In Search of Effectful Dependent Types

Matthijs Vákár
Magdalen College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy
Trinity 2017

To my parents,
Hilde and László,

and grandparents,
Gé, Dick, Imi and László,

for their extraordinary selflessness.

Time's fun when you're having flies.

— Kermit the Frog

Acknowledgements

Personal

This thesis is the result of many years of academic and social support from far more people than I could list here.

I would like to thank Samson Abramsky for giving me the fantastic and unique opportunity of almost limitless freedom to pursue my academic interests under his guidance and encouragement for the past years and for sharing his experience and wisdom on how to navigate academic life whenever I needed advice. In addition to Samson, I've had the joy of having Radha Jagadeesan as a collaborator and, effectively but unofficially, cosupervisor. His optimism and excitement about our project as well as his kindness as a person were a huge source of support for me. I would like to point out that I (acting as primary author) was lucky to produce the work on coherence space and game semantics for dependent types included in this thesis in collaboration with both Radha and Samson.

I am very thankful to Nick Benton for taking me on as an intern at Microsoft Research Cambridge, despite my theoretical background, and for patiently and in a fun way teaching me so much about practical computer science and software engineering. My thanks go out also to my examiners, Aleks Kissinger, Kobi Kremnitzer, Guy McCusker, Luke Ong and Sam Staton, as well as the numerous anonymous conference and journal referees who read my work and provided impressively precise and useful feedback. I learned a lot from visiting Bath, Birmingham, Bristol and Paris, which I owe to Fanny He, Neel Krishnaswami, James Ladyman and Alexis Saurin. I would further like to thank Hongseok Yang and Paul Levy for the discussions on programming language theory and Urs Schreiber for explaining to me his thoughts on linear dependent type theory. Further, it's been lots of fun and a great learning experience to get to collaborate (and lift weights!) with my good friend Neil Dhir.

More broadly, I am thankful to the Departments of Computer Science – particularly the Quantum Group –, of Statistics and of Engineering at the University of Oxford for providing such a fascinating and friendly academic environment. Particularly, my experience in Oxford would have been much less enjoyable and educational if it hadn't been for the many discussions of logic over coffee with Alex Kavvos, Kohei Kishida and Norihiro Yamada. I am grateful to Destiny Chen

and Julie Sheppard for their spectacular administrative support and for making sure I never failed to leave their offices with a smile. I am highly indebted to my friendly colleagues at MSR Cambridge, like Jonathan Balkind, Tony Hoare and Claudio Russo, who made my time there a real treat.

Before my time in Oxford, I was very fortunate to be inspired and supported in my ambition to pursue a doctorate, by many of the excellent professors I've been lucky enough to have been taught by. In particular, I am very thankful to Heinz Hanßmann, Jan Hogendijk, Peter Johnstone, Corry Samson and Paul Ziche for being such marvellous academic rôle models. Similarly, I am highly indebted to my friends Dejan Gajic and Joost Nuiten, whose intelligence and work ethic I always hugely admired during our time together in Amersfoort and Utrecht.

It's been an especially incredible gift to get to spend this period of academic and personal development in an inspiring environment like Oxford. I am very thankful for the stunning physical environment, all the interesting academic events, but most of all for the totally extraordinary people I've had the privilege to meet there. In particular, I feel I've been very blessed to have wonderful friends here like Carlos, Christoph, Claudia, Jerome, Karine, Marieke, Molly, Paul and Santhy, as well as the support of the Oxford Thich Nhat Hanh sangha, the Clarendon Scholars community and my friendly housemates at 20 Tyndale Road. During the end of my time in Oxford, it was so meaningful and uplifting to write up together with Jenna and to be inspired by her optimism and empathy.

Finally, I cannot imagine what my life would have looked like without my caring family and my long-time friends Bart, Carien, Ewout, Hambo, Julius, Meike, Pieter, Temple, Victor and my Descartes College chums. I am very grateful to Gina for her love during many of the past years. Thanks to all of you for putting up with me! I can only hope to have given you as much joy and support as you have given me over the years.

Institutional

I am enormously grateful to the EPSRC, the Clarendon Fund and the Department of Computer Science at the University of Oxford for funding this endeavour. Many diagrams in this thesis were produced using Paul Taylor's commutative diagrams package.

There are only two kinds of programming languages: those people always bitch about and those nobody uses.

— Bjarne Stroustrup

Abstract

Real world programming languages crucially depend on the availability of computational effects to achieve programming convenience and expressive power as well as program efficiency. Logical frameworks rely on predicates, or dependent types, to express detailed logical properties about entities. According to the Curry-Howard correspondence, programming languages and logical frameworks should be very closely related. However, a language that has both good support for real programming and serious proving is still missing from the programming languages zoo. We believe this is due to a fundamental lack of understanding of how dependent types should interact with computational effects. In this thesis, we make a contribution towards such an understanding, with a focus on semantic methods.

