A category-theoretic account of program modules

Eugenio Moggi*
em@lfcs.edinburgh.ac.uk
LFCS, University of Edinburgh, EH9 3JZ Edinburgh, UK

May 31, 1994

Abstract

The type-theoretic explanation of modules proposed to date (for programming languages like ML) is unsatisfactory, because it does not capture that evaluation of type-expressions is *independent* from evaluation of program-expressions. We propose a new explanation based on "programming languages as indexed categories" and illustrates how ML can be extended to support higher order modules, by developing a category-theoretic semantics for a calculus of modules with dependent types. The paper outlines also a methodology, which may lead to a modular approach in the study of programming languages.

Introduction

The addition of module facilities to programming languages is motivated by the need to provide a better environment for the development and maintenance of large programs. Nowadays many programming languages include such facilities. Throughout the paper Standard ML (see [Mac85, HMM86, MTH90]) is taken as representative for these languages. The implementation of module facilities has been based mainly on an operational understanding. More recently, a type-theoretic understanding of ML-modules has been proposed, which is based on a type theory with dependent types and a cumulative hierarchy of two universes U_1 and U_2 s.t. $U_1{:}\,U_2$ and $U_1 \subset U_2$ (see [Mac86, HM88]). The explanation of ML-modules according to this understanding goes as follows:

- *ML-signatures* are elements of U_2 built from U_1 and types (i.e. elements of U_1) by the Σ-type constructor, while *ML-structures* are elements of ML-signature, namely tuples made of values (i.e. elements of types) and types

*This paper has been written in Cambridge and Paris thanks to the financial support given by ESPRIT Basic Research Action No. 3003 (Categorical Logic In Computer Science) and the Ecole Normale Supérieure in Paris.

- *ML-functor signatures* are elements of U_2 of the form $\Pi s\colon sig_1.sig_2(s)$, where sig_1 and sig_2 are ML-signatures, while *ML-functors* are elements of ML-functor signatures, namely functions from ML-structures to ML-structures.

However, the explanation of ML-functors as functions is problematic in relation to type-checking at compile-time. This becomes apparent when trying to define *higher order modules* as done in XML (see [HM88]). Let τ be a type and $f\colon(\Pi x\colon\tau.U_1)$ be an ML-functor variable, which may occur in the body of an higher order module.

- It is not clear what the meaning of the type-expression $fM\colon U_1$ is, when the program-expression $M\colon\tau$ diverges. This may happen in ML, because functions can be defined by recursion.

- Type equality becomes undecidable as soon as recursive types are allowed (as in ML). For instance, if $\tau = \tau \to \tau$, then $fM_1 = fM_2\colon U_1 \iff M_1 = M_2\colon\tau$ and the second equality is undecidable, since it amounts to $\beta\eta$-conversion between untyped λ-terms.

These two problems seem to jeopardise decidability of type-checking. XML avoids these problems by requiring expressions to be *total* and equality of expression to be decidable, but then the theory hardly captures the *essence* of ML, while ML avoids them by banning higher order modules, so that the only ML-functors $\lambda x\colon\tau.M$ of ML-signature $\Pi x\colon\tau.U_1$ are those where x is not free in M.

> We take **independence** of type-expressions from program-expressions as essential feature of programming languages (such as PASCAL, core ML, ADA), which should be *respected* by module facilities. Remark 5.1 clarifies how independence relates to *phase distinction* and *type-checking*.

We capture independence of types from values categorically, by viewing a programming language as an indexed category, which suggests an obvious definition of category of modules (see Section 4). A concrete outcome of our analysis is that ML-functors should be viewed as pairs of functions and not as functions (from pairs to pairs), e.g. ML-functors from $\Sigma t_1\colon U_1.\tau_1(t_1)$ to $\Sigma t_2\colon U_1.\tau_2(t_2)$ *are* elements of

$$\Sigma f\colon U_1 \to U_2.(\Pi t_1\colon U_1.\tau_1(t_1) \to \tau_2(ft_1))$$

and not elements of $(\Sigma t_1\colon U_1.\tau_1(t_1)) \to (\Sigma t_2\colon U_1.\tau_2(t_2))$, as in [Mac86, HM88].

> We give also a more **abstract characterisation** of independence (see Definition 7.3 and Theorem 7.5) and **explain** dependent sums and products at the level of ML-signatures in terms of dependent sums and products at the level of kinds and type schemas (see Theorem 7.9).

Pragmatic issues and general methodology

The objective of this paper is not only to describe a language supporting higher order modules, but also to propose a general methodology for studying programming languages and to suggest how the independence of type-expressions from program-expressions and program modules may fit into it. From this perspective Category Theory is particularly appropriate. This methodology tries to address a deep-rooted problem in the study of programming languages (and other areas): the lack of *modularity*. In a modular approach the key concept to investigate should be the "addition of features to a language" rather than specific "toy languages" (i.e. languages with only a few features).

Syntax-independent view. The first ingredient of such a methodology should be to abstract as much as possible from the concrete presentation of a language, so that one can focus only on the underlying "mathematical structures". This is standard practice in Categorical Logic (see [KR77, Pitar]), where theories are identified with categories having certain additional structure. We follow a similar paradigm for programming languages. In particular, we propose to identify a programming language where type-expressions are evaluated independently from program-expressions with an indexed category $\mathcal{C}: \mathcal{B}^{op} \to \mathbf{Cat}$ s.t.

- type-expressions are morphisms in $\mathcal{B}$, while
- program-expressions are morphisms in the fibers $\mathcal{C}[X]$.

We introduce a language HML, where type-expressions are independent from program-expressions, and show how it can be viewed as an indexed category. In [HMM90] this and related languages (in particular a calculus for higher order modules with dependent signatures) are investigated in greater depth and with more emphasis on pragmatic issues, such as decidability of type-checking. While here we investigate only the category-theoretic foundations for these languages.

Program modules. If programming languages are indexed categories (as outlined above), then it is natural to expect that program modules should live in a category where the type-expressions and program-expressions coexist. There is a standard construction, due to Grothendieck, which transforms an indexed category $\mathcal{C}: \mathcal{B}^{op} \to \mathbf{Cat}$ into a *fibration* $\pi_{\mathcal{C}}: \mathcal{GC} \to \mathcal{B}$. We take $\mathcal{GC}$ as the *category of modules* for the programming language $\mathcal{C}$. Once we have established the correct link between programming language concepts and category-theoretic notions, it is mainly a matter of letting standard category-theoretic machinery do the rest, e.g. tell us what it means to have higher order modules. More specifically, we will show that the category of HML-modules has the structure needed to interpret a calculus with dependent types, like the calculus λ_{mod}^{ML} studied in [HMM90].

2-categorical approach. The second ingredient of a general methodology should be ways to combine features. This is a very difficult problem to solve in generality, because it depends on the way features *interact*. For instance:

- they could be *unrelated*, like products and coproducts, or

- one feature could be defined in terms of the other, e.g. the definition of function space relies on having products.

Here we focus on "adding one feature *on top* of another". The motivating example is to make precise the idea of a notion of computation (see [Mog89b]) which respects the independence of type-expressions from program-expressions, or more formally the idea of a monad over an indexed category (which captures the independence of type-expressions from program-expressions). A simpler example of such a combination is the notion of topological group, which amounts to a group in the category of topological spaces. The strategy suggested by the second example is to look for a category of topological spaces (the first feature) and generalise the definition of group (the second feature) over a set to that of group over an object in a category (with finite products). By applying (mutatis mutandis) the same strategy to the first example, we have to generalise the definition of monad over a category to that of monad over an object in a 2-category and show that indexed categories form a 2-category. Summarising, the key ideas of the 2-categorical approach to programming languages are:

- programming languages (with certain features) are objects of a 2-category $\mathcal{C}$ (which will depend on the features one is interested in)

- an additional feature is an instance in $\mathcal{C}$ of a 2-categorical concept.

Summary

Section 1 reviews the basics of 2-category theory (see also [KS74]).
Section 2 reviews the basics of indexed-category theory, and defines the 2-category of indexed categories. We use indexed categories to capture the independence of type-expressions from program-expressions.
Section 3 reviews briefly the category-theoretic semantics of several typed lambda-calculi and discusses how various features of programming languages can be described categorically.
Section 4 describes the Grothendieck construction, i.e. how to go from a programming language, viewed as an indexed category $\mathcal{C}$, to its category $\mathcal{GC}$ of modules.
Section 5 introduces HML, a language similar to ML with type-expressions independent from program-expressions, and describes the category $\mathcal{GC}$ of HML-modules. The Appendix gives a complete description of HML, while Section 7 investigates additional structure on the category $\mathcal{GC}$.
Section 6 reviews the category-theoretic semantics of dependent types and the Calculus of Constructions (see [CH88]) following quite closely [HP89], but using *categories with attributes* instead of classes of display maps.
Section 7 investigates dependent kinds and dependent type schemas in HML, using the machinery set up in Section 6. More specifically, it describes the effect of *independence* of constructor-expressions from program-expressions at the level of categories with attributes, and relates dependency at the level of modules to dependency at the level of constructors and programs.

1 Preliminaries on 2-categories

Both categories and $\mathcal{B}$-indexed categories can be viewed as objects of suitable 2-categories, **Cat** and $\mathbf{ICat}(\mathcal{B})$ respectively. This view is particularly useful for giving definitions involving $\mathcal{B}$-indexed categories (and proving their properties) by analogy with categories. In fact, it is just a matter of rephrasing familiar concepts, like monad or adjunction, in the formal language of 2-categories. We recall the definitions of 2-category, 2-functor and 2-natural transformation (see also [KS74]).

Definition 1.1 *A* **2-category** $\mathcal{C}$ *is a* **Cat***-enriched category, i.e.*

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

Notation 1.2 An object f of $\mathcal{C}(c_1, c_2)$ is called a **1-morphism**, while an arrow σ is called a **2-morphism**. We write $\_;\_$ for $comp(\_, \_)$

$$\begin{tikzcd} c_1 \arrow[r, shift left=3, "f_1"] \arrow[r, shift right=3, "f_1'"'] \arrow[r, phantom, "\Downarrow \sigma_1"] & c_2 \arrow[r, shift left=3, "f_2"] \arrow[r, shift right=3, "f_2'"'] \arrow[r, phantom, "\Downarrow \sigma_2"] & c_3 \end{tikzcd} \quad \overset{\_;\_}{\longmapsto} \quad \begin{tikzcd} c_1 \arrow[r, shift left=3, "f_1;f_2"] \arrow[r, shift right=3, "f_1';f_2'"'] \arrow[r, phantom, "\Downarrow \sigma_1;\sigma_2"] & c_3 \end{tikzcd}$$

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

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

Example 1.3 The canonical example of a 2-category is **Cat** itself (see [Mac71]):

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

Definition 1.4 *A* **2-functor** *F from $\mathcal{C}_1$ to $\mathcal{C}_2$ is a mapping*

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

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

Definition 1.5 *If F_1 and F_2 are 2-functors from $\mathcal{C}_1$ to $\mathcal{C}_2$, then a* **2-natural transformation** *τ from F_1 to F_2 is a family $\langle F_1c \overset{\tau_c}{\to} F_2c | c \in \mathrm{Obj}(\mathcal{C}_1)\rangle$ of 1-morphisms s.t.*

$$
\begin{tikzcd}
c \arrow[d, bend right] \arrow[d, bend left] \arrow[d, phantom, "\overset{\sigma}{\Rightarrow}"] \\
c'
\end{tikzcd}
\quad \Longrightarrow \quad
\begin{tikzcd}
F_1c \arrow[r, "\tau_c"] \arrow[d, bend right] \arrow[d, bend left] \arrow[d, phantom, "\overset{F_1\sigma}{\Rightarrow}"] & F_2c \arrow[d, bend right] \arrow[d, bend left] \arrow[d, phantom, "\overset{F_2\sigma}{\Rightarrow}"] \\
F_1c' \arrow[r, "\tau_{c'}"] & F_2c'
\end{tikzcd}
$$

i.e. the functors $\tau_c; F_2\_$ and $F_1\_;\tau_{c'}$ from $\mathcal{C}_1(c, c')$ to $\mathcal{C}_2(F_1c, F_2c')$ are equal.

Adjunctions are the basic tool to define data-types, while monads are used to model computations (see [Mog89b]). Their definition can be rephrased in the language of 2-categories and most of their properties can be proved in such a formal setting (see [Str72]), so these standard tools can be applied in a different 2-category, e.g. that of indexed categories (see Section 2).

Definition 1.6 *Let c and c' be objects of a 2-category $\mathcal{C}$.*

- *A* **monad** *over c is a triple (T, η, μ) s.t.*

$$
\begin{tikzcd}
c \arrow[r, "T"] & c
\end{tikzcd}
\qquad
\begin{tikzcd}
c \arrow[d, bend right, "id_c"'] \arrow[d, bend left, "T"] \arrow[d, phantom, "\overset{\eta}{\Rightarrow}"] \\
c
\end{tikzcd}
\qquad
\begin{tikzcd}
c \arrow[d, bend right, "T;T"'] \arrow[d, bend left, "T"] \arrow[d, phantom, "\overset{\mu}{\Rightarrow}"] \\
c
\end{tikzcd}
$$

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

- *An* **adjunction** *from c to c' is a quadruple (F, G, η, ϵ) s.t.*

$$
\begin{tikzcd}
c \arrow[r, bend left, "F"] & c' \arrow[l, bend left, "G"]
\end{tikzcd}
\qquad
\begin{tikzcd}
c \arrow[d, bend right, "id_c"'] \arrow[d, bend left, "F;G"] \arrow[d, phantom, "\overset{\eta}{\Rightarrow}"] \\
c
\end{tikzcd}
\qquad
\begin{tikzcd}
c' \arrow[d, bend right, "G;F"'] \arrow[d, bend left, "id_{c'}"] \arrow[d, phantom, "\overset{\epsilon}{\Rightarrow}"] \\
c'
\end{tikzcd}
$$

  $(\eta;F)\cdot(F;\epsilon) = \mathrm{id}_F$ *and* $(G;\eta)\cdot(\epsilon;G) = \mathrm{id}_G$.

It is obvious from the definition above, that the 2-categorical notions of monad and adjunction are preserved by 2-functors and that in the 2-category **Cat** they amount to familiar definitions.

2 Indexed categories and programming languages

In this section we define the 2-category $\mathbf{ICat}(\mathcal{B})$ of $\mathcal{B}$-indexed categories. Indexed categories model only one feature of a strongly typed programming language, namely that expressions are partitioned into two groups, type-expressions and program-expressions, and that the former are independent from the latter. Section 3 will discuss how to model other features by additional structure over an indexed category.

In Categorical Logic there is a similar use of indexed categories to capture that types and terms of first-order logic are given independently from formulas and proofs (see [See83]), while to enforce the principle of formulas-as-types one must be able to map fibers down to the base (see [See84]).

The general definition of indexed category is fairly complicated, since it involves the notion of canonical isomorphism. However, for representing languages it is more appropriate to use a stricter definition of $\mathcal{B}$-indexed category (e.g. see [See84, See87]), namely a functor from $\mathcal{B}^{op}$ to $\mathbf{Cat}$, where $\mathcal{B}$ is a small category and $\mathbf{Cat}$ is the category of small categories and functors.

Definition 2.1 *Given a small category $\mathcal{B}$, the 2-category $\mathbf{ICat}(\mathcal{B})$ of $\mathcal{B}$-indexed categories is defined as follows:*

- *an object (indexed category) is a functor $\mathcal{C}: \mathcal{B}^{op} \to \mathbf{Cat}$*

$$\begin{tikzcd} X & & \mathcal{C}[X] \arrow[d, "{\mathcal{C}[f]}"] \\ Y \arrow[u, "f"] \arrow[r, phantom, "\overset{\mathcal{C}}{\longmapsto}"] & & \mathcal{C}[Y] \end{tikzcd}$$

- *a 1-morphism (indexed functor) from $\mathcal{C}_1$ to $\mathcal{C}_2$ is a natural transformation $F: \mathcal{C}_1 \dot{\to} \mathcal{C}_2$, i.e. a family $\langle F[X]: \mathcal{C}_1[X] \to \mathcal{C}_2[X] | X \in \mathcal{B}\rangle$ of functors s.t.*

$$\begin{tikzcd} X & & \mathcal{C}_1[X] \arrow[r, "{F[X]}"] \arrow[d, "{\mathcal{C}_1[f]}"'] & \mathcal{C}_2[X] \arrow[d, "{\mathcal{C}_2[f]}"] \\ Y \arrow[u, "f"] \arrow[r, phantom, "\Longrightarrow"] & & \mathcal{C}_1[Y] \arrow[r, "{F[Y]}"'] & \mathcal{C}_2[Y] \end{tikzcd}$$

  i.e. for every $f: Y \to X$ in $\mathcal{B}$ the functors $\mathcal{C}_1[f]; F[Y]$ and $F[X]; \mathcal{C}_2[f]$ are equal.

