Dependent Types and Fibred Computational Effects

Citation for published version:
Ahman, D, Ghani, N & Plotkin, GD 2016, Dependent Types and Fibred Computational Effects. in Foundations of Software Science and Computation Structures: 19th International Conference, FOSSACS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9634, Springer Berlin Heidelberg, pp. 36-54, 19th International Conference of Foundations of Software Science and Computation Structures , Eindhoven, Netherlands, 2/04/16. DOI: 10.1007/978-3-662-49630-5_3

Digital Object Identifier (DOI):
10.1007/978-3-662-49630-5_3

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Foundations of Software Science and Computation Structures

Dependent Types and Fibred Computational Effects

Danel Ahman[1]* Neil Ghani[2] Gordon D. Plotkin[1]

[1] LFCS, University of Edinburgh, Scotland
[2] MSP Group, University of Strathclyde, Scotland

Abstract. We study the interplay between dependent types and general computational effects. We define a language with both value types and terms, and computation types and terms, where types depend only on value terms. We use computational Σ-types to account for type-dependency in the sequential composition of computations. Our language design is justified by a natural class of categorical models. We account for both algebraic and non-algebraic effects. We also show how to extend the language with general recursion, using continuous families of cpos.

1 Introduction

While dependent types have proven very useful on their own, for both programming and theorem proving, one also seeks a general way to combine them with computational effects, such as I/O, state, continuations, or recursion, so as to write more practical, concise, or clearer programs. However, despite the study of each of the two fields being well advanced, their combination presents difficulties, as recognised already by Moggi [25].

One puzzling problem is what to make of a type $A(M)$ if M can raise an effect. We do not know a general denotational semantics for such types, though there may be one (there is one for local names [28]). Pragmatically, the situation depends on the nature of the computational effects considered. For effects not requiring interaction with the program runtime, such as local names [28] or general recursion [7], one need not restrict M, as computing $A(M)$ then only depends on static information. However, this does not work well in general, as some effects, e.g., I/O, do crucially depend on interaction with the program runtime, and so then will the computation of $A(M)$. As one consequence, extending a coarse-grained language, e.g., Moggi's computational lambda calculus [24], with dependent types seems not to give a general solution, as the natural elimination rule for Π-types produces types of the form $A(M)$ (and cf. Levy [20, §12.4.1]).

A natural move to try to solve these problems is to allow types to depend only on value terms. This is the route we take in this paper and it does lead to a natural general semantics. While at first such a choice might seem limiting, we

* This work was funded by a University of Edinburgh PhD scholarship and by the scholarship program Kristjan Jaak, which is funded and managed by Archimedes Foundation in collaboration with the Estonian Ministry of Education and Research.

recover dependency on the statically available information about computational effects by inspecting the structure of thunked computations.

To ensure that types depend only on values, we make a clear distinction between values and computations, using separate classes of value types A and computation types $\underline{C}$, as in Call-By-Push-Value (CBPV) [20] and the Enriched Effect Calculus (EEC) [9]. Then the variables in types range only over value types, as desired. However, a further problem then arises: the usual typing rule

$$\frac{\Gamma \vdash M : FA \quad \Gamma, x\!:\!A \vdash N(x) : \underline{C}(x)}{\Gamma \vdash M \texttt{ to } x \texttt{ in } N(x) : \underline{C}(x)} \; (*)$$

for sequential composition of computations is not correct as x may occur freely in $\underline{C}(x)$ in the conclusion. An evident move here is to prohibit x from occurring freely in $\underline{C}$, as, e.g., advocated by Levy [20, §12.4.1] and Brady [5]. However, this seems too restrictive: to make the most of dependent types, it is desirable for the type of an effectful program to depend on values produced by preceding computations. As an example, consider combining monadic parsing [16] with dependent types and applying it to the parsing of well-typed syntax. Then it is natural to decompose the parsing of function applications into a parser for the function and a parser for the argument, where the type of the latter crucially depends on the domain of the type of the parsed function.

The approach we take is to keep the restricted form of (*), but to also introduce *computational Σ-types* $\Sigma x\!:\!A.\underline{C}(x)$, whose use is inspired by the algebraic treatment of computational effects [31]. To explain these, suppose computation types denote algebras for the algebraic theory of boolean-valued read-only store [35, §III.A]. Now let us consider the composite effectful program

$$\big((\texttt{return } 2) \,?\, (\texttt{return } 3)\big) \texttt{ to } x \texttt{ in } M(x)$$

where $x\!:\!\mathsf{Nat} \vdash M(x) : \underline{C}(x)$. Here we see that, after looking up the bit with ?, this program either evaluates as $M(2)$, which has type $\underline{C}(2)$, or as $M(3)$, which has type $\underline{C}(3)$. So the whole computation yields an element of the coproduct of algebras $\underline{C}(2) + \underline{C}(3)$. This pattern also recurs with other computational effects, and dependency on value types other than natural numbers: hence the computational Σ-types. These types enable us to type sequential composition by "closing-off" the free variable x in $\underline{C}(x)$: using the hypothesis of (*) we derive

$$\Gamma, x\!:\!A \vdash \langle x, N(x)\rangle : \Sigma x\!:\!A.\underline{C}(x)$$

using the introduction rule for computational Σ-types, and then derive

$$\Gamma \vdash M \texttt{ to } x \texttt{ in } \langle x, N(x)\rangle : \Sigma x\!:\!A.\underline{C}(x)$$

using the restricted form of (*).

We aim to put these ideas together at a *foundational* level, comparable with the levels at which the two fields have been studied separately. In contrast, the existing work on combining dependent types with first-class computational effects has concerned only *particular* kinds of effects (e.g., [7, 26, 28, 37]). Other authors have used dependent types in existing languages to *represent* effects using DSLs (e.g., [5, 13, 23]). There has, however, been no work combining dependent types with *general, first class* computational effects.

In Section 2, we define a small dependently-typed language with computational effects, combining features from Martin-Löf type theory (MLTT) [22] and computational languages separating values and computations, such as CBPV or EEC. The language design we propose is justified in Section 3 in terms of a sound and complete interpretation in a class of categorical models naturally combining i) comprehension categories arising from the semantics of dependent types; and ii) adjunctions arising from the semantics of computational effects. In Section 4, we extend the language and its semantics with algebraic effects. In Section 5, we extend the language and its semantics with general recursion.

2 A dependently-typed effectful language

We now define the syntax and equational theory of our dependently-typed effectful language, making a clear distinction between values and computations, at both type and term levels: the value fragment of our language is based on MLTT; the computational fragment is greatly influenced by CBPV and EEC.

Variables. We assume two countable sets: of *value variables*, ranged over by $x, y, \ldots$; and of *computation variables*, ranged over by $z, \ldots$. The former are treated as in MLTT: they are intuitionistic, and enjoy structural rules of weakening and contraction. The latter, on the other hand, are treated linearly, as in EEC, and play a crucial role in the elimination rule for computational Σ-types.

Types. Our *value types* $A, B, \ldots$ are given as in MLTT, except for the type $U\underline{C}$ of thunks, familiar from CBPV. To keep the presentation simple and focussed on the computational fragment of our language, we omit value sum types and general inductive types, both of which are easily added. Our *computation types* $\underline{C}, \underline{D}, \ldots$ generalise those of CBPV and EEC. The grammar of types is

$$A ::= \mathsf{Nat} \mid 1 \mid \Sigma x{:}A.B \mid \Pi x{:}A.B \mid \mathsf{Id}_A(V,W) \mid U\underline{C} \qquad \underline{C} ::= FA \mid \Sigma x{:}A.\underline{C} \mid \Pi x{:}A.\underline{C}$$

Here, FA is the type of computations returning values of type A. The computational Σ- and Π-types can be viewed as the natural dependently-typed generalisations of EEC's computational tensor and function types: $!A \otimes \underline{C}$, $A \to \underline{C}$. Rules defining well-formed value contexts $\vdash \Gamma$, value types $\Gamma \vdash A$ and computation types $\Gamma \vdash \underline{C}$ are given in Figure 1.

Terms. We let $V, W, \ldots$ to range over *value terms*. These are given as in MLTT, except for thunked computations $\mathtt{thunk}\, M$, familiar from CBPV. Compared to CBPV, we include both value functions and complex values to accommodate pure programs on which the types of our language could depend. In order to define effectful programs, we further distinguish between *computation terms*, ranged over by $M, N, \ldots$ and *homomorphism terms*, ranged over by $K, L, \ldots$. The three classes of terms are given by the following grammar:

$$\begin{aligned}
V &::= x \mid \mathsf{zero} \mid \mathsf{succ}\, V \mid \mathsf{rec}_{(x).A}(V_z, (y_1, y_2).V_s, W) \mid \star \mid \lambda x{:}A.V \mid V W \mid \langle V, W \rangle \mid \\
&\quad\;\; \mathtt{fst}\, V \mid \mathtt{snd}\, V \mid \mathtt{refl}\; V \mid \mathtt{ind}_A((x_1, x_2, x_3).B, (y).W, V_1, V_2, V_p) \mid \mathtt{thunk}\; M \\
M &::= \mathtt{return}\; V \mid M \;\mathtt{to}\; x \;\mathtt{in}\; N \mid \mathtt{force}\; V \mid \lambda x{:}A.M \mid M V \mid \langle V, M \rangle \mid M \;\mathtt{to}\; \langle x, z \rangle \;\mathtt{in}\; K \\
K &::= z \mid K \;\mathtt{to}\; x \;\mathtt{in}\; M \mid \lambda x{:}A.K \mid K V \mid \langle V, K \rangle \mid K \;\mathtt{to}\; \langle x, z \rangle \;\mathtt{in}\; L
\end{aligned}$$

Well-formed value types

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Nat}} \qquad \frac{\vdash \Gamma}{\Gamma \vdash 1} \qquad \frac{\Gamma, x\!:\!A \vdash B}{\Gamma \vdash \Sigma x\!:\!A.B} \qquad \frac{\Gamma, x\!:\!A \vdash B}{\Gamma \vdash \Pi x\!:\!A.B} \qquad \frac{\Gamma \vdash V : A \quad \Gamma \vdash W : A}{\Gamma \vdash \mathsf{Id}_A(V, W)} \qquad \frac{\Gamma \vdash \underline{C}}{\Gamma \vdash U\underline{C}}$$

Well-formed value contexts

$$\frac{}{\vdash \cdot} \qquad \frac{\vdash \Gamma \quad x \notin \mathit{Vars}(\Gamma) \quad \Gamma \vdash A}{\vdash \Gamma, x\!:\!A}$$

Well-formed computation types

$$\frac{\Gamma \vdash A}{\Gamma \vdash FA} \qquad \frac{\Gamma, x\!:\!A \vdash \underline{C}}{\Gamma \vdash \Sigma x\!:\!A.\underline{C}} \qquad \frac{\Gamma, x\!:\!A \vdash \underline{C}}{\Gamma \vdash \Pi x\!:\!A.\underline{C}}$$

Fig. 1. Well-formed contexts and types.

Our *computation terms* share many similarities with those in CBPV, but additionally include the introduction and elimination rules for computational Σ-types, given by pairing $\langle V, M\rangle$ and pattern-matching $M \texttt{ to } \langle x, z\rangle \texttt{ in } K$, whose syntax is similar to that for computational tensor types $!A \otimes \underline{C}$ in EEC.

As with the linear terms of EEC, our *homomorphism terms* contain computation variables z used linearly. From an operational perspective, the typing rules for homomorphism terms ensure that a computation bound to z always happens first in a well-typed term containing it. So, when eliminating a pair $\langle V, M\rangle$, M always happens before the rest of K in the compound term $\langle V, M\rangle \texttt{ to } \langle x, z\rangle \texttt{ in } K$, thus preserving the intended left-to-right evaluation order. This use of computation variables ensures that homomorphism terms denote algebra homomorphisms in the examples based on Eilenberg-Moore algebras of monads, or on algebraic effects [31]: hence the name. Similar forms of linearity are also present in CBPV with stacks [20, §2.3.4]; indeed, homomorphism terms can be viewed as a programmer-friendly syntax for dependently-typed stack terms.

Well-typed value terms $\Gamma \vdash V : A$, well-typed computation terms $\Gamma \vdash M : \underline{C}$, and well-typed homomorphism terms $\Gamma \,|\, z\!:\!\underline{C} \vdash K : \underline{D}$ are given in Figure 2.

The linear use of computation variables is also reminiscent of recent work on combining dependent and linear types in languages with distinguished intuitionistic and linear fragments [19, 38]. That work is designed to capture the adjunction models of intuitionistic linear logic [3]; in contrast, our language is designed to capture the computational nature of certain adjunctions.

Equations. We equip our language with an equational theory, consisting of equations between well-formed types, written $\Gamma \vdash A = B$ and $\Gamma \vdash \underline{C} = \underline{D}$; and well-typed terms, written $\Gamma \vdash V = W : A$, $\Gamma \vdash M = N : \underline{C}$ and $\Gamma \,|\, z\!:\!\underline{C} \vdash K = L : \underline{D}$. The equations between types consist of reflexivity equations for Nat and 1, and congruence rules for all the other type formers. We omit the equations between value terms as they are standard from MLTT with natural numbers and intensional identity types [22]. The rules for equations between computation and homomorphism terms are given in Figure 3. We leave the well-typedness assumptions about the constituent terms implicit, and omit type conversion, equivalence, and congruence rules. Many of the equations in Figure 3 are familiar from EEC, modulo the dependent typing. Compared to other computational languages, such as [24], standard equations that may seem missing from the theory, such as associativity of sequential composition, are in fact derivable.

Some meta-theory. The substitution of value terms for value variables has a straightforward mutually recursive definition. We write $A[V/x]$ for the substitution of V for x in A. The substitution of computation and homomorphism

Type conversions for value and computation terms

$$\frac{\Gamma \vdash V : A \quad \Gamma \vdash A = B}{\Gamma \vdash V : B} \qquad \frac{\Gamma \vdash M : \underline{C} \quad \Gamma \vdash \underline{C} = \underline{D}}{\Gamma \vdash M : \underline{D}} \qquad \frac{\Gamma \mid z{:}\underline{C} \vdash K : \underline{D}_1 \quad \Gamma \vdash \underline{D}_1 = \underline{D}_2}{\Gamma \mid z{:}\underline{C} \vdash K : \underline{D}_2}$$

Well-typed value terms

$$\frac{\vdash \Gamma \quad x{:}A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash M : \underline{C}}{\Gamma \vdash \texttt{thunk}\ M : U\underline{C}} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \star : 1} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{zero} : \mathsf{Nat}} \qquad \frac{\Gamma \vdash V : \mathsf{Nat}}{\Gamma \vdash \mathsf{succ}\, V : \mathsf{Nat}}$$

$$\frac{\Gamma, x{:}\mathsf{Nat} \vdash A \quad \Gamma \vdash V_z : A[\mathsf{zero}/x] \quad \Gamma, y_1{:}\mathsf{Nat}, y_2{:}A[y_1/x] \vdash V_s : A[\mathsf{succ}\, y_1/x] \quad \Gamma \vdash W : \mathsf{Nat}}{\Gamma \vdash \mathsf{rec}_{(x).A}(V_z, (y_1, y_2).V_s, W) : A[V/x]}$$

$$\frac{\Gamma \vdash V : A \quad \Gamma, x{:}A \vdash B \quad \Gamma \vdash W : B[V/x]}{\Gamma \vdash \langle V, W \rangle : \Sigma x{:}A.B} \qquad \frac{\Gamma \vdash V : \Sigma x{:}A.B}{\Gamma \vdash \texttt{fst}\, V : A} \qquad \frac{\Gamma \vdash V : \Sigma x{:}A.B}{\Gamma \vdash \texttt{snd}\, V : B[\texttt{fst}\, V/x]}$$

$$\frac{\Gamma, x{:}A \vdash V : B}{\Gamma \vdash \lambda x{:}A.V : \Pi x{:}A.B} \qquad \frac{\Gamma \vdash V : \Pi x{:}A.B \quad \Gamma \vdash W : A}{\Gamma \vdash VW : B[W/x]} \qquad \frac{\Gamma \vdash V : A}{\Gamma \vdash \texttt{refl}\ V : \mathsf{Id}_A(V, V)}$$

$$\frac{\begin{array}{c}\Gamma, x_1{:}A, x_2{:}A, x_3{:}\mathsf{Id}_A(x_1, x_2) \vdash B \quad \Gamma, y{:}A \vdash W : B[y/x_1, y/x_2, \texttt{refl}\ y/x_3] \\ \Gamma \vdash V_1 : A \quad \Gamma \vdash V_2 : A \quad \Gamma \vdash V_p : \mathsf{Id}_A(V_1, V_2)\end{array}}{\Gamma \vdash \texttt{ind}_A((x_1, x_2, x_3).B, (y).W, V_1, V_2, V_p) : B[V_1/x_1, V_2/x_2, V_p/x_3]}$$

Well-typed computation terms

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \texttt{return}\ V : FA} \qquad \frac{\Gamma \vdash M : FA \quad \Gamma \vdash \underline{C} \quad \Gamma, x{:}A \vdash N : \underline{C}}{\Gamma \vdash M\ \texttt{to}\ x\ \texttt{in}\ N : \underline{C}} \qquad \frac{\Gamma, x{:}A \vdash M : \underline{C}}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.\underline{C}}$$

$$\frac{\Gamma \vdash M : \Pi x{:}A.\underline{C} \quad \Gamma \vdash V : A}{\Gamma \vdash MV : \underline{C}[V/x]} \qquad \frac{\Gamma \vdash V : A \quad \Gamma, x{:}A \vdash \underline{C} \quad \Gamma \vdash M : \underline{C}[V/x]}{\Gamma \vdash \langle V, M \rangle : \Sigma x{:}A.\underline{C}}$$

$$\frac{\Gamma \vdash M : \Sigma x{:}A.\underline{C} \quad \Gamma \vdash \underline{D} \quad \Gamma, x{:}A \mid z{:}\underline{C} \vdash K : \underline{D}}{\Gamma \vdash M\ \texttt{to}\ \langle x, z \rangle\ \texttt{in}\ K : \underline{D}} \qquad \frac{\Gamma \vdash V : U\underline{C}}{\Gamma \vdash \texttt{force}\ V : \underline{C}}$$

Well-typed homomorphism terms

$$\frac{\Gamma \vdash \underline{C}}{\Gamma \mid z{:}\underline{C} \vdash z : \underline{C}} \qquad \frac{\Gamma \mid z_1{:}\underline{C} \vdash K : \Sigma x{:}A.\underline{D}_1 \quad \Gamma \vdash \underline{D}_2 \quad \Gamma, x{:}A \mid z_2{:}\underline{D}_1 \vdash L : \underline{D}_2}{\Gamma \mid z_1{:}\underline{C} \vdash K\ \texttt{to}\ \langle x, z_2 \rangle\ \texttt{in}\ L : \underline{D}_2}$$

$$\frac{\Gamma \vdash V : A \quad \Gamma, x{:}A \vdash \underline{D} \quad \Gamma \mid z{:}\underline{C} \vdash K : \underline{D}[V/x]}{\Gamma \mid z{:}\underline{C} \vdash \langle V, K \rangle : \Sigma x{:}A.\underline{D}} \qquad \frac{\Gamma \vdash \underline{C} \quad \Gamma, x{:}A \mid z{:}\underline{C} \vdash K : \underline{D}}{\Gamma \mid z{:}\underline{C} \vdash \lambda x{:}A.K : \Pi x{:}A.\underline{D}}$$

$$\frac{\Gamma \mid z{:}\underline{C} \vdash K : FA \quad \Gamma \vdash \underline{D} \quad \Gamma, x{:}A \vdash M : \underline{D}}{\Gamma \mid z{:}\underline{C} \vdash K\ \texttt{to}\ x\ \texttt{in}\ M : \underline{D}} \qquad \frac{\Gamma \mid z{:}\underline{C} \vdash K : \Pi x{:}A.\underline{D} \quad \Gamma \vdash V : A}{\Gamma \mid z{:}\underline{C} \vdash KV : \underline{D}[V/x]}$$

Fig. 2. Well-typed terms.

terms for computation variables is also routine, recursing only in the sub-terms where linearly used computation variables can appear. We write $K[M/z]$ for the substitution of M for z in K. The following theorem is proved by induction on the derivations of the judgments of well-formed types, terms, and equations.

Theorem 1. *Weakening, exchange, and substitution rules for value variables are admissible for all judgments of our language, e.g.:*

$$\frac{\Gamma_1, \Gamma_2 \mid z{:}\underline{C} \vdash K : \underline{D} \quad x \notin \mathit{Vars}(\Gamma_1) \quad x \notin \mathit{Vars}(\Gamma_2) \quad \Gamma_1 \vdash A}{\Gamma_1, x{:}A, \Gamma_2 \mid z : \underline{C} \vdash K : \underline{D}}$$

$$\frac{\Gamma_1, x{:}A, y{:}B, \Gamma_2 \mid z : \underline{C} \vdash K : \underline{D} \quad \Gamma_1 \vdash B}{\Gamma_1, y{:}B, x{:}A, \Gamma_2 \mid z : \underline{C} \vdash K : \underline{D}} \qquad \frac{\Gamma_1, x{:}A, \Gamma_2 \mid z{:}\underline{C} \vdash K : \underline{D} \quad \Gamma_1 \vdash V : A}{\Gamma_1, \Gamma_2[V/x] \mid z{:}\underline{C}[V/x] \vdash K[V/x] : \underline{D}[V/x]}$$

In addition, further substitution rules for computation variables are admissible for judgments $\Gamma \mid z{:}\underline{C} \vdash K : \underline{D}$ and $\Gamma \mid z{:}\underline{C} \vdash K = L : \underline{D}$. These rules cover the substitution of computation or homomorphism terms for computation variables.

Equations involving thunking and forcing

$$\Gamma \vdash \texttt{thunk}\ (\texttt{force}\ V) = V : U\underline{C} \qquad \Gamma \vdash \texttt{force}\ (\texttt{thunk}\ M) = M : \underline{C}$$

Equations between well-typed computation terms

$$\Gamma \vdash \texttt{return}\ V\ \texttt{to}\ x\ \texttt{in}\ M = M[V/x] : \underline{C} \qquad \Gamma \vdash M\ \texttt{to}\ x\ \texttt{in}\ K[\texttt{return}\ x/z] = K[M/z] : \underline{C}$$
$$\Gamma \vdash \langle V, M\rangle\ \texttt{to}\ \langle x, z\rangle\ \texttt{in}\ K = K[V/x, M/z] : \underline{D} \qquad \Gamma \vdash (\lambda x\!:\!A.M)V = M[V/x] : \underline{C}[V/x]$$
$$\Gamma \vdash M\ \texttt{to}\ \langle x, z_2\rangle\ \texttt{in}\ K[\langle x, z_2\rangle/z_1] = K[M/z_1] : \underline{D} \qquad \Gamma \vdash M = \lambda x\!:\!A.(Mx) : \Pi x\!:\!A.\underline{C}$$

Equations between well-typed homomorphism terms

$$\Gamma \mid z_1\!:\!\underline{C} \vdash K\ \texttt{to}\ x\ \texttt{in}\ L[\texttt{return}\ x/z_2] = L[K/z_2] : \underline{D}$$
$$\Gamma \mid z_1\!:\!\underline{C} \vdash \langle V, K\rangle\ \texttt{to}\ \langle x, z_2\rangle\ \texttt{in}\ L = L[V/x, K/z_2] : \underline{D}_2 \qquad \Gamma \mid z_1\!:\!\underline{C} \vdash (\lambda x\!:\!A.K)V = K[V/x] : \underline{D}[V/x]$$
$$\Gamma \mid z_1\!:\!\underline{C} \vdash K\ \texttt{to}\ \langle x, z_3\rangle\ \texttt{in}\ L[\langle x, z_3\rangle/z_2] = L[K/z_2] : \underline{D}_2 \qquad \Gamma \mid z_1\!:\!\underline{C} \vdash K = \lambda x\!:\!A.(Kx) : \Pi x\!:\!A.\underline{D}$$

Fig. 3. Fragment of the equational theory.

3 Denotational semantics

The denotational semantics of our language is based on standard fibred category theory. To make our work more accessible, we recall some preliminaries of this theory and suggest [18] for more details. Fibred category theory provides a natural framework for developing the semantics of dependently-typed languages, where: i) functors model type-dependency; ii) split fibrations model substitution; and iii) closed comprehension categories model Σ- and Π-types. The ideas we develop can also be expressed in terms of other models of dependent types, such as categories with families, or categories with attributes [15, 27].

3.1 Fibred category theory preliminaries

Fibrations. Given a functor $p : \mathcal{E} \longrightarrow \mathcal{B}$, a morphism $g : A \longrightarrow B$ is called a *Cartesian lifting* of $f : X \longrightarrow Y$ if $p(g) = f$ and for all $i : C \longrightarrow B$ and $j : p(C) \longrightarrow X$, such that $p(i) = f \circ j$ in $\mathcal{B}$, there exists a unique $h : C \longrightarrow A$ over j such that $g \circ h = i$. The functor $p : \mathcal{E} \longrightarrow \mathcal{B}$ is called a *fibration* if for every B in $\mathcal{E}$ and $f : X \longrightarrow p(B)$ in $\mathcal{B}$ there exists a Cartesian lifting $g : A \longrightarrow B$ of f in $\mathcal{E}$. A morphism $f : A \longrightarrow B$ in $\mathcal{E}$ is called *vertical* if $p(A) = p(B) = X$ and $p(f) = \mathsf{id}_X$. For any X in $\mathcal{B}$, we write $\mathcal{E}_X$ for the *fibre over* X, i.e., for the subcategory of $\mathcal{E}$ consisting of objects over X and vertical morphisms. A fibration is called *cloven* if it comes with a choice of Cartesian liftings. We write $\overline{f}(B) : f^*(B) \longrightarrow B$ for the chosen Cartesian lifting of $f : X \longrightarrow p(B)$. In cloven fibrations, every $\mathcal{B}$-morphism $f : X \longrightarrow Y$ determines a *reindexing functor* $f^* : \mathcal{E}_Y \longrightarrow \mathcal{E}_X$, satisfying $(\mathsf{id}_X)^* \cong \mathsf{id}_{\mathcal{E}_X}$ and $(g \circ f)^* \cong f^* \circ g^*$. A cloven fibration is said to be *split* if these two isomorphisms are identities.

Given split fibrations $p : \mathcal{V} \longrightarrow \mathcal{B}$ and $q : \mathcal{C} \longrightarrow \mathcal{B}$, a *split fibred functor* $F : p \longrightarrow q$ is given by a functor $F : \mathcal{V} \longrightarrow \mathcal{C}$, such that $q \circ F = p$ and F preserves the chosen Cartesian morphisms on-the-nose. Given two split fibred functors $F, G : p \longrightarrow q$, a *split fibred natural transformation* $\alpha : F \Rightarrow G$ is given by a natural transformation $\alpha : F \Rightarrow G$, in which every component of α is vertical. A *split fibred adjunction* $F \dashv U : q \longrightarrow p$ is given by split fibred functors $F : p \longrightarrow q$ and $U : q \longrightarrow p$, together with split fibred natural transformations $\eta : \mathsf{id}_{\mathcal{V}} \longrightarrow U \circ F$ and $\varepsilon : F \circ U \longrightarrow \mathsf{id}_{\mathcal{C}}$, subject to the standard unit-counit laws for adjunctions.

Comprehension categories. A *(split) comprehension category with unit* is given by a (split) fibration $p : \mathcal{E} \longrightarrow \mathcal{B}$, together with a comprehension-admitting terminal object functor $1 : \mathcal{B} \longrightarrow \mathcal{E}$, i.e., 1 has a right adjoint $\{-\} : \mathcal{E} \longrightarrow \mathcal{B}$; it is said to be *full* when the functor $A \overset{\pi_{(-)}}{\mapsto} p(\varepsilon_A^{1 \dashv \{-\}}) : \mathcal{E} \longrightarrow \mathcal{B}^{\rightarrow}$ is full and faithful. For all A in $\mathcal{E}$, the $\mathcal{B}$-morphism $\pi_A : \{A\} \longrightarrow p(A)$ is called a *projection map*. The corresponding reindexing functor $\pi_A^* : \mathcal{E}_{p(A)} \longrightarrow \mathcal{E}_{\{A\}}$ is called the *weakening functor*. For every comprehension category with unit $p : \mathcal{E} \longrightarrow \mathcal{B}$, we have an isomorphism $\mathcal{E}_{p(A)}(1_{p(A)}, A) \;\cong\; \{g : p(A) \longrightarrow \{A\} \,|\, \pi_A \circ g = \mathsf{id}_{p(A)}\}$, for all A in $\mathcal{E}$. As a notational convention, we write $\mathsf{s}(f) : p(A) \longrightarrow \{A\}$ for the section corresponding to the global element $f : 1_{p(A)} \longrightarrow A$, given by $\{f\} \circ \eta_A^{1 \dashv \{-\}}$.

A comprehension category with unit $p : \mathcal{E} \longrightarrow \mathcal{B}$ is said to have *dependent products* (resp. *weak dependent sums*) when the weakening functors π_A^* have right adjoints Π_A (resp. left adjoints Σ_A), for all A in $\mathcal{E}$, satisfying the Beck-Chevalley condition: for all Cartesian morphisms $f : A \longrightarrow B$, the canonical natural transformation $(p(f))^* \circ \Pi_B \longrightarrow \Pi_A \circ \{f\}^*$ (resp. $\Sigma_A \circ \{f\}^* \longrightarrow (p(f))^* \circ \Sigma_B$) is an isomorphism. A comprehension category with unit $p : \mathcal{E} \longrightarrow \mathcal{B}$ is said to have *strong dependent sums* when it has weak dependent sums, s.t. for all B in $\mathcal{E}_{\{A\}}$, the morphism $\{\overline{\pi_A}(\Sigma_A B) \circ \eta_B^{\Sigma_A \dashv \pi_A^*}\} : \{B\} \longrightarrow \{\Sigma_A B\}$ is an isomorphism.

Split closed comprehension categories. In order to define fibred adjunction models in Section 3.2, we use a particularly well-behaved class of comprehension categories (from the perspective of interpreting type theory), namely, those that are split and closed. A *split closed comprehension category* (SCCompC) is a split full comprehension category with unit $p : \mathcal{E} \longrightarrow \mathcal{B}$, where the base category $\mathcal{B}$ has a terminal object; the fibred terminal objects are preserved on-the-nose by reindexing; and which has dependent products and strong dependent sums, for which the isomorphisms in the Beck-Chevalley conditions are identities.

Natural numbers. A SCCompC $p : \mathcal{E} \longrightarrow \mathcal{B}$ is said to support *weak natural numbers* if there exists an object $\mathbb{N}$ in $\mathcal{E}_1$ and vertical morphisms $\mathsf{zero} : 1_1 \longrightarrow \mathbb{N}$, $\mathsf{succ} : \mathbb{N} \longrightarrow \mathbb{N}$, s.t. for all X in $\mathcal{B}$, A in $\mathcal{E}_{\{!_X^*(\mathbb{N})\}}$, $h_z : 1_X \longrightarrow (\mathsf{s}(!_X^*(\mathsf{zero})))^*(A)$ in $\mathcal{E}_X$ and $h_s : 1_{\{A\}} \longrightarrow \pi_A^*(\{!_X^*(\mathsf{succ})\}^*(A))$ in $\mathcal{E}_{\{A\}}$, there exists $h : 1_{\{!_X^*(\mathbb{N})\}} \longrightarrow A$ in $\mathcal{E}_{\{!_X^*(\mathbb{N})\}}$, satisfying $(\mathsf{s}(!_X^*(\mathsf{zero})))^*(h) = h_z$ and $\pi_A^*(\{!_X^*(\mathsf{succ})\}^*(h)) = h_s$.

We present $\mathbb{N}$ axiomatically rather than using weak initial algebras since our language and its models do not assume coproducts. Moreover, discussing the semantics of inductive types and their fibred induction principles in full generality [10] would digress too much from our central theme.

Identity types. Following the axiomatic presentation given by Warren [39], a SCCompC $p : \mathcal{E} \longrightarrow \mathcal{B}$ is said to support *identity types*, if, for all A in $\mathcal{E}$, there exists an object Id_A in $\mathcal{E}_{\{\pi_A^*(A)\}}$, and $\mathtt{r}_A : 1_{\{A\}} \longrightarrow \delta_A^*(\mathsf{Id}_A)$ in $\mathcal{E}_{\{A\}}$, such that for all B in $\mathcal{E}_{\{\mathsf{Id}_A\}}$ and $f : 1_{\{A\}} \longrightarrow (\mathsf{s}(\mathtt{r}_A))^*(\{\overline{\delta_A}(\mathsf{Id}_A)\}^*(B))$ in $\mathcal{E}_{\{A\}}$, there exists $\mathtt{i}_{A,B}(f) : 1_{\{\mathsf{Id}_A\}} \longrightarrow B$ in $\mathcal{E}_{\{\mathsf{Id}_A\}}$, satisfying $(\mathsf{s}(\mathtt{r}_A))^*(\{\overline{\delta_A}(\mathsf{Id}_A)\}^*(\mathtt{i}_{A,B}(f))) = f$. These identity types are also required to satisfy a split Beck-Chevalley condition: for all Cartesian morphisms $f : A \longrightarrow B$, we must have $\{f'\}^*(\mathsf{Id}_B) = \mathsf{Id}_A$ in $\mathcal{E}_{\{\pi_A^*(A)\}}$, where $f' : \pi_A^*(A) \longrightarrow \pi_B^*(B)$ is the unique mediating morphism over $\{f\}$, arising from $\overline{\pi_B}(B) : \pi_B^*(B) \longrightarrow B$ being a Cartesian morphism. As in [18,

§9.3.5], the *diagonal morphisms* δ_A arise from pullback squares of the form

$$\begin{tikzcd}
\{A\} \arrow[r, dashed, "\delta_A"'] \arrow[rr, bend left, "\mathsf{id}_{\{A\}}"] \arrow[dr, bend right, "\mathsf{id}_{\{A\}}"'] & \{\pi_A^*(A)\} \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \{A\} \arrow[d, "\pi_A"] \\
 & \{A\} \arrow[r, "\pi_A"'] & p(A)
\end{tikzcd}$$

3.2 Interpretation of our language in fibred adjunction models

A *fibred adjunction model* is given by a SCCompC $p : \mathcal{V} \longrightarrow \mathcal{B}$, a split fibration $q : \mathcal{C} \longrightarrow \mathcal{B}$, and a split fibred adjunction $F \dashv U : q \longrightarrow p$, such that p supports identity types and weak natural numbers (in the sense of Section 3.1), and q supports split dependent products and sums with respect to p as depicted in

$$\begin{tikzcd}
\mathcal{V} \arrow[r, bend left, "F"] \arrow[r, phantom, "\perp"] \arrow[d, bend right=60, "p"'] \arrow[d, bend left=60, "\{-\}"] & \mathcal{C} \arrow[l, bend left, "U"] \arrow[dl, bend left, "q"] \\
\mathcal{B} \arrow[u, "1" description] &
\end{tikzcd}$$

$p \dashv 1 \dashv \{-\}$

The split dependent products and sums in q, with respect to p, are defined as left and right adjoints to the weakening functors $\pi_A^* : \mathcal{C}_{p(A)} \longrightarrow \mathcal{C}_{\{A\}}$, required to satisfy the analogues of the split Beck-Chevalley conditions from Section 3.1.

Given a SCCompC $p : \mathcal{V} \longrightarrow \mathcal{B}$ that supports identity types and weak natural numbers, we can always pick the identity adjunction $\mathsf{id}_{\mathcal{V}} \dashv \mathsf{id}_{\mathcal{V}} : \mathcal{V} \longrightarrow \mathcal{V}$ to construct a corresponding "effect-free" fibred adjunction model. Further, we can construct a restricted form of adjunction models (without identity types) from models of EEC with weak natural numbers [9], i.e., from $\mathcal{D}$-enriched adjunctions $F_{\text{EEC}} \dashv U_{\text{EEC}} : \mathcal{E} \longrightarrow \mathcal{D}$, where $\mathcal{D}$ is Cartesian closed and has a weak NNO, $\mathcal{E}$ is $\mathcal{D}$-enriched and has all $\mathcal{D}$-tensors and -cotensors. These models are based on a computational variant $q : \mathsf{s}(\mathcal{D}, \mathcal{E}) \longrightarrow \mathcal{D}$ of the the simple fibration $p : \mathsf{s}(\mathcal{D}) \longrightarrow \mathcal{D}$ [18, Thm. 10.5.5]. In particular, the objects of $\mathsf{s}(\mathcal{D}, \mathcal{E})$ are pairs $(X, \underline{C})$ of a $\mathcal{D}$-object X and a $\mathcal{E}$-object $\underline{C}$; and the morphisms $(X, \underline{C}) \longrightarrow (Y, \underline{D})$ are pairs (f, g) of morphisms, with $f : X \longrightarrow Y$ in $\mathcal{D}$ and $g : X \otimes \underline{C} \longrightarrow \underline{D}$ in $\mathcal{E}$.

Interpretation. Following Hoffmann [15] and Streicher [36], we define the interpretation of our language in fibred adjunction models by defining a partial interpretation function $[\![-]\!]$ simultaneously on *pre-contexts*, *pre-types* and *pre-terms*. We do so because of the well-known issue in dependently-typed languages: the derivations of well-formed types and well-typed terms are not unique because of the type conversion rules, as given for our language in Figure 2.

We interpret a pre-context Γ as an object $[\![\Gamma]\!]$ in $\mathcal{B}$, given by

$$[\![\cdot]\!] = 1 \qquad\qquad [\![\Gamma, x\!:\!A]\!] = \{[\![\Gamma; A]\!]\} \quad \text{if } x \notin \Gamma$$

We interpret a value pre-type $\Gamma; A$ as an object $[\![\Gamma; A]\!]$ in $\mathcal{V}_{[\![\Gamma]\!]}$ and a computation pre-type $\Gamma; \underline{C}$ as an object $[\![\Gamma; \underline{C}]\!]$ in $\mathcal{C}_{[\![\Gamma]\!]}$. The definition is given by mapping the syntactic type formers to the corresponding semantic structures:

$$[\![\varGamma;\mathsf{Nat}]\!] = !^*_{[\![\varGamma]\!]}(\mathbb{N}) \quad [\![\varGamma;1]\!] = 1_{[\![\varGamma]\!]} \quad [\![\varGamma;\mathsf{Id}_A(V,W)]\!] = (\langle \mathsf{s}([\![\varGamma;V]\!]), \mathsf{s}([\![\varGamma;W]\!])\rangle)^*(\mathsf{Id}_{[\![\varGamma;A]\!]})$$

$$[\![\varGamma;\varSigma x\!:\!A.B]\!] = \varSigma_{[\![\varGamma;A]\!]}([\![\varGamma,x\!:\!A;B]\!]) \quad [\![\varGamma;\varPi x\!:\!A.B]\!] = \varPi_{[\![\varGamma;A]\!]}([\![\varGamma,x\!:\!A;B]\!])$$

$$[\![\varGamma;U\underline{C}]\!] = U([\![\varGamma;\underline{C}]\!]) \quad [\![\varGamma;FA]\!] = F([\![\varGamma;A]\!])$$

$$[\![\varGamma;\varSigma x\!:\!A.\underline{C}]\!] = \varSigma_{[\![\varGamma;A]\!]}([\![\varGamma,x\!:\!A;\underline{C}]\!]) \quad [\![\varGamma;\varPi x\!:\!A.\underline{C}]\!] = \varPi_{[\![\varGamma;A]\!]}([\![\varGamma,x\!:\!A;\underline{C}]\!])$$

In the interpretation of the identity type $\mathsf{Id}_A(V,W)$, the pairing morphism $\langle \mathsf{s}([\![\varGamma;V]\!]), \mathsf{s}([\![\varGamma;W]\!])\rangle : [\![\varGamma]\!] \longrightarrow \{\pi^*_{[\![\varGamma;A]\!]}([\![\varGamma;A]\!])\}$ is the unique mediating morphism into the pullback square from Section 3.1, for $\mathsf{s}([\![\varGamma;V]\!])$ and $\mathsf{s}([\![\varGamma;W]\!])$.

We interpret a value pre-term $\varGamma;V$ as a morphism $[\![\varGamma;V]\!] : 1_{[\![\varGamma]\!]} \longrightarrow A$ in $\mathcal{V}_{[\![\varGamma]\!]}$, for some A in $\mathcal{V}_{[\![\varGamma]\!]}$; a computation pre-term $\varGamma;M$ as a morphism $[\![\varGamma;M]\!] : 1_{[\![\varGamma]\!]} \longrightarrow U(\underline{C})$ in $\mathcal{V}_{[\![\varGamma]\!]}$, for some $\underline{C}$ in $\mathcal{C}_{[\![\varGamma]\!]}$; and a homomorphism pre-term $\varGamma;\underline{C};K$ as a morphism $[\![\varGamma;\underline{C};K]\!] : [\![\varGamma;\underline{C}]\!] \longrightarrow \underline{D}$ in $\mathcal{C}_{[\![\varGamma]\!]}$, for some $\underline{D}$ in $\mathcal{C}_{[\![\varGamma]\!]}$. Similarly to types, the interpretation is again straightforward, mapping syntactic term formers to their semantic counterparts. We omit most of the interpretation of terms and only show the cases for return and sequential composition as representative examples, given by the following defining rules

$$\frac{[\![\varGamma;V]\!] = 1_{[\![\varGamma]\!]} \xrightarrow{f} A}{[\![\varGamma;\texttt{return}\ V]\!] = 1_{[\![\varGamma]\!]} \xrightarrow{f} A \xrightarrow{\eta_A} U(F(A))}$$

$$\frac{\begin{array}{ll} [\![\varGamma;M]\!] = 1_{[\![\varGamma]\!]} \xrightarrow{f} U(F([\![\varGamma;A]\!])) & \text{for some value type } A \\ [\![\varGamma,x\!:\!A;N]\!] = 1_{[\![\varGamma,x:A]\!]} \xrightarrow{g} U(\pi^*_{[\![\varGamma;A]\!]}(\underline{C})) & \text{for some } \underline{C} \text{ in } \mathcal{C}_{[\![\varGamma]\!]} \end{array}}{[\![\varGamma;M\ \texttt{to}\ x\ \texttt{in}\ N]\!] = 1_{[\![\varGamma]\!]} \xrightarrow{f} U(F([\![\varGamma;A]\!]) \xrightarrow{U(g^\dagger)} U(\underline{C})}$$

where $g^\dagger$ is derived from g by recalling that U is a split fibred functor and therefore commutes with reindexing; using the adjunction $\varSigma_{[\![\varGamma;A]\!]} \dashv \pi^*_{[\![\varGamma;A]\!]}$; recalling that the fibred terminal objects are preserved by reindexing; noticing that $\varSigma_{[\![\varGamma;A]\!]}(\pi^*_{[\![\varGamma;A]\!]}(1_{[\![\varGamma]\!]})) \cong [\![\varGamma;A]\!]$ in $\mathcal{V}_{[\![\varGamma]\!]}$; and using the adjunction $F \dashv U$.

Following Hoffmann [15] and Streicher [36], we prove the correctness of the interpretation function we defined above as the following soundness result.

Theorem 2 (Soundness). *The interpretation function is defined on all well-formed contexts, well-formed types and well-typed terms. The interpretation also identifies types and terms that are equal in the equational theory. For example,*
if $\varGamma \vdash A$, *then* $[\![\varGamma;A]\!]$ *is an object in* $\mathcal{V}_{[\![\varGamma]\!]}$*; and*
if $\varGamma \vdash V = W : A$, *then* $[\![\varGamma;V]\!] = [\![\varGamma;W]\!] : 1_{[\![\varGamma]\!]} \longrightarrow [\![\varGamma;A]\!]$ *in* $\mathcal{V}_{[\![\varGamma]\!]}$.

As in [15, 36], the proof of soundness relies on lemmas relating weakening and substitution to reindexing in fibred adjunction models; we omit them here.

The classifying model. We now show that the interpretation of our language in fibred adjunction models is complete by constructing its classifying model.

First, in the classifying fibred adjunction model the objects of $\mathcal{B}$ are given by equivalence classes of well-formed value contexts $\varGamma$. The morphisms $\varGamma_1 \longrightarrow \varGamma_2$

are given by equivalence classes of tuples of value terms $\boldsymbol{V} = (V_1, \ldots, V_m)$, where $\Gamma_2 = y_1 : B_1, \ldots, y_m : B_m$ and $\Gamma_1 \vdash V_i : B_i[V_1/y_1, \ldots, V_{i-1}/y_{i-1}]$, for all $1 \leq i \leq m$.

Next, the objects of the total category $\mathcal{V}$ are given by equivalence classes of value types $\Gamma \vdash A$ and its morphisms $\Gamma_1 \vdash A \longrightarrow \Gamma_2 \vdash B$ by equivalence classes of tuples of value terms $(\boldsymbol{V}, V)$, where $\boldsymbol{V}$ are typed as in $\mathcal{B}$, and V is typed as $\Gamma_1, x : A \vdash V : B[\boldsymbol{V}/\Gamma_2]$. The objects and morphisms of $\mathcal{C}$ are defined similarly: as equivalence classes of computation types $\Gamma \vdash \underline{C}$; and as equivalence classes of tuples of terms $(\boldsymbol{V}, K)$, where K is typed as $\Gamma_1 \,|\, z : \underline{C} \vdash K : \underline{D}[\boldsymbol{V}/\Gamma_2]$. The fibrations p and q are defined by context projections, i.e., by $p(\Gamma \vdash A) = \Gamma$.

The various adjunctions involved in the definition of fibred adjunction models are defined in terms of their syntactic counterparts. For example, the split fibred adjunction $F \dashv U : q \longrightarrow p$ is defined using the types FA and $U\underline{C}$, given by

$$\begin{array}{ll} F(\Gamma \vdash A) = \Gamma \vdash FA & F(\boldsymbol{V}, V) = (\boldsymbol{V}, z \texttt{ to } y \texttt{ in return } V[y/x]) \\ U(\Gamma \vdash \underline{C}) = \Gamma \vdash U\underline{C} & U(\boldsymbol{V}, K) = (\boldsymbol{V}, \texttt{thunk } (K[\texttt{force } x/z])) \end{array}$$

The identity types and natural numbers are also given in terms of their syntactic counterparts, e.g., the object $\mathsf{Id}_{\Gamma \vdash A}$ in $\mathcal{V}_{\{\pi^*_{\Gamma \vdash A}(\Gamma \vdash A)\}}$ is $\Gamma, x : A, y : A \vdash \mathsf{Id}_A(x, y)$.

Proposition 1. *The above definitions, based on the syntax of our language, constitute a fibred adjunction model, called the classifying model of our language.*

Finally, we can use this result to prove the completeness of the interpretation.

Theorem 3 (Completeness). *If two types or two terms of our language are identified in all fibred adjunction models, they are equal in the equational theory.*

3.3 Fibred adjunction models based on the families fibration

We now discuss some examples of fibred adjunction models based on the prototypical model of dependent types, the families fibration $p : \mathsf{Fam}(\mathcal{D}) \longrightarrow \mathsf{Set}$. The objects of $\mathsf{Fam}(\mathcal{D})$ are given by pairs (X, A) of a set X and an X-indexed family of $\mathcal{D}$-objects $A : X \longrightarrow ob(\mathcal{D})$; the morphisms $(X, A) \longrightarrow (Y, B)$ are pairs $(f, \{g_x\}_{x \in X})$ of a function $f : X \longrightarrow Y$ and a X-indexed family of $\mathcal{D}$-morphisms $\{g_x : A(x) \longrightarrow B(f(x))\}_{x \in X}$. The functor p is defined by first projection, i.e. by $p(X, A) = X$. In fact, p is a split fibration: the reindexing functors f^* are defined by pre-composition, i.e. by $f^*(Y, B) = (X, B \circ f)$ for all $f : X \longrightarrow Y$. In our examples, we take $\mathcal{D}$ to be Set. In this case, p is a SCCompC [18, §10.5]. The examples we discuss below are instances of the following general result, building on the fact that adjunctions can be lifted to families fibrations [18, Ex. 1.8.7 (i)].

Theorem 4. *Given $F \dashv U : \mathcal{E} \longrightarrow \mathsf{Set}$, such that $\mathcal{E}$ has both set-indexed products and coproducts, the fibrations $p : \mathsf{Fam}(\mathsf{Set}) \longrightarrow \mathsf{Set}$ and $q : \mathsf{Fam}(\mathcal{E}) \longrightarrow \mathsf{Set}$, together with the pointwise lifting of $F \dashv U$, determine a fibred adjunction model.*

In Theorem 4, the set-indexed products and coproducts in $\mathcal{E}$ are assumed to exist in order to define the dependent products and dependent sums in q as

$$\begin{array}{l} \Pi_{(X,A)}(\coprod_{x \in X} A(x), \underline{C}) = (X, x \mapsto \prod_{a \in A(x)} \underline{C}(x, a)) \\ \Sigma_{(X,A)}(\coprod_{x \in X} A(x), \underline{C}) = (X, x \mapsto \coprod_{a \in A(x)} \underline{C}(x, a)) \end{array}$$

The lifting $\hat{F} \dashv \hat{U}$ of $F \dashv U$ is defined by composition, i.e., $\hat{F}(X, A) = (X, F \circ A)$.

The first collection of examples we discuss are based on Eilenberg-Moore algebras (EM-algebras) for a monad (T, η, μ) on Set. As standard, we write Set^T for the category of EM-algebras. Its objects are given by pairs (X, α) of a set X and a function $\alpha : TX \longrightarrow X$ such that $\alpha \circ T(\alpha) = \alpha \circ \mu_X$ and $\alpha \circ \eta_X = \mathsf{id}_X$; its morphisms $(X, \alpha) \longrightarrow (Y, \beta)$ are given by functions $f : X \longrightarrow Y$ such that $\beta \circ T(f) = f \circ \alpha$. There is a canonical EM-adjunction $F^T \dashv U^T : \mathsf{Set}^T \longrightarrow \mathsf{Set}$.

The category Set^T has both set-indexed products and coproducts. In fact, for any monad (T, η, μ) on Set, Set^T is complete and cocomplete because Set is complete, cocomplete and regular, and all epimorphisms in it are split (cf. [4, Thm. 4.3.5]). The set-indexed products in Set^T can be defined from the set-indexed products in Set by $\prod_{i \in I}(X_i, \alpha_i) = (\prod_{i \in I} X_i, \langle \alpha_i \circ T(\mathsf{proj}_i) \rangle_{i \in I})$. As shown by Linton [21], the set-indexed coproduct $\coprod_{i \in I}(X_i, \alpha_i)$ in Set^T can be defined as the reflexive coequalizer $e : F^T(\coprod_{i \in I} X_i) \longrightarrow \coprod_{i \in I}(X_i, \alpha_i)$ of the diagram

$$\begin{tikzcd}
F^T(\coprod_{i \in I} X_i) \arrow[r, "F^T([\mathsf{inj}_i \circ \eta_{X_i}]_{i \in I})"] & F^T(\coprod_{i \in I} TX_i) \arrow[r, shift left, "F^T([\mathsf{inj}_i \circ \alpha_i]_{i \in I})"] \arrow[r, shift right, "\mu_{\coprod_{i \in I} X_i} \circ F^T([T(\mathsf{inj}_i)]_{i \in I})"'] & F^T(\coprod_{i \in I} X_i)
\end{tikzcd}$$

Corollary 1. *For any monad (T, η, μ) on* Set*, the EM-adjunction $F^T \dashv U^T$ and the families fibration p :* $\mathsf{Fam}(\mathsf{Set}) \longrightarrow \mathsf{Set}$ *determine a fibred adjunction model.*

A particularly well-behaved collection of fibred adjunction models arises from the algebraic treatment of computational effects [31], namely, from monads arising from countable Lawvere theories. These are exactly the monads on Set that are of countable rank [33, Thm. 2.8]. As discussed in [17], such monads (and, therefore, also the corresponding fibred adjunction models) combine easily, in terms of combining the operations and equations of the corresponding countable Lawvere theories. For constructing the fibred adjunction models, we recall that every countable Lawvere theory $\mathcal{L}$ induces a category $\mathsf{Mod}(\mathcal{L}, \mathsf{Set})$ of models of $\mathcal{L}$ in Set, with an associated forgetful functor $U_{\mathcal{L}} : \mathsf{Mod}(\mathcal{L}, \mathsf{Set}) \longrightarrow \mathsf{Set}$ [33]. As Set is locally countably presentable [1], $U_{\mathcal{L}}$ has a left adjoint $F_{\mathcal{L}}$, inducing an equivalence of categories between $\mathsf{Mod}(\mathcal{L}, \mathsf{Set})$ and $\mathsf{Set}^{T_{\mathcal{L}}}$, for $T_{\mathcal{L}} = U_{\mathcal{L}} \circ F_{\mathcal{L}}$. Importantly for us, $\mathsf{Mod}(\mathcal{L}, \mathsf{Set})$ is both complete and cocomplete: in addition to combining the equivalence with [4, Thm. 4.3.5], this also follows from $T_{\mathcal{L}}$ having countable rank and Set being locally countably presentable [1, Thm. 2.78].

Corollary 2. *For any countable Lawvere theory $\mathcal{L}$, the adjunction $F_{\mathcal{L}} \dashv U_{\mathcal{L}}$ and the families fibration p :* $\mathsf{Fam}(\mathsf{Set}) \longrightarrow \mathsf{Set}$ *determine a fibred adjunction model.*

Finally, we add two computationally motivated examples arising from Theorem 4 and decompositions of monads (T, η, μ) on Set into adjunctions other than $F^T \dashv U^T$. In particular, we consider the continuations monad $R^{R^{(-)}}$ and the global state monad $((-) \times S)^S$. These monads can be decomposed into the adjunctions $R^{(-)} \dashv R^{(-)} : \mathsf{Set}^{\mathrm{op}} \longrightarrow \mathsf{Set}$ and $(-) \times S \dashv (-)^S : \mathsf{Set} \longrightarrow \mathsf{Set}$, where $\mathsf{Set}^{\mathrm{op}}$ inherits its set-indexed products and coproducts trivially from Set.

Corollary 3. *The adjunctions $R^{(-)} \dashv R^{(-)}$ and $(-) \times S \dashv (-)^S$ together with the families fibration p :* $\mathsf{Fam}(\mathsf{Set}) \longrightarrow \mathsf{Set}$ *determine fibred adjunction models.*

4 Extending the language with algebraic effects

Until now we have have not said how computational effects, such as I/O, state, exceptions, etc., arise in our language and how programmers can program with them. In this section, we make the source of computational effects explicit, by drawing ideas from the algebraic treatment of computational effects [31].

4.1 Algebraic effects in the syntax

We begin by assuming we are given a *collection of typed operation symbols*

$$\mathsf{op} : (x_{\mathsf{in}} : I) \longrightarrow O$$

where we call $\cdot \vdash I$ the *input type* and $x_{\mathsf{in}} : I \vdash O$ the *output type* of op. We restrict I and O to be *pure value types*, i.e., value types that do not contain U.

We add such operation symbols to our language by extending the syntax of computation terms: for each operation symbol $\mathsf{op} : (x_{\mathsf{in}} : I) \longrightarrow O$, we add algebraic operations $\mathsf{op}^{\underline{C}}_V(x.M)$, for all $\underline{C}$, and a generic effect genop_V, typed as

$$\frac{\Gamma \vdash V : I \quad \Gamma \vdash \underline{C} \quad \Gamma, x{:}O[V/x_{\mathsf{in}}] \vdash M : \underline{C}}{\Gamma \vdash \mathsf{op}^{\underline{C}}_V(x.M) : \underline{C}} \qquad \frac{\Gamma \vdash V : I}{\Gamma \vdash \mathsf{genop}_V : F(O[V/x_{\mathsf{in}}])}$$

These operations and generic effects are in 1-to-1 correspondence, as in [30].

The operation symbols we consider can also come equipped with a *collection of equations*, describing their intended computational behaviour. We extend our language with the corresponding equations between computation terms.

Note that we allow the output types of operation symbols to depend on input values. This additional type-dependency is useful for giving more concise representations of collections of standard, simply-typed operations. For example, we can use a boolean-indexed type family and two operation symbols

$$\mathsf{lookup} : (x_{\mathsf{in}} : \mathsf{Bool}) \longrightarrow (\texttt{if } x_{\mathsf{in}} \texttt{ then } \mathsf{Nat} \texttt{ else } \mathsf{Bool})$$
$$\mathsf{update} : (x_{\mathsf{in}} : \Sigma x{:}\mathsf{Bool}.(\texttt{if } x \texttt{ then } \mathsf{Nat} \texttt{ else } \mathsf{Bool})) \longrightarrow 1$$

as a concise syntax for the four operations of global state with two locations:

$$\mathsf{lookup}_{\mathtt{tt}} : 1 \longrightarrow \mathsf{Nat} \quad \mathsf{update}_{\mathtt{tt}} : \mathsf{Nat} \longrightarrow 1 \quad \mathsf{lookup}_{\mathtt{ff}} : 1 \longrightarrow \mathsf{Bool} \quad \mathsf{update}_{\mathtt{ff}} : \mathsf{Bool} \longrightarrow 1$$

In order to make the computation terms $\mathsf{op}^{\underline{C}}_V(x.M)$ behave like algebraic operations, we extend our language with a *general algebraicity equation*:

$$\frac{\mathsf{op} : (x_{\mathsf{in}} : I) \longrightarrow O \qquad \Gamma \vdash V : I \qquad \Gamma, x{:}O[V/x_{\mathsf{in}}] \vdash M : \underline{C} \qquad \Gamma \,|\, z{:}\underline{C} \vdash K : \underline{D}}{\Gamma \vdash K[\mathsf{op}^{\underline{C}}_V(x.M)/z] = \mathsf{op}^{\underline{D}}_V(x.K[M/z]) : \underline{D}} \; (**)$$

Using $(**)$, we can easily prove equations familiar from languages with algebraic effects, e.g. the algebraicity equation for sequential composition from [34, §3.3]:

$$\Gamma \vdash \mathsf{op}^{\underline{C}}_V(x.(M \texttt{ to } y \texttt{ in } N)) = \mathsf{op}^{FA}_V(x.M) \texttt{ to } y \texttt{ in } N : \underline{C}$$

We omit handlers of algebraic effects [32] from this paper as a full account, in which the equational properties of handlers are derived from those of homomorphism terms, i.e., from $(**)$, involves extending our language with a further computation type former $\langle A, \{V_{\mathsf{op}}\}_{\mathsf{op}} \rangle$ for user-defined algebras. To make handlers first-class, one extends the language with the type $\underline{C} \multimap \underline{D}$ of homomorphisms, familiar from EEC. We will report on this extension separately in future work.

4.2 Algebraic effects in the semantics

Following the definition of algebraic operations for a monad in terms of its EM-algebras [30, §6], we say that a *fibred adjunction model supports algebraic operations* if for all $\mathsf{op} : (x_{\mathsf{in}} \,{:}\, I) \longrightarrow O$, there exist vertical morphisms

$$[\![\mathsf{op}]\!]_{\underline{C}} : \Sigma_{!^*_{q(\underline{C})}([\![I]\!])}(\Pi_{\{\overline{!_{q(\underline{C})}}([\![I]\!])\}^*([\![O]\!])}(\pi^*_{\{\overline{!_{q(\underline{C})}}([\![I]\!])\}^*([\![O]\!])}(\pi^*_{!^*_{q(\underline{C})}([\![I]\!])}(U(\underline{C}))))) \longrightarrow U(\underline{C})$$

that are natural in $\underline{C}$. When combining this naturality with the fact that U is a split fibred functor, it can be shown that the $[\![\mathsf{op}]\!]_{\underline{C}}$'s are preserved by reindexing. If the given collection of operation symbols also comes with associated equations, these vertical morphisms are additionally required to satisfy these equations.

The classifying model from Section 3.2 can be straightforwardly extended: for any $\Gamma = x_1 \,{:}\, A_1, \ldots, x_n \,{:}\, A_n$ and $\Gamma \vdash \underline{C}$, the vertical morphisms $[\![\mathsf{op}]\!]_{\Gamma \vdash \underline{C}}$ are

$$[\![\mathsf{op}]\!]_{\Gamma \,\vdash\, \underline{C}} = \big(x_1, \ldots, x_n, \texttt{thunk}\ (\mathsf{op}^{\underline{C}}_{\texttt{fst}\, x'}(x.\texttt{force}\ ((\texttt{snd}\, x')\, x)))\big)$$

where x' has type $\Sigma x_{\mathsf{in}} \,{:}\, I.\Pi x \,{:}\, O.U\underline{C}$. The naturality of these is proved by using the equations describing the interaction between thunking and forcing, in combination with the general algebraicity equation $(**)$ from Section 4.1.

Finally, we note how algebraic operations can be characterised in the fibred adjunction models based on the families fibration and EM-algebras of a monad.

Proposition 2. *The fibred adjunction models from Section 3.3, based on the EM-algebras of a monad* (T, η, μ) *on* Set*, support algebraic operations if for all operations* $\mathsf{op} : (x_{\mathsf{in}} \,{:}\, I) \longrightarrow O$ *there exists a family of natural transformations*

$$\Big\{\ [\![\mathsf{op}]\!]_i : (U^T(-))^{[\![O]\!](\langle \star, i \rangle)} \longrightarrow U^T(-)\ \Big\}_{i \in [\![I]\!](\star)}$$

5 Extending the language with general recursion

We now show how to extend our language with general recursion, considering it as a computational effect to keep the MLTT fragment of our language effect-free.

5.1 Recursion in the syntax

We start by extending our language with a new computation term, the *fixed point operation* $\mu x \,{:}\, U\underline{C}.M$. The corresponding typing rule is given by

$$\frac{\Gamma, x \,{:}\, U\underline{C} \vdash M : \underline{C}}{\Gamma \vdash \mu x \,{:}\, U\underline{C}.M : \underline{C}}$$

We also extend the language's equational theory with unfolding of fixed points:

$$\Gamma \vdash \mu x \,{:}\, U\underline{C}.M = M[\texttt{thunk}\ (\mu x \,{:}\, U\underline{C}.M)/x] : \underline{C}$$

Finally, we also alter the definition of identity types $\mathsf{Id}_{A_{\mathsf{disc}}}(V, W)$, restricting them to be over *discrete value types*, to be able to interpret this extended language in models based on continuous families of cpos. These discrete types are

$$A_{\mathsf{disc}}, B_{\mathsf{disc}} \ ::= \ \mathsf{Nat} \mid \Sigma x \,{:}\, A_{\mathsf{disc}}.B_{\mathsf{disc}} \mid 1 \mid \Pi x \,{:}\, A.B_{\mathsf{disc}} \mid \mathsf{Id}_{A_{\mathsf{disc}}}(V, W)$$

5.2 Domain-theoretic semantics for recursion

We build the denotational semantics for the language with recursion around the *SCCompC of continuous families* $p : \mathsf{CFam}(\mathcal{CPO}) \longrightarrow \mathcal{CPO}$ [18, §10.6]. Compared to [18], we use ω-complete partial orders instead of directed-complete partial orders, because the former constitute a locally countably presentable category, whereas the latter do not [1, Ex. 1.14 (4)]; and we need local presentability for fibred adjunction models based on the algebraic treatment of computational effects. An overview of the relevant domain theory can be found in [11, 29].

We recall that the objects of $\mathsf{CFam}(\mathcal{CPO})$ are pairs (X, A) of a cpo X and a continuous functor $A : X \longrightarrow \mathcal{CPO}^{\mathrm{EP}}$ (a *continuous family*), treating X as a category and valued in the category of embedding-projection pairs. The morphisms $(X, A) \longrightarrow (Y, B)$ are pairs $(f, \{g_x\}_{x \in |X|})$ of a continuous function $f : X \longrightarrow Y$ and a family of continuous functions $\{g_x : A(x) \longrightarrow B(f(x))\}_{x \in |X|}$, satisfying

$$x_1 \sqsubseteq_X x_2 \implies B(f(x_1 \sqsubseteq_X x_2))^e \circ g_{x_1} \sqsubseteq_{B(f(x_2))^{A(x_1)}} g_{x_2} \circ A(x_1 \sqsubseteq_X x_2)^e$$
$$\langle x_n \rangle \text{ incr. } \omega\text{-chain} \implies g_{\bigvee_n x_n} = \bigvee_n \big(B(f(x_n \sqsubseteq_X \bigvee_n x_n))^e \circ g_{x_n} \circ A(x_n \sqsubseteq_X \bigvee_n x_n)^p\big)$$

The dependent products and strong dependent sums are defined by using the cpo-indexed products $\prod_X A$ and coproducts $\coprod_X A$ in $\mathcal{CPO}$, which are given by

$$\textstyle\coprod_X A = \big(\coprod_{x \in |X|} |A(x)|, \langle x_1, a_1 \rangle \sqsubseteq \langle x_2, a_2 \rangle \text{ iff } x_1 \sqsubseteq_X x_2 \text{ and } A(x_1 \sqsubseteq_X x_2)^e(a_1) \sqsubseteq a_2\big)$$
$$\textstyle\prod_X A = \big(\{f : X \longrightarrow \coprod_X A \,|\, \mathsf{fst} \circ f = \mathsf{id}_X\}, f_1 \sqsubseteq f_2 \text{ iff } \forall x \in |X|.\, f_1(x) \sqsubseteq_{\coprod_X A} f_2(x)\big)$$

For identity types, we require A to be a continuous family of discrete cpos, matching the changes we made in the syntax of our language, and then define

$$\mathsf{Id}_{(X,A)} = \big(\{\pi^*_{(X,A)}(X, A)\}, \langle x, a, a' \rangle \mapsto \textstyle\coprod_{\{\star \,|\, a = a'\}} 1\big)$$

The discreteness of A is necessary for $\langle x, a, a' \rangle \mapsto \coprod_{\{\star \,|\, a = a'\}} 1$ to constitute a continuous functor: if A would not be discrete, then from $\langle x_1, a_1, a'_1 \rangle \sqsubseteq \langle x_2, a_2, a'_2 \rangle$ and $a_1 = a'_1$ it would not follow that $a_2 = a'_2$, and vice versa, which we need for defining the embedding-projection pair between $\coprod_{\{\star \,|\, a_1 = a'_1\}} 1$ and $\coprod_{\{\star \,|\, a_2 = a'_2\}} 1$.

For modeling computation terms involving recursion, we assume a $\mathcal{CPO}$-enriched monad (T, η, μ) on $\mathcal{CPO}$, such that its EM-algebras are pointed and the morphisms between them are strict; or equivalently, we assume that the given monad supports a least zero-ary algebraic operation, in the sense of [30, §6]. For modeling computational Σ-types, we further assume that $\mathcal{CPO}^T$ has reflexive coequalizers. We can then model our computation types in the split fibration $q : \mathsf{CFam}(\mathcal{CPO}^T) \longrightarrow \mathcal{CPO}$, defined analogously to p above. Monads satisfying these conditions arise naturally from the algebraic treatment of computational effects: from $\mathcal{CPO}$-enriched countable Lawvere theories [17] with a least constant.

The dependent products and sums in q, with respect to p, are defined using the cpo-indexed products and coproducts in $\mathcal{CPO}^T$. First, the cpo-indexed product $\prod_X \underline{C}$ in $\mathcal{CPO}^T$ is directly inherited from $\mathcal{CPO}$, being defined on the carrier $\prod_X (U^T \circ \underline{C})$. On the other hand, analogously to Section 3.3, the cpo-indexed coproduct $\coprod_X \underline{C}$ cannot be defined simply by taking $\coprod_X (U^T \circ \underline{C})$ as the carrier. Instead, we construct $\coprod_X \underline{C}$ as the reflexive coequalizer for a diagram similar to the one used in Section 3.3, with the difference that here we would use the free algebras over cpo-indexed coproducts rather than over set-indexed coproducts.

Despite having a more complex categorical definition, the cpo-indexed coproducts in $\mathcal{CPO}^T$ have the same universal property as those in $\mathcal{CPO}$. We first recall Jacobs's remark [18, §10.6] that $\coprod_X A$ results from applying the Grothendieck construction to A. By a result of Gray [12], $\coprod_X A$ can also be understood as the oplax colimit of A in $\mathcal{CPO}$. In $\mathcal{CPO}^T$, the situation is analogous:

Proposition 3. *$\coprod_X \underline{C}$ is the oplax colimit of $\underline{C} : X \longrightarrow (\mathcal{CPO}^T)^{EP}$ in $\mathcal{CPO}^T$.*

Similarly to Section 3.3, the split fibred adjunction $F \dashv U : q \longrightarrow p$ is defined from $F^T \dashv U^T$ by post-composition, i.e., by setting $F(X, A) = (X, F^T \circ A)$, where $F^T \circ A$ is a continuous functor because of the $\mathcal{CPO}$-enrichment of F^T and the limit-colimit coincidences in $\mathcal{CPO}^{\mathrm{EP}}$ and $(\mathcal{CPO}^T)^{\mathrm{EP}}$. U is defined analogously.

We interpret our language as discussed in Section 3.2, except for recursion:

$$[\![\mu x\!:\!U\underline{C}.M]\!] = \big(\mathsf{id}_{[\![\Gamma]\!]}, \{x \in |1| \mapsto \mu(g_\gamma)\}_{\gamma \in |[\![\Gamma]\!]|}\big) \,:\, ([\![\Gamma]\!], \gamma \mapsto 1) \longrightarrow ([\![\Gamma]\!], U^T \circ [\![\underline{C}]\!])$$

where we use the least fixed points $\mu(g_\gamma)$ of the family of continuous functions $\{g_\gamma : U^T([\![\underline{C}]\!])(\gamma) \longrightarrow U^T([\![\underline{C}]\!])(\gamma)\}_{\gamma \in |[\![\Gamma]\!]|}$, determined by a vertical morphism $([\![\Gamma]\!], U^T \circ [\![\underline{C}]\!]) \longrightarrow ([\![\Gamma]\!], U^T \circ [\![\underline{C}]\!])$ we derive from $[\![M]\!]$. The least fixed points of these continuous functions g_γ are guaranteed to exist because our assumptions about $\mathcal{CPO}^T$ make every $U^T \circ [\![\underline{C}]\!]$ into a continuous family of pointed cpos.

Theorem 5. *Given a monad (T, η, μ) on $\mathcal{CPO}$ satisfying the conditions given in this section, the fibred adjunction model built from $p : \mathsf{CFam}(\mathcal{CPO}) \longrightarrow \mathcal{CPO}$ and $F^T \dashv U^T$ is a model of the equational theory extended with fixed point unfolding.*

Finally, we note that the other obvious candidate $\mathsf{cod} : \mathcal{CPO}^{\rightarrow} \longrightarrow \mathcal{CPO}$, even if made split [14, 8], is not a SCCompC, because of [18, Thm. 10.5.5] and:

Proposition 4. *$\mathcal{CPO}$ is not locally Cartesian closed.*

In particular, the condition that every base change functor has to have a right adjoint fails because some of these functors do not preserve all colimits, e.g., given a non-empty cpo X, the pullback of the epimorphism $n \mapsto n : \mathbb{N}_= \longrightarrow \mathbb{N}_\omega$ in $\mathcal{CPO}/\mathbb{N}_\omega$ along the constant map $x \mapsto \omega : X \longrightarrow \mathbb{N}_\omega$ is not an epimorphism.

6 Conclusions and Future Work

We addressed the problem of finding a mathematically natural combination of dependent types and computational effects. We were motivated by: i) the success similar foundations have had in driving the study of computational effects in the simply-typed setting; and ii) the success of dependently-typed programming in generating a number of concrete attempts to combine dependent types with computational effects. Our solution is mathematically natural, combining comprehension categories, arising from the semantics of dependent types, with adjunctions, arising from the semantics of computational effects. It is also general, covering a variety of algebraic and non-algebraic effects, and can be extended to accommodate general recursion. For future work, a natural next step is to investigate operational semantics, leading towards an implementation. We are also working on a fibred generalisation of Atkey's parametrised notions of computation [2], aiming at a semantic account of the effects in Idris [5, 6].

References

[1] Adamek, J., Rosicky, J.: Locally Presentable and Accessible Categories. No. 189 in London Mathematical Society Lecture Note Series, Cambridge Univ. Press (1994)

[2] Atkey, R.: Algebras for parameterised monads. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) Proc. of 3rd Int. Conf. on Algebra and Coalgebra in Computer Science, CALCO 2009. LNCS, vol. 5728, pp. 3–17. Springer (2009)

[3] Benton, P.: A mixed linear and non-linear logic: Proofs, terms and models. In: Pacholski, L., Tiuryn, J. (eds.) Proc. of 9th Int. Wksh. Computer Science Logic, CSL'95. LNCS, vol. 933, pp. 121–135. Springer (1995)

[4] Borceux, F.: Handbook of Categorical Algebra, vol. 2: Categories and Structures. Cambridge Univ. Press (1994)

[5] Brady, E.: Programming and reasoning with algebraic effects and dependent types. In: Morrisett, G., Uustalu, T. (eds.) Proc. of 18th Int. Conf. on Functional Programming, ICFP'13. pp. 133–144. ACM (2013)

[6] Brady, E.: Resource-dependent algebraic effects. In: Hage, J. (ed.) Proc. of 15th Symp. on Trends in Functional Programming, TFP'14. LNCS, vol. 8843, pp. 18–33. Springer (2015)

[7] Casinghino, C., Sjöbergberg, V., Weirich, S.: Combining proofs and programs in a dependently typed language. In: Sewell, P. (ed.) Proc. of 41st Ann. Symp. on Principles of Programming Languages, POPL'14. pp. 33–45. ACM (2014)

[8] Clairambault, P., Dybjer, P.: The biequivalence of locally cartesian closed categories and Martin-Löf type theories. In: Ong, L. (ed.) Proc. of 10th Int. Conf. on Typed Lambda Calculi and Applications, TLCA 2011. LNCS, vol. 6690, pp. 91–106. Springer (2011)

[9] Egger, J., Møgelberg, R.E., Simpson, A.: The enriched effect calculus: syntax and semantics. J. Log. Comput. 24(3), 615–654 (2014)

[10] Ghani, N., Johann, P., Fumex, C.: Indexed induction and coinduction, fibrationally. Logical Methods in Computer Science 9(3) (2013)

[11] Gierz, G., Hofmann, K.H., Keimel, K., Lawson, J.D., Mislove, M., Scott, D.S.: Continuous Lattices and Domains. Cambridge Univ. Press (2003)

[12] Gray, J.W.: The categorical comprehension scheme. In: Proc. of Category Theory, Homology Theory and Their Applications III, Lecture Notes in Mathematics, vol. 99, pp. 242–312. Springer (1969)

[13] Hancock, P., Setzer, A.: Interactive programs in dependent type theory. In: Clote, P., Schwichtenberg, H. (eds.) Proc of 14th Int. Wksh. Computer Science Logic, CSL 2000. LNCS, vol. 1862, pp. 317–331. Springer (2000)

[14] Hofmann, M.: On the interpretation of type theory in locally cartesian closed categories. In: Pacholski, L., Tiuryn, J. (eds.) Proc. of 8th Int. Wksh. Computer Science Logic, CSL'94. LNCS, vol. 933, pp. 427–441. Springer (1994)

[15] Hofmann, M.: Syntax and semantics of dependent types. In: Pitts, A.M., Dybjer, P. (eds.) Semantics and Logics of Computation, pp. 79–130. CUP (1997)

[16] Hutton, G., Meijer, E.: Monadic parsing in Haskell. J. Funct. Program. 8(4), 437–444 (1998)

[17] Hyland, M., Plotkin, G., Power, J.: Combining effects: Sum and tensor. Theor. Comput. Sci. 357(1–3), 70–99 (2006)

[18] Jacobs, B.: Categorical Logic and Type Theory. No. 141 in Studies in Logic and the Foundations of Mathematics, North Holland, Amsterdam (1999)

[19] Krishnaswami, N.R., Pradic, P., Benton, N.: Integrating linear and dependent types. In: Walker, D. (ed.) Proc. of 42nd Ann. Symp. on Principles of Programming Languages, POPL'15. pp. 17–30. ACM (2015)

[20] Levy, P.B.: Call-By-Push-Value: A Functional/Imperative Synthesis, Semantics Structures in Computation, vol. 2. Springer (2004)

[21] Linton, F.: Coequalizers in categories of algebras. In: Eckmann, B. (ed.) Proc. of Seminar on Triples and Categorical Homology Theory, LNCS, vol. 80, pp. 75–90. Springer (1969)

[22] Martin-Löf, P.: An intuitionisitc theory of types, predicative part. In: Rose, E., J.C., S. (eds.) Proc. of Logic Colloquium 1973. pp. 73–118. North-Holland (1975)

[23] McBride, C.: Functional pearl: Kleisli arrows of outrageous fortune. J. Funct. Program. (To appear)

[24] Moggi, E.: Computational lambda-calculus and monads. In: Parikh, R. (ed.) Proc. of 4th Ann. Symp. on Logic in Computer Science, LICS'89. pp. 14–23. IEEE (1989)

[25] Moggi, E.: Notions of computation and monads. Inf. Comput. 93(1), 55–92 (1991)

[26] Nanevski, A., Morrisett, G., Birkedal, L.: Hoare type theory, polymorphism and separation. J. Funct. Program. 18(5-6), 865–911 (2008)

[27] Pitts, A.M.: Categorical Logic. In: Abramsky, S., Gabbay, D.M., Maibaum, T.S.E. (eds.) Handbook of Logic in Computer Science, Volume 5. Algebraic and Logical Structures, chap. 2, pp. 39–128. Oxford University Press (2000)

[28] Pitts, A.M., Matthiesen, J., Derikx, J.: A dependent type theory with abstractable names. In: Mackie, I., Ayala-Rincon, M. (eds.) Proc. of 9th Wksh. on Logical and Semantic Frameworks, with Applications, LSFA 2014. ENTCS, vol. 312, pp. 19–50. Elsevier (2015)

[29] Plotkin, G.: Pisa notes (on domain theory) (1983), available online

[30] Plotkin, G., Power, J.: Semantics for algebraic operations. In: Brookes, S., Mislove, M. (eds.) Proc. of 17th Conf. on the Mathematical Foundations of Programming Semantics, MFPS XVII. ENTCS, vol. 45, pp. 332–345. Elsevier (2001)

[31] Plotkin, G.D., Power, J.: Notions of computation determine monads. In: Nielsen, M. (ed.) Proc. of 5th Int. Conf. on Foundations of Software Science and Computation Structures, FOSSACS 2002. LNCS, vol. 2303, pp. 342–356. Springer (2002)

[32] Plotkin, G.D., Pretnar, M.: Handling algebraic effects. Logical Methods in Computer Science 9(4:23) (2013)

[33] Power, J.: Countable Lawvere theories and computational effects. In: Seda, A.K., Hurley, T., Schellekens, M., an Airchinnigh, M.M., Strong, G. (eds.) Proc. of 3rd Irish Conf. on the Mathematical Foundations of Computer Science and Information Technology, MFCSIT 2004. ENTCS, vol. 161, pp. 59–71. Elsevier (2006)

[34] Pretnar, M.: The Logic and Handling of Algebraic Effects. Ph.D. thesis, School of Informatics, University of Edinburgh (2010)

[35] Staton, S.: Instances of computational effects: An algebraic perspective. In: Kupferman, O. (ed.) Proc. of 28th Ann. Symp. on Logic in Computer Science, LICS 2013. pp. 519–519. IEEE (June 2013)

[36] Streicher, T.: Semantics of Type Theory. Birkhäuser (1991)

[37] Swamy, N., Weinberger, J., Schlesinger, C., Chen, J., Livshits, B.: Verifying higher-order programs with the Dijkstra monad. In: Flanagan, C. (ed.) Proc. of 34th Conf. on Programming Language Design and Implementation, PLDI'13. pp. 387–398. ACM (2013)

[38] Vákár, M.: A categorical semantics for linear logical frameworks. In: Pitts, A. (ed.) Proc. of 18th Int. Conf. on Foundations of Software Science and Computation Structures, FoSSaCS 2015, LNCS, vol. 9034, pp. 102–116. Springer (2015)

[39] Warren, M.A.: Homotopy Theoretic Aspects of Constructive Type Theory. Ph.D. thesis, Department of Philosophy, Carnegie Mellon University (2008)