Our first line of work concerns a dependently typed version of linear logic (which can be seen as a calculus for commutative effects). We develop a dependently typed dual intuitionistic linear logic as well as a sound and complete categorical semantics using certain indexed monoidal categories satisfying a comprehension axiom. We present a range of models, based on monoidal families, commutative effects, a double gluing construction, domains and strict functions and coherence spaces.

Our second line of work develops a game semantics for dependent type theory, which had so far been missing altogether. We show that, if we work with deterministic well-bracketed history-free winning strategies, the semantics satisfies a full and faithful completeness result with respect to call-by-name dependent type theory for a hierarchy of types built from certain finite inductive families. We show that by relaxing the notion of strategy, we can further model various effects rather than the pure type theory.

Our final line of work explores a generalisation of Levy's call-by-push-value (CBPV) to encompass dependent types. We show that the syntax of CBPV naturally extends to a calculus we call dCBPV- in which types are allowed to depend on values but not computations. We show it has an elegant categorical semantics and a well-behaved operational semantics and that it admits a wide range of models arising from indexed monads on models of pure dependent type theory and from models of linear dependent type theory. By contrast with the simply typed situation, however, it does not suffice to encode call-by-value and call-by-name versions of dependent type theories with unrestricted effects. To obtain those, we need a richer calculus dCBPV+ with a Kleisli extension principle for dependent functions, which turns out to be less well-behaved from a semantic point of view.

(...) Livet maa forstaaes baglaends. Men (...) maa leves forlaends.

Life can only be understood backwards; but it must be lived forwards.

— Søren Kierkegaard

Contents

Logic, like whiskey, loses its beneficial effect when taken in too large quantities.

— Lord Dunsany

1 Introduction

1.1 Motivation

1.1.1 The Limits of Logic and the Conception of Computers

Logic and computer science have been intimately related since the latter's early days [1–3]. Indeed, the precise modern concept of computability[1] was rapidly formalised in the early 1930s by a group of logicians, motivated, at least in part, by questions in foundations of mathematics like Hilbert's Entscheidungsproblem. Particularly notable is that a wide range of formalisations of the concept of computability were proposed in short succession, many of which were proven to be equivalent in the so-called Church-Turing thesis. This "confluence of notions" of computation included but was by no means limited to

- Herbrand-Gödel computable functions (or general **recursive** functions), a scheme for axiomatising effectively computable functions, introduced in the aftermath of Gödel's study of his incompleteness theorems, which demonstrated the limits of axiomatic systems to formalise mathematics;

[1] Following centuries of more informal descriptions of special cases of algorithms and computing machines, dating back at least to Euclid.

- Church's λ-calculus, a formal language for defining functions that can now be seen as a failed attempt at providing a foundation of mathematics: it turned out to be inconsistent as a **logic**; in hindsight, one could argue that this was one of the first real **programming languages**, however, for writing **algorithms** or **programs** rather than **proofs**; modern functional languages still closely resemble it;

- Kleene's μ-recursive functions, which clearly show how the expressive power of general computable functions can be obtained from the weaker previously studied scheme of primitive recursive functions: by adding a minimisation operator, closely related to the fixpoint combinators definable in the (untyped) λ-calculus;

- Turing machines, giving a universal notion of **hardware** on which computation can be performed.

Of course, it would still take more than a decade of clever engineering to transform this theoretical groundwork into a working practical computer. For excellent accounts of this fascinating history, we refer the reader to [1–3].

1.1.2 Terms and Types

To restore the logical consistency of his system, Church introduced devices called **types**[2], to classify the **terms** of his λ-calculus (the **programs**, if we view the calculus as a programming language). From a modern point of view, we can think of types as providing guarantees about a term (algorithm), for instance by putting certain (**extensional**) restrictions on the **inputs** it takes and the **outputs** it produces or (**intensional**) restrictions on **the manner** in which the outputs are computed from the inputs.

Types were originally introduced by Church for foundational reasons to restore the consistency of the λ-calculus as a logic. We must remember that the untyped

[2]The inconsistency is caused by self-application which allows us to construct Russell's paradox in the untyped λ-calculus. Note that types had been previously introduced by Russell already to circumvent the same paradox in Cantor's naive set theory.