- *Given 1-morphisms F_1 and F_2 from $\mathcal{C}_1$ to $\mathcal{C}_2$, a 2-morphism (indexed natural transformation) from F_1 to F_2 is a family $\langle \sigma[X]: F_1[X] \dot{\to} F_2[X] | X \in \mathcal{B}\rangle$ of* *natural transformations s.t.*

$$
\begin{tikzcd}
X & & \mathcal{C}_1[X] \arrow[r, bend left] \arrow[r, bend right] \arrow[r, phantom, "\Downarrow \sigma[X]"] \arrow[d, "{\mathcal{C}_1[f]}"'] & \mathcal{C}_2[X] \arrow[d, "{\mathcal{C}_2[f]}"] \\
Y \arrow[u, "f"] & \Longrightarrow & \mathcal{C}_1[Y] \arrow[r, bend left] \arrow[r, bend right] \arrow[r, phantom, "\Downarrow \sigma[Y]"] & \mathcal{C}_2[Y]
\end{tikzcd}
$$

i.e. for every $f\colon Y \to X$ in $\mathcal{B}$ the natural transformations $\mathcal{C}_1[f];\sigma[Y]$ and $\sigma[X];\mathcal{C}_2[f]$ are equal.

The definition of monad and adjunction for $\mathcal{B}$-indexed categories are particular instances of the 2-categorical definitions. The following proposition characterises an adjunction in $\mathbf{ICat}(\mathcal{B})$ as a family of adjunctions in $\mathbf{Cat}$ satisfying the *Beck-Chevalley condition.*

Proposition 2.2 ([PS78]) *Given a $\mathcal{B}$-indexed functor G from $\mathcal{C}_2$ to $\mathcal{C}_1$, an adjunction (F, G, η, ϵ) from $\mathcal{C}_1$ to $\mathcal{C}_2$ amounts to having a family*

$$\langle (F[b], G[b], \eta[b], \epsilon[b]) | b \in \mathcal{B} \rangle$$

satisfying the following properties:

loc $(F[b], G[b], \eta[b], \epsilon[b])$ *is an adjunction from $\mathcal{C}_1[b]$ to $\mathcal{C}_2[b]$, for every $b \in \mathcal{B}$;*

BC *the natural transformation $(\eta[b_1]; \mathcal{C}_1[f]; F[b_2]) \cdot (F[b_1]; \mathcal{C}_2[f]; \epsilon[b_2])$ from $\mathcal{C}_1[f]; F[b_2]$ to $F[b_1]; \mathcal{C}_2[f]$ is the identity, for every $f\colon b_2 \to b_1$ in $\mathcal{B}$.*

Remark 2.3 The condition loc means that for every $f\colon Y \to X$

$$
\begin{tikzcd}
\mathcal{C}_2[X] \arrow[d, "{\mathcal{C}_2[f]}"'] \arrow[r, phantom, "\perp"] & \mathcal{C}_1[X] \arrow[l, shift left=2, "{G[X]}"] \arrow[d, "{\mathcal{C}_1[f]}"] \\
\mathcal{C}_2[Y] \arrow[r, phantom, "\perp"] & \mathcal{C}_1[Y] \arrow[l, shift left=2, "{G[Y]}"]
\end{tikzcd}
\quad\text{with}\quad \mathcal{C}_2[X] \xrightarrow{F[X]} \mathcal{C}_1[X],\ \mathcal{C}_2[Y] \xrightarrow{F[Y]} \mathcal{C}_1[Y]
$$

and the square involving only the Gs commute, since G is an indexed functor. The square involving only Fs does not commute, in general, though there is a natural transformation

$$\begin{tikzcd}
\mathcal{C}_2[X] \arrow[d, "\mathcal{C}_2[f]"'] & \mathcal{C}_1[X] \arrow[l, "F[X]"'] \arrow[d, "\mathcal{C}_1[f]"] \arrow[dl, phantom, "\Uparrow \tau"] \\
\mathcal{C}_2[Y] & \mathcal{C}_1[Y] \arrow[l, "F[Y]"]
\end{tikzcd}$$

where $\tau_c \in \mathcal{C}_2(F[Y](\mathcal{C}_1[f]c), \mathcal{C}_2[f](F[X]c))$ is given by the following construction:

- take the unit $\eta[X]_c \in \mathcal{C}_1[X](c, G[X](F[X]c))$ of the adjunction $G[X] \vdash F[X]$
- take its image g in $\mathcal{C}_1[Y](\mathcal{C}_1[f]c, \mathcal{C}_1[f](G[X](F[X]c)))$ via the functor $\mathcal{C}_1[f]$
- g is in $\mathcal{C}_1[Y](\mathcal{C}_1[f]c, G[Y](\mathcal{C}_2[f](F[X]c)))$, because the indexed functor G commutes with substitution, i.e. $\mathcal{C}_2[f]; G[Y] = G[X]; \mathcal{C}_1[f]$
- τ_c is the morphism in $\mathcal{C}_2[Y](F[Y](\mathcal{C}_1[f]c), \mathcal{C}_2[f](F[X]c))$ corresponding to g via the natural isomorphism $F[Y](\_); \epsilon[Y]_b$ from $\mathcal{C}_1[Y](a, G[Y]b)$ to $\mathcal{C}_2[Y](F[Y]a, b)$, where $\epsilon[Y]$ is the counit for the adjunction $G[Y] \vdash F[Y]$

and the condition BC requires that τ defined above is the identity. The Beck-Chevalley condition in [PS78] requires only that the natural transformation given above is a *canonical isomorphism*, but we have adopted a *strict* notion of indexed category, where canonical isomorphisms are identities.

3 Intermezzo

At this point we review the category-theoretic structures used for interpreting some typed λ-calculi and discuss the additional structures needed to model various features of programming languages.

- Hyperdoctrines model the proof theory of intuitionistic first order logic (see [See83]). They are indexed-categories $\mathcal{C}: \mathcal{B}^{op} \to \mathbf{Cat}$, where morphisms in the base correspond to terms and morphisms in the fibers correspond to derivations. Moreover, the base $\mathcal{B}$ has finite products, the fibers $\mathcal{C}[X]$ are bicartesian closed, the functors $\mathcal{C}[f]$ preserve such structure and for every first projection $\pi_1^{b_1,b}: b_1 \times b \to b_1$ in $\mathcal{B}$ the functor $\mathcal{C}[\pi_1^{b_1,b}]$ has right adjoint $\forall[b_1]_b$ and left adjoint $\exists[b_1]_b$ (corresponding to universal and existential quantification over b) satisfying the Beck-Chevalley condition. The definition of universal and existential quantification in an hyperdoctrine $\mathcal{C}$ could be rephrased in terms of categories with attributes over $\mathcal{GC}$ (see Definition 6.11), provided both the base and the fibers have terminal objects and enough pullbacks.

- Locally cartesian closed categories model intuitionistic type theory with *equality types* (see [See84]). They amount to *identifying* the two levels of an hyperdoctrine, intuitively propositions and types are identified.

- Contextual categories, class of display maps, D-categories and categories with attributes provide *essentially equivalent* accounts of dependent types. Unlike the approach based on locally cartesian closed categories, they give a general category-theoretic understanding of dependent types (see Section 6).

- PL Categories model the higher order lambda calculus, or equivalently the proof theory of higher order intuitionistic propositional calculus (see [See87]). They are hyperdoctrines with an object $\Omega \in \mathcal{B}$ (the type of propositions) s.t. the set of objects of $\mathcal{C}[X]$ is $\mathcal{B}(X, \Omega)$, and for any $X \in \mathcal{B}$ a distinguished exponential Ω^X (the type of predicates over X).

- Monads can be used to model *notions of computation* (see [Mog89b, Mogar]). Computational types are easily accommodated in the simply typed λ-calculus, but it is still unclear how they fit with dependent types.

  > We believe that a proper understanding of computational types in a calculus of dependent types will clarify the semantics of *sharing constraints* and *generativity*, which at the moment is given only operationally (see [MTH90]).

Other features of programming languages, besides those of main interest for the paper, can be modelled as follows.

- A distinguished object Ω in the base category corresponds to the *kind* of all types, while exponentials Ω^X allow the interpretation of higher order type-constructors.

- Computations at *run-time* are modelled by a monad in the 2-category $\mathbf{ICat}(B)$. Since monads are a 2-categorical concept, it is clear how to define monads over indexed categories.

- Data-types (like products, sums, functional types, ...) are modelled by the usual adjunctions but in the 2-category $\mathbf{ICat}(B)$ instead of $\mathbf{Cat}$.

  > This is not quite right, because function spaces (and dependent products) are given via an adjunction with parameter. A 2-categorical reformulation may have to rely on fibrations in a 2-category, which is far from *simple* (see [Str73]).

- Polymorphic types are modelled like universal quantifiers (in Hyperdoctrines), while abstract data-types are modelled like existential quantifiers (see [MP88]).

Remark 3.1

- The requirement $\mathrm{Obj}(\mathcal{C}[X]) = \mathcal{B}(X, \Omega)$ for the kind Ω of all types is not always justified in relation to programming languages. For instance, in ML there are types and type schemas. Types correspond to elements of Ω, while type schemas correspond to objects in the fiber categories. In Section 5 we introduce a language which does not identify types and type schemas. The inclusion of types into type schemas is modelled by an object $t \in \mathcal{C}[\Omega]$ (the generic type), so that a type expression $f: X \to \Omega$ (with a free variable of kind X) corresponds to the type schema $\mathcal{C}[f](t)$ in $\mathcal{C}[X]$. When $\mathrm{Obj}(\mathcal{C}[X]) = \mathcal{B}(X, \Omega)$, the generic type is simply $\mathrm{id}_\Omega \in \mathcal{C}[\Omega]$.

- A general understanding of dependent types is essential for explaining dependent types in the category of modules in terms of dependent types in the base and fiber categories. For instance, there are non-trivial dependencies at the level of ML-signatures, even though core ML does not have dependent types.

  The semantics of dependent types is based on a special kind of indexed categories (fibrations), where it is possible to go back and forth from one level to the other (see definition of D-category in [Ehr88]). Such a possibility of moving back and forth *contradicts* the independence of type-expressions from program-expressions, we will consider instead an indexed category $\mathcal{C}$ with two D-category structures, one for the base and one for the fibers (see Section 7).

4 The category of modules

The 2-category $\mathbf{ICat}(\mathcal{B})$ is isomorphic to the 2-category of split $\mathcal{B}$-fibrations (see [Ben85]). Since $\mathcal{B}$-fibrations are functors with codomain $\mathcal{B}$ satisfying certain additional properties, the 2-category of $\mathcal{B}$-fibrations is a 2-subcategory of $\mathbf{Cat}{\downarrow}\mathcal{B}$ and the 2-embedding, mapping a $\mathcal{B}$-indexed category $\mathcal{C}$ to the corresponding $\mathcal{B}$-fibration $\pi_\mathcal{C}: \mathcal{GC} \to \mathcal{B}$, can be viewed as a 2-functor from $\mathbf{ICat}(\mathcal{B})$ to $\mathbf{Cat}{\downarrow}\mathcal{B}$. For our purposes we need only the 2-functor $\mathcal{G}$ from $\mathbf{ICat}(\mathcal{B})$ to $\mathbf{Cat}$, mapping a programming language $\mathcal{C}$ to its category of modules $\mathcal{GC}$. In Section 5 we will define the category of modules for HML.

Definition 4.1 *The 2-functor $\mathcal{G}$ from $\mathbf{ICat}(\mathcal{B})$ to $\mathbf{Cat}$ is defined as follows:*

- *if $\mathcal{C}$ is an indexed category, then $\mathcal{GC}$ is the category s.t.*

$$\begin{array}{ll} \langle X_1, c_1\rangle & \text{where } X_1 \in \mathcal{B} \text{ and } c_1 \in \mathcal{C}[X_1] \\ \big\downarrow \langle f, g\rangle & \text{where } f \in \mathcal{B}(X_1, X_2) \text{ and } g \in \mathcal{C}[X_1](c_1, \mathcal{C}[f]c_2) \\ \langle X_2, c_2\rangle & \text{where } X_2 \in \mathcal{B} \text{ and } c_2 \in \mathcal{C}[X_2] \end{array}$$

  identity over $\langle X, c\rangle$ is $\langle \mathrm{id}_X, \mathrm{id}_c\rangle$, composition of $\langle f_1, g_1\rangle$ and $\langle f_2, g_2\rangle$ is $\langle f_1; f_2, g_1; \mathcal{C}[f_1]g_2\rangle$

- *if F is an indexed functor from $\mathcal{C}_1$ to $\mathcal{C}_2$, then $\mathcal{G}F$ is the functor s.t.*

$$\begin{array}{ccc} \langle X, c\rangle & & \langle X, F[X]c\rangle \\ \big\downarrow \langle f, g\rangle \ \text{in } \mathcal{GC}_1 & \overset{\mathcal{G}F}{\longmapsto} & \big\downarrow \langle f, F[X]g\rangle \ \text{in } \mathcal{GC}_2 \\ \langle X', c'\rangle & & \langle X', F[X']c'\rangle \end{array}$$

- *if F_1 and F_2 are indexed functors from $\mathcal{C}_1$ to $\mathcal{C}_2$ and τ is an indexed natural transformation from F_1 to F_2, then $\mathcal{G}\tau$ is the natural transformation s.t.*

$$\langle X, c\rangle \text{ in } \mathcal{GC}_1 \qquad \stackrel{\mathcal{G}\tau}{\longmapsto} \qquad \langle X, F_1[X]c\rangle \xrightarrow{\langle \mathrm{id}_X, \tau[X]_c\rangle} \langle X, F_2[X]c\rangle \text{ in } \mathcal{GC}_2$$

Remark 4.2 After defining the general construction which maps a programming language, viewed as an indexed category $\mathcal{C}$, to its category $\mathcal{GC}$ of modules, we can investigate how additional structure on $\mathcal{GC}$ depends on (is induced by) additional structure on $\mathcal{C}$ and/or the base category $\mathcal{B}$. For instance, an indexed monad (T, η, μ) over $\mathcal{C}$, which corresponds to a notion of run-time computation, induces a monad over $\mathcal{GC}$ by simply taking its image w.r.t. the 2-functor $\mathcal{G}$, more precisely $(\mathcal{G}T)(\langle b, c\rangle) = \langle b, T[b]c\rangle$.

5 An example: HML

In this section we define a language, HML (for higher-order ML), which extends the one given in [Mog89a]. The main features of HML are:

- Independence of type-expressions from program-expressions (as in system $F\omega$), enforced *syntactically* by having two levels of judgements (and contexts), so that HML can be viewed as an indexed category.
- Dependent kinds and *type schemas* (as in the Calculus of Constructions CC described in [HP89]). More precisely, HML has dependent sums and products for kinds and type schemas, and type schemas universally and existentially quantified over kinds.

  This enable us to analyse in full generality dependent types at the level of modules, that are necessary for giving a type-theoretic account of sharing constraints (see [MTH90]), since these constraints specify *equality* of (a limited form of) program-expressions.
- Distinction between types and type schemas (as in ML), so that having both proper dependency at the level of type schemas (necessary to accommodate sharing constraints) and independence of type-expressions from program-expressions (as in $F\omega$) does not lead to inconsistency.

A study of the additional structure on the category $\mathcal{GC}$ of HML-modules is postponed to Section 7, where such structure is compared to that for the Calculus of Constructions (as described in [HP89]).

Remark 5.1 The study of dependent types at the level of modules is relevant to the calculus λ^{ML}_{mod} studied in [HMM90]. Though λ^{ML}_{mod} has an important restriction, type schemas are independent from the evaluation of program-expressions, which is enforced by replacing closure w.r.t. Σ- and Π-types with closure w.r.t. products and function spaces only. Such a restriction is essential to prove that "type-checking can be done at compile-time" (even for the calculus of modules).

Our notion of *independence* of type-expressions from program-expressions is related to *phase distinction* as introduced in [Car88] "... the execution of a program is carried out in two *phases*: a type-checking phase (*compile-time*) and an execution phase (*run-time*)". Independence and phase distinction coincide, when types and type schemas are identified, but in general phase distinction is a stronger requirement than independence (provided types are a *subset* of type schemas).

In λ_{mod}^{ML} phase distinction (and termination of the type-checking phase) is achieved by having type schemas independent from program-expressions. In HML program-expressions may occur in type schemas (at least potentially), so *the only way* to guaranty termination of type-checking is to use a decidable approximation of equality for program-expressions.

Overview. HML has four syntactic classes:

- Kinds $\Delta \vdash \mathrm{k}$, whose raw syntax is

  $$\mathrm{k} \in \mathit{Kind}::= 1 \mid \Omega \mid (\Sigma v{:}\,\mathrm{k}_1.\mathrm{k}_2) \mid (\Pi v{:}\,\mathrm{k}_1.\mathrm{k}_2)$$

  where Δ is a constructor context, i.e. a sequence $v_1{:}\,\mathrm{k}_1, \ldots, v_m{:}\,\mathrm{k}_m$.

- Constructors $\Delta \vdash u{:}\,\mathrm{k}$, whose raw syntax is

  $$u \in \mathit{Constr}::= v \mid 1 \mid \times \mid \rightarrow \mid * \mid \langle u_1, u_2\rangle \mid \pi_i(u) \mid (\lambda v{:}\,\mathrm{k}.u) \mid u_1(u_2)$$

  Besides the constants $1{:}\,\Omega$ and $\times, \rightarrow{:}\,\Omega \rightarrow \Omega \rightarrow \Omega$ we could have considered also $\forall, \exists{:}\,(\mathrm{k} \rightarrow \Omega) \rightarrow \Omega$, if types were closed w.r.t. universal and existential quantification over kind (like $F\omega$). As common practice, we write $u_1 \rightarrow u_2$ instead of $\rightarrow u_1 u_2$ (similarly for $\times$) and τ for a constructor of kind Ω.

- Type schemas $\Delta; \Gamma \vdash \sigma$, whose raw syntax is

  $$\sigma \in \mathit{Schema}::= 1 \mid \mathrm{set}(\tau) \mid (\Sigma x{:}\,\sigma_1.\sigma_2) \mid (\Pi x{:}\,\sigma_1.\sigma_2) \mid (\exists v{:}\,\mathrm{k}_1.\sigma_2) \mid (\forall v{:}\,\mathrm{k}_1.\sigma_2)$$

  where Γ is a term context, i.e. a sequence $x_1{:}\,\sigma_1, \ldots, x_n{:}\,\sigma_n$.

- Terms $\Delta; \Gamma \vdash e{:}\,\sigma$, whose raw syntax is

  $$\begin{array}{rl} e \in \mathit{Term}::= & x \mid * \mid \langle e_1, e_2\rangle \mid \pi_i(e) \mid (\lambda x{:}\,\sigma.e) \mid e_1(e_2) \mid \\ & (u, e) \mid (\mathrm{let}\,(v, x){=}e\,\mathrm{in}\,e') \mid (\lambda v{:}\,\mathrm{k}.e) \mid e(u) \end{array}$$

For each syntactic class there are two forms of judgements: formation judgements and equality judgements. Moreover, there are two auxiliary forms of formation judgements for contexts.

	formation	*equality*
constr. contexts	$\Delta \vdash$	
kinds	$\Delta \vdash \mathrm{k}$	$\Delta \vdash \mathrm{k}_1 = \mathrm{k}_2$
constructors	$\Delta \vdash u{:}\,\mathrm{k}$	$\Delta \vdash u_1 = u_2{:}\,\mathrm{k}$
term contexts	$\Delta; \Gamma \vdash$	
schemas	$\Delta; \Gamma \vdash \sigma$	$\Delta; \Gamma \vdash \sigma_1 = \sigma_2$
terms	$\Delta; \Gamma \vdash e{:}\,\sigma$	$\Delta; \Gamma \vdash e_1 = e_2{:}\,\sigma$

Remark 5.2 There are many similarities between HML and the Calculus of Constructions CC described in [HP89], but one crucial difference (tightly linked to the distinction between types and type schemas): in HML the formation and equality rules for constructors and kinds are independent from the rules for schemas and terms, while in CC all forms of judgement are interdependent.

In an indexed category the distinction between types and type schemas is easily captured as follows: types correspond to elements of an object Ω in the base category, type schemas correspond to objects in the fiber categories. The identification of types and type schemas (typical of $F\omega$ and CC) amounts to having a one-one correspondence between morphisms from b to Ω in the base category $\mathcal{B}$ and objects in the fiber $\mathcal{C}[b]$ over b. In HML a type τ can be made into a type schema $\mathrm{set}(\tau)$, so types are type schemas, but not the other way around.

Inference Rules. The inference rules of HML are partitioned in two sets s.t. the first is independent from the second (see Appendix). The first set of rules is for deriving formation and equality judgements for kinds and constructors, which amounts to Martin-Löf predicative theory of dependent types with a kind constant Ω and constructor constants 1, $\times$ and $\rightarrow$ (of appropriate kind). The second set of rules is for deriving formation and equality judgements for type schemas and terms. It is similar to the rules in [HP89] for types and terms of the Calculus of Constructions.

Categorical view. The calculus HML can be viewed as an indexed category according to a standard term-model construction (see [See87]), where constructors (up to constructor equality) are morphisms in the base category and terms are morphisms in the fiber categories.

Definition 5.3 HML *can be viewed as an indexed category* $\mathcal{C}\colon \mathcal{B}^{op} \rightarrow \mathbf{Cat}$.

- *The base category* $\mathcal{B}$ *is defined as follows:*
  - *objects are equivalence classes* $[\mathrm{k}]$ *of kinds, i.e.* $\{\mathrm{k}'|\emptyset \vdash \mathrm{k} = \mathrm{k}'\}$
  - *morphisms from* $[\mathrm{k}_1]$ *to* $[\mathrm{k}_2]$ *are equivalence classes* $[u]_{\mathrm{k}_1,\mathrm{k}_2}$ *of constructors, i.e.* $\{u'|v\colon \mathrm{k}_1 \vdash u = u'\colon \mathrm{k}_2\}$, *where it does not matter what one chooses as representative for* $[\mathrm{k}_1]$ *and* $[\mathrm{k}_2]$
  - $[u_1]_{\mathrm{k}_1,\mathrm{k}_2}$ *followed by* $[u_2]_{\mathrm{k}_2,\mathrm{k}_3}$ *is* $[[u_1/v]u_2]_{\mathrm{k}_1,\mathrm{k}_3}$
- *If* $[\mathrm{k}]$ *is an object of* $\mathcal{B}$, *then the category* $\mathcal{C}[[\mathrm{k}]]$ *is defined as follows:*
  - *objects are equivalence classes* $[\sigma]_\mathrm{k}$ *of schemas, i.e.* $\{\sigma'|v\colon \mathrm{k}; \emptyset \vdash \sigma = \sigma'\}$
  - *morphisms from* $[\sigma_1]_\mathrm{k}$ *to* $[\sigma_2]_\mathrm{k}$ *are equivalence classes* $[e]_{\mathrm{k};\sigma_1,\sigma_2}$ *of terms, i.e.* $\{e'|v\colon \mathrm{k}; x\colon \sigma_1 \vdash e = e'\colon \sigma_2\}$
  - $[e_1]_{\mathrm{k};\sigma_1,\sigma_2}$ *followed by* $[e_2]_{\mathrm{k};\sigma_2,\sigma_3}$ *is* $[[e_1/x]e_2]_{\mathrm{k};\sigma_1,\sigma_3}$
- *If* $f = [u]_{\mathrm{k}_1,\mathrm{k}_2}$ *is a morphism from* $[\mathrm{k}_1]$ *to* $[\mathrm{k}_2]$ *in* $\mathcal{B}$, *then* $\mathcal{C}[f]$ *is the functor from* $\mathcal{C}[[\mathrm{k}_2]]$ *to* $\mathcal{C}[[\mathrm{k}_1]]$ *defined as follows:*

– $[\sigma]_{\mathrm{k}_2}$ *is mapped to* $[[u/v]\sigma]_{\mathrm{k}_1}$ *and*

– $[e]_{\mathrm{k}_2;\sigma_1,\sigma_2}$ *is mapped to* $[[u/v]e]_{\mathrm{k}_1;[u/v]\sigma_1,[u/v]\sigma_2}$

The indexed category $\mathcal{C}$ has additional structure, but we will study it as additional structure on the category $\mathcal{GC}$ of modules, so that it can be compared more easily with the categorical structure of the Calculus of Constructions considered in [HP89].

Definition 5.4 *The category* $\mathcal{GC}$ *of* HML*-modules can be described as follows:*

- *objects are pairs* $\langle [\mathrm{k}], [\sigma]_{\mathrm{k}} \rangle$*, denoted by* $[v\colon \mathrm{k}; \sigma]$
- *morphisms from* $[v\colon \mathrm{k}_1; \sigma_1]$ *to* $[v\colon \mathrm{k}_2; \sigma_2]$ *are pairs* $\langle [u]_{\mathrm{k}_1,\mathrm{k}_2}, [e]_{\mathrm{k}_1;\sigma_1,[u/v]\sigma_2} \rangle$ *denoted by* $[(v\colon \mathrm{k}_1; x\colon \sigma_1).\langle u; e\rangle]_{\mathrm{k}_2;\sigma_2}$
- $[(v\colon \mathrm{k}_1; x\colon \sigma_1).\langle u_1; e_1\rangle]_{\mathrm{k}_2;\sigma_2}$ *followed by* $[(v\colon \mathrm{k}_2; x\colon \sigma_2).\langle u_2; e_2\rangle]_{\mathrm{k}_3;\sigma_3}$ *is the pair* $[(v\colon \mathrm{k}_1; x\colon \sigma_1).\langle [u_1/v]u_2; [u_1, e_1/v, x]e_2\rangle]_{\mathrm{k}_3;\sigma_3}$

Remark 5.5 At this point we can outline the correspondence between ML-modules and $\mathcal{GC}$. Since we have not yet defined all the relevant structure on $\mathcal{GC}$, we can consider only *closed* signatures, structures and functors (i.e. without reference to non-local variables). This restriction will be dropped in Remark 7.11.

- A closed ML-signature corresponds to an object of $\mathcal{GC}$, e.g.

  signature *sig* = (**sig** type v; val $x\colon \sigma$ **end**)

  corresponds to the object $[v\colon \Omega; \sigma]$.

- A closed ML-structure of closed signature *sig* corresponds to an element of $[v\colon \Omega; \sigma]$, e.g.

  structure S = (**struct** type $v = u$; val $x = e$ **end**)

  corresponds to the morphism $[(v\colon 1; x\colon 1).\langle u; e\rangle]_{\Omega;\sigma}$ from the terminal object $[v\colon 1; 1]$ to $[v\colon \Omega; \sigma]$.

- A closed ML-functor, whose parameter and result signatures are closed, corresponds to a morphism of $\mathcal{GC}$, e.g.

  functor $F(S\colon sig_1)\colon sig_2 =$
  struct type $v = [S.v/v]u$; val $x = [S.v, S.x/v, x]e$ **end**

  corresponds to the morphism $[(v\colon \Omega; x\colon \sigma_1).\langle u; e\rangle]_{\Omega;\sigma_2}$ from $[v\colon \Omega; \sigma_1]$ to $[v\colon \Omega; \sigma_2]$.

While our explanation of ML-signatures and ML-structures matches the type-theoretic intuition in [Mac86, HM88], there is a major difference in the understanding of ML-functors:

> for us it is essential that ML-types (including those in the body of an ML-functor) do not depend on values, otherwise it would not be possible to associate a morphism in $\mathcal{GC}$ to an ML-functor. On the other hand, such a property is ignored in [Mac86, HM88].

Note that this property holds even for HML, despite having dependent kinds and type schemas.

6 A categorical treatment of dependent types

Up to now there is no agreement on what is the *best* way of looking at dependent types categorically (see [Car78, See84, Tay87, HP89, Ehr88, Str88, Cur89, Pit89, Jac90, Pitar]). We follow the approach based on *categories with attributes* (see [Car78] and also [Cur89, Pit89, Pitar]), which avoids certain *limitations* of classes of display maps (see [Tay87, HP89]) and the *unnecessary generality* of D-categories (see [Ehr88]). This section reviews the concepts involved in the definition of categorical model for the Calculus of Construction (see Definition 6.16), namely: category with attributes, generic type, unit types, dependent sums and products, universal and existential types, embedding between categories with attributes. Moreover, it introduces some operations on categories with attributes (see Definition 6.14).

Remark 6.1 In the introduction we advocate formulating concepts 2-categorically. Since the concept of fibration can be formulated 2-categorically (see [Str73]), it is possible to formulate *categories with attributes* and other concepts introduced below 2-categorically, too. However, we have not done so, because it seemed too complicated and unintelligible. Nevertheless, a 2-categorical formulation is essential to define *indexed categories with attributes* and explain how type dependency in the category of modules is induced by type dependency in the base and the fibers via the 2-functor $\mathcal{G}$ (given by the Grothendieck construction).

Definition 6.2 *A* **category with attributes** *is specified by a quadruple* $(\mathcal{C}, \mathcal{D}, G, \mathrm{p})$ *made of the following data:*

- *a category* $\mathcal{C}$ *with a terminal object* 1*,*
- *a discrete* $\mathcal{C}$*-indexed category* $\mathcal{D}$ *(we identify sets with discrete categories)*
- *a natural transformation* p

$$\begin{tikzcd} \mathcal{G}\mathcal{D} \arrow[r, shift left=2, "G"] \arrow[r, shift right=2, "\pi_{\mathcal{D}}"'] \arrow[r, phantom, "\Downarrow \mathrm{p}"] & \mathcal{C} \end{tikzcd} \quad \textit{in } \mathbf{Cat}$$

s.t. for all $f: Y \to X$ *and* $a \in \mathcal{D}[X]$

$$\begin{tikzcd} Y \cdot f^* a \arrow[r, "f \cdot a"] \arrow[d, "\mathrm{p}_{f^* a}"'] \arrow[dr, phantom, very near start, "\lrcorner"] & X \cdot a \arrow[d, "\mathrm{p}_a"] \\ Y \arrow[r, "f"'] & X \end{tikzcd}$$

i.e. the square is a pullback, where we write

– f^*a *for* $\mathcal{D}[f](a)$,
– $X \cdot a$ *for the* **context extension** $G(\langle X, a\rangle)$,
– $f \cdot a$ *for* $G(\langle f, \mathrm{id}_{\mathcal{D}[f]a}\rangle)$,
– p_a *for the* **context projection** p *at* $\langle X, a\rangle$.

Remark 6.3 A general outline of the interpretation of judgements for a calculus of dependent types in a category with attributes goes as follows:

	Judgement	Interpretation
context	$\Gamma \vdash$	object X of $\mathcal{C}$
type	$\Gamma \vdash \sigma$	element a of $\mathcal{D}[X]$
term	$\Gamma \vdash e{:}\,\sigma$	section f of p_a, i.e. $f{:}\,X \to X \cdot a$ s.t. $f;\mathrm{p}_a = \mathrm{id}_X$
type equality	$\Gamma \vdash \sigma = \sigma'$	$a = a'$
term equality	$\Gamma \vdash e = e'{:}\,\sigma$	$f = f'$

The terminal object 1 is used to interpret the empty context.

Given a category with attributes we define the following categories and functors:

Definition 6.4 (Slice category) *Let* $X \in \mathcal{C}$, *then the* **slice category** $\mathcal{C}/X$ *has as objects the morphisms with codomain* X *and as morphisms from* $f_1{:}\,Y_1 \to X$ *to* $f_2{:}\,Y_2 \to X$ *the* $g{:}\,Y_1 \to Y_2$ *s.t.*

$$\begin{tikzcd}
Y_1 \arrow[r, "g"] \arrow[dr, "f_1"'] & Y_2 \arrow[d, "f_2"] \\
& X
\end{tikzcd}$$

Let $a \in \mathcal{D}[X]$, *then* $\_\cdot a{:}\,\mathcal{C}/X \to \mathcal{C}/(X\cdot a)$ *is the functor mapping an object* $f{:}\,Y \to X$ *onto* $f \cdot a{:}\,Y \cdot f^*a \to X \cdot a$ *and a morphism* g *from* f_1 *to* f_2 *onto the unique* $g \cdot a$ *s.t.*

$$\begin{tikzcd}
& Y_1 \arrow[r, "g"] & Y_2 \arrow[d, "f_2"] \\
Y_1 \cdot f_1^* a \arrow[ur, "\mathrm{p}"] \arrow[r, dashed, "g \cdot a"] \arrow[dr, "f_1 \cdot a"'] & Y_2 \cdot f_2^* a \arrow[ur, "\mathrm{p}"] \arrow[d, "f_2 \cdot a"] & X \\
& X \cdot a \arrow[ur, "\mathrm{p}"'] &
\end{tikzcd}$$

Definition 6.5 (Relative slice category) *Let $X \in \mathcal{C}$, then the* **relative slice category** $\mathcal{C}/_{\mathcal{D}}X$ *has as objects the elements of $\mathcal{D}[X]$ and as morphisms from a to b the $g\colon X \cdot a \to X \cdot b$ s.t.*

$$\begin{tikzcd}
X \cdot a \arrow[r, "g"] \arrow[dr, "\mathrm{p}"'] & X \cdot b \arrow[d, "\mathrm{p}"] \\
& X
\end{tikzcd}$$

Let $f\colon Y \to X$, then $f^*\colon \mathcal{C}/_{\mathcal{D}}X \to \mathcal{C}/_{\mathcal{D}}Y$ is the functor mapping an object a onto f^*a and a morphism g from a to b onto the unique f^*g s.t.

$$\begin{tikzcd}
& X \cdot a \arrow[r, "g"] & X \cdot b \arrow[d, "\mathrm{p}"] \\
Y \cdot f^*a \arrow[ur, "f \cdot a"] \arrow[r, dashed, "f^*g"] \arrow[dr, "\mathrm{p}"'] & Y \cdot f^*b \arrow[ur, "f \cdot b"] \arrow[d, "\mathrm{p}"] & X \\
& Y \arrow[ur, "f"'] &
\end{tikzcd}$$

The following definition gives a categorical characterisation of certain types constructors, dependent sums and products, in terms of universal properties and *commutativity with substitution*, i.e. Beck-Chevalley condition.

Definition 6.6 *We say that a category with attributes $(\mathcal{C}, \mathcal{D}, G, \mathrm{p})$ has*

- **generic type** $(U \in \mathcal{C}, t \in \mathcal{D}[U]) \overset{\Delta}{\iff}$
  for every $X \in \mathcal{C}$ and $a \in \mathcal{D}[X]$ exists unique $f\colon X \to U$ s.t. $a = f^*t$
- **units** $1[X] \in \mathcal{D}[X]$, *for $X \in \mathcal{C}$* $\overset{\Delta}{\iff}$
  - *for every $f\colon Y \to X$*

$$\begin{tikzcd}
Y \cdot 1' \arrow[r, "f \cdot 1"] \arrow[d, "\mathrm{p}"'] & X \cdot 1 \arrow[d, "\mathrm{p}"] \\
Y \arrow[r, "f"'] & X
\end{tikzcd}$$

where $1 \triangleq 1[X]$ and $1' \triangleq 1[Y] = f^*1$

– *for every* $f\colon Y \to X$ *exists unique* ! *s.t.*

$$\begin{tikzcd} Y \arrow[r, dashed, "!"] \arrow[dr, "f"'] & X \cdot 1 \arrow[d, "\mathrm{p}"] \\ & X \end{tikzcd}$$

or equivalently $!\colon X \xrightarrow{\sim} X \cdot 1$, *i.e.* ! *is an iso, from* id_X *to* p_1 *in* $\mathcal{C}/X$.

- **sums** $\Sigma[X]a.b \in \mathcal{D}[X]$ *with unit* $\eta[X]_{a,b}$ *for* $X \in \mathcal{C}$, $a \in \mathcal{D}[X]$ *and* $b \in \mathcal{D}[X \cdot a]$

$$\begin{tikzcd} X \cdot a \cdot b \arrow[r, "\eta"] \arrow[dr, "\mathrm{p}"'] & X \cdot \Sigma \cdot \mathrm{p}^* a \arrow[r, "\mathrm{p}"] \arrow[d, "\mathrm{p} \cdot a"] & X \cdot \Sigma \arrow[d, "\mathrm{p}"] \\ & X \cdot a \arrow[r, "\mathrm{p}"'] & X \end{tikzcd}$$

where $\Sigma \triangleq \Sigma[X]a.b$ *and* $\eta \triangleq \eta[X]_{a,b} \overset{\Delta}{\iff}$

– *for every* $f\colon Y \to X$

$$\begin{tikzcd} & X \cdot a \cdot b \arrow[r, "\eta"] & X \cdot \Sigma \cdot \mathrm{p}^* a \arrow[r, "\mathrm{p}"] & X \cdot \Sigma \arrow[d, "\mathrm{p}"] \\ Y \cdot a' \cdot b' \arrow[ur, "f \cdot a \cdot b"] \arrow[r, dashed, "\eta'"] \arrow[dr, "\mathrm{p}"'] & Y \cdot \Sigma' \cdot \mathrm{p}^* a' \arrow[ur, "f \cdot \Sigma \cdot \mathrm{p}^* a"] \arrow[r, "\mathrm{p}"] \arrow[d, "\mathrm{p} \cdot a'"] & Y \cdot \Sigma' \arrow[ur, "f \cdot \Sigma"] \arrow[d, "\mathrm{p}"] & X \\ & Y \cdot a' \arrow[r, "\mathrm{p}"'] & Y \arrow[ur, "f"'] & \end{tikzcd}$$

where $a' \triangleq f^* a$, $b' \triangleq (f \cdot a)^* b$, $\Sigma' \triangleq \Sigma[Y]a'.b' = f^* \Sigma$ *and* $\eta' \triangleq \eta[Y]_{a',b'}$

– *there is a natural isomorphism* $\mathcal{C}/X(\mathrm{p}_\Sigma, \_) \dot{\cong} \mathcal{C}/(X \cdot a)(\mathrm{p}_b, \_ \cdot a)$ *given by*

$$\begin{tikzcd} X \cdot \Sigma \arrow[r, "g"] \arrow[dr, "\mathrm{p}"'] & Y \arrow[d, "f"] \\ & X \end{tikzcd} \quad g \overset{\sim}{\mapsto} h \triangleq \eta; (g \cdot a) \quad \begin{tikzcd} X \cdot a \cdot b \arrow[r, "h"] \arrow[dr, "\mathrm{p}"'] & Y \cdot f^* a \arrow[d, "f \cdot a"] \\ & X \cdot a \end{tikzcd}$$

or equivalently $s \triangleq \eta; \mathrm{p}\colon X \cdot a \cdot b \xrightarrow{\sim} X \cdot \Sigma$ *from* $\mathrm{p}_b; \mathrm{p}_a$ *to* p_Σ *in* $\mathcal{C}/X$.

- **products** $\Pi[X]a.b \in \mathcal{D}[X]$ *with counit* $\epsilon[X]_{a,b}$ *for* $X \in \mathcal{C}$, $a \in \mathcal{D}[X]$ *and* $b \in \mathcal{D}[X \cdot a]$

$$\begin{tikzcd}
X \cdot a \cdot b \arrow[dr, "\mathrm{p}"'] & X \cdot \Pi \cdot \mathrm{p}^* a \arrow[l, "\epsilon"'] \arrow[r, "\mathrm{p}"] \arrow[d, "\mathrm{p} \cdot a"] & X \cdot \Pi \arrow[d, "\mathrm{p}"] \\
& X \cdot a \arrow[r, "\mathrm{p}"'] & X
\end{tikzcd}$$

where $\Pi \triangleq \Pi[X]a.b$ *and* $\epsilon \triangleq \epsilon[X]_{a,b} \overset{\Delta}{\iff}$

  - *for every* $f\colon Y \to X$

$$\begin{tikzcd}
& X \cdot a \cdot b & X \cdot \Pi \cdot \mathrm{p}^* a \arrow[l, "\epsilon"'] \arrow[r, "\mathrm{p}"] & X \cdot \Pi \arrow[d, "\mathrm{p}"] \\
Y \cdot a' \cdot b' \arrow[ur, "f \cdot a \cdot b"] \arrow[dr, "\mathrm{p}"'] & Y \cdot \Pi' \cdot \mathrm{p}^* a' \arrow[l, dashed, "\epsilon'"'] \arrow[ur, "f \cdot \Pi \cdot \mathrm{p}^* a"] \arrow[r, "\mathrm{p}"] \arrow[d, "\mathrm{p} \cdot a'"] & Y \cdot \Pi' \arrow[ur, "f \cdot \Pi"] \arrow[d, "\mathrm{p}"] & X \\
& Y \cdot a' \arrow[r, "\mathrm{p}"'] & Y \arrow[ur, "f"'] &
\end{tikzcd}$$

  where $a' \triangleq f^* a$, $b' \triangleq (f \cdot a)^* b$, $\Pi' \triangleq \Pi[Y]a'.b' \triangleq f^*\Pi$ *and* $\epsilon' \triangleq \epsilon[Y]_{a',b'}$

  - *there is a natural isomorphism* $\mathcal{C}/X(\_, \mathrm{p}_\Pi) \dot{\cong} \mathcal{C}/(X \cdot a)(\_ \cdot a, \mathrm{p}_b)$ *given by*

$$\begin{tikzcd}
X \cdot \Pi \arrow[dr, "\mathrm{p}"'] & Y \arrow[l, "g"'] \arrow[d, "f"] \\
& X
\end{tikzcd}
\quad g \overset{\sim}{\mapsto} h \triangleq (g \cdot a); \epsilon \quad
\begin{tikzcd}
X \cdot a \cdot b \arrow[dr, "\mathrm{p}"'] & Y \cdot f^* a \arrow[l, "h"'] \arrow[d, "f \cdot a"] \\
& X \cdot a
\end{tikzcd}$$

Remark 6.7 The definitions of units and sums are essentially equivalent to those in [HP89], but our definitions of generic type and products are stronger.

- In the definition of generic type [HP89] demands only existence of $f\colon X \to U$, while we demand also uniqueness. Our definition describes better the intended property of a generic type, especially in *syntactic models*, and it seems more appropriate, when there is a canonical choice of pullbacks.

- In the definition of products [HP89] demands $\mathcal{C}/X(\_, \mathrm{p}_\Pi) \dot{\cong} \mathcal{C}/(X \cdot a)(\_ \cdot a, \mathrm{p}_b)$ for _ ranging only over $\mathrm{p}_c\colon X \cdot c \to X$ (i.e. it uses $\mathcal{C}/_{\mathcal{D}}\_$ instead of $\mathcal{C}/\_$), while we let _ range over arbitrary $f\colon Y \to X$. Note that the *absoluteness* of products (see 2.8 of [HP89]) becomes a simple consequence of our definition.

The unit η for sums and the counit ϵ for products have a simple description as *context realisation* in a theory of dependent types:

- η is the realisation $\langle x, \langle a, b\rangle, a\rangle$ of the context $[x{:}\, X, y{:}\, (\Sigma a{:}\, A.B), a{:}\, A]$ in the context $[x{:}\, X, a{:}\, A, b{:}\, B]$;
- ϵ is the realisation $\langle x, a, f(a)\rangle$ of the context $[x{:}\, X, a{:}\, A, b{:}\, B]$ in the context $[x{:}\, X, f{:}\, (\Pi a{:}\, A.B), a{:}\, A]$.

Example 6.8 A category $\mathcal{C}$ can be made into a trivial category with attributes (having units, sums and products) by defining:

- $\mathcal{D}[X] = \{*\}$
- $X \cdot * = X$
- $f \cdot * = f$
- $\mathrm{p}_* = \mathrm{id}_X$

Example 6.9 A category $\mathcal{C}$ with finite products can be made into a category with attributes, having units and sums, by defining:

- $\mathcal{D}[X] = \mathrm{Obj}(\mathcal{C})$
- $X \cdot a = X \times a$
- $f \cdot a = f \times \mathrm{id}_a$
- $\mathrm{p}_a = \pi_1^{X,a}$
- $1[X] = 1$
- $\Sigma[X]a.b = a \times b$

Moreover, if $\mathcal{C}$ has exponentials, then as a category with attributes it has also products $\Pi[X]a.b = b^a$. In summary, a cartesian closed category can be viewed as a category with attributes having units, sums and products.

Example 6.10 The category **Cat** of small categories can be made into a category with attributes (having units, sums and products) by defining:

- $\mathcal{D}[\mathcal{B}] = \mathrm{Obj}(\mathbf{Cat}^{\mathcal{B}^{op}})$, i.e. the class of $\mathcal{B}$-indexed categories
- $\mathcal{B} \cdot \mathcal{C} = \mathcal{G}\mathcal{C}$
- $(F \cdot \mathcal{C})\langle X, c\rangle = \langle FX, c\rangle$
- $\mathrm{p}_{\mathcal{C}} = \pi_{\mathcal{C}}$

Note that the relative slice category $\mathbf{Cat}/_{\mathcal{D}}\mathcal{B}$ is not equivalent to $\mathbf{ICat}(\mathcal{B})$. However, the two are equivalent on discrete $\mathcal{B}$-indexed categories.

Following [HP89], we define when a category with attributes $(\mathcal{C}, \mathcal{E})$ has universal and existential quantification along projections corresponding to another category with attributes $(\mathcal{C}, \mathcal{D})$ (possibly the same).

Definition 6.11 *Given a category with attributes $(\mathcal{C}, \mathcal{D}, G, \mathrm{p})$, we say that another category with attributes $(\mathcal{C}, \mathcal{E}, G, \mathrm{p})$ has*

- **∃-quantifiers** $\exists[X]a.b \in \mathcal{E}[X]$ *with unit* $\eta[X]_{a,b}$ *for* $X \in \mathcal{C}$, $a \in \mathcal{D}[X]$ *and* $b \in \mathcal{E}[X \cdot a]$

$$\begin{tikzcd}
X \cdot a \cdot b \arrow[r, "\eta"] \arrow[dr, "\mathrm{p}"'] & X \cdot a \cdot \mathrm{p}^*\exists \arrow[r, "\mathrm{p} \cdot \exists"] \arrow[d, "\mathrm{p}"] & X \cdot \exists \arrow[d, "\mathrm{p}"] \\
 & X \cdot a \arrow[r, "\mathrm{p}"'] & X
\end{tikzcd}$$

where $\exists \triangleq \exists[X]a.b$ *and* $\eta \triangleq \eta[X]_{a,b} \overset{\Delta}{\iff}$

  – *for every* $f \colon Y \to X$

$$\begin{tikzcd}
 & X \cdot a \cdot b \arrow[r, "\eta"] & X \cdot a \cdot \mathrm{p}^*\exists \arrow[r, "\mathrm{p} \cdot \exists"] & X \cdot \exists \arrow[d, "\mathrm{p}"] \\
Y \cdot a' \cdot b' \arrow[ur, "f \cdot a \cdot b"] \arrow[r, dashed, "\eta'"] \arrow[dr, "\mathrm{p}"'] & Y \cdot a' \cdot \mathrm{p}^*\exists' \arrow[ur, "f \cdot a \cdot \mathrm{p}^*\exists"] \arrow[r, "\mathrm{p} \cdot \exists'"] \arrow[d, "\mathrm{p}"] & Y \cdot \exists' \arrow[ur, "f \cdot \exists"] \arrow[d, "\mathrm{p}"] & X \\
 & Y \cdot a' \arrow[r, "\mathrm{p}"'] & Y \arrow[ur, "f"'] &
\end{tikzcd}$$

where $a' \triangleq f^*a$, $b' \triangleq (f \cdot a)^*b$, $\exists' \triangleq \exists[Y]a'.b' = f^*\exists$ *and* $\eta' \triangleq \eta[Y]_{a',b'}$

  – *there is a natural isomorphism* $\mathcal{C}/_{\mathcal{E}}X(\exists, \_) \stackrel{\cdot}{\cong} \mathcal{C}/_{\mathcal{E}}(X \cdot a)(b, \mathrm{p}_a^*\_)$ *given by*

$$\begin{tikzcd}
X \cdot \exists \arrow[r, "g"] \arrow[dr, "\mathrm{p}"'] & X \cdot c \arrow[d, "\mathrm{p}"] \\
 & X
\end{tikzcd}
\quad g \overset{\sim}{\mapsto} h \triangleq \eta; (\mathrm{p}_a^* g) \quad
\begin{tikzcd}
X \cdot a \cdot b \arrow[r, "h"] \arrow[dr, "\mathrm{p}"'] & X \cdot a \cdot \mathrm{p}^* c \arrow[d, "\mathrm{p}"] \\
 & X \cdot a
\end{tikzcd}$$

– *the morphism* $s \triangleq \eta; (\mathrm{p}\cdot\exists)\colon X\cdot a\cdot b \to X\cdot\exists$ *is* **orthogonal** *to the set* $\{\mathrm{p}_c | Y \in \mathcal{C}\ ,\ c \in \mathcal{E}[Y]\}$*, i.e. for all* $Y \in \mathcal{C}$ *and* $c \in \mathcal{E}[Y]$

$$\text{if}\quad \begin{tikzcd} \cdot \arrow[r, "s"] \arrow[d, "g"'] & \cdot \arrow[d, "f"] \\ \cdot \arrow[r, "\mathrm{p}_c"'] & \cdot \end{tikzcd} \quad \text{then } \exists! h \text{ s.t.} \quad \begin{tikzcd} \cdot \arrow[r, "s"] \arrow[d, "g"'] & \cdot \arrow[d, "f"] \arrow[dl, "h"'] \\ \cdot \arrow[r, "\mathrm{p}_c"'] & \cdot \end{tikzcd}$$

- **∀-quantifiers** $\forall[X]a.b \in \mathcal{E}[X]$ *with counit* $\epsilon[X]_{a,b}$ *for* $X \in \mathcal{C}\ ,\ a \in \mathcal{D}[X]$ *and* $b \in \mathcal{E}[X\cdot a]$

$$\begin{tikzcd} X\cdot a\cdot b \arrow[dr, "\mathrm{p}"'] & X\cdot\forall\cdot \mathrm{p}^*a \arrow[l, "\epsilon"'] \arrow[r, "\mathrm{p}"] \arrow[d, "\mathrm{p}\cdot a"] & X\cdot\forall \arrow[d, "\mathrm{p}"] \\ & X\cdot a \arrow[r, "\mathrm{p}"'] & X \end{tikzcd}$$

where $\forall \triangleq \forall[X]a.b$ *and* $\epsilon \triangleq \epsilon[X]_{a,b} \overset{\Delta}{\iff}$

they satisfy, mutatis mutandis, the requirements for products in Definition 6.6

Remark 6.12 The definition of ∃-quantifier is essentially equivalent to that in [HP89], but our definition of ∀-quantifiers is stronger in the same way as our definition of product is (see Remark 6.7).

The unit η for ∃-quantifiers, the morphism s used in the orthogonality condition and the counit ϵ for ∀-quantifier have a simple description as *context realisation* in a theory of dependent types:

- η is the realisation $\langle x, a, (a, b)\rangle$ of the context $[x\colon X, a\colon A, y\colon (\exists a\colon A.B)]$ in the context $[x\colon X, a\colon A, b\colon B]$;
- s is the realisation $\langle x, (a, b)\rangle$ of the context $[x\colon X, y\colon (\exists a\colon A.B)]$ in the context $[x\colon X, a\colon A, b\colon B]$;
- ϵ is the realisation $\langle x, a, f(a)\rangle$ of the context $[x\colon X, a\colon A, b\colon B]$ in the context $[x\colon X, f\colon (\forall a\colon A.B), a\colon A]$.

The orthogonality condition means simply that there is a one-one correspondence between terms of type C (classified by $\mathcal{E}$) in the context $[x\colon X, y\colon (\exists a\colon A.B)]$ and terms of type $[(a, b)/y]C$ in the context $[x\colon X, a\colon A, b\colon B]$. The need for the orthogonality condition in the definition of ∃-quantifiers was realised by M. Hyland and A. Pitts. In special cases orthogonality follows from the other two conditions:

- When $\mathcal{D} = \mathcal{E}$ and $\mathcal{E}$ has sums. In fact, $(\Sigma[X]a.b)$ and $(\exists[X]a.b)$ are isomorphic, so s is an iso, and isos are orthogonal to any class of morphisms.

- When $\mathcal{E}$ is constant, i.e. when types do not depend on values like in $F\omega$. In this case orthogonality follows from the natural isomorphism of the second condition.

There is an asymmetry between the definition of $\exists$- and $\forall$-quantifiers, since in the former the universal property is given in terms of relative slice categories, while in the latter it is given in terms of slice categories. We have chosen to formulate the universal property for $\forall$-quantifiers in terms of slice categories because of the *absoluteness* result for dependent products established in 2.8 of [HP89]. When $\mathcal{D} = \mathcal{E}$, it is obvious that the definitions of products and $\forall$-quantifiers for $\mathcal{E}$ coincide, while sums and $\exists$-quantifiers can both be defined and different.

For categories with attributes the set-theoretic notion of inclusion between classes of display maps has to be replaced by a more complex one.

Definition 6.13 (Embedding)
Given two categories with attributes $(\mathcal{C}, \mathcal{D}, G, \mathrm{p})$ *and* $(\mathcal{C}, \mathcal{E}, G, \mathrm{p})$*, an* **embedding** *of the first into the second is a pair* (In, in)*, where* $In\colon \mathcal{D} \to \mathcal{E}$ *is a* $\mathcal{C}$*-indexed functor (between discrete indexed categories) and* $in\colon G \dot\to (\mathcal{G}(In); G)\colon \mathcal{GD} \to \mathcal{C}$ *is a natural isomorphism s.t. for* $X \in \mathcal{C}$ *and* $a \in \mathcal{D}[X]$ *the following diagram in* $\mathcal{C}$ *commutes*

$$\begin{tikzcd} X \cdot a \arrow[r, "in_{X,a}"] \arrow[d, "\mathrm{p}_a"'] & X \cdot In(a) \arrow[dl, "\mathrm{p}_{In(a)}"] \\ X & \end{tikzcd}$$

Embeddings preserve (not necessarily on the nose) units, sums, products and $\forall$-quantifiers, but may not preserve $\exists$-quantifiers.

Definition 6.14 *Let* $(\mathcal{C}, \mathcal{D}, G, \mathrm{p})$ *and* $(\mathcal{C}, \mathcal{E}, G, \mathrm{p})$ *be two categories with attributes,* **parallel composition** $(\mathcal{C}, \mathcal{D}\|\mathcal{E}, G, \mathrm{p})$ *and* **juxtaposition** $(\mathcal{C}, \mathcal{D} \cdot \mathcal{E}, G, \mathrm{p})$ *are the categories with attributes defined as follows:*

- $(\mathcal{D}\|\mathcal{E})[X] = (\mathcal{D}[X] \times \mathcal{E}[X])$
  $X \cdot \langle a, b\rangle = X \cdot a \cdot \mathrm{p}_a^* b$
  $f \cdot \langle a, b\rangle = f \cdot a \cdot \mathrm{p}_a^* b$
  $\mathrm{p}_{\langle a,b\rangle} = \mathrm{p}_{\mathrm{p}_a^* b}; \mathrm{p}_a$, *i.e. the diagonal filling of the pullback square*

$$\begin{tikzcd} X \cdot a \cdot \mathrm{p}_a^* b \arrow[r, "\mathrm{p}_a \cdot b"] \arrow[d, "\mathrm{p}_{\mathrm{p}_a^* b}"'] \arrow[dr, "\mathrm{p}_{\langle a,b\rangle}"] & X \cdot b \arrow[d, "\mathrm{p}_b"] \\ X \cdot a \arrow[r, "\mathrm{p}_a"'] & X \end{tikzcd}$$

- $(\mathcal{D} \cdot \mathcal{E})[X] = (\Sigma a \in \mathcal{D}[X].\mathcal{E}[X \cdot a])$
  $X \cdot \langle a, b\rangle = X \cdot a \cdot b$
  $f \cdot \langle a, b\rangle = f \cdot a \cdot b$
  $\mathrm{p}_{\langle a,b\rangle} = \mathrm{p}_b; \mathrm{p}_a$

Remark 6.15 The intended meaning of the operations described above is:

- context extension in $\mathcal{D}\|\mathcal{E}$ means context extension by $\mathcal{D}$ and $\mathcal{E}$ *in parallel*,
- context extension in $\mathcal{D} \cdot \mathcal{E}$ means context extension by $\mathcal{D}$ *and then* by $\mathcal{E}$.

These operation can also be viewed as binary functors on the category $\mathbf{Attr}(\mathcal{C})$ of categories with attributes over $\mathcal{C}$ and embedding. From this perspective the binary functors corresponding to parallel composition and juxtaposition are part of a monoidal structure over $\mathbf{Attr}(\mathcal{C})$ with unit 1, where $1[X] = \{*\}$ (see Example 6.8). Moreover, parallel composition is symmetric, while juxtaposition is not. Note also that a category with attributes having units and sums corresponds to a monoid $units\colon 1 \to \mathcal{D} \leftarrow \mathcal{D} \cdot \mathcal{D}\colon sums$ in the monoidal category $\mathbf{Attr}(\mathcal{C})$ with juxtaposition as tensor product. The operations of parallel composition and juxtaposition can be viewed also as functors on the poset $\mathbf{Disp}(\mathcal{C})$ of classes of display maps over $\mathcal{C}$ ordered by inclusion, and it is easier to look at them in these terms. There are other operations on classes of display maps worth mentioning:

- $\_ \times \_$ and $\_ + \_$ (i.e. product and coproduct in $\mathbf{Disp}(\mathcal{C})$).

We summarise the categorical semantics of the Calculus of Constructions given in [HP89] using the terminology introduced in this section.

Definition 6.16 *A model of* CC *is specified by a category $\mathcal{B}$ with a terminal object* 1 *and two structures $\mathcal{R}$ and $\mathcal{A}$ of category with attributes on $\mathcal{B}$ s.t.*

- *$\mathcal{R}$ is embedded in $\mathcal{A}$;*
- *$\mathcal{R}$ has a generic type (U, t) s.t. $U = 1 \cdot a$ for some $a \in \mathcal{A}[1]$;*
- *$\mathcal{R}$ and $\mathcal{A}$ have units, sums and products;*
- *$\mathcal{R}$ has $\exists$- and $\forall$-quantifiers along context projections in $\mathcal{A}$.*

Remark 6.17 $\mathcal{R}$ and $\mathcal{A}$ correspond to context extensions by a type and a kind respectively. The embedding of $\mathcal{R}$ into $\mathcal{A}$ means that types are included in kinds, while $U = 1 \cdot a$ for some $a \in \mathcal{A}[1]$ means that there is a kind of all types.

The definition above is almost equivalent to that in Summary 2.13 of [HP89]. We have only dropped the requirement that every $X \in \mathcal{B}$ is (up to isomorphism) of the form $1 \cdot a$ for some $a \in \mathcal{A}[1]$. In this way it is left open what can be declared in a context besides variables ranging over a kind (or a type).

7 Independence and HML-modules

In this section we analyse the structure over the category $\mathcal{GC}$ of HML-modules (see Definition 5.4) necessary for the interpretation of a calculus with dependent types.

- First, we consider two categories with attributes over $\mathcal{GC}$, $\mathcal{D}$ and $\mathcal{E}$, corresponding to dependent kinds and dependent type schemas.
- Second, we define *independence* for categories with attributes (see Definition 7.3) and prove that $\mathcal{D}$ is independent from $\mathcal{E}$.
- Finally, we prove various technical lemmas on independence leading to Theorem 7.9, which infer properties of $\mathcal{D}\cdot\mathcal{E}$, corresponding (we claim) to dependent signatures, from similar properties of $\mathcal{D}$ and $\mathcal{E}$.

In analogy with the categorical semantics of the Calculus of Constructions (see Definition 6.16) we define two categories with attributes over the category of HML-modules, $\mathcal{D}$ and $\mathcal{E}$, corresponding to context extension by a kind and a type schema.

Definition 7.1 *The category $\mathcal{GC}$ is equipped with two structures $\mathcal{D}$ and $\mathcal{E}$ of category with attributes, defined as follows*

- $\mathcal{D}[\langle X,c\rangle]$ *is the set of equivalence classes* $[\mathrm{k}']_{\mathrm{k}}$, *i.e.* $\{\mathrm{k}''|v{:}\,\mathrm{k}\vdash \mathrm{k}'=\mathrm{k}''\}$
  $\langle f,g\rangle^{*}d=[[u/v]\mathrm{k}']_{k_1}$
  $\langle X,c\rangle\cdot d=[v{:}\,(\Sigma v{:}\,\mathrm{k}.\mathrm{k}');\hat{\sigma}]$
  $\langle f,g\rangle\cdot d=[(v{:}\,(\Sigma v{:}\,\mathrm{k}_1.[u/v]\mathrm{k}');x{:}\,\hat{\sigma_1}).\langle\langle\hat{u},\pi_2 v\rangle;\hat{e}\rangle]_{(\Sigma v:\mathrm{k}.\mathrm{k}');\hat{\sigma}}$
  $\mathrm{p}_d=[(v{:}\,(\Sigma v{:}\,\mathrm{k}.\mathrm{k}');x{:}\,\hat{\sigma}).\langle\pi_1 v;x\rangle]_{\mathrm{k};\sigma}$
  where $\langle X,c\rangle=[v{:}\,\mathrm{k};\sigma]\in\mathcal{GC}$, $d=[\mathrm{k}']_{\mathrm{k}}\in\mathcal{D}[\langle X,c\rangle]$
  $\langle f,g\rangle=[(v{:}\,\mathrm{k}_1;x{:}\,\sigma_1).\langle u;e\rangle]_{\mathrm{k};\sigma}$ *morphism from* $\langle X_1,c_1\rangle$ *to* $\langle X,c\rangle$
  $\hat{\_}$ *is a shorthand for* $[\pi_1 v/v]\_$

- $\mathcal{E}[\langle X,c\rangle]$ *is the set of equivalence classes* $[\sigma']_{\mathrm{k};\sigma}$, *i.e.* $\{\sigma''|v{:}\,\mathrm{k};x{:}\,\sigma\vdash\sigma'=\sigma''\}$
  $\langle f,g\rangle^{*}e=[[u,e/v,x]\sigma']_{k_1;\sigma_1}$
  $\langle X,c\rangle\cdot e=[v{:}\,\mathrm{k};(\Sigma x{:}\,\sigma.\sigma')]$
  $\langle f,g\rangle\cdot e=[(v{:}\,\mathrm{k}_1;x{:}\,(\Sigma x{:}\,\sigma_1.[u,e/v,x]\sigma')).\langle u;\langle[\pi_1 x/x]e,\pi_2 x\rangle\rangle]_{\mathrm{k};(\Sigma x:\sigma.\sigma')}$
  $\mathrm{p}_e=[(v{:}\,\mathrm{k};x{:}\,(\Sigma x{:}\,\sigma.\sigma')).\langle v;\pi_1 x\rangle]_{\mathrm{k};\sigma}$
  where $\langle X,c\rangle=[v{:}\,\mathrm{k};\sigma]\in\mathcal{GC}$, $e=[\sigma']_{\mathrm{k};\sigma}\in\mathcal{E}[\langle X,c\rangle]$
  $\langle f,g\rangle=[(v{:}\,\mathrm{k}_1;x{:}\,\sigma_1).\langle u;e\rangle]_{\mathrm{k};\sigma}$ *morphism from* $\langle X_1,c_1\rangle$ *to* $\langle X,c\rangle$

Proposition 7.2 *the category $\mathcal{GC}$ has a terminal object* 1 *and the two structures $\mathcal{E}$ and $\mathcal{D}$ of category with attributes on $\mathcal{GC}$ are s.t.*

- *$\mathcal{E}$ and $\mathcal{D}$ have units, sums and products;*
- *$\mathcal{E}$ has $\exists$- and $\forall$-quantifiers along context projections in $\mathcal{D}$.*

Proof The definitions of terminal object, units, sums, products and quantifiers are quite obvious, therefore they will be only sketched. The required properties can be reformulated as equations and proved using the inference rules for HML.

- Given $X = [v\colon \mathrm{k}; \sigma] \in \mathcal{GC}$, $d_1 = [\mathrm{k}_1]_{\mathrm{k}} \in \mathcal{D}[X]$ and $d_2 = [\mathrm{k}_2]_{\Sigma v:\mathrm{k}.\mathrm{k}_1} \in \mathcal{D}[X \cdot d_1]$, the unit in $\mathcal{D}[X]$ is $[1]_{\mathrm{k}}$, the sum and product of d_2 indexed over d_1 are

  $(\Sigma[X]d_1.d_2) \triangleq [\Sigma v_1\colon \mathrm{k}_1.[\langle v, v_1\rangle/v]\mathrm{k}_2]_{\mathrm{k}}$

  $(\Pi[X]d_1.d_2) \triangleq [\Pi v_1\colon \mathrm{k}_1.[\langle v, v_1\rangle/v]\mathrm{k}_2]_{\mathrm{k}}$

- Given $X = [v\colon \mathrm{k}; \sigma] \in \mathcal{GC}$, $e_1 = [\sigma_1]_{\mathrm{k};\sigma} \in \mathcal{E}[X]$ and $e_2 = [\sigma_2]_{\mathrm{k};\Sigma x:\sigma.\sigma_1} \in \mathcal{E}[X \cdot e_1]$, the unit in $\mathcal{E}[X]$ is $[1]_{\mathrm{k};\sigma}$, the sum and product of e_2 indexed over e_1 are

  $(\Sigma[X]e_1.e_2) \triangleq [\Sigma x_1\colon \sigma_1.[\langle x, x_1\rangle/x]\sigma_2]_{\mathrm{k};\sigma}$

  $(\Pi[X]e_1.e_2) \triangleq [\Pi x_1\colon \sigma_1.[\langle x, x_1\rangle/x]\sigma_2]_{\mathrm{k};\sigma}$

- Given $X = [v\colon \mathrm{k}; \sigma] \in \mathcal{GC}$, $d_1 = [\mathrm{k}_1]_{\mathrm{k}} \in \mathcal{D}[X]$ and $e_2 = [\sigma_2]_{(\Sigma v:\mathrm{k}.\mathrm{k}_1);\hat{\sigma}} \in \mathcal{E}[X \cdot d_1]$, the $\exists$- and $\forall$-quantifier of e_2 along p_{d_1} are

  $(\exists[X]d_1.e_2) \triangleq [\exists v_1\colon \mathrm{k}_1.[\langle v, v_1\rangle/v]\sigma_2]_{\mathrm{k};\sigma}$

  $(\forall[X]d_1.e_2) \triangleq [\forall v_1\colon \mathrm{k}_1.[\langle v, v_1\rangle/v]\sigma_2]_{\mathrm{k};\sigma}$

∎

The structures $\mathcal{D}$ and $\mathcal{E}$ fail to give a model of the Calculus of Construction for two reasons: $\mathcal{E}$ is not included in $\mathcal{D}$, and $\mathcal{E}$ doesn't have a generic type (U, t) s.t. $U = 1 \cdot d$ for some $d \in \mathcal{D}[1]$. The second reason could be circumvented by using $F\omega$ instead of HML, since in $F\omega$ types and type schemas are identified. However, the first reason is strongly related to independence of constructor-expressions from term-expressions, which is a feature of $F\omega$, too. In HML independence in enforced syntactically, by forbidding terms in kinds and constructors (see Section 5). Before studying any further the category of HML-modules, we characterise independence at a more abstract level.

Definition 7.3 (Independence) *Given two structures $\mathcal{D}$ and $\mathcal{E}$ of category with attributes over a category $\mathcal{C}$, we say that $\mathcal{D}$ is* **independent** *from $\mathcal{E}$ iff for every $X \in \mathcal{C}$, $e \in \mathcal{E}[X]$ and $d \in \mathcal{D}[X]$*

- *the mapping $\mathcal{D}[\mathrm{p}_e]\colon \mathcal{D}[X] \to \mathcal{D}[X \cdot e]$ is a bijection*
- *there is a natural isomorphism $\mathcal{C}/X(\_, \mathrm{p}_d) \dot{\cong} \mathcal{C}/(X \cdot e)(\_ \cdot e, \mathrm{p}_d \cdot e)$ given by*

$$\begin{tikzcd} Y \arrow[r, "g"] \arrow[dr, "f"'] & X \cdot d \arrow[d, "\mathrm{p}"] \\ & X \end{tikzcd} \quad g \overset{\sim}{\mapsto} h \triangleq (g \cdot e) \quad \begin{tikzcd} Y \cdot f^*e \arrow[r, "h"] \arrow[dr, "f \cdot e"'] & X \cdot d \cdot \mathrm{p}^*e \arrow[r, "\sim"] \arrow[d, "\mathrm{p} \cdot e"] & X \cdot e \cdot \mathrm{p}^*d \arrow[dl, "\mathrm{p}"] \\ & X \cdot e & \end{tikzcd}$$

Remark 7.4 Since $\mathrm{p}_d \cdot e$ and $\mathrm{p}_{\mathrm{p}_e^* d}$ are isomorphic in $\mathcal{C}/(X \cdot e)$ and context projections can be pulled back along any morphism, then the second condition amounts to saying that there is a bijection between sections of p_d and section of $\mathrm{p}_{\mathrm{p}_e^* d}$. In summary independence of $\mathcal{D}$ from $\mathcal{E}$ means that types and terms *classified* by $\mathcal{D}$ are invariant w.r.t. context extensions by types *classified* by $\mathcal{E}$.

Theorem 7.5 (Independence for HML**)** *$\mathcal{D}$ is independent from $\mathcal{E}$, where $\mathcal{D}$ and $\mathcal{E}$ are the categories with attributes on $\mathcal{GC}$ given in Definition 7.1.*

Proof Indeed, we prove that $\mathcal{D}$ is independent from $\mathcal{E}$ whenever $\mathcal{D}$ is *induced* by a category with attributes over the base $\mathcal{B}$ and p_e is of the form $\langle \mathrm{id}_X, g\rangle$ for every $X \in \mathcal{B}$ and $e \in \mathcal{E}[X]$.

It is obvious from the definition that $\mathcal{D}[\langle X, c\rangle]$ depends only from X and $\mathcal{D}[\langle f, g\rangle]$ depends only from f. Since p_e is of the form $\langle \mathrm{id}_X, g\rangle$ and $\mathcal{D}[\langle \mathrm{id}_X, \mathrm{id}_c\rangle]$ is the identity, then $\mathcal{D}[\mathrm{p}_e]$ must be the identity and this amounts to the first requirement for independence.

The natural isomorphism $\mathcal{GC}/\langle X, c\rangle(\_, \mathrm{p}_d) \dot{\cong} \mathcal{GC}/(\langle X, c\rangle \cdot e)(\_ \cdot e, \mathrm{p}_d \cdot e)$, demanded in the second requirement for independence, is a consequence of the following facts:

- p_d is $\langle f, \mathrm{id}_{f^* c}\rangle$ for some f and c
- $\langle f, \mathrm{id}_{f^* c}\rangle \cdot e$ is $\langle f, \mathrm{id}_{f^* c'}\rangle$ for some c'
- $\mathcal{GC}/\langle X, c\rangle(\langle f', g'\rangle, \langle f, \mathrm{id}_{f^* c}\rangle) \dot{\cong} \mathcal{B}/X(f', f)$, as a morphism in the first hom-set must be of the form $\langle h, g'\rangle$ for some h s.t. $f' = h; f$.

∎

In the sequel we establish some basic facts about independence.

Lemma 7.6 *If $\mathcal{D}$ is independent from $\mathcal{E}$, then p_e is orthogonal to p_d for every $X, Y \in \mathcal{C}$, $d \in \mathcal{D}[X]$ and $e \in \mathcal{E}[Y]$, i.e.*

$$\textit{if}\quad \begin{tikzcd} \cdot \arrow[r, "\mathrm{p}_e"] \arrow[d, "g"'] & \cdot \arrow[d, "f"] \\ \cdot \arrow[r, "\mathrm{p}_d"'] & \cdot \end{tikzcd} \quad \textit{then } \exists! h \textit{ s.t.} \quad \begin{tikzcd} \cdot \arrow[r, "\mathrm{p}_e"] \arrow[d, "g"'] & \cdot \arrow[d, "f"] \arrow[dl, "h"] \\ \cdot \arrow[r, "\mathrm{p}_d"'] & \cdot \end{tikzcd}$$

Proof Because of Remark 7.4 we can assume that $X = Y$, $f = \mathrm{id}_X$ and show that

$$\text{if}\quad \begin{tikzcd} \cdot \arrow[r, "\mathrm{p}_e"] \arrow[d, "g"'] & \cdot \\ \cdot \arrow[ur, "\mathrm{p}_d"'] & \end{tikzcd} \quad \text{then } \exists! h \text{ section of } \mathrm{p}_d \text{ s.t.} \quad \begin{tikzcd} \cdot \arrow[r, "\mathrm{p}_e"] \arrow[d, "g"'] & \cdot \arrow[dl, "h"] \\ \cdot & \end{tikzcd}$$

which amounts to a bijection between $g \in \mathcal{C}/X(\mathrm{p}_e, \mathrm{p}_d)$ and $h \in \mathcal{C}/X(\mathrm{id}_X, \mathrm{p}_d)$ given by $g = \mathrm{p}_e; h$. Such a bijection can be given in two steps:

- the bijection between $g \in \mathcal{C}/X(\mathrm{p}_e, \mathrm{p}_d)$ and $h' \in \mathcal{C}/X(\mathrm{id}_{X \cdot e}, \mathrm{p}_d \cdot e)$ given by $g = h'; \mathrm{p}_{\mathrm{p}_d^* e}$, since

$$\begin{tikzcd}
\cdot \arrow[r, "\mathrm{p}_d \cdot e"] \arrow[d, "\mathrm{p}_{\mathrm{p}_d^* e}"'] \arrow[dr, phantom, "\lrcorner", very near start] & \cdot \arrow[d, "\mathrm{p}_e"] \\
\cdot \arrow[r, "\mathrm{p}_d"'] & \cdot
\end{tikzcd}$$

- the bijection between $h' \in \mathcal{C}/X(\mathrm{id}_{X \cdot e}, \mathrm{p}_d \cdot e)$ and $h \in \mathcal{C}/X(\mathrm{id}_X, \mathrm{p}_d)$ given by the second requirement in the definition of independence, as $\mathrm{id}_{X \cdot e} = \mathrm{id}_X \cdot e$.

We skip the check that composition of these two bijections is the desired one ∎

Lemma 7.7 *If $\mathcal{D}$ is independent from $\mathcal{E}$, then for every $X \in \mathcal{C}$, $e_1 \in \mathcal{E}[X]$ and $d_2 \in \mathcal{D}[X \cdot e_1]$ there exists $d_1 \in \mathcal{D}[X]$ and $e_2 \in \mathcal{E}[X \cdot d_1]$ s.t.*

$$\begin{tikzcd}
\cdot \arrow[r, "\sim"] \arrow[d, "\mathrm{p}_{e_2}"'] \arrow[dr, phantom, "\lrcorner", very near start] & \cdot \arrow[r, "\mathrm{p}_{d_2}"] & \cdot \arrow[d, "\mathrm{p}_{e_1}"] \\
\cdot \arrow[rr, "\mathrm{p}_{d_1}"'] & & \cdot
\end{tikzcd}$$

Proof Because of the first condition in the definition of independence, there exists unique $d_1 \in \mathcal{D}[X]$ s.t. $d_2 = \mathcal{D}[\mathrm{p}_{e_1}]d_1$. Let $e_2 \in \mathcal{E}[X \cdot d_1]$ be $\mathcal{E}[\mathrm{p}_{d_1}]e_1$. To show that d_1 and e_2 satisfy the requirement, use the fact that the following squares are pullbacks for the same pair of morphisms

$$\begin{tikzcd}
\cdot \arrow[r, "\mathrm{p}_{d_1} \cdot e_1"] \arrow[d, "\mathrm{p}_{e_2}"'] & \cdot \arrow[d, "\mathrm{p}_{e_1}"] \\
\cdot \arrow[r, "\mathrm{p}_{d_1}"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "\mathrm{p}_{d_2}"] \arrow[d, "\mathrm{p}_{e_1} \cdot d_1"'] & \cdot \arrow[d, "\mathrm{p}_{e_1}"] \\
\cdot \arrow[r, "\mathrm{p}_{d_1}"'] & \cdot
\end{tikzcd}$$

∎

Remark 7.8 The proof of the Lemma essentially says that the two categories with attributes $\mathcal{E} \cdot \mathcal{D}$ and $\mathcal{E} \| \mathcal{D}$, as given in Definition 6.14, are *equivalent*. Since independence implies also that context projections for $\mathcal{E}$ are orthogonal to those for $\mathcal{D}$, then the factorisation of $\mathrm{p}_{d_2}; \mathrm{p}_{e_1}$ given by the Lemma is unique (up to isomorphism). Another consequence of independence is that $\mathcal{D}$ has quantifiers along context projections for $\mathcal{E}$, but they are *uninteresting*. In fact, $(\exists[X]e_1.d_2) = (\forall[X]e_1.d_2) = d_1$, where d_1 is that given by the Lemma.

Finally, we derive properties of the juxtaposition $\mathcal{D} \cdot \mathcal{E}$ (see Definition 6.14) under the assumption that $\mathcal{D}$ is independent from $\mathcal{E}$.

Theorem 7.9 *Given two categories with attributes $\mathcal{D}$ and $\mathcal{E}$ s.t.*

- *$\mathcal{D}$ is independent from $\mathcal{E}$;*
- *$\mathcal{E}$ and $\mathcal{D}$ have units, sums and products;*
- *$\mathcal{E}$ has $\exists$- and $\forall$-quantifiers along context projections in $\mathcal{D}$;*

then the juxtaposition $\mathcal{A} = \mathcal{D} \cdot \mathcal{E}$ satisfies the following properties

- *$\mathcal{E}$ has $\exists$- and $\forall$-quantifiers along context projections in $\mathcal{A}$;*
- *$\mathcal{A}$ has units, sums and products.*

Proof We give only a sketch, and write a_i for an element $\langle d_i, e_i \rangle \in \mathcal{A}[\_]$.

- Given $X \in \mathcal{C}$, $a_1 \in \mathcal{A}[X]$ and $e_2 \in \mathcal{E}[X \cdot a_1]$, then
  the $\exists$- and $\forall$-quantifier of e_2 along p_{a_1} are
  $(\exists[X]a_1.e_2) \triangleq \exists[X]d_1.(\Sigma[X \cdot d_1]e_1.e_2)$ and
  $(\forall[X]a_1.e_2) \triangleq \forall[X]d_1.(\Pi[X \cdot d_1]e_1.e_2)$
- Given $X \in \mathcal{C}$, $a_1 \in \mathcal{A}[X]$ and $a_2 \in \mathcal{A}[X \cdot a_1]$, then
  the unit in $\mathcal{A}[X]$ is $1[X] \triangleq \langle 1[X], 1[X \cdot 1[X]] \rangle$
  the sum and product of a_2 indexed over a_1 are
  $(\Sigma[X]a_1.a_2) \triangleq \langle (\Sigma[X]d_1.d_3), (s^{-1})^*(\Sigma[X \cdot d_1 \cdot d_3]e_3.e_2) \rangle$
  $(\Pi[X]a_1.a_2) \triangleq \langle d, \forall[X \cdot d]\mathrm{p}_d^* d_1.\epsilon^*(\Pi[X \cdot d_1 \cdot d_3]e_3.e_2) \rangle$
  where $d_3 \in \mathcal{D}[X \cdot d_1]$ and $e_3 \in \mathcal{E}[X \cdot d_1 \cdot d_3]$ are s.t. (see Lemma 7.7)[1]

$$\begin{tikzcd} \cdot \arrow[r, "\mathrm{p}_{d_2}"] \arrow[d, "\mathrm{p}_{e_3}"'] \arrow[dr, phantom, "\lrcorner", very near start] & \cdot \arrow[d, "\mathrm{p}_{e_1}"] \\ \cdot \arrow[r, "\mathrm{p}_{d_3}"'] & \cdot \end{tikzcd}$$

  $s\colon X \cdot d_1 \cdot d_3 \xrightarrow{\sim} X \cdot \Sigma[X]d_1.d_3$ is the isomorphism for sums (see Definition 6.6)
  $d \in \mathcal{D}[X]$ is the product $(\Pi[X]d_1.d_3)$, and
  $\epsilon\colon X \cdot d \cdot \mathrm{p}_d^* d_1 \to X \cdot d_1 \cdot d_3$ is the counit for products (see Definition 6.6).

∎

[1] For simplicity, we that the isomorphism in the top-left corner is the identity.

The importance of Theorem 7.9 rests on the observation that, when $\mathcal{D}$ and $\mathcal{E}$ are the categories with attributes given in Definition 7.1, then $\mathcal{A} = \mathcal{D} \cdot \mathcal{E}$ is the category with attributes corresponding to **context extension by a signature**. This claim is justified by looking at a more concrete definition of $\mathcal{A}$ and by completing the correspondence between ML-modules and the category $\mathcal{GC}$ given in Remark 5.5.

Definition 7.10 *The category $\mathcal{GC}$ is equipped with a structure $\mathcal{A}$ of category with attributes, defined as follows*

- $\mathcal{A}[\langle X, c\rangle]$ *is the set of equivalence classes* $[v'{:}\,\mathrm{k}'; \sigma']_{\mathrm{k};\sigma}$, *i.e.* $\{\langle \mathrm{k}'', \sigma''\rangle | v{:}\,\mathrm{k} \vdash \mathrm{k}' = \mathrm{k}''$ *and* $v{:}\,\mathrm{k}, v'{:}\,\mathrm{k}'; x{:}\,\sigma \vdash \sigma' = \sigma''\}$

  $\langle f, g\rangle^* a = [v'{:}\,[u/v]\mathrm{k}'; [u, e/v, x]\sigma']_{k_1;\sigma_1}$

  $\langle X, c\rangle \cdot a = [v{:}\,(\Sigma v{:}\,\mathrm{k}.\mathrm{k}'); (\Sigma x{:}\,\hat{\sigma}.[\pi_2 v/v']\hat{\sigma}')]$

  $\langle f, g\rangle \cdot a = [(v{:}\,(\Sigma v{:}\,\mathrm{k}_1.[u/v]\mathrm{k}'); x{:}\,(\Sigma x{:}\,\hat{\sigma_1}.[\hat{u}, \pi_2 v, \hat{e}/v, v', x]\sigma')).$
  $\langle\langle \hat{u}, \pi_2 v\rangle; \langle[\pi_1 x/x]\hat{e}, \pi_2(x)\rangle\rangle]_{(\Sigma v:\mathrm{k}.\mathrm{k}');(\Sigma x:\hat{\sigma}.[\pi_2 v/v']\hat{\sigma}')}$

  $\mathrm{p}_a = [(v{:}\,(\Sigma v{:}\,\mathrm{k}.\mathrm{k}'); x{:}\,(\Sigma x{:}\,\hat{\sigma}.[\pi_2 v/v']\hat{\sigma}')).\langle \pi_1 v; \pi_1 x\rangle]_{\mathrm{k},\sigma}$

  where $\langle X, c\rangle = [v{:}\,\mathrm{k}; \sigma] \in \mathcal{GC}$, $a = [v'{:}\,\mathrm{k}'; \sigma']_{\mathrm{k};\sigma} \in \mathcal{A}[\langle X, c\rangle]$

  $\langle f, g\rangle = [(v{:}\,\mathrm{k}_1; x{:}\,\sigma_1).\langle u; e\rangle]_{\mathrm{k};\sigma}$ *morphism from* $\langle X_1, c_1\rangle$ *to* $\langle X, c\rangle$

  $\hat{\_}$ *is a shorthand for* $[\pi_1 v/v]\_$

Remark 7.11 There is a bijection between $\mathcal{A}[\langle X, c\rangle]$ and $(\mathcal{D} \cdot \mathcal{E})[\langle X, c\rangle]$, namely

$a = [v'{:}\,\mathrm{k}'; \sigma']_{\mathrm{k};\sigma} \in \mathcal{A}[\langle X, c\rangle]$ corresponds to the pair $\langle d, e\rangle$, where $d = [\mathrm{k}']_{\mathrm{k}} \in \mathcal{D}[\langle X, c\rangle]$ and $e = [[\pi_2 v/v']\hat{\sigma}']_{(\Sigma v:\mathrm{k}.\mathrm{k}');\hat{\sigma}} \in \mathcal{E}[\langle X, c\rangle \cdot d]$.

The notation for $a \in \mathcal{A}[\langle X, c\rangle]$ is suggestive of ML-signatures. To make the correspondence with ML-modules easier to express we introduce a more suggestive notation also for sections.

Given $\langle X, c\rangle = [v{:}\,\mathrm{k}; \sigma] \in \mathcal{GC}$ and $a = [v'{:}\,\mathrm{k}'; \sigma']_{\mathrm{k};\sigma} \in \mathcal{A}[\langle X, c\rangle]$, we write $[(v{:}\,\mathrm{k}; x{:}\,\sigma).\langle v_i = u; x_i = e\rangle]_{\mathrm{k}';[v_i/v']\sigma'}$ for the section

$$[(v{:}\,\mathrm{k}; x{:}\,\sigma).\langle\langle v, u\rangle; \langle x, e\rangle\rangle]_{(\Sigma v:\mathrm{k}.\mathrm{k}');(\Sigma x:\hat{\sigma}.[\pi_2 v/v_i]\hat{\sigma}')}$$

of p_a, where v_i and x_i can be choosen arbitrarily and $\hat{\_}$ stands for $[\pi_1 v/v]\_$.

Indeed every section of p_a can be written in this way.

Using the structure $\mathcal{A}$ over $\mathcal{GC}$ we can revise the correspondence given in Remark 5.5 to account for type dependency at the level of ML-modules. In ML structures and signatures must always be considered relatively to a *context* Γ for constructor and value variables, specifying kind and type of all (relevant) free variables. A context Γ can be thought as a closed signature, and therefore it corresponds to an object $\langle X, c\rangle = [v{:}\,\mathrm{k}; \sigma]$ of $\mathcal{GC}$ (see Remark 5.5).

- An ML-signature corresponds to an element of $\mathcal{A}[\langle X, c\rangle]$, e.g.

  signature sig = (**sig** type v'; val $x'{:}\,\sigma'$ **end**)

  corresponds to the element $a = [v'{:}\,\Omega; \sigma']_{\mathrm{k};\sigma}$.

- An ML-structure of signature *sig* corresponds to a section, which given a realisation for Γ extends it to a realisation for Γ extended with *sig*, e.g.

$$\textbf{structure } S = (\textbf{struct} \text{ type } v' = u; \text{ val } x' = e \textbf{ end})$$

corresponds to the section $[(v\colon \mathrm{k}; x\colon \sigma).\langle v' = u; x' = e\rangle]_{\Omega;\sigma'}$ of p_a.

- An ML-functor, with parameter signature sig_1 and result signature sig_2 (possibly depending on sig_1), corresponds to a section of $\mathrm{p}_{(\Pi[\langle X,c\rangle]a_1.a_2)}$, where a_i corresponds to sig_i, e.g.

$$\begin{array}{l}\textbf{functor } F(S\colon sig_1)\colon sig_2 = \\ \quad \textbf{struct} \text{ type } v' = [S.v'/v']u; \text{ val } x' = [S.v', S.x'/v', x']e \textbf{ end}\end{array}$$

corresponds to the section

$$[(v\colon \mathrm{k}; x\colon \sigma).\langle F = (\lambda v'\colon \Omega.u); G = (\lambda v'\colon \Omega.\lambda x'\colon \sigma_1.e)\rangle]_{(\Pi v'\colon\Omega.\Omega);(\Pi v'\colon T.\Pi x'\colon\sigma_1.[Fv'/v']\sigma_2)}$$

8 Conclusion and further research

In this paper we have investigated program modules in relation to independence of type-expressions from program-expressions (which had been overlooked in previous accounts) and type dependency. In our investigation we have abstracted, as far as possible, from the syntax and tried to work at a great level of generality. In fact, our understanding of program modules applies to any programming language which can be viewed as an indexed category (possibly with some additional structure). The main advantages of this approach are its language-independence and the ability to reformulate unclear questions, like "when does a language *support* higher order modules?", in terms of simple and precise concepts, namely "is the Grothendieck construction $\mathcal{GC}$ a cartesian closed category?".

In Remarks 5.5 and 7.11 we briefly outlined how ML-modules fit in the categorical account of program modules. ML, like other programming language, has many other aspects that we do not address here, and some of them have no satisfactory theoretical account, yet. Our analysis is not just an exposition of program modules for theoretically minded people, instead we expect that it will have a feedback on programming languages. Good module facilities are essential for programming in the large, and there seems to be a lot of space for improvement in this area. Bridging the gap between theory and practice could be rather difficult, since one needs to address also syntactic and pragmatic issues. However, [HMM90] has already made a step in this direction, by looking at calculi for program modules *consistently* with the category-theoretic account given in this paper.

Finally, we mention some related areas of research (see also the introduction):

- **Sharing constraints**. Sharing constraints specify that two structure identifiers denote the *same* structure. They were proposed by Dave MacQueen and are incorporated in Standard ML (see [MTH90]). There is already a clean understanding of sharing constraints in terms of *names* and *generativity*, which is used in the definition of Standard ML.

In our opinion there should be a more general explanation of sharing constraints based on a calculus for dependent types (as in Martin-Löf Type Theory) and computations (as in [Mogar]), which would remove some of the current limitations, e.g. only structure identifiers can be used in sharing constraints. In such a calculus one would expect that the subtypes $\{[a]\}$ and $T\{a\}$ of TA (where $a \in A$) are different. The first type is the singleton containing only the computation $[a]$ (which does not do anything except returning the value a). The second type is the set of computations which can do whatever they like, but at the end they can only return the value a. The latter alternative seems a more appropriate to account for sharing constraints.

- **Modular approach**. In this paper we have focused our attention on one aspect of programming languages. However, in the introduction we stressed the need for combining features, and how a 2-categorical setting could help. We do not believe in a *mechanical* way of finding the right combining of features, a trial and error methodology is more likely. The main contribution we expect from a 2-categorical view of programming languages is a small set of *strategies* to guide in such a search.

- **Partial evaluation**. If indexed categories capture independence of a class of expressions from another, perhaps they ought to capture evaluation of *constant* expressions at compile-time, as done by optimising compilers. More precisely, expressions evaluated at compile-time should be morphisms in the base category. Therefore for every object N in the fiber over 1 we have to introduce an object N' in the base, which *classifies* the expressions of type N computable at compile-time. The inclusion of N' into N can be achieved by having an element $c: 1 \to N$ in the fiber over N', corresponding to the *generic expression* of type N computable at compile-time. We have not investigated whether this way of looking at partial evaluation has any useful applications.

- **Categorical semantics of dependent types**. This paper has introduced some operations on categories with attributes and classes of display maps (see Definition 6.2 and Remark 6.15) and the concept of independence (see Definition 7.3), that were not present in the literature.

  Though these concepts were motivated by a specific application, we believe that they deserve further study, e.g. it is not clear whether there are analogues of Theorem 7.9 for the other operations on categories with attributes, and could be useful in analysing and comparing type theories. For instance, Theorem 7.9 essentially says that any model of $F\omega$ induces a model of CC; this seems related to Berardi-Mohring's translation from CC to $F\omega$ (see [Ber89, Moh89]).

Acknowledgements

All my thanks to Dave MacQueen, who introduced me to ML-modules; Rod Burstall, Luca Cardelli, Bob Harper, Tony Hoare, John Mitchell, Pierre-Louis Curien, Martin Hyland, Andy Pitts (and other members of the CLICS project) for useful discussions; Pierre-Louis Curien, Eike Ritter and the referees gave valuable comments on previous drafts. I used Paul Taylor's package for commutative diagrams.

Appendix: HML inference rules

We write $[e_1, \ldots, e_n/x_1, \ldots, x_n]e$ for the *parallel substitution* in e of all variables $x_1, \ldots, x_n$ by the expressions $e_1, \ldots, e_n$.

Given a context Γ we write $\mathrm{DV}(\Gamma)$ for the set of variables *declared* in Γ and, if x is a variable in $\mathrm{DV}(\Gamma)$, then we write $\Gamma(x)$ for the (unique) kind or type schema assigned to x in Γ.

Compile-time inference rules

Constructor context formation rules $\Delta \vdash$

$$\emptyset \quad \frac{}{\emptyset \vdash}$$

$$v \quad \frac{\Delta \vdash \mathrm{k}}{\Delta, v{:}\, \mathrm{k} \vdash} \quad v \notin \mathrm{DV}(\Delta)$$

Kind formation rules $\Delta \vdash \mathrm{k}$

$$\Omega \quad \frac{\Delta \vdash}{\Delta \vdash \Omega}$$

$$1 \quad \frac{\Delta \vdash}{\Delta \vdash 1}$$

$$\Sigma \quad \frac{\Delta \vdash \mathrm{k}_1 \qquad \Delta, v{:}\, \mathrm{k}_1 \vdash \mathrm{k}_2}{\Delta \vdash (\Sigma v{:}\, \mathrm{k}_1.\mathrm{k}_2)}$$

$$\Pi \quad \frac{\Delta \vdash \mathrm{k}_1 \qquad \Delta, v{:}\, \mathrm{k}_1 \vdash \mathrm{k}_2}{\Delta \vdash (\Pi v{:}\, \mathrm{k}_1.\mathrm{k}_2)}$$

Kind equality rules $\Delta \vdash \mathrm{k}_1 = \mathrm{k}_2$

The type equality rules of the predicative theory of dependent types (see [HP89]).

Constructor formation rules $\Delta \vdash u\colon \mathrm{k}$

$$v \quad \frac{\Delta \vdash}{\Delta \vdash v\colon \mathrm{k}} \quad \mathrm{k} = \Delta(v)$$

$$\text{unit} \quad \frac{\Delta \vdash}{\Delta \vdash 1\colon \Omega}$$

$$\text{prod} \quad \frac{\Delta \vdash}{\Delta \vdash \times\colon \Omega \to \Omega \to \Omega}$$

$$\text{fun} \quad \frac{\Delta \vdash}{\Delta \vdash \to\colon \Omega \to \Omega \to \Omega}$$

$$1\text{I} \quad \frac{\Delta \vdash}{\Delta \vdash *\colon 1}$$

$$\Sigma\text{I} \quad \frac{\Delta, v\colon \mathrm{k}_1 \vdash \mathrm{k}_2 \qquad \Delta \vdash u_1\colon \mathrm{k}_1 \qquad \Delta \vdash u_2\colon [u_1/v]\mathrm{k}_2}{\Delta \vdash \langle u_1, u_2\rangle\colon (\Sigma v\colon \mathrm{k}_1.\mathrm{k}_2)}$$

$$\Sigma\text{E.1} \quad \frac{\Delta \vdash u\colon (\Sigma v\colon \mathrm{k}_1.\mathrm{k}_2)}{\Delta \vdash \pi_1(u)\colon \mathrm{k}_1}$$

$$\Sigma\text{E.2} \quad \frac{\Delta \vdash u\colon (\Sigma v\colon \mathrm{k}_1.\mathrm{k}_2)}{\Delta \vdash \pi_2(u)\colon [\pi_1(u)/v]\mathrm{k}_2}$$

$$\Pi\text{I} \quad \frac{\Delta, v\colon \mathrm{k}_1 \vdash u\colon \mathrm{k}_2}{\Delta \vdash (\lambda v\colon \mathrm{k}_1.u)\colon (\Pi v\colon \mathrm{k}_1.\mathrm{k}_2)}$$

$$\Pi\text{E} \quad \frac{\Delta \vdash u\colon (\Pi v\colon \mathrm{k}_1.\mathrm{k}_2) \qquad \Delta \vdash u_1\colon \mathrm{k}_1}{\Delta \vdash u(u_1)\colon [u_1/v]\mathrm{k}_2}$$

$$:\text{-eq} \quad \frac{\Delta \vdash u\colon \mathrm{k}_1 \qquad \Delta \vdash \mathrm{k}_1 = \mathrm{k}_2}{\Delta \vdash u\colon \mathrm{k}_2}$$

Constructor equality rules $\Delta \vdash u_1 = u_2\colon \mathrm{k}$

The term equality rules of the predicative theory of dependent types (see [HP89]).

Run-time inference rules

Term context formation rules $\Delta; \Gamma \vdash$

$$\emptyset \quad \frac{\Delta \vdash}{\Delta; \emptyset \vdash}$$

$$x \quad \frac{\Delta; \Gamma \vdash \sigma}{\Delta; \Gamma, x\colon \sigma \vdash} \quad x \notin \mathrm{DV}(\Delta, \Gamma)$$

Type schema formation rules $\Delta;\Gamma\vdash\sigma$

$$\text{type}\quad\frac{\Delta;\Gamma\vdash\qquad\Delta\vdash\tau\colon\Omega}{\Delta;\Gamma\vdash\mathrm{set}(\tau)}$$

$$1\quad\frac{\Delta;\Gamma\vdash}{\Delta;\Gamma\vdash 1}$$

$$\Sigma\quad\frac{\Delta;\Gamma\vdash\sigma_1\qquad\Delta;\Gamma,x\colon\sigma_1\vdash\sigma_2}{\Delta;\Gamma\vdash(\Sigma x\colon\sigma_1.\sigma_2)}$$

$$\Pi\quad\frac{\Delta;\Gamma\vdash\sigma_1\qquad\Delta;\Gamma,x\colon\sigma_1\vdash\sigma_2}{\Delta;\Gamma\vdash(\Pi x\colon\sigma_1.\sigma_2)}$$

$$\forall\quad\frac{\Delta;\Gamma\vdash\qquad\Delta,v\colon\mathrm{k};\Gamma\vdash\sigma}{\Delta;\Gamma\vdash(\forall v\colon\mathrm{k}.\sigma)}$$

$$\exists\quad\frac{\Delta;\Gamma\vdash\qquad\Delta,v\colon\mathrm{k};\Gamma\vdash\sigma}{\Delta;\Gamma\vdash(\exists v\colon\mathrm{k}.\sigma)}$$

Type schema equality rules $\Delta;\Gamma\vdash\sigma_1=\sigma_2$

Similar to the type equality rule for the Calculus of Constructions (see [HP89]), plus the following rules saying that set(_) commutes with products and function spaces:

$$1.=\quad\frac{\Delta;\Gamma\vdash}{\Delta;\Gamma\vdash\mathrm{set}(1)=1}$$

$$\times.=\quad\frac{\Delta;\Gamma\vdash\qquad\Delta\vdash\tau_1\colon\Omega\qquad\Delta\vdash\tau_2\colon\Omega}{\Delta;\Gamma\vdash\mathrm{set}(\tau_1\times\tau_2)=(\Sigma x\colon\mathrm{set}(\tau_1).\mathrm{set}(\tau_2))}$$

$$\to.=\quad\frac{\Delta;\Gamma\vdash\qquad\Delta\vdash\tau_1\colon\Omega\qquad\Delta\vdash\tau_2\colon\Omega}{\Delta;\Gamma\vdash\mathrm{set}(\tau_1\to\tau_2)=(\Pi x\colon\mathrm{set}(\tau_1).\mathrm{set}(\tau_2))}$$

Term formation rules $\Delta;\Gamma\vdash e\colon\sigma$

$$x\quad\frac{\Delta;\Gamma\vdash}{\Delta;\Gamma\vdash x\colon\sigma}\quad\sigma=\Gamma(x)$$

$$1\mathrm{I}\quad\frac{\Delta;\Gamma\vdash}{\Delta;\Gamma\vdash *\colon 1}$$

$$\Sigma\mathrm{I}\quad\frac{\Delta;\Gamma,x\colon\sigma_1\vdash\sigma_2\qquad\Delta;\Gamma\vdash e_1\colon\sigma_1\qquad\Delta;\Gamma\vdash e_2\colon[e_1/x]\sigma_2}{\Delta;\Gamma\vdash\langle e_1,e_2\rangle\colon(\Sigma x\colon\sigma_1.\sigma_2)}$$

$$\Sigma\mathrm{E}.1\quad\frac{\Delta;\Gamma\vdash e\colon(\Sigma x\colon\sigma_1.\sigma_2)}{\Delta;\Gamma\vdash\pi_1(e)\colon\sigma_1}$$

$$\Sigma\text{E.2} \quad \frac{\Delta; \Gamma \vdash e\colon (\Sigma x\colon \sigma_1.\sigma_2)}{\Delta; \Gamma \vdash \pi_2(e)\colon [\pi_1(e)/x]\sigma_2}$$

$$\Pi\text{I} \quad \frac{\Delta; \Gamma, x\colon \sigma_1 \vdash e\colon \sigma_2}{\Delta; \Gamma \vdash (\lambda x\colon \sigma_1.e)\colon (\Pi x\colon \sigma_1.\sigma_2)}$$

$$\Pi\text{E} \quad \frac{\Delta; \Gamma \vdash e\colon (\Pi x\colon \sigma_1.\sigma_2) \qquad \Delta; \Gamma \vdash e_1\colon \sigma_1}{\Delta; \Gamma \vdash e(e_1)\colon [e_1/x]\sigma_2}$$

$$\forall\text{I} \quad \frac{\Delta; \Gamma \vdash \qquad \Delta, v\colon \mathrm{k}; \Gamma \vdash e\colon \sigma}{\Delta; \Gamma \vdash (\Lambda v\colon \mathrm{k}.e)\colon (\forall v\colon \mathrm{k}.\sigma)}$$

$$\forall\text{E} \quad \frac{\Delta; \Gamma \vdash e\colon (\forall v\colon \mathrm{k}.\sigma) \qquad \Delta \vdash u\colon \mathrm{k}}{\Delta; \Gamma \vdash e(u)\colon [u/v]\sigma}$$

$$\exists\text{I} \quad \frac{\Delta, v\colon \mathrm{k}; \Gamma \vdash \sigma \qquad \Delta \vdash u\colon \mathrm{k} \qquad \Delta; \Gamma \vdash e\colon [u/v]\sigma}{\Delta; \Gamma \vdash (u, e)\colon (\exists v\colon \mathrm{k}.\sigma)}$$

$$\exists\text{E} \quad \frac{\begin{array}{l}\Delta; \Gamma \vdash e\colon (\exists v\colon \mathrm{k}.\sigma) \\ \Delta; \Gamma, z\colon (\exists v\colon \mathrm{k}.\sigma) \vdash \sigma' \qquad \Delta, v\colon \mathrm{k}; \Gamma, x\colon \sigma \vdash e'\colon [(v, x)/z]\sigma'\end{array}}{\Delta; \Gamma \vdash (\mathrm{let}\,(v, x){=}e \,\mathrm{in}\, e')\colon [e/z]\sigma'}$$

$$\text{:-eq} \quad \frac{\Delta; \Gamma \vdash e\colon \sigma_1 \qquad \Delta; \Gamma \vdash \sigma_1 = \sigma_2}{\Delta; \Gamma \vdash e\colon \sigma_2}$$

Term equality rules $\Delta; \Gamma \vdash e_1 = e_2\colon \sigma$

Similar to the type equality rule for the Calculus of Constructions (see [HP89]), namely the general rules for a congruence and the following $\beta\eta$-rules:

$$1.\eta \quad \frac{\Delta; \Gamma \vdash e\colon 1}{\Delta; \Gamma \vdash * = e\colon 1}$$

$$\Sigma.\beta.1 \quad \frac{\Delta; \Gamma, x\colon \sigma_1 \vdash \sigma_2 \qquad \Delta; \Gamma \vdash e_1\colon \sigma_1 \qquad \Delta; \Gamma \vdash e_2\colon [e_1/x]\sigma_2}{\Delta; \Gamma \vdash \pi_1(\langle e_1, e_2 \rangle) = e_1\colon \sigma_1}$$

$$\Sigma.\beta.2 \quad \frac{\Delta; \Gamma, x\colon \sigma_1 \vdash \sigma_2 \qquad \Delta; \Gamma \vdash e_1\colon \sigma_1 \qquad \Delta; \Gamma \vdash e_2\colon [e_1/x]\sigma_2}{\Delta; \Gamma \vdash \pi_2(\langle e_1, e_2 \rangle) = e_2\colon [e_1/x]\sigma_2}$$

$$\Sigma.\eta \quad \frac{\Delta; \Gamma \vdash e\colon (\Sigma x\colon \sigma_1.\sigma_2)}{\Delta; \Gamma \vdash \langle \pi_1(e), \pi_2(e) \rangle = e\colon (\Sigma x\colon \sigma_1.\sigma_2)}$$

$$\Pi.\beta \quad \frac{\Delta; \Gamma, x\colon \sigma_1 \vdash e_2\colon \sigma_2 \qquad \Delta; \Gamma \vdash e_1\colon \sigma_1}{\Delta; \Gamma \vdash (\lambda x\colon \sigma_1.e_2)(e_1) = [e_1/x]e_2\colon [e_1/x]\sigma_2}$$

$$\Pi.\eta \quad \frac{\Delta; \Gamma \vdash e\colon (\Pi x\colon \sigma_1.\sigma_2)}{\Delta; \Gamma \vdash (\lambda x\colon \sigma_1.e(x)) = e\colon (\Pi x\colon \sigma_1.\sigma_2)}$$

$$\forall.\beta \quad \frac{\Delta; \Gamma \vdash \qquad \Delta, v\colon \mathrm{k}; \Gamma \vdash e\colon \sigma \qquad \Delta \vdash u\colon \mathrm{k}}{\Delta; \Gamma \vdash (\Lambda v\colon \mathrm{k}.e)(u) = [u/v]e\colon [u/v]\sigma}$$

$$\forall.\eta \quad \frac{\Delta; \Gamma \vdash e\colon (\forall v\colon \mathrm{k}.\sigma)}{\Delta; \Gamma \vdash (\Lambda v\colon \mathrm{k}.e(v)) = e\colon (\forall v\colon \mathrm{k}.\sigma)}$$

$$\exists.\beta \quad \frac{\begin{array}{l}\Delta, v\colon \mathrm{k}; \Gamma \vdash \sigma \\ \Delta \vdash u\colon \mathrm{k} \qquad \Delta; \Gamma \vdash e\colon [u/v]\sigma \\ \Delta; \Gamma, z\colon (\exists v\colon \mathrm{k}.\sigma) \vdash \sigma' \qquad \Delta, v\colon \mathrm{k}; \Gamma, x\colon \sigma \vdash e'\colon [(v,x)/z]\sigma'\end{array}}{\Delta; \Gamma \vdash (\mathrm{let}\,(v,x){=}(u,e)\,\mathrm{in}\,e') = [u,e/v,x]e'\colon [(u,e)/z]\sigma'}$$

$$\exists.\eta \quad \frac{\Delta; \Gamma \vdash e\colon (\exists v\colon \mathrm{k}.\sigma) \qquad \Delta; \Gamma, z\colon (\exists v\colon \mathrm{k}.\sigma) \vdash e'\colon \sigma'}{\Delta; \Gamma \vdash (\mathrm{let}\,(v,x){=}e\,\mathrm{in}\,[(v,x)/z]e') = [e/z]e'\colon [e/z]\sigma'}$$

References

[Ben85] J. Benabou. Fibred categories and the foundation of naive category theory. *Journal of Symbolic Logic*, 50, 1985.

[Ber89] S. Berardi. *Type Dependency and Constructive Mathematics.* PhD thesis, Universitá di Torino, 1989.

[Car78] J. Cartmell. *Generalized Algebraic Theories and Contextual Categories.* PhD thesis, University of Oxford, 1978.

[Car88] L. Cardelli. Phase distinction in type theory. Draft 4/1/88, DEC SRC, 1988.

[CH88] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 73(2/3), 1988.

[Cur89] P.-L. Curien. Alpha-conversion, conditions on variables and categorical logic. *Studia Logica*, 3, 1989.

[Ehr88] T. Ehrhard. A categorical semantics of constructions. In *3rd LICS Conf.* IEEE, 1988.

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

[HMM86] R. Harper, D. MacQueen, and R. Milner. Standard ML. Technical Report ECS-LFCS-86-2, Edinburgh Univ., Dept. of Comp. Sci., 1986.

[HMM90] R. Harper, J. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *17th POPL*. ACM, 1990.

[HP89] J.M.E. Hyland and A.M. Pitts. The theory of constructions: Categorical semantics and topos-theoretic models. *Contemporary Mathematics*, 92, 1989.

[Jac90] B. Jacobs. Comprehension categories and the semantics of type dependency. June 90, Dept. of Computer Science, Univ. of Nijmegen, 1990.

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

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

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

[Mac85] D. MacQueen. Modules for standard ML. *Polymorphism*, 2, 1985.

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

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

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

[Mogar] E. Moggi. Notions of computations as monads. *Information and Computation*, to appear.

[Moh89] C. Mohring. Extracting Fω's programs from proofs in the calculus of constructions. In *16th POPL*. ACM, 1989.

[MP88] J.C. Mitchell and G.D. Plotkin. Abstract types have existential type. *ACM Trans. on Progr. Lang. and Sys.*, 10(3), 1988.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT press, 1990.

[Pit89] A.M. Pitts. Categorical semantics of dependent types. Talk given at SRI Menlo Park and at the Logic Colloquium in Berlin, 1989.

[Pitar] A.M. Pitts. Categorical logic. In Samson Abramsky, Dov M. Gabbay, and Tom S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume III*, chapter 3.10. Oxford University Press, to appear.

[PS78] R. Pare and D. Schumacher. Abstract families and the adjoint functor theorems. In P.T. Johnstone and R. Pare, editors, *Indexed Categories and their Applications*, volume 661 of *Lecture Notes in Mathematics*. Springer Verlag, 1978.

[See83] R.A.G. Seely. Hyperdoctrines, natural deduction and the Beck condition. *Zeitschr. f. math. Logik und Grundlagen d. Math.*, 29, 1983.

[See84] R.A.G. Seely. Locally cartesian closed categories and type theory. *Math. Proc. Camb. Phil. Soc.*, 95, 1984.

[See87] R.A.G. Seely. Categorical semantics for higher order polymorphic lambda calculus. *Journal of Symbolic Logic*, 52(2), 1987.

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

[Str73] R. Street. Fibrations and Yoneda's lemma in a 2-category. In *Category Seminar*, volume 420 of *Lecture Notes in Mathematics*. Springer Verlag, 1973.

[Str88] T. Streicher. *Correctness and Completeness of a Semantics of the Calculus of Constructions with respect to Interpretation in Doctrines of Constructions*. PhD thesis, University of Passau, 1988.

[Tay87] P. Taylor. *Recursive Domains, Indexed Category Theory and Polymorphism*. PhD thesis, University of Cambridge, 1987.