCC (viz. polymorphism) that cannot be given a simple set theoretic semantics. This led to the development of realizability models based on ω-sets. In [Luo82], Luo interprets types as ω-sets: `Type(j)` is interpreted as the full subcategory ω-*Sets*(j) of ω-*Sets* whose objects have carrier in the universe V_{k_j}. Then ω-*Sets* must be sufficiently large as to contain all the ω-*Sets*(j) and V_k comes to the rescue.

`Prop` is interpreted as the category *PER* of partial equivalence relations on N, whose good closure properties allow impredicative quantification over propositions. Such quantification is modelled in the fibration of PERs over ω-*Sets*, where the ω-set P_0 of all PERs is a generic object. Note that *PER* is a small category and P_0 is in ω-*Sets*(0), thus validating the axiom `Prop:Type(0)`.

Notation. We write ω-*Sets* for the category of ω-sets whose carrier is a set in the universe V_k. The categories ω-*Sets*(j), for $j = 0, 1 \ldots$, are similarly defined from universes V_{k_j} and they are full subcategories of ω-*Sets*. We write ω-*Sets*$_0(j)$ for the ω-set $(obj(\omega\text{-}Sets(j)), \omega \times obj(\omega\text{-}Sets(j)))$ of objects at level j and similarly ω-*Sets*$_1(j)$ for the object of arrows. Together, they form an internal subcategory of any of the ω-*Sets*$(j+1) \ldots \omega$-*Sets*. We may refer to ω-*Sets*(j) as the intended *external* interpretation of `Type(j)` and to ω-*Sets*$_0(j)$ as the *internal* one.

□

Let Dom_0 be an ω-set providing the objects of the internal category *Dom* of predomains; the externalization map `E:Dom->Type(0)` is interpreted as a morphism of the form $Dom_0 \to \omega\text{-}Sets_0(0)$. The arrows of *Dom* are elements of the $(Dom_0 \times Dom_0)$-indexed family $\{a : \omega\text{-}Sets_1(0) \mid Dom(a) = E(X) \wedge cod(a) = E(Y)\}$ obtained by pulling back $\langle Dom, cod\rangle : \omega\text{-}Sets_1(0) \to \omega\text{-}Sets_0(0) \times \omega\text{-}Sets_0(0)$ along $[\![E]\!] \times [\![E]\!]$. The category ω-*Sets* has all finite limits because it is a full reflective subcategory of a complete category and hence such a pullback exists and it provides suitable domain and codomain maps for *Dom*.

To get inductive types it is enough to choose *PER* as *Dom*. *PER* is cartesian closed and the inclusion into ω-*Sets* factorizing through the equivalence $\Psi : PER \to Mod$ between *PER* and the category of modest sets is a full and faithful functor; $[\![E]\!]$ is the object map of such inclusion. Since *PER* is a small complete category, one gets initial T-algebras from the limit to the forgetful functor $PER^T \to PER$ (see [Lam93]).

To get fixed points we must give some domain structure to our objects. A suitable condition to ask for is *extensionality* which induces a canonical partial order in the domains of PERs (where antisymmetry holds modulo the relation). Moreover, any map between extensional PERs is continuous with respect to a suitable notion of ω-chain. So we further restrict *PER* and consider, as category of predomains, the algebraically complete subcategory *ExP* of extensional PERs studied in [FMRS90].

ExP is a small-complete cartesian closed category. This is because any full reflective subcategory of a small-complete category is also small-complete. Then, *ExP* gets all small limits from *PER* and hence it is suitable for modelling inductive types. As for cartesian closedness, we must make sure that, for A and B in *ExP*, the extensional PER $A \Rightarrow B$, as defined in [FMRS90], is complete. A way of doing this is by brute force, that is by showing that $sup(\mu) : A \Rightarrow B$ for any ascending sequence μ in $A \Rightarrow B$.

There is a more abstract proof based on the internal version of the above result on limits. Firstly, *ExP* is an internal category in ω-*Sets*, just like *PER*. If $\mathcal{D}$ is internally complete in ω-*Sets*, then the fibration $[\mathcal{D}] : \Sigma(\mathcal{D}) \to \omega\text{-}Sets$ (see [Jac91, 1.4.4]) is complete (see [Pho92, 2.3.5]), so that we get exponentials B^A in $\mathcal{D}$ as $\Pi_A B$. Then, the fact that *PER* is complete relative to ω-*Sets* ([Pho92, 4.3.20]) fires the above implications and gives us exponentials in *ExP*.

Finally, we axiomatize fixed points only for endomorphisms on types of programs $A \Rightarrow TB$ which we assume to have a least element. Such fixed points always exist since any map between extensional PERs is continuous.

8 Directions for further research

We have presented three applications of calculi based on monads and considered a variety of theoretical matters concerning the use of monads for modelling computation. Here, we list some issues that we believe deserve further attention.

Evaluation relations. In chapter 4, we obtained evaluation relations from an endofunctor and first order quantification; we argued that such relations are useful both semantically and proof-theoretically. In order to use evaluation relations in a program logic based on the computational metalanguage, we need general axioms relating them with the operations *val* and *let*. For example, while $M \Leftarrow \mathit{val}(M)$ can be expected to hold of most notions of computation under the interpretation (4.8) of $\Leftarrow$, not so for the rule

$$\frac{M \Leftarrow E \qquad N \Leftarrow F(M)}{N \Leftarrow \mathit{let}(F, E)}$$

proposed in [CP92] to weaken the notion of evaluation in Fix-Logic, whose soundness depends on strong assumptions on *let*. In a weaker version of the above rule, the hypothesis $N \Leftarrow F(M)$ can be replaced by $\mathit{val}(N) = F(M)$. We have not yet investigated the strength of this or other axiomatizations.

Another question to be considered is how $\Leftarrow$, as in (4.8), depends on lifting. From theorem 4.8.2, we reckon that (4.8) can be weaker (in the sense of producing a larger relation) than (4.5) when the lifting monad $(\_)_\perp$ does not classify all predicates of the logic. For example, if $(\_)_\perp$ is obtained from the trivial class of admissible monos including only isomorphisms, so that $X_\perp \cong X$, we have that $x \Leftarrow_1 w$ always holds. Then, in the case of the powerset monad, we obtain $* \Leftarrow_1 \emptyset$, even though $* \notin \emptyset$. This raises the question of how can $\Leftarrow$ be strengthened or weakened depending on the choice of $(\_)_\perp$ and whether this dependency can be exploited to capture different notions of evaluation.

A third question to investigate concerning the use of evaluation relations is whether, by exploiting the idea of propositions-as-types, they can be used to combine computational and dependent types. Computational types do not mix well with dependent types. To see this, consider the formation rule for *let*:

$$\frac{\Gamma \vdash M : T\tau \quad \Gamma, x : \tau \vdash N : T\sigma}{\Gamma \vdash \mathit{let}\ x \Leftarrow M\ \mathit{in}\ N : T\sigma}$$

If the type σ depends on the variable $x : \tau$, the type $T\sigma$ in the conclusion of this rule would incorrectly still depend on x. Instead, such an x should be bound to a possible value produced by M, as it is in $\mathit{let}\ x \Leftarrow M\ \mathit{in}\ N$. A relation $\Leftarrow_A$, viewed as the type of all pairs of $x : A$ and $w : TA$ such that $x \Leftarrow_A w$, can be used in combination with dependent products and sums to bind variables of type A to the result of A-computations. In view of the formulae (4.6) and (4.7), two natural candidates are:

$$\begin{aligned} \Pi\, x \Leftarrow M.\, \sigma(x) &\stackrel{\text{def}}{=} \Pi\, x : A.\, (x \Leftarrow_A M) \to \sigma(x) \\ \Sigma\, x \Leftarrow M.\, \sigma(x) &\stackrel{\text{def}}{=} \Sigma\, x : A.\, (x \Leftarrow_A M) \times \sigma(x). \end{aligned}$$

Logical relations and computation. Logical relations are widely used in the study of lambda calculi [Tai67,Plo73,Sta85,MM85,Mit90,Her93] and programming language semantics [Rey83,Abr90a,AJ91,MR91,MS92,OT93].

In [Her93], a categorical view of logical predicates for the simply typed lambda calculus is given in terms of fibrations with logical structure. Let $p : \mathcal{E} \to \mathcal{B}$ be a fibred cartesian closed category with indexed products over a cartesian closed category $\mathcal{B}$; in this case, $\mathcal{E}$ is also cartesian closed and p strictly preserves the cartesian closed structure [Her93, 3.3.11]. The idea is that $\mathcal{B}$ is a category of semantic objects, $\mathcal{E}$ is a category of predicates and the internal language of p provides a fragment of first order logic, with implication, conjunction and universal quantification, to talk about the semantic objects. Given a cartesian closed category $\mathcal{L}$, viewed as a theory of the simply typed lambda calculus, and a structure-preserving functor $[\![\_]\!] : \mathcal{L} \to \mathcal{B}$, viewed as an interpretation of $\mathcal{L}$ in $\mathcal{B}$, a family $\mathcal{P} = \{\mathcal{P}_\sigma\}_{\sigma \in |\mathcal{L}|}$ of logical predicates, indexed by the types of $\mathcal{L}$, is defined by a structure preserving functor $\mathcal{P} : \mathcal{L} \to \mathcal{E}$ such that $p \circ \mathcal{P} = [\![\_]\!]$.

To convince oneself that such a $\mathcal{P}$ is "logical," consider the predicate $\mathcal{P}_{\sigma\to\tau}$ over $[\![\sigma \to \tau]\!] = [\![\sigma]\!] \to [\![\tau]\!]$; since $\mathcal{P}$ preserves the cartesian closed structure, we have that $\mathcal{P}_{\sigma\to\tau}$ is equivalent to $\mathcal{P}_\sigma \to \mathcal{P}_\tau$, which, in the internal language of p is written:

$$f : \sigma \to \tau \vdash \forall\, x : \sigma.\ \mathcal{P}_\sigma(x) \supset \mathcal{P}_\tau(f(x)).$$

However, when computation is involved, Kleisli exponentials are used to interpret arrow types and hence the above picture must change. In chapter 3, programs of type $\sigma \to \tau$ were interpreted as elements of $[\![\sigma]\!] \to T[\![\tau]\!]$ and hence, in order to define the formal approximation relation $\leq_{\sigma\to\tau}$ *logically* (section 3.6), we used a relation $\preceq_\tau$ extending $\leq_\tau$ to computations. We believe that a general categorical picture of logical relations involving computational types should emerge from an abstract understanding of the relation between the two families $\leq$ and $\preceq$.

Once such a picture is available, one can tackle the more ambitious goal of extending monad constructors to logical relations in the attempt to modularize the study of proof-theoretic and semantic properties of computation. For example, if $\leq$ and $\preceq$ are related by an action $\tilde{T}$ of the underlying functor of the monad T on a relational structure, corresponding families of relations could be obtained for the notion of computation $\mathcal{F}(T)$ by applying an "action constructor" $\tilde{\mathcal{F}}$ to $\tilde{T}$.

Recursive predicates and computation. In section 5.4, `while`-programs were extended to include *top-level* assertions and, in section 5.5, this simple form of annotated programs were used in a proof of partial correctness. General annotated `while`-programs are obtained by including annotated programs (p, θ) in the syntax of `while`-programs. However, in order to express the weakest precondition of a `while`-loop whose body includes assertions, a fixed point operator on predicates is needed:

$$wp\,(\texttt{while}\ B\ \texttt{do}\ C\ \texttt{od}) \ \stackrel{\text{def}}{=}\ \mu P.\, B \supset (wp\,(C) \wedge [C]\ P).$$

In [BG87], A. Blass and Y. Gurevich deal with annotated `while`-programs by augmenting a universal quantification-free fragment of first order logic with inductive definitions of predicates. The *Existential Fixed-Point Logic* thus obtained allows them to remove the *expressivity* hypothesis in Cook's completeness theorem for Hoare logic [Coo78]. According to the authors, the need for such an hypothesis in Cook's proof indicates that first order logic is not the best one for dealing with programs. An interesting issue that we would like to explore in further research is the interaction between evaluation modalities and recursive predicates. This would also call for a comparison between a version of EL extended with a μ operator and existing modal μ-calculi (see [Sti92]).

Duality. Relative interpretations of HML theories are characterized by theorem 6.8.12 as morphisms in the slice category $LFP/\lambda\overline{\omega}$-Cat. We argued that this characterization can be adapted to any formal language whose models are described by a finite limit theory:

Theorem 8.0.1 *Any lfp category $\mathcal{M}$ fully embeds in the opposite of $LFP/\mathcal{M}$ via the inclusion $I : \mathcal{M}^{op} \to LFP/\mathcal{M}$ mapping X to $?_X/\mathcal{M} : X/\mathcal{M} \to \emptyset/\mathcal{M} \cong \mathcal{M}$.*

In order to turn the above theorem into a duality result, one should single out, by the intrinsic properties of its objects, a full subcategory $\mathcal{H}$ of $LFP/\mathcal{M}$ such that the above inclusion I restricts to $\mathcal{H}$ and is part of an adjoint equivalence $\mathcal{M}^{op} \cong \mathcal{H}$. What we know of the objects $G : \mathcal{A} \to \mathcal{M}$ of such an $\mathcal{H}$ is that they are *monadic* functors (because so are the codomain functors $cod_X : X/\mathcal{M} \to \mathcal{M}$) such that $\mathcal{A} \cong G(\emptyset)/\mathcal{M}$.

Semantic constructors and logics. The theory of semantic constructors, which was developed in chapter 6 for *equational* theories of computation, should be extended to richer logics, such as EL. Theorem 5.1.10 is an example where we ask whether a property expressed in EL is preserved by a semantic constructor. The closure of such constructors with respect to satisfaction of formulae should be investigated in further research. Related questions are whether the setting of synthetic domain theory that we adopted in chapter 4 is the most appropriate for studying the behaviour of semantic constructors with respect to logics and whether the standard interpretation of logic is beneficial in this context.

A The metalanguage HML

In the simply typed lambda calculus, types can be described separately from the terms. However, in some type theories where terms and types are mutually dependent, the formation rules of the ones cannot be given independently from the formation rules of the others. In HML, the integration of entities of different syntactic nature in a unique formal system is pushed further by including, besides types, objects such as signatures and theories that traditionally belong to the metatheory of the formal system.

Let s be a symbol in the set $\{sig, theory, kind, context, scheme, prop\}$; the prase "$\alpha\ s$" in the conclusion of an HML sequent states the well formedness of an object α of the syntactic category corresponding to s, that is: signatures, theories, kinds, contexts, type schemes and formulae. Among these symbols, *kind* and *scheme* are "variable universes," in the sense that a kind or a type scheme, that is an α such that $(\alpha\ s)$ is derivable, for $s \in \{kind, scheme\}$, can be used in a context as the range of a variable. An object u of kind k is called an *operator*, while an object e of type scheme σ is called a *term*. The well formedness of operators and terms is stated by sequents whose conclusion is $u : k$ and $e : \sigma$ respectively. Besides judgements of well formedness, HML features equality judgements on operators and schemes, and truth judgements involving formulae of first order logic with equality.

Note that *prop* could also be viewed as a variable universe: although there are no explicit expressions whose "type" is a formula, a truth judgement $\Gamma; \Delta \vdash_{\Sigma,\mathcal{T}} \phi$ (see below) asserting the truth of ϕ under the assumptions in Δ, can be viewed as introducing an invisible witness of ϕ. In this framework, the assumptions in Δ can be viewed as extending the context Γ and the axioms in the theory $\mathcal{T}$ as extending the signature Σ.

Type schemes depend on kinds, in the sense that, in a judgement $\Gamma, v : k \vdash \sigma\ \mathit{scheme}$, σ may contain a free occurrence of v of kind k. Note that a scheme $\Gamma \vdash \sigma\ \mathit{scheme}$ cannot contain any variable $x : \sigma$ ranging over a scheme that may be in Γ, that is: schemes do not depend on schemes. Formulae can contain both operator and term variables. *Dependency relations* between sorts [Jac91, 2.1.1] are often used to present type theories in view of a study of abstract models based on fibrations.

HML is inspired by the type theory described in [Mog91a]; the following definition comes, with minor changes, from [CM93], to which we added the raw syntax and the rules (add-P), (eqn) and (P).

Let Var_O and Var_E be disjoint countable sets of variables and let Const_K, Const_O, Const_S, Const_E and Const_P be disjoint sets of constants. The sets of *raw kinds* ($\mathcal{K}$), *operators* ($\mathcal{U}$), *type schemes* ($\mathcal{S}$), *terms* ($\mathcal{E}$), *formulae* ($\mathcal{P}$), *contexts* ($\mathfrak{C}$), *signatures* ($\mathfrak{S}$) and *theories* ($\mathfrak{T}$) of HML are given by the following grammar:

$$
\begin{array}{lcl}
k & ::= & K \mid 1 \mid k_1 \times k_2 \mid k_1 \Rightarrow k_2 \\
u & ::= & v \mid C \mid \ * \ \mid \langle u_1, u_2 \rangle \mid \pi_i(u) \mid \lambda\, v : k.\, u \mid u_1(u_2) \\
\sigma & ::= & u \mid S(u) \mid 1 \mid \sigma_1 \times \sigma_2 \mid \sigma_1 \Rightarrow \sigma_2 \mid \Pi\, v : k.\, \sigma \\
e & ::= & x \mid c \mid \ * \ \mid \langle e_1, e_2 \rangle \mid \pi_i(e) \mid \lambda\, x : \sigma.\, e \mid e_1(e_2) \mid \Lambda\, v : k.\, e \mid e[u] \\
\phi & ::= & e_1 =_\sigma e_2 \mid P(u, e) \mid \mathit{false} \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \supset \phi_2 \mid \forall\, v : k.\, \phi \mid \exists\, v : k.\, \phi \mid \\
 & & \forall\, x : \sigma.\, \phi \mid \exists\, x : \sigma.\, \phi \\
\Gamma & ::= & \emptyset \mid \Gamma, v : k \mid \Gamma, x : \sigma \\
\Sigma & ::= & \emptyset \mid \Sigma, K : \mathit{kind} \mid \Sigma, C : k \mid \Sigma, S : k \Rightarrow \mathit{scheme} \mid \Sigma, c : \sigma \mid \\
 & & \Sigma, P : (v : k)\sigma \Rightarrow \mathit{prop} \\
\mathcal{T} & ::= & \emptyset \mid \mathcal{T}, \phi
\end{array}
$$

where $v \in \mathit{Var}_O$, $x \in \mathit{Var}_E$, $K \in \mathit{Const}_K$, $C \in \mathit{Const}_O$, $S \in \mathit{Const}_S$, $c \in \mathit{Const}_E$ and $P \in \mathit{Const}_P$. The above grammar defines the *raw syntax* of HML.

The usual notions of free variable, substitution and α-conversion applies to raw operators, schemes, terms and formulae. The only nonstandard notation occurs in signatures in the arity of predicates $P : (v : k)\sigma \Rightarrow \mathit{prop}$; this indicates that σ has at most one free variable v of kind k.

The sequents of HML come in different shapes, depending on the kind of judgement they make. In the following table, the expressions in the first column define the set of judgements whose elements are classified in the second column and written as in the third.

$\mathfrak{S} \times \mathcal{K}$	kind formation	$\vdash_\Sigma k\ \mathit{kind}$
$\mathfrak{S} \times \mathfrak{C} \times \mathcal{U} \times \mathcal{K}$	operator formation	$\Gamma \vdash_\Sigma u : k$
$\mathfrak{S} \times \mathfrak{C} \times \mathcal{S}$	scheme formation	$\Gamma \vdash_\Sigma \sigma\ \mathit{scheme}$
$\mathfrak{S} \times \mathfrak{C} \times \mathcal{E} \times \mathcal{S}$	term formation	$\Gamma \vdash_\Sigma e : \sigma$
$\mathfrak{S} \times \mathfrak{C} \times \mathcal{P}$	formula formation	$\Gamma \vdash_\Sigma \phi\ \mathit{prop}$
$\mathfrak{S} \times \mathfrak{C}$	context formation	$\vdash_\Sigma \Gamma\ \mathit{context}$
$\mathfrak{S} \times \mathfrak{C} \times \mathcal{U} \times \mathcal{U} \times \mathcal{K}$	operator equality	$\Gamma \vdash_\Sigma u_1 = u_2 : k$
$\mathfrak{S} \times \mathfrak{C} \times \mathcal{S} \times \mathcal{S}$	scheme equality	$\Gamma \vdash_\Sigma \sigma_1 = \sigma_2$
$\mathfrak{S} \times \mathfrak{T} \times \mathfrak{C} \times \mathcal{P}^* \times \mathcal{P}$	truth	$\Gamma, \Delta \vdash_{\Sigma,\mathcal{T}} \phi$

where $\Delta \in \mathcal{P}^*$ is a possibly empty sequence of raw formulae. The type theory HML is defined by the following inference rules over the above sets of judgements:

Signatures

$$\text{start}\ \frac{}{\vdash \emptyset\ \mathit{sig}}$$

$$\text{add-}K\ \frac{\vdash \Sigma\ \mathit{sig}}{\vdash \Sigma, K : \mathit{kind}\ \ \mathit{sig}}\ K \notin \mathit{dom}(\Sigma)$$

$$\text{add-}C\ \frac{\vdash_\Sigma k\ \mathit{kind}}{\vdash \Sigma, C : k\ \ \mathit{sig}}\ C \notin \mathit{dom}(\Sigma)$$

$$\text{add-}S\ \frac{\vdash_\Sigma k\ \mathit{kind}}{\vdash \Sigma, S : k \Rightarrow \mathit{scheme}\ \ \mathit{sig}}\ S \notin \mathit{dom}(\Sigma)$$

$$\text{add-}c\ \frac{\emptyset \vdash_\Sigma \sigma\ \mathit{scheme}}{\vdash \Sigma, c : \sigma\ \mathit{sig}}\ c \notin \mathit{dom}\,(\Sigma)$$

$$\text{add-}P\ \frac{v : k \vdash_\Sigma \sigma\ \mathit{scheme}}{\vdash \Sigma, P : (v : k)\sigma \Rightarrow \mathit{prop}\ \ \mathit{sig}}\ P \notin \mathit{dom}\,(\Sigma)$$

Theories

$$\emptyset\ \frac{\vdash \Sigma\ \mathit{sig}}{\vdash_\Sigma \emptyset\ \mathit{theory}} \qquad \text{add-prop}\ \frac{\vdash_\Sigma \mathcal{T}\ \mathit{theory} \qquad \vdash_\Sigma \phi\ \mathit{prop}}{\vdash_\Sigma \mathcal{T}, \phi\ \ \mathit{theory}}$$

Kinds

$$K\ \frac{\vdash \Sigma\ \mathit{sig}}{\vdash_\Sigma K\ \mathit{kind}}\ \Sigma(K) = \mathit{kind} \qquad 1\ \frac{\vdash \Sigma\ \mathit{sig}}{\vdash_\Sigma 1\ \mathit{kind}} \qquad \Omega\ \frac{\vdash \Sigma\ \mathit{sig}}{\vdash_\Sigma \Omega\ \mathit{kind}}$$

$$\times\ \frac{\vdash_\Sigma k_1\ \mathit{kind} \qquad \vdash_\Sigma k_2\ \mathit{kind}}{\vdash_\Sigma k_1 \times k_2\ \mathit{kind}} \qquad \Rightarrow\ \frac{\vdash_\Sigma k_1\ \mathit{kind} \qquad \vdash_\Sigma k_2\ \mathit{kind}}{\vdash_\Sigma k_1 \Rightarrow k_2\ \mathit{kind}}$$

Contexts

$$\emptyset\ \frac{\vdash \Sigma\ \mathit{sig}}{\vdash_\Sigma \emptyset\ \mathit{context}}$$

$$\text{add-}v\ \frac{\vdash_\Sigma \Gamma\ \mathit{context} \qquad \vdash_\Sigma k\ \mathit{kind}}{\vdash_\Sigma \Gamma, v : k\ \ \mathit{context}}\ v \notin \mathit{dom}\,(\Gamma)$$

$$\text{add-}x\ \frac{\vdash_\Sigma \Gamma\ \mathit{context} \qquad \Gamma \vdash_\Sigma \sigma\ \mathit{scheme}}{\vdash_\Sigma \Gamma, x : \sigma\ \ \mathit{context}}\ x \notin \mathit{dom}\,(\Gamma)$$

Operators

$$v\ \frac{\vdash_\Sigma \Gamma\ \mathit{context}}{\Gamma \vdash_\Sigma v : k}\ \Gamma(v) = k \qquad C\ \frac{\vdash_\Sigma \Gamma\ \mathit{context}}{\Gamma \vdash_\Sigma C : k}\ \Sigma(C) = k \qquad 1\mathrm{I}\ \frac{\vdash_\Sigma \Gamma\ \mathit{context}}{\Gamma \vdash_\Sigma * : 1}$$

$$\times\mathrm{I}\ \frac{\Gamma \vdash_\Sigma u_1 : k_1 \qquad \Gamma \vdash_\Sigma u_2 : k_2}{\Gamma \vdash_\Sigma \langle u_1, u_2\rangle : k_1 \times k_2} \qquad \times\mathrm{E}\ \frac{\Gamma \vdash_\Sigma u : k_1 \times k_2}{\Gamma \vdash_\Sigma \pi_i(u) : k_i}$$

$$\Rightarrow\mathrm{I}\ \frac{\Gamma, v : k_1 \vdash_\Sigma u : k_2}{\Gamma \vdash_\Sigma (\lambda\, v : k_1.\, u) : k_1 \Rightarrow k_2} \qquad \Rightarrow\mathrm{E}\ \frac{\Gamma \vdash_\Sigma u : k_1 \Rightarrow k_2 \qquad \Gamma \vdash_\Sigma u_1 : k_1}{\Gamma \vdash_\Sigma u(u_1) : k_2}$$

Operator equality

Operator equality is the congruence generated by the following rules:

$$1.\eta\ \frac{\Gamma \vdash_\Sigma u : 1}{\Gamma \vdash_\Sigma * = u : 1}$$

$$\times.\beta\ \frac{\Gamma \vdash_\Sigma u_1 : k_1 \qquad \Gamma \vdash_\Sigma u_2 : k_2}{\Gamma \vdash_\Sigma \pi_i\langle u_1, u_2\rangle = u_i : k_i}$$

$$\times.\eta\ \frac{\Gamma \vdash_\Sigma u : k_1 \times k_2}{\Gamma \vdash_\Sigma \langle \pi_1(u), \pi_2(u)\rangle = u : k_1 \times k_2}$$

$$\Rightarrow.\beta\ \frac{\Gamma, v : k_1 \vdash_\Sigma u_2 : k_2 \qquad \Gamma \vdash_\Sigma u_1 : k_1}{\Gamma \vdash_\Sigma (\lambda\, v : k_1.\, u_2)(u_1) = [u_1/v]u_2 : k_2}$$

$$\Rightarrow.\eta\ \frac{\Gamma \vdash_\Sigma u : k_1 \Rightarrow k_2}{\Gamma \vdash_\Sigma (\lambda\, v : k_1.\, u(v)) = u : k_1 \Rightarrow k_2}$$

Type schemes

$$1\ \frac{\vdash_\Sigma \Gamma\ \mathit{context}}{\Gamma \vdash_\Sigma 1\ \mathit{scheme}} \qquad S\ \frac{\Gamma \vdash_\Sigma u : k}{\Gamma \vdash_\Sigma S(u)\ \mathit{scheme}}\ \Sigma(S) = k \Rightarrow \mathit{scheme}$$

$$\text{type}\ \frac{\Gamma \vdash_\Sigma u : \Omega}{\Gamma \vdash_\Sigma u\ \mathit{scheme}} \qquad \times\ \frac{\Gamma \vdash_\Sigma \sigma_1\ \mathit{scheme} \qquad \Gamma \vdash_\Sigma \sigma_2\ \mathit{scheme}}{\Gamma \vdash_\Sigma \sigma_1 \times \sigma_2\ \mathit{scheme}}$$

$$\Rightarrow\ \frac{\Gamma \vdash_\Sigma \sigma_1\ \mathit{scheme} \qquad \Gamma \vdash_\Sigma \sigma_2\ \mathit{scheme}}{\Gamma \vdash_\Sigma \sigma_1 \Rightarrow \sigma_2\ \mathit{scheme}} \qquad \Pi\ \frac{\Gamma, v : k \vdash_\Sigma \sigma\ \mathit{scheme}}{\Gamma \vdash_\Sigma (\Pi\, v : k.\, \sigma)\ \mathit{scheme}}$$

Type scheme equality

Type scheme equality is the congruence induced by operator equality.

Terms

$$x\ \frac{\vdash_\Sigma \Gamma\ \mathit{context}}{\Gamma \vdash_\Sigma x : \sigma}\ \Gamma(x) = \sigma \qquad c\ \frac{\vdash_\Sigma \Gamma\ \mathit{context}}{\Gamma \vdash_\Sigma c : \sigma}\ \Sigma(c) = \sigma \qquad 1\mathrm{I}\ \frac{\vdash_\Sigma \Gamma\ \mathit{context}}{\Gamma \vdash_\Sigma * : 1}$$

$$\times\mathrm{I}\ \frac{\Gamma \vdash_\Sigma e_1 : \sigma_1 \qquad \Gamma \vdash_\Sigma e_2 : \sigma_2}{\Gamma \vdash_\Sigma \langle e_1, e_2 \rangle : \sigma_1 \times \sigma_2} \qquad \times\mathrm{E}\ \frac{\Gamma \vdash_\Sigma e : \sigma_1 \times \sigma_2}{\Gamma \vdash_\Sigma \pi_i(e) : \sigma_i}$$

$$\Rightarrow\mathrm{I}\ \frac{\Gamma, x : \sigma_1 \vdash_\Sigma e : \sigma_2}{\Gamma \vdash_\Sigma (\lambda\, x : \sigma_1.\, e) : \sigma_1 \Rightarrow \sigma_2} \qquad \Rightarrow\mathrm{E}\ \frac{\Gamma \vdash_\Sigma e : \sigma_1 \Rightarrow \sigma_2 \qquad \Gamma \vdash_\Sigma e_1 : \sigma_1}{\Gamma \vdash_\Sigma e(e_1) : \sigma_2}$$

$$\Pi\mathrm{I}\ \frac{\Gamma, v : k \vdash_\Sigma e : \sigma}{\Gamma \vdash_\Sigma (\Lambda\, v : k.\, e) : (\Pi\, v : k.\, \sigma)} \qquad \Pi\mathrm{E}\ \frac{\Gamma \vdash_\Sigma e : (\Pi\, v : k.\, \sigma) \qquad \Gamma \vdash_\Sigma u : k}{\Gamma \vdash_\Sigma e[u] : [u/v]\sigma}$$

$$\text{conv}\ \frac{\Gamma \vdash_\Sigma e : \sigma_1 \qquad \Gamma \vdash_\Sigma \sigma_1 = \sigma_2}{\Gamma \vdash_\Sigma e : \sigma_2}$$

Formulae

Formulae are obtained by applying the usual logical operators of first order predicate calculus to the atomic formulae generated by the following rules:

$$\text{eqn}\ \frac{\Gamma \vdash_\Sigma e_1 : \sigma \qquad \Gamma \vdash_\Sigma e_2 : \sigma}{\Gamma \vdash_\Sigma e_1 =_\sigma e_2\ \mathit{prop}}$$

$$P\ \frac{\Gamma, v : k \vdash_\Sigma \sigma\ \mathit{scheme} \quad \Gamma \vdash_\Sigma u : k \quad \Gamma \vdash_\Sigma e : [u/v]\sigma}{\Gamma \vdash_\Sigma P(u, e)\ \mathit{prop}}\ \Sigma(P) = (v : k)\sigma \Rightarrow \mathit{prop}$$

Truth inference

The inference rules for truth judgements include the standard rules of intuitionistic predicate calculus with equality. Axioms are introduced by the rule:

$$\text{axiom}\ \frac{\vdash_\Sigma \mathcal{T}\ \mathit{theory}}{\Gamma \vdash_{\Sigma,\mathcal{T}} \phi}\ \phi \in \mathcal{T}$$

The equational fragment of HML is the congruence generated by operator equality and the following rules:

$$1.\eta\ \frac{\vdash_\Sigma \mathcal{T}\ \mathit{theory} \qquad \Gamma \vdash_\Sigma e : 1}{\Gamma \vdash_{\Sigma,\mathcal{T}} * =_1 e}$$

$$\times.\beta\ \frac{\vdash_\Sigma \mathcal{T}\ \mathit{theory} \qquad \Gamma \vdash_\Sigma e_1 : \sigma_1 \qquad \Gamma \vdash_\Sigma e_2 : \sigma_2}{\Gamma \vdash_{\Sigma,\mathcal{T}} \pi_i\langle e_1, e_2\rangle =_{\sigma_i} e_i}$$

$$\times.\eta\ \frac{\vdash_\Sigma \mathcal{T}\ \mathit{theory} \qquad \Gamma \vdash_\Sigma e : \sigma_1 \times \sigma_2}{\Gamma \vdash_{\Sigma,\mathcal{T}} \langle \pi_1(e), \pi_2(e)\rangle =_{\sigma_1 \times \sigma_2} e}$$

$$\Rightarrow.\beta\ \frac{\vdash_\Sigma \mathcal{T}\ \mathit{theory} \qquad \Gamma, x:\sigma_1 \vdash_\Sigma e_2:\sigma_2 \qquad \Gamma \vdash_\Sigma e_1:\sigma_1}{\Gamma \vdash_{\Sigma,\mathcal{T}} (\lambda\, x:\sigma_1.\, e_2)e_1 =_{\sigma_2} [e_1/x]e_2}$$

$$\Rightarrow.\eta\ \frac{\vdash_\Sigma \mathcal{T}\ \mathit{theory} \qquad \Gamma \vdash_\Sigma e:\sigma_1 \Rightarrow \sigma_2}{\Gamma \vdash_{\Sigma,\mathcal{T}} \lambda\, x:\sigma_1.\, e(x) =_{\sigma_1\Rightarrow\sigma_2} e}$$

$$\Pi.\beta\ \frac{\vdash_\Sigma \mathcal{T}\ \mathit{theory} \qquad \Gamma, v:k \vdash_\Sigma e:\sigma \qquad \Gamma \vdash_\Sigma u:k}{\Gamma \vdash_{\Sigma,\mathcal{T}} (\Lambda\, v:k.\, e)[u] =_{[u/v]\sigma} [u/v]e}$$

$$\Pi.\eta\ \frac{\vdash_\Sigma \mathcal{T}\ \mathit{theory} \qquad \Gamma \vdash_\Sigma e:(\forall v:k.\sigma)}{\Gamma \vdash_{\Sigma,\mathcal{T}} \Lambda\, v:k.\, e[v] =_{\Pi v:k.\sigma} e}$$

$$\text{conv-eq}\ \frac{\Gamma \vdash_{\Sigma,\mathcal{T}} e_1 =_{\sigma_1} e_2 \qquad \Gamma \vdash_\Sigma \sigma_1 = \sigma_2}{\Gamma \vdash_{\Sigma,\mathcal{T}} e_1 =_{\sigma_2} e_2}$$

B *Dom*, the category of discourse

```
[Dom1:Type = Sigma [X:Dom](Sigma [Y:Dom](E X)->E Y)];
[Dom2:Type = Sigma [X:Dom](Sigma [Y:Dom](Sigma [Z:Dom]
                                     indprod ((E X)->E Y) (E Y)->E Z))];

(*  defn  mkarr = ... : {X,Y|Dom}((E X)->E Y)->Dom1  (internalization) *)

[d0 = sig_pi1|Dom|[X:Dom](sigma|Dom|[Y:Dom](E X)->E Y):Dom1->Dom];
[d1 = [Z: Dom1]sig_pi1 (sig_pi2 Z):Dom1->Dom];
[i = [X:Dom] mkarr ([x:E X] x):Dom->Dom1];
[o = [fg:Dom2] mkarr ([x:E (sig_pi1 fg)]
                       Snd (sig_pi2 (sig_pi2 (sig_pi2 fg)))
                        (Fst (sig_pi2 (sig_pi2 (sig_pi2 fg))) x)):Dom2->Dom1];
```

C Signatures and theories

```
[Sum = <sum:Dom->Dom->Dom>
       <in1:{X1,X2|Dom}(E X1)->E(sum X1 X2)>
       <in2:{X1,X2|Dom}(E X2)->E(sum X1 X2)>
       <case:{X1,X2,X|Dom}
                ((E X1)->E X)->
                ((E X2)->E X)->
                (E(sum X1 X2))->E X> Unit ];

[AX_INL = {S:Sum}{X1,X2,X|Dom}{f:(E X1)->E X}{g:(E X2)->E X}
                 Eq ([x1:E X1] (case S f g (in1 S x1))) f];
[AX_INR = {S:Sum}{X1,X2,X|Dom}{f:(E X1)->E X}{g:(E X2)->E X}
                 Eq ([x2:E X2] (case S f g (in2 S x2))) g];
[AX_CASE = {S:Sum}{X1,X2,X|Dom}{h: (E(sum S X1 X2)) -> (E X)}
                 Eq (case S ([x:(E X1)]h(in1 S x)) ([x:(E X2)]h(in2 S x))) h];

[Prod = <prod:Dom->Dom->Dom>
        <pin:{X1,X2|Dom}(E X1)->(E X2)->E(prod X1 X2)>
        <pout1:{X1,X2|Dom}(E (prod X1 X2))->E X1>
        <pout2:{X1,X2|Dom}(E (prod X1 X2))->E X2> Unit ];

[Exponential = <arr: Dom->Dom->Dom>
               <lmb: {X,Y|Dom}((E X)->E Y)->E(arr X Y)>
               <app: {X,Y|Dom}(E (arr X Y))->(E X)->E Y> Unit];
```

The usual axioms for products and exponentials can be defined as for sums.

```
[Ind = <mu:Functor->Dom>
       <alg:{F|Functor}(E (FT F(mu F)))->E(mu F)>
       <it:{F|Functor}{X|Dom}
           ((E(FT F X))->E X) -> (E (mu F)) -> E X> Unit ];

[AX_WEAK_INITIAL = {Z:Ind}{F|Functor}{X|Dom}
                   {a:(E (FT F X)) -> E X}
                     Eq ([z: E (FT F (mu Z F))] it Z a (alg Z z))
                        ([z: E (FT F (mu Z F))] a (str F (it Z a) z))];

[AX_UNIVERSAL = {Z:Ind}{F|Functor}{X|Dom}
                   {a:(E (FT F X)) -> E X}
                   {g:(E (mu Z F)) -> E X}
                     ({z: E(FT F (mu Z F))} Eq (g(alg Z z)) (a (str F g z)))
                     -> Eq ([x: E (mu Z F)] it Z a x) g];
```

The operations in `Ind` are shown to satisfy Lambek's lemma, that is: the initial F-algebra `alg` is an isomorphism with inverse `coalg = it(F(alg))`:

```
[coalg [Z:Ind][F|Functor] = [x:E (mu Z F)] it Z (str F(alg Z|F)) x];

              (* Lemma LM_LAMBEK1: it(alg)=id
                 Lemma LM_LAMBEK2: coalg;alg=id
                 Lemma LM_LAMBEK3: alg;coalg=id *)

[LM_LAMBEK1 = {Z:Ind}{F|Functor}
      Eq ([x:E (mu Z F)]x) ([x:E (mu Z F)] it Z (alg Z|F) x)];

Goal LM_LAMBEK1;
Intros;
Claim {z:E (FT F (mu Z F))} Eq (alg Z z)(alg Z|F (str F ([y:E (mu Z F)]y) z));
Qrepl ax_universal Z (alg Z|F) ([y:E (mu Z F)]y) ?+1;
Immed;
Intros;
(* Due to a LEGO bug, "Qrepl ax_str_id..." would not work here *)
Refine (Eq_sym(ax_str_id F|(mu Z F)))
        ([k:(E (FT F (mu Z F)))->E (FT F (mu Z F))] P1 (alg Z (k z))) H1;
Save lm_Lambek1;


[LM_LAMBEK2 = {Z:Ind}{F|Functor}
      Eq ([x:E (mu Z F)]x) ([x:E (mu Z F)] alg Z (coalg Z x))];
[LM_LAMBEK3 = {Z:Ind}{F|Functor}
             Eq ([z:E (FT F(mu Z F))] z)
                ([z:E (FT F(mu Z F))] coalg Z (alg Z z))];
```

The proofs of `LM_LAMBEK2` and `LM_LAMBEK3` are similar to the one above.

D Encoding ML_T

```
[Grd: Type];

Inductive [Ty: Type] Constructors
  [GR: Grd->Ty]
  [AR: Ty->Ty->Ty]
  [TT: Ty->Ty];

[Con = list Ty]
[Con_elim = list_elim Ty];

Inductive [Var:Con->Ty->Type] Constructors
  [Vo: {c:Con}{t:Ty}Var (cons t c) t]
  [Weak: {c|Con}{s:Ty}{t|Ty}(Var c t)->Var (cons s c) t];

Inductive [Exp:Con->Ty->Type] Constructors
  [VAR: {c|Con}{t|Ty}(Var c t)->Exp c t]
  [LMB: {c|Con}{s,t|Ty}(Exp (cons s c) t)->Exp c (AR s t)]
  [APP: {c|Con}{s,t|Ty}(Exp c (AR s t))->(Exp c s)->Exp c t]
  [LET: {c|Con}{s,t|Ty}(Exp (cons s c) (TT t))->(Exp c (TT s))->Exp c (TT t)]
  [VAL: {c|Con}{t|Ty}(Exp c t)->Exp c (TT t)];
```

E Interpretation of ML_T

```
[I_Grd: Grd->Dom];

[I_Ty [Ex:Exponential] [M:Monad]: Ty->Dom = Ty_elim
  ([_|Ty]Dom)
  I_Grd
  ([_,_|Ty][X,Y:Dom]arr Ex X Y)
  ([_|Ty][X:Dom]T M X)];

[I_Con [P:Prod] [Ex:Exponential] [M:Monad]: Con->Dom = Con_elim
  ([_|Con]Dom)
  (one:Dom)
  ([t:Ty][_:Con][X:Dom] prod P X (I_Ty Ex M t))];

[I_Var [P:Prod] [Ex:Exponential] [M:Monad] = Var_elim
  ([c|Con][t|Ty][_:Var c t](E (I_Con P Ex M c))->E (I_Ty Ex M t))
  ([c:Con][t:Ty]pout2 P|(I_Con P Ex M c)|(I_Ty Ex M t))
  ([c|Con][s:Ty][t|Ty][_:Var c t][F:(E (I_Con P Ex M c))->E (I_Ty Ex M t)]
    [x: E (prod P (I_Con P Ex M c) (I_Ty Ex M s))] F (pout1 P x))];
```

```
[I_Exp [P:Prod] [Ex:Exponential] [M:Monad] =
  [I_Ty=I_Ty Ex M] [I_Con=I_Con P Ex M] [I_Var=I_Var P Ex M] Exp_elim
  ([c|Con][t|Ty][_:Exp c t](E (I_Con c))->E (I_Ty t))

  (* variables *)
  ([c|Con][t|Ty][v:Var c t] I_Var v)

  (* lambda abstraction *)
  ([c|Con][s,t|Ty][_:Exp (cons s c) t]
    [F:(E (I_Con (cons s c)))->E (I_Ty t)]
    [x: E (I_Con c)] lmb Ex ([y: E (I_Ty s)] F (pin P x y)))

  (* application *)
  ([c|Con][s,t|Ty][_:Exp c (AR s t)][_:Exp c s]
    [F:(E (I_Con c))->E (I_Ty (AR s t))]
    [G:(E (I_Con c))->E (I_Ty s)]
    [x:E (I_Con c)] app Ex (F x) (G x))

  (* composition (let) *)
  ([c|Con][s,t|Ty][_:Exp (cons s c) (TT t)][_:Exp c (TT s)]
    [F:(E (I_Con (cons s c)))->E (I_Ty (TT t))]
    [G:(E (I_Con c))->E (I_Ty (TT s))]
    [x:E (I_Con c)] let M ([y:E (I_Ty s)] F(pin P x y)) (G x))

  (* lifting (val) *)
  ([c|Con][t|Ty][_:Exp c t][F:(E (I_Con c))->E (I_Ty t)]
    [x:E (I_Con c)]val M (F x))];
```

F Correctness of the exception constructor

```
Goal {S:Sum}{Exn:Dom}{M:Monad}AX_RUNIT (Exception_Constructor S Exn M);
Normal;
Intros;
Equiv P ([x:E X]([z:E(sum S X Exn)]
                 (let M (case S f ([x'2:E Exn] val M (in2 S x'2)))
                        (val M z)))(in1 S x));
Qrepl Eq_sym (ax_runit M|(sum S X Exn)|(sum S Y Exn)|
                      (case S f ([x'2:E Exn] val M (in2 S x'2))));
Qrepl ax_inl
      S|X|Exn|(T M (sum S Y Exn))|f|([x'2:E Exn] val M (in2 S x'2));
Immed;
```

```
Goal {S:Sum}{Exn:Dom}{M:Monad}AX_ASSOC (Exception_Constructor S Exn M);
Normal;
Intros;
Qrepl Eq_sym (ax_assoc M|(sum S X Exn)|(sum S Y Exn)|(sum S Z Exn)|
              c|(case S f ([x:E Exn] val M (in2 S x)))|
              (case S g ([x:E Exn] val M (in2 S x))));
Claim {S|Sum}{M|Monad}{X,Y,A,B|Dom}{k:(E A)->(E (T M B))}
      {f:(E X)->(E (T M A))}{g: (E Y)->(E (T M A))}
      Eq ([w: E (sum S X Y)] let M k (case S f g w))
         (case S ([x: E X] let M k (f x))
         ([y: E Y] let M k (g y)));
Qrepl ?+1|S|M|X|Exn|(sum S Y Exn)|(sum S Z Exn)|
         (case S g ([x'2:E Exn] val M (in2 S x'2)))|
         f|([x'2:E Exn] val M (in2 S x'2));
Equiv P (let M (case S
                  ([x:E X]let M (case S g ([x'2:E Exn] val M (in2 S x'2)))
                                (f x))
                  ([y:E Exn]
                    ([z: E (sum S Y Exn)]
                      let M (case S g ([x'2:E Exn] val M (in2 S x'2)))
                            (val M z))(in2 S y)))
                 c);
Qrepl Eq_sym (ax_runit M|(sum S Y Exn)|(sum S Z Exn)|
                    (case S g ([x'2:E Exn] val M (in2 S x'2))));
Qrepl ax_inr S|Y|Exn|(T M (sum S Z Exn))|g|([x'2:E Exn] val M (in2 S x'2));
Immed;
Normal;
Intros;
Equiv P1 (case S1 ([x:E X1]([z:E (T M1 A)] let M1 k z)(f1 x))
     ([y:E Y1]([z:E (T M1 A)] let M1 k z)(g1 y)));
Qrepl Sum_comp|S1|X1|Y1|(T M1 A)|(T M1 B)|
      f1|g1|([z:E (T M1 A)] let M1 k z);
Immed;
```

G The operator pand of parallel composition

Let $\mathcal{F}$ be the constructor of resumptions. Given `M:Monad`, let functions `ROR` and `RY` apply uniform redefinitions to the operations of structures `N:(Nondetm M)` and `F:Fix (T M)` to produce operations of structures `(`$\mathcal{F}$`N):(Nondetm `$\mathcal{F}$`M)` and `(`$\mathcal{F}$`F):Fix (T `$\mathcal{F}$`M)`. The operator `pand` of parallel composition for computations with resumptions is defined as follows:

```
[pand [S:Sum] [P:Prod] [Z:Ind] [M:Monad] [N:Nondetm M] [F:Fix (T M)] =
  [resumptions = Resumptions S Z M]
  [Res: Dom->Dom = resumptions.1]
  [Rval: {X|Dom}(E X)->E (Res X) = resumptions.2.1]
  [Rlet: {X1,X2|Dom}((E X1)->E (Res X2))->(E (Res X1))->E (Res X2) =
         resumptions.2.2.1]
  [CH: {X1,X2|Dom}((E X1)->E (Res X2))->((E (Res X1))->E (Res X2))->
         (E (Res X1))->E (Res X2) = resumptions.2.2.2.2.1]
  [orr = ROR S Z M N]
  [yrr =RY S Z M F]
  [X,Y|Dom] [w:E(Res X)] [z:E(Res Y)]
         (((yrr [h:(E (prod P (Res X)(Res Y)))->E (Res (prod P X Y))]
                [v:E (prod P (Res X)(Res Y))]
                [v1:E (Res X)=pout1 P v] [v2:E (Res Y)=pout2 P v]
                orr (CH ([x:E X] Rlet ([y:E Y] Rval (pin P x y)) v2)
                        ([w:E (Res X)] h (pin P w v2)) v1)
                    (CH ([y:E Y] Rlet ([x:E X] Rval (pin P x y)) v1)
                        ([z:E (Res Y)] h (pin P v1 z)) v2)) (pin P w z)):
                                  E (Res (prod P X Y)))];
```

H Commutativity of pand

```
Goal THM_COMMUTATIVITY;

Expand THM_COMMUTATIVITY pand pand;

intros;

[CH=resumptions.2.2.2.2.1][orr=ROR S Z M N][yrr=RY S Z M F];

Intros 1;

Equiv P1
  (([X,Y|Dom][f:(E X)->E Y]Rlet
   ([x:E X]Rval (f x))) (pswap P|X1|X2)
  ((yrr ([h:(E (prod P (Res1 X1) (Res1 X2)))->E (Res1 (prod P X1 X2))]
       [v:E (prod P (Res1 X1) (Res1 X2))][v1=pout1 P v][v2=pout2 P v]
       orr (CH ([x:E X1]Rlet ([y:E X2]Rval (pin P x y)) v2)
       ([w'16:E (Res1 X1)]h (pin P w'16 v2)) v1)
           (CH ([y:E X2]Rlet ([x:E X1]Rval (pin P x y)) v1)
       ([z'16:E (Res1 X2)]h (pin P v1 z'16)) v2))
  (pin P w z))));

Qrepl Eq_sym (lm_pswap1 P z w);
```

```
Equiv P1
(([K:(E (prod P (Res1 X1) (Res1 X2)))->E (Res1 (prod P X1 X2))]
   [v:E (prod P (Res1 X2) (Res1 X1))]
([X,Y|Dom][f:(E X)->E Y]Rlet ([x:E X]Rval (f x))) (pswap P|X1|X2)
(K (pswap P v)))
  (yrr ([h:(E (prod P (Res1 X1) (Res1 X2)))->E (Res1 (prod P X1 X2))]
        [v:E (prod P (Res1 X1) (Res1 X2))][v1=pout1 P v][v2=pout2 P v]
  orr (CH ([x:E X1]Rlet ([y:E X2]Rval (pin P x y)) v2)
([w'16:E (Res1 X1)]h (pin P w'16 v2)) v1)
      (CH ([y:E X2]Rlet ([x:E X1]Rval (pin P x y)) v1)
([z'16:E (Res1 X2)]h (pin P v1 z'16)) v2)))
  (pin P z w));

Claim STRICT
([K:(E (prod P (Res1 X1) (Res1 X2)))->E (Res1 (prod P X1 X2))]
 [v:E (prod P (Res1 X2) (Res1 X1))]
 ([X,Y|Dom][f:(E X)->E Y]Rlet ([x:E X]Rval (f x))) (pswap P|X1|X2)
 (K (pswap P v)));

Claim
  [h = ([K:(E (prod P (Res1 X1) (Res1 X2)))->E (Res1 (prod P X1 X2))]
[v:E (prod P (Res1 X2) (Res1 X1))]
([X,Y|Dom][f:(E X)->E Y]Rlet ([x:E X]Rval (f x))) (pswap P|X1|X2)
(K (pswap P v)))]
  [f = ([h:(E (prod P (Res1 X1) (Res1 X2)))->E (Res1 (prod P X1 X2))]
        [v:E (prod P (Res1 X1) (Res1 X2))][v1=pout1 P v][v2=pout2 P v]
  orr (CH ([x:E X1]Rlet ([y:E X2]Rval (pin P x y)) v2)
([w'16:E (Res1 X1)]h (pin P w'16 v2)) v1)
      (CH ([y:E X2]Rlet ([x:E X1]Rval (pin P x y)) v1)
```

```
([z'16:E (Res1 X2)]h (pin P v1 z'16)) v2))]
  [g = ([h:(E (prod P (Res1 X2) (Res1 X1)))->E (Res1 (prod P X2 X1))]
        [v:E (prod P (Res1 X2) (Res1 X1))][v1=pout1 P v][v2=pout2 P v]
  orr (CH ([x:E X2]Rlet ([y:E X1]Rval (pin P x y)) v2)
([w'16:E (Res1 X2)]h (pin P w'16 v2)) v1)
      (CH ([y:E X1]Rlet ([x:E X2]Rval (pin P x y)) v1)
([z'16:E (Res1 X1)]h (pin P v1 z'16)) v2))]
  {k:(E (prod P (Res1 X1) (Res1 X2)))->E (Res1 (prod P X1 X2))}
Eq (h (f k)) (g (h k));


Qrepl Eq_sym (lm_Res_uniformity S Z M F (ax_uniformity F)
  ([K:(E (prod P (Res1 X1) (Res1 X2)))->E (Res1 (prod P X1 X2))]
   [v:E (prod P (Res1 X2) (Res1 X1))]
([X,Y|Dom][f:(E X)->E Y]Rlet ([x:E X]Rval (f x))) (pswap P|X1|X2)
(K (pswap P v)))
  ?+1
  ([h:(E (prod P (Res1 X1) (Res1 X2)))->E (Res1 (prod P X1 X2))]
        [v:E (prod P (Res1 X1) (Res1 X2))][v1=pout1 P v][v2=pout2 P v]
  orr (CH ([x:E X1]Rlet ([y:E X2]Rval (pin P x y)) v2)
([w'16:E (Res1 X1)]h (pin P w'16 v2)) v1)
      (CH ([y:E X2]Rlet ([x:E X1]Rval (pin P x y)) v1)
([z'16:E (Res1 X2)]h (pin P v1 z'16)) v2))
  ([h:(E (prod P (Res1 X2) (Res1 X1)))->E (Res1 (prod P X2 X1))]
        [v:E (prod P (Res1 X2) (Res1 X1))][v1=pout1 P v][v2=pout2 P v]
  orr (CH ([x:E X2]Rlet ([y:E X1]Rval (pin P x y)) v2)
([w'16:E (Res1 X2)]h (pin P w'16 v2)) v1)
      (CH ([y:E X1]Rlet ([x:E X2]Rval (pin P x y)) v1)
([z'16:E (Res1 X1)]h (pin P v1 z'16)) v2))
  ?+2
```

```
  (pin P z w));

Immed;

Refine ax_strict
([resumptions=Resumptions S Z M][Res'3=resumptions.1]
[Rval=resumptions.2.1][Rlet=resumptions.2.2.1]
[K:(E (prod P (Res'3 X1) (Res'3 X2)))->E (Res'3 (prod P X1 X2))]
[v:E (prod P (Res'3 X2) (Res'3 X1))]Rlet ([x:E (prod P X1 X2)]Rval
(pswap P|X1|X2 x)) (K (pswap P v)));

intros;

Refine Eq_sym;

Expand f g h;

Refine (ax_extensionality
 ([v:E (prod P (Res1 X2) (Res1 X1))][v1=pout1 P v][v2=pout2 P v]
  orr (CH ([x:E X2]Rlet ([y:E X1]Rval (pin P x y)) v2)
  ([w'16:E (Res1 X2)]Rlet
([x:E (prod P X1 X2)]Rval (pswap P|X1|X2 x))
(k (pswap P (pin P w'16 v2)))) v1)
      (CH ([y:E X1]Rlet ([x:E X2]Rval (pin P x y)) v1)
  ([z'16:E (Res1 X1)]Rlet
([x:E (prod P X1 X2)]Rval (pswap P|X1|X2 x))
(k (pswap P (pin P v1 z'16)))) v2))
 ([v:E (prod P (Res1 X2) (Res1 X1))]
   Rlet ([x:E (prod P X1 X2)]Rval (pswap P|X1|X2 x))
```

```
([v1=pout1 P (pswap P v)][v2=pout2 P (pswap P v)]
 orr (CH ([x:E X1]Rlet ([y:E X2]Rval (pin P x y)) v2)
([w'16:E (Res1 X1)]k (pin P w'16 v2)) v1)
     (CH ([y:E X2]Rlet ([x:E X1]Rval (pin P x y)) v1)
([z'16:E (Res1 X2)]k (pin P v1 z'16)) v2))));

Intros v P2 H1;

Qrepl Eq_sym(lm_pswap2 P v);

Qrepl Eq_sym(lm_pswap3 P v);

Qrepl Eq_sym (ax_or_naturality
  (Rnd S Z M N)(pswap P|X1|X2)
  (CH ([x:E X1]Rlet ([y:E X2]Rval (pin P x y)) (pout1 P v))
      ([w'16:E (Res1 X1)]k (pin P w'16 (pout1 P v)))(pout2 P v))
  (CH ([y:E X2]Rlet ([x:E X1]Rval (pin P x y)) (pout2 P v))
      ([z'16:E (Res1 X2)]k (pin P (pout2 P v) z'16))(pout1 P v)));

Qrepl ax_ch_naturality S Z M (pswap P|X1|X2)
  ([x:E X1]Rlet ([y:E X2]Rval (pin P x y)) (pout1 P v))
  ([w'16:E (Res1 X1)]k (pin P w'16 (pout1 P v)))
  (pout2 P v);

Qrepl ax_ch_naturality S Z M (pswap P|X1|X2)
  ([y:E X2]Rlet ([x:E X1]Rval (pin P x y)) (pout2 P v))
  ([z'16:E (Res1 X2)]k (pin P (pout2 P v) z'16))
  (pout1 P v);
```

```
Qrepl (ax_extensionality
  ([y:E X2] Rlet ([x:E (prod P X1 X2)]Rval (pswap P|X1|X2 x))
(Rlet ([x:E X1]Rval (pin P x y)) (pout2 P v)))
  ([y:E X2] (Rlet ([x:E X1]Rlet ([u:E (prod P X1 X2)] Rval (pswap P|X1|X2 u))
(Rval (pin P x y)))
    (pout2 P v)))
  ([y:E X2](Eq_sym
(ax_assoc (Resumptions_Constructor S Z M)
(pout2 P v)
([x:E X1]Rval (pin P x y))
([x:E (prod P X1 X2)]Rval (pswap P|X1|X2 x))))));

Equiv P2
 (orr (CH ([x:E X1] Rlet ([x:E (prod P X1 X2)]Rval (pswap P|X1|X2 x))
(Rlet ([y:E X2]Rval (pin P x y)) (pout1 P v)))
 ([w'16:E (Res1 X1)]Rlet([x:E (prod P X1 X2)]Rval(pswap P|X1|X2 x))
(k (pin P w'16 (pout1 P v)))) (pout2 P v))
      (CH ([y:E X2]
   (Rlet ([x:E X1](([p: E (prod P X1 X2)]Rlet
    ([u:E (prod P X1 X2)] Rval (pswap P|X1|X2 u))
    (Rval p)) (pin P x y)))
(pout2 P v)))
  ([z'16:E (Res1 X2)]Rlet([x:E (prod P X1 X2)]Rval(pswap P|X1|X2 x))
(k (pin P (pout2 P v) z'16))) (pout1 P v)));

Qrepl Eq_sym (ax_runit (Resumptions_Constructor S Z M)
  ([u:E (prod P X1 X2)] Rval (pswap P|X1|X2 u)));

Claim {P:Prod}{X1,X2|Dom}
```

```
  Eq ([x:E X1][y:E X2]pswap P|X1|X2 (pin P x y))
     ([x:E X1][y:E X2] pin P y x);

Equiv P2
 (orr (CH ([x:E X1] Rlet ([x:E (prod P X1 X2)]Rval (pswap P|X1|X2 x))
(Rlet ([y:E X2]Rval (pin P x y)) (pout1 P v)))
 ([w'16:E (Res1 X1)]Rlet([x:E (prod P X1 X2)]Rval(pswap P|X1|X2 x))
(k (pin P w'16 (pout1 P v)))) (pout2 P v))
      (CH (([K:(E X1)->(E X2)->E (prod P X2 X1)]
([y:E X2]Rlet ([x:E X1]Rval (K x y))(pout2 P v)))
    ([a:E X1][b:E X2]pswap P|X1|X2 (pin P a b)))
  ([z'16:E (Res1 X2)]Rlet([x:E (prod P X1 X2)]Rval(pswap P|X1|X2 x))
(k (pin P (pout2 P v) z'16))) (pout1 P v)));

Qrepl ?+1 P|X1|X2;

Qrepl (ax_extensionality
  ([x:E X1] Rlet ([x:E (prod P X1 X2)]Rval (pswap P|X1|X2 x))
(Rlet ([y:E X2]Rval (pin P x y)) (pout1 P v)))
  ([x:E X1] (Rlet ([y:E X2]Rlet ([u:E (prod P X1 X2)] Rval (pswap P|X1|X2 u))
(Rval (pin P x y)))
    (pout1 P v)))
  ([x:E X1](Eq_sym
(ax_assoc (Resumptions_Constructor S Z M)
(pout1 P v)
([y:E X2]Rval (pin P x y))
([x:E (prod P X1 X2)]Rval (pswap P|X1|X2 x))))));

Equiv P2
```

```
  (orr (CH ([x:E X1]
    (Rlet ([y:E X2](([p: E (prod P X1 X2)]Rlet
     ([u:E (prod P X1 X2)]Rval (pswap P|X1|X2 u))
     (Rval p)) (pin P x y)))
 (pout1 P v)))
   ([w'16:E (Res1 X1)]Rlet
([x:E (prod P X1 X2)]Rval (pswap P|X1|X2 x))
(k (pin P w'16 (pout1 P v)))) (pout2 P v))
       (CH ([y:E X2]Rlet ([x:E X1]Rval (pin P y x))(pout2 P v))
   ([z'16:E (Res1 X2)]Rlet([x:E (prod P X1 X2)]Rval(pswap P|X1|X2 x))
(k (pin P (pout2 P v) z'16))) (pout1 P v)));

Qrepl Eq_sym (ax_runit (Resumptions_Constructor S Z M)
  ([u:E (prod P X1 X2)] Rval (pswap P|X1|X2 u)));

Equiv P2
  (orr (CH (([K:(E X1)->(E X2)->E (prod P X2 X1)]
([x:E X1]Rlet ([y:E X2]Rval (K x y))(pout1 P v)))
     ([a:E X1][b:E X2]pswap P|X1|X2 (pin P a b)))
   ([w'16:E (Res1 X1)]Rlet
([x:E (prod P X1 X2)]Rval (pswap P|X1|X2 x))
(k (pin P w'16 (pout1 P v)))) (pout2 P v))
       (CH ([y:E X2]Rlet ([x:E X1]Rval (pin P y x))(pout2 P v))
   ([z'16:E (Res1 X2)]Rlet([x:E (prod P X1 X2)]Rval(pswap P|X1|X2 x))
(k (pin P (pout2 P v) z'16))) (pout1 P v)));

Qrepl ?+1 P|X1|X2;

Qrepl (ax_or_commutativity (Rnd S Z M N)
```

```
(CH ([x:E X1]Rlet ([y:E X2]Rval (pin P y x)) (pout1 P v))
   ([w'16:E (Res1 X1)]Rlet
([x:E (prod P X1 X2)]Rval (pswap P|X1|X2 x))
(k (pin P w'16 (pout1 P v)))) (pout2 P v))
(CH ([y:E X2]Rlet ([x:E X1]Rval (pin P y x)) (pout2 P v))
    ([z'16:E (Res1 X2)]Rlet
([x:E (prod P X1 X2)]Rval (pswap P|X1|X2 x))
(k (pin P (pout2 P v) z'16))) (pout1 P v)));

Equiv P2
  (orr (CH ([y:E X2]Rlet ([x:E X1]Rval (pin P y x)) (pout2 P v))
    (([K:(E (Res1 X2))->E (prod P (Res1 X1) (Res1 X2))]
      [z'16:E (Res1 X2)]Rlet
([x:E (prod P X1 X2)]Rval (pswap P|X1|X2 x))
(k (K z'16)))([z:E (Res1 X2)]pin P (pout2 P v) z))
    (pout1 P v))
(CH ([x:E X1]Rlet ([y:E X2]Rval (pin P y x)) (pout1 P v))
   ([w'16:E (Res1 X1)]Rlet
([x:E (prod P X1 X2)]Rval (pswap P|X1|X2 x))
(k (pin P w'16 (pout1 P v)))) (pout2 P v)));

Qrepl ax_extensionality
([z:E (Res1 X2)]pin P (pout2 P v) z)
([z:E (Res1 X2)]pswap P (pin P z (pout2 P v)))
([z:E (Res1 X2)] Eq_sym(lm_pswap1 P z (pout2 P v)));

Equiv P2
  (orr (CH ([y:E X2]Rlet ([x:E X1]Rval (pin P y x)) (pout2 P v))
   ([z'16:E (Res1 X2)]Rlet
```

```
([x:E (prod P X1 X2)]Rval (pswap P|X1|X2 x))
(k (pswap P (pin P z'16 (pout2 P v))))) (pout1 P v))
(CH ([x:E X1]Rlet ([y:E X2]Rval (pin P y x)) (pout1 P v))
     (([K:(E (Res1 X1))->E (prod P (Res1 X1) (Res1 X2))]
[w'16:E (Res1 X1)]Rlet
([x:E (prod P X1 X2)]Rval (pswap P|X1|X2 x))
(k (K w'16)))([w:E (Res1 X1)]pin P w (pout1 P v)))
(pout2 P v)));

Qrepl ax_extensionality
([w:E (Res1 X1)]pin P w (pout1 P v))
([w:E (Res1 X1)]pswap P (pin P (pout1 P v) w))
([w:E (Res1 X1)] Eq_sym(lm_pswap1 P (pout1 P v) w));

Immed;

Intros PP A B;

Refine ax_extensionality
([x:E A][y:E B]pswap PP|A|B (pin PP x y))
([x:E A][y:E B]pin PP y x);

Intros x;
Refine ax_extensionality
([y:E B]pswap PP (pin PP x y))
([y:E B]pin PP y x);
Intros y;
Qrepl Eq_sym (lm_pswap1 PP x y);
Refine Eq_refl;
```

I Preservation of uniformity

```
Goal {S:Sum}{Z:Ind}{M:Monad}{F:Fix (T M)}
        (AX_UNIFORMITY F) -> AX_UNIFORMITY (Rfix S Z M F);

Expand AX_UNIFORMITY Rfix Yop RY Uniform_Redefinition2;

Intros;

Equiv P
 (h ([w:E X1]alg Z|(RF S (Monad_to_Functor M) X2)
     (Yop F|X1|(sum S X2 (Res S Z M X2))
([h:(E X1)->E (Unwind S Z M X2)][x1:E X1] coalg Z
   (f ([x:E X1]alg Z|(RF S (Monad_to_Functor M) X2) (h x)) x1))
w)) x);

Qrepl Eq_push (lm_Lambek2 Z|(RF S (Monad_to_Functor M) X4))
 (h ([w:E X1]alg Z|(RF S (Monad_to_Functor M) X2)
     (Yop F ([h:(E X1)->E (Unwind S Z M X2)][x1:E X1] coalg Z
(f ([x:E X1]alg Z|(RF S (Monad_to_Functor M) X2) (h x)) x1))
      w)) x);

Equiv P (alg Z
 (([K:(E X1)->E (Unwind S Z M X2)][x3: E X3]
   (coalg Z (h ([w:E X1]alg Z|(RF S (Monad_to_Functor M) X2) (K w)) x3)))
```

```
   (Yop F ([h:(E X1)->E (Unwind S Z M X2)][x1:E X1] coalg Z
(f ([x:E X1]alg Z|(RF S (Monad_to_Functor M) X2) (h x))
 x1))) x));

Claim STRICT ([K:(E X1)->E (Unwind S Z M X2)][x3: E X3]
  (coalg Z (h ([w:E X1]alg Z|(RF S (Monad_to_Functor M) X2) (K w)) x3)));

Claim {j:(E X1)->E (Unwind S Z M X2)}
  [h1 = [K:(E X1)->E (Unwind S Z M X2)][x3: E X3]
   (coalg Z (h ([w:E X1]alg Z|(RF S (Monad_to_Functor M) X2) (K w)) x3))]
  [f1 = [h:(E X1)->E (Unwind S Z M X2)][x1:E X1]
coalg Z (f ([x:E X1]alg Z|(RF S (Monad_to_Functor M) X2) (h x)) x1)]
  [g1 = [k:(E X3)->E (Unwind S Z M X4)][x3:E X3]
coalg Z (g ([x:E X3]alg Z|(RF S (Monad_to_Functor M) X4) (k x)) x3)]
  (Eq (h1 (f1 j)) (g1 (h1 j)));

Qrepl Eq_sym
 (H ([K:(E X1)->E (Unwind S Z M X2)][x3: E X3]
   (coalg Z (h ([w:E X1]alg Z|(RF S (Monad_to_Functor M) X2) (K w)) x3)))
  ?+1
  ([h:(E X1)->E (Unwind S Z M X2)][x1:E X1]
coalg Z (f ([x:E X1]alg Z|(RF S (Monad_to_Functor M) X2) (h x)) x1))
  ([k:(E X3)->E (Unwind S Z M X4)][x3:E X3]
coalg Z (g ([x:E X3]alg Z|(RF S (Monad_to_Functor M) X4) (k x)) x3))
  ?+2 x);

Immed;

Refine ax_strict
```

```
  ([K:(E X1)->E (Unwind S Z M X2)][x3:E X3]
    coalg Z (h ([w:E X1]alg Z|(RF S (Monad_to_Functor M) X2) (K w)) x3));

Intros;

Equiv P1 [x3:E X3]coalg Z (g ([x:E X3]
  ([z:E (Res S Z M X4)] alg Z|(RF S (Monad_to_Functor M) X4) (coalg Z z))
  (h ([w:E X1]alg Z|(RF S (Monad_to_Functor M) X2) (j w)) x)) x3);

Qrepl (Eq_sym (lm_Lambek2 Z|(RF S (Monad_to_Functor M) X4)));

Qrepl Eq_sym (ax_eta (h ([w:E X1]alg Z|(RF S (Monad_to_Functor M) X2) (j w))));

Equiv P1 (([K:(E X3)->E (Res S Z M X4)][x3:E X3]coalg Z (K x3))
(g (h ([w:E X1]alg Z|(RF S (Monad_to_Functor M) X2) (j w)))));

Qrepl Eq_sym (H2 ([w:E X1]alg Z|(RF S (Monad_to_Functor M) X2) (j w)));

Qrepl ax_eta (f ([w:E X1]alg Z|(RF S (Monad_to_Functor M) X2) (j w)));

Equiv P1 ([x3:E X3]coalg Z
  (h (([K:(E (Res S Z M X2))-> E (Res S Z M X2)]
       [x:E X1] K (f ([w:E X1]alg Z|(RF S (Monad_to_Functor M) X2) (j w)) x))
      ([y:E (Res S Z M X2)]y)) x3));

Qrepl (lm_Lambek2 Z|(RF S (Monad_to_Functor M) X2));

Immed;
```

Bibliography

[AA74] L. Aiello and M. Aiello. Proving Program Correctness in LCF. In *Programming symposium, Lecture Notes in Computer Science, vol. 19*, pages 59–71. Springer-Verlag, 1974.

[AAW77] L. Aiello, M. Aiello, and R. W. Weyrauch. Pascal in LCF: Semantics and Examples of Proof. *Theoretical Computer Science*, 5:135–177, 1977.

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

[Abr90a] S. Abramsky. Abstract interpretation, logical relations and Kan extensions. *Journal of Logic and Computation*, 1:5–40, 1990.

[Abr90b] S. Abramsky. The Lazy λ-calculus. In D. A. Turner, editor, *Logical Foundations of Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[Abr91] S. Abramsky. Domain Theory in Logical Form. *Annals of Pure and Applied Logic*, 51:1–77, 1991.

[Acz88] P. Aczel. Non-Well-Founded Sets, 1988. CSLI Lecture Notes, 14, Stanford University.

[AGM] S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors. *Semantic Structures*, volume 3. Oxford University Press, Oxford, England, . To appear.

[AJ91] S. Abramsky and T. P. Jensen. A relational approach to strictness analysis for higher order polymorphic functions. In *Proceedings 20th ACM Symp. on Principles of Programming Languages*, 1991.

[AJM94] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF (extended abstract). In *Proceedings TACS '94*, number 789 in LNCS. Springer-Verlag, 1994.

[Alt92] T. Altenkirch. Brewing strong normalization proofs with lego. LFCS Report Series ECS-LFCS-92-230, Laboratory for the Foundations of Computer Science, Computer Science Department, University of Edinburgh, 1992.

[Apt81] K.R. Apt. Ten years of Hoare's Logic: a survey - part 1. *ACM TOPLS*, 3, 4, 1981.

[AR94] J. Adámek and J. Rosický. *Locally Presentable and Accessible Categories.* LMS Lecture Notes Series 189. Cambridge University Press, 1994.

[BA91] B. Bloom and L. Aceto. Turning sos rules into equations. Technical report, Stiching Mathematisch Centrum, CWI, 1991. CS-R9218.

[Bar92] Henk Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*. Oxford University Press, 1992.

[Ben86] Ermanno Bencivenga. Free Logics. In Gabbay and Guenthner [GG86], pages 373–426. Volume 3.

[Ber91] Stefano Berardi. Girard's normalization proof in lego. Unpublished draft, 1991.

[BG85] R. Burstall and J. Goguen. A Study in the Foundations of Program Methodology: Specifications, Institutions, Charters and Parchments. In *Proc. of Summer Workshop on Category Theory and Computer Programming.* University of Surrey, 1985.

[BG87] A. Blass and Y. Gurevich. Existential fixed-point logic. In E. Börger, editor, *Computation Theory and Logic*, pages 20–36. LNCS 270, Springer-Verlag, 1987.

[Blo88] B. Bloom. Can lcf be topped? Flat lattice models of typed lambda calculus. In *IEEE Symposium on Logic in Computer Science*, pages 282–290. Computer Society Press, 1988.

[BS92] J. Bradfield and C. Stirling. Local model checking for infinite state spaces. *Theoretical Computer Science*, 96:157–174, 1992.

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

[BW90] M. Barr and C. Wells. *Category theory for computing science.* Prentice-Hall, London, 1990.

[CAB+86] R.L. Constable, S.F. Allen, H.M. Bromley, et al. *Implementing Mathematics with the NuPRL Proof Development System.* Prentice Hall, 1986.

[Cen94] P. Cenciarelli. A modular development of denotational semantics in LEGO. Presented at the Types for Proofs and Programs Workshop, Båstad, June 1994.

[Cen95] P. Cenciarelli. Partial correctness in Evaluation Logic. In P. Dybier and R. Pollack, editors, *Proceedings of the Joint CLiCS-Types Workshop on Categories and Type Theory, Göteborg, January 95.* Programming Methodology Group, Göteborg University and Chalmers University of Technology, May 1995. Report 85.

[CF94] R. Cartwright and M. Felleisen. Extensible denotational languge specifications. In *Proceedings of TACS 94*. LNCS 789, 1994.

[CH85] T. Coquand and G. Huet. Constructions: a higher order proof system for mechanizing mathematics. In *Proc. EUROCAL 85*. Springer LNCS 203, 1985.

[CJKP94] A. Carboni, G. Janelidze, G. M. Kelly, and R Paré. On localization and stabilization for factorization systems. *??*, 1994.

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

[Coo78] S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Computing*, 7:70–90, 1978.

[CP92] R.L. Crole and A.M. Pitts. New Foundations for Fixpoint Computations: FIX-hyperdoctrines and the FIX-logic. *Information and Computation*, 98:171–210, 1992.

[deB70] N.G. deBruijn. The mathematical language automath. In *Symposium in Automatic Demonstration, Lecture Notes in Mathematics*, volume Vol. 125, pages 29–61. Springer-Verlag, 1970.

[DFH$^+$93] Dowek, Felty, Herbelin, Huet, Murthy, Parent, Paulin-Mohring, and Werner. The Coq proof assistant user's guide. Technical report, INRIA-Roquencourt, February 1993.

[dG95] Philippe de Groote. A simple calculus of exception handling. In *Proc. TLCA-95, Edinburgh*. Springer-Verlag, 1995.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[EK66] S. Eilenberg and G. M. Kelly. Closed categories. In S. Eilenberg, D.K. Harrison, S. MacLane, and H. Rohrl, editors, *Proceedings of the Conference on Categorical Algebra*. Springer-Verlag, 1966. La Jolla, 1965.

[End77] H.B. Enderton. *Elements of set theory.* Academic Press, 1977.

[Fef95] S. Feferman. Definedness. In *Proceedings of the mini-conference on Partial Functions and Programming: Foundational Questions*, May 1995. U. C. Irvine.

[Fio94a] M. P. Fiore. First steps on the representation of domains. Manuscript (available from http://www.dcs.ed.ac.uk/home/mf/path.dvi), December 1994.

[Fio94b] M.P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps.* PhD thesis, Department of Computer Science, Edinburgh University, 1994. Also published as CST-113-94.

[FK72] P.J. Freyd and G. M. Kelly. Categories of continuous functors. *J. of Pure and Applied Algebra*, 2:169–191, 1972.

[Flo67] R. W. Floyd. Assigning Meanings to Programs. *Proc. Symp. App. Math.*, 19, 1967.

[FMRS90] P.J. Freyd, P. Mulry, G. Rosolini, and D. Scott. Extensional pers. In *Proc. of LICS '90.* IEEE, 1990.

[FP94] M.P. Fiore and G.D. Plotkin. An axiomatisation of computationally adequate domain theoretic models of FPC. In *9^{th} LICS Conf.* IEEE, Computer Society Press, 1994.

[Fre72] P. Freyd. Aspects of topoi. *Bull. Austral. Math. Soc.*, 7, 1972.

[Fre90] P.J. Freyd. Recursive types reduced to inductive types. In *Proc. of LICS '90*, pages 498–507. IEEE Computer Society Press, 1990.

[Fre91] P. Freyd. Algebraically complete categories. In A. Carboni, M.C. Pedicchio, and G. Rosolini, editors, *Category Theory - Proc. of the Int'l Conf. held in*

Como, Italy, July 1990, volume 1488 of *Lecture Notes in Mathematics*, pages 95–104. Springer-Verlag, Berlin, 1991.

[FS90] P.J. Freyd and A. Scedrov. *Categories, Allegories.* North-Holland, Amsterdam, 1990.

[GG86] D. Gabbay and F. Guenthner, editors. *Handbook of Philisophical Logic.* Kluwer Academic Publishers, 1986.

[Gir72] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre superieur.* PhD thesis, Université de Paris VII, 1972.

[GMW79] M.J. Gordon, A.J. Milner, and C.P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science.* Springer Verlag, 1979.

[Gor79] M.J. Gordon. *The denotational description of programming languages.* Springer Verlag, New York, 1979.

[Gri81] D. Gries. *The Science of Programing.* Springer-Verlag, 1981.

[GU75] P. Gabriel and F. Ulmer. Lokal Präsentierbare kategorien. In *Lecture notes in mathematics, vol.445*, Berlin, 1975. Springer.

[Gun92] C.A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* The MIT Press, 1992.

[Gur91] D.J. Gurr. *Semantic Frameworks for Complexity.* PhD thesis, Department of Computer Science, Edinburgh University, 1991. Also published as CST-72-91.

[HA80] M. C. B. Hennessy and E. A. Ashcroft. A mathematical semantics for a nondeterministic typed λ-calculus. *Theoretical Computer Science*, 11:227–245, 1980.

[Her93] C. A. Hermida. *Fibrations, Logical Predicates and Indeterminates.* PhD thesis, Department of Computer Science, Edinburgh University, 1993. Also published as CST-103-93.

[Hes92] W. H. Hesselink. *Programs, Recursion and Unbounded Choice.* Cambridge University Press, 1992.

[HHP87] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204, 1987.

[HJP80] J.M.E. Hyland, P.T. Johnstone, and A.M. Pitts. Tripos theory. *Math. Proc. Camb. Phil. Soc.*, 88:205–232, 1980.

[HL93] P. Harper and M. Lillibridge. Explicit Polymorphism and CPS Conversion. In *Proc. 20th PoPL conference.* Association for Computing Machinery, 1993.

[HO94] J.M.E Hyland and C.-H.L. Ong. Full abstraction for PCF: dialogue games and innocent strategies. Preprint, 1994.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12:567–580, 1969.

[HP89] J.M.E. Hyland and A.M. Pitts. The theory of constructions: categorical semantics and topos-theoretic models. *Contemporary Mathematics*, 92:137–199, 1989.

[HP90] H. Huwig and A. Poigné. A note on inconsistency caused by fixed points in a cartesian closed category. *Theoretical Computer Science*, 73:101–112, 1990.

[Hyl82] J.M.E. Hyland. The effective topos. In A.S. Troelstra and D. van Dalen, editors, *The L.E.J. Brouwer Centenary Symposium.* North-Holland, 1982.

[Hyl90] J.M.E. Hyland. First steps in synthetic domain theory. In A. Carboni, M.C. Pedicchio, and G. Rosolini, editors, *Category Theory*, volume 1488 of *Lecture Notes in Mathematics*, pages 95–104. Springer-Verlag, 1990.

[Jac91] B.P.F. Jacobs. *Categorical type theory.* PhD thesis, Katholieke Universiteit Nijmegen, 1991.

[Jut77] L.S. Jutting. *Checking Landau's Grundlagen in the AUTOMATH System.* PhD thesis, Eindhoven University, Netherlands, 1977.

[Kah88] G. Kahn. Natural Semantics. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 237–258, Amsterdam, 1988. Elsevier Science Publishers B. V. (North-Holland).

[Kea75] O. Keane. Abstract horn theories. In F.W. Lawvere, C. Maurer, and G.C. Wraith, editors, *Model theory and topoi*, pages 15–50, Berlin, 1975. Springer. Lecture notes in mathematics, vol.445.

[Kel82] G. M. Kelly. *Basic concepts of enriched category theory.* London Mathematical Society Lecture Note 64. Cambridge University Press, 1982.

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

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

[KR77] A. Kock and G.E. Reyes. Doctrines in categorical logic. In J. Barwise, editor, *Handbook of mathematical logic*, pages 283–313. North-Holland, 1977.

[Laf88] Y. Lafont. The linear abstract machine. *Theoretical Computer Science*, 59, 1988.

[Lam68] J. Lambek. A fixed point theorem for complete categories. *Math. Zeitschr.*, 103:151–161, 1968.

[Lam93] J. Lambek. Least fixpoints of endofunctors of cartesian closed categories. *Math. Struct. in Comp. Science*, 3:229–257, 1993.

[Law69] F. W. Lawvere. Adjointness in foundations. *Dialectica*, 23, No.3/4:281–296, 1969.

[Law70] F. W. Lawvere. Equality in hyperdoctrines and comprehension schema as an adjoint. In *Applications of Category Theory*, 1970. Proceedings of the American Mathematical Society Symposia on Pure Mathematics XVII.

[Law75] F. W. Lawvere. Introduction. In F.W. Lawvere, C. Maurer, and G.C. Wraith, editors, *Model theory and topoi*, pages 3–14, Berlin, 1975. Springer. Lecture notes in mathematics, vol.445.

[LHJ95] S. Liang, P. Hudak, and M. Jones. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd Symposium on Principles of Programming Languages*, pages 333–343. ACM Press, January 1995. San Francisco.

[Lil95] Mark Lillibridge. Exceptions are Strictly More Powerful than Call/CC. Technical Report CMU-CS-95-178, Carnegie Mellon University, 1995.

[Lin66] F.E.J. Linton. Some Aspects of Equational Categories. In S. Eilenberg, D.K. Harrison, S. MacLane, and H. Rohrl, editors, *Proceedings of the Conference on Categorical Algebra*, pages 84–94. Springer-Verlag, 1966. La Jolla, 1965.

[LP92] Z. Luo and R. Pollack. Lego Proof Development System: User's Manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, Dpt. of Computer Science, 1992.

[LS86] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic.* Number 7 in Cambridge studies in advanced mathematics. Cambridge University Press, 1986.

[Luo82] Z. Luo. *An extended calculus of constructions.* PhD thesis, Department of Computer Science, Edinburgh University, 1982. Also published as CST-65-90.

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

[Mag92] Lena Magnusson. The new implementation of alf. In *Workshop on Logical Frameworks*, 1992.

[Mey88] A.R. Meyer. Semantical paradigms: Notes for an invited lecture. In *IEEE Symposium on Logic in Computer Science*, pages 236–253. Computer Society Press, 1988.

[Mil72] R. Milner. Implementation and application of scott's logic for computable functions. In *ACM conference on Proving Assertions about Programs*, pages 1–6. Ellis Horwood Limited, 1972.

[Mil77] R. Milner. Fully abstract models of typed lambda λ-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[Mil83] R. Milner. Calculi for Syncrony and Asyncrony. *Theoretical Computer Science*, 25:267–310, 1983.

[Mit90] J. C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume II, pages 365 – 458. Elsevier Science Publishers, 1990.

[MM85] J. C. Mitchell and A. R. Meyer. Second order logical relations. In *Logics of Programs*, number 193 in LNCS, pages 225–236. Springer-Verlag, 1985.

[Moga] E. Moggi. A general semantics for evaluation logic. Unpublished manuscript.

[Mogb] E. Moggi. A semantics for Evaluation Logic. To appear in *Fundamenta Informaticae*.

[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.

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

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

[Mog91a] E. Moggi. A category-theoretic account of program modules. *Math. Struct. in Comp. Science*, 1:103–139, 1991.

[Mog91b] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[Mog91c] E. Moggi. A modular approach to denotational semantics. Invited talk, Conference on Category Theory and Computer Science, Paris 1991.

[Mog95a] E. Moggi. Metalanguages and applications. To appear in: *Semantics and Logics of Computation*, CISM Lecture Notes, Springer-Verlag. Presented at the Summer School on Semantics and Logics of Computation, Newton Institute, Cambridge, 25-29 September 1995.

[Mog95b] E. Moggi. Towards a framework for programming languages (and program logics). In P. Dybier and R. Pollack, editors, *Proceedings of the Joint CLiCS-Types Workshop on Categories and Type Theory, Göteborg, January 95*. Programming Methodology Group, Göteborg University and Chalmers University of Technology, May 1995. Report 85.

[Mos88] Peter D. Mosses. The modularity of action semantics. Technical Report DAIMI IR-75, Computer Science Department, Aarhus University, 1988.

[Mos90] Peter D. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier Science Publishers, 1990.

[Mos92] Peter D. Mosses. *Action Semantics*. Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[MR76] M. Makkai and G.E. Reyes. Model theoretic methods in the theory of topoi and related categories. *Acad. Polon. Sci.*, 24:379–392, 1976.

[MR91] Q. Ma and J. C. Reynolds. Types, abstraction and parametric polymorphism 2. In *Mathematical Foundations of Programming Languages*, LNCS. Springer-Verlag, 1991.

[MS92] J. Mitchell and A. Shedrov. Notes on sconing and relators. In *Computer Science Logic '92*, volume 702 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[MW72] R. Milner and R. Weyhrauch. Proving compiler correctness in a mechanized logic. *Machine Intelligence*, 7:51–70, 1972.

[Neu44] J. Von Neumann. The EDAVAC report. In *Computer from PASCAL to Von Neumann*, ed. H. Goldstain, Princeton University Press, Princeton, NJ, 1972, 1944.

[New75] M. Newey. Formal semantics of LISP with applications to program correctness, 1975. Artificial Intelligence Memo 257, Stanford.

[OT92] P.W. O'Hearn and R.D. Tennent. Semantics of local variables. In *Applications of Categories in Computer Science*, volume 177 of *Lecture Notes Series*. Cambridge University Press, 1992.

[OT93] P.W. O'Hearn and R.D. Tennent. Relational parametricity and loval variables. In *Proc. 20th PoPL conference*. Association for Computing Machinery, 1993.

[PA93] G. Plotkin and M. Abadi. A Logic for Parametric Polymorphism. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications, Proceedings of TLCA '93*. Springer LNCS 664, 1993.

[Pau87] L. Paulson. *Logic and Computation*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1987.

[Pav90] Duško Pavlović. *Predicates and fibrations*. PhD thesis, Rijksuniversiteit Utrecht, 1990.

[Pho90] W.K.-S. Phoa. Effective domains and intrinsic structure. In *Proc. of 5th Annual Symposium on Logic in Computer Science*, 1990.

[Pho92] Wesley Phoa. An introduction to fibrations, topos theory, the effective topos and modest sets. Technical Report ECS-LFCS-92-208, University of Edinburgh, LFCS, 1992.

[Pit] A. M. Pitts. Relational properties of domains. A revised version of Cambridge Computer Laboratory Technical Report Number 321. To appear in Information and Computation.

[Pit82] A.M. Pitts. The secret category theory lecture notes. Unpublished manuscript, 1982.

[Pit91] A.M. Pitts. Evaluation logic. In G. Birtwistle, editor, *Workshops in Computing 283*. 4th Higher Order Workshop, Banf. 1990, Springer Verlag, 1991.

[Pit93] A.M. Pitts. *Computational adequacy via 'mixed' inductive definitions*, volume 802 of *Lecture Notes in Computer Science*. Springer Verlag, 1993. Proc. 9th Int. Conf. on Mathematical Foundations of Programming Language Semantics, New Orleans.

[Plo73] Gordon Plotkin. Lambda-definability and logical relations. Technical report, School of Artificial Intelligence, University of Edinburgh, 1973.

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

[Plo77] Gordon Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5, 1977.

[Plo81] Gordon Plotkin. A Structural Approach to Operational Semantics, 1981. Aarhus University, Department of Computer Science, DAIMI FN-19.

[Plo85a] G.D. Plotkin. Denotational semantics with partial functions. Lecture at C.S.L.I. Summer School, 1985.

[Plo85b] Gordon Plotkin. Types and Partial Functions, 1985. Lecture notes, Department of Computer Science, University of Edinburgh.

[Plo91a] Gordon Plotkin. Notes on a metalanguage, 1991. Domain Theory Lecture Notes (extending "The Stanford Manuscript"), University of Edinburgh.

[Plo91b] Gordon Plotkin. "The Stanford Manuscript", 1991. Domain Theory Lecture Notes, University of Edinburgh (LFCS).

[Plo93] G.D. Plotkin. Second order type theory and recursion. Slides of a talk given at the Scott Fest, February 1993.

[Pow94] A. J. Power. Why tricategories? Technical Report ECS-LFCS-94-289, Department of Computer Science, Edinburgh University, 1994.

[Rey74] J.C. Reynolds. *Towards a Theory of Type Structure*, volume 19 of *Lecture Notes in Computer Science*. Springer-Verlag, 1974. Proc. Colloque sur la Programmation.

[Rey81] J.C. Reynolds. *The craft of programming*. Prentice Hall, 1981.

[Rey83] J.C. Reynolds. Types, abstraction and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing 83*, pages 513–523. North-Holland, Amsterdam, 1983.

[Rob95] E.P. Robinson. Note on the presentation of enriched monads. Unpublished manuscript, 1995.

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

[RS] B. Reus and T. Streicher. Naïve synthetic domain theory - a logical approach. unpublished manuscript.

[SB83] D. Sannella and R. Burstall. Structured theories in LCF. Technical Report CSR-129-83, Department of Computer Science, Edinburgh University, 1983. Also appeared in Proceedings of the 8th Colloquium on Trees in Algebra and Programming, L'Aquila, Italy, 1983.

[Sco69] D. Scott. A Type-theoretical Alternative to CHUCH, ISWIM, OWHY. (Also in *Theoretical Computer Science*, 121, 411-440, 1993), 1969.

[See82] R.A.G. Seely. Review of topos theory. *Journal of Symbolic Logic*, 47, 1982.

[See87] R.A.G. Seely. Categorical semantics for higher order polymorphic lambda calculus. *Journal of Symbolic Logic*, 52(2), 1987.

[SS71] D. Scott and S. Strachey. *Towards a Mathematical Semantics for Computer Languages.* Oxford Univ. Comp. Lab, PRG 6, 1971.

[ST87] D. Sannella and A. Tarlecki. Toward a formal development of programs from algebraic specifications: implementations revisited. In H. Ehrig and al., editors, *TAPSOFT 87*, volume 250 of *London Mathematical Society Lecture Note Series.* Springer Verlag, 1987.

[ST89] D. Sannella and A. Tarlecki. Toward formal developmento of ml programs: foundations and methodology. Technical Report ECS-LFCS-89-71, Department of Computer Science, Edinburgh University, 1989. Extended abstract in *Proc. Colloq. on Concurrent Issues in Programming Languages*, Joint Conf. in Theory and Practice of Software Development (TAPSOFT), Barcelona, 1989. LNCS 352, pp. 357.

[Sta85] R. Statman. Logical relations and the typed lambda calculus. *Inform. and Control*, 65:85–97, 1985.

[Sti92] Colin Stirling. Modal and Temporal Logics. In D. Gabbay S. Abramsky and T. Maibaum, editors, *Handbook of Logic in Computer Science.* Oxford University Press, 1992.

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

[Tai67] W. W. Tait. Intentional interpretation of functionals of finite type. *J. Symbolic Logic*, 32:198–212, 1967.

[Tay91] P. Taylor. The fixed point property in synthetic domain theory. In *Proc. of 6th Annual Symposium on Logic in Computer Science*, 1991.

[Tay92] Paul Taylor. Domains and duality. unpublished manuscript, 1992.

[Ten] R. D. Tennent. Denotational semantics. In Abramsky et al. [AGM]. To appear.

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

[Vic89] S. Vickers. *Topology via logic.* Cambridge University Press, Cambridge, 1989.

[Wad76] C. P. Wadsworth. The relationship between λ-expressions and their denotations in Scott's model of the λ-calculus. *SIAM J. Comput.*, September 1976.

[Wad91] Philip Wadler. Is there a use for linear logic? In *Symposium on Parallel Evaluation and Semantics Based Program Manipulation (PEPM), Yale University.* ACM/IFIP, June 1991.

[Wad92] Philip Wadler. The essence of functional programming. In *Proc. 19th Annual Symposium on Principles of Programming Languages.* ACM SIGPLAN-SIGACT, January 1992.

[Wec92] W. Wechler. Universal Algebra for Computer Scientists. In W. Brawer, G. Rozenberg, and A. Salomaa, editors, *EATC Monographs on Theoretical Computer Science.* Springer-Verlag, 1992.

[Wey72] R. W. Weyhrauch. Program Semantics and Correctness in a Mechanized Logic. In *Proc. 1st USA-Japan Computer Conf., Tokyo*, 1972.

[Win85] G. Winskel. A complete proof system for sccs with modal assertions. In S. N. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science*, pages 392–410. Springer-Verlag, 1985. Lecture Notes in Computer Science No. 206.

Index