Type theory	**Programming**	**Intuitionistic Logic**
Type A	(Data) Type A	Proposition A
Term $b : B$	Program b with output of type B	Proof b with conclusion B
Typing context $x_1 : A_1, \ldots, x_n : A_n$	Inputs of type $A_1, \ldots, A_n$	Assumptions $A_1, \ldots, A_n$
Conversion $b \rightsquigarrow b' : B$	Execution $b \rightsquigarrow b' : B$	Proof normalization $b \rightsquigarrow b' : B$
Product type $A \times B$	Type of pairs of type A and B	Conjunction $A \wedge B$
Sum type $A + B$	Disjoint union of types A and B	Disjunction $A \vee B$
Function type $A \Rightarrow B$	Type of (first class) functions from A to B	Implication $A \Rightarrow B$
Singleton type 1	`void` (Type of returning commands)	True
Empty type 0	`error` (Type of non-returning commands)	False
Parametric polymorphism Π_A	Generics	2^{nd}-order quantification $\forall_A$

Figure 1.1: An informal sketch of some instances of the Curry-Howard correspondence.

λ-calculus was already a fine (albeit primitive) programming language! It is perhaps surprising, therefore, that types have turned out to be of huge practical value in software development, the main reason being that simple type annotations happen to catch many of the most common bugs introduced by programmers before the program is run. Moreover, types provide a useful abstraction of programs that helps programmers think about their code and certainly make it much easier to read code written by others. It is not a coincidence that the top four programming languages in the TIOBE Index of popular programming languages (Java, C, C++ and C#) all have a strongly enforced type system [4].

The motivations for types outlined in the previous paragraph are rather pragmatic in nature. A more principled motivation for types comes from the **Curry-Howard correspondence**, which suggests that some **type theories**, the simply typed λ-calculus being the prime example, can be interpreted both as a programming language and as an (intuitionistic[3]) formal logic.

Informally, a type theory is a calculus for constructing terms (the **programs** or **proofs**) in a compositional way, starting from certain basic building blocks, subject to the restrictions on their inputs and outputs (or assumptions and conclusions) imposed by the types (or **propositions**). It can also be used to reason about the equality and conversion (execution/evaluation behaviour or proof normalization) of these terms. This dual reading of a type theory as a programming language and a logic is very roughly summarised in figure 1.1.

[3]In the sense that the principle of double negation elimination does not hold: not not A does not imply A. Note that such a formalism is strictly more expressive than classical logic as the latter is precisely the fragment of the former consisting of the doubly negated propositions.

In particular, simple type theory (or the simply-typed λ-calculus) can not only be viewed as a primitive programming language, but also as a formalism for writing (natural deduction style) proofs for intuitionistic (implicational[4]) propositional logic.

1.1.3 Programming Requires More Terms: Effects

The simply typed λ-calculus is a rather unexpressive programming language, even when enriched with ground types for booleans and natural numbers (the so-called Gödel system T). Indeed, it can only define primitive recursive functionals (a generalization of the class of primitive recursive functions to a system with higher types), in particular functions that always terminate, and we do not have the power of general recursion available: it is not Turing complete. In fact, just as important in practice as mere **expressive power**[5] is the **practical convenience** that general recursion schemes provide for programmers: some primitive recursive functionals can be defined more conveniently using general recursion, as we do not have the burden of proving termination[6].

The reason the untyped λ-calculus was inconsistent as a logic turned out to be that the absence of types made so-called fixpoint combinators definable. We now know (as was already foreshadowed by Kleene's μ-recursive functions) that these are the crucial ingredient on top of primitive recursion in defining general computable functions.

One can explicitly add such fixpoint combinators to the syntax of a simply typed λ-calculus to have the expressive power of general recursion in a typed setting. However, the resulting language, known as PCF if we start from system T, is again inconsistent as a logic, as programs involving a fixpoint combinator do not

[4]Of course, we can add product and sum types to the simple λ-calculus to get a correspondence with full intuitionistic propositional logic.

[5]For instance, it is well-known that such a language of total functions cannot define its own interpreter. [5]

[6]For instance, writing a program that computes the same function as the simplex algorithm for linear programming would be possible in a language without general recursion by looping over the (finite) number of vertices in the polytope. However, such a solution would be more effort to implement and less efficient than the usual general recursive solution using a while loop as it would involve at the very least computing from the problem specification the number of vertices in the simplex.

correspond to acceptable proofs. Such programs which do not correspond to logical proofs are often called **effectful** and they are of crucial importance in real world software development. By contrast, programs corresponding to proofs are called **(purely) functional**. As purely functional code tends to be less error-prone and easier to reason about, a lot can be said in its favour.

However, in software engineering practice, pure functionality is often too much of a restriction, for reasons of efficiency, expressivity or mere programmer convenience. We have already seen the example of general recursion, which is an important feature in real programming languages. Another example of a class of effects are those that are introduced to give the programmer the option of more explicit low-level manipulation of the way the program is executed on the available hardware. A prime concrete example is the explicit manipulation of memory (or **state**). This can lead to a reduced (time and space) resource consumption. It can also make a certain algorithm easier to understand and implement for the programmer[7]. In other cases, effectful behaviour is an essential part of the specification of a program: for instance, we may want a program to generate a random number, to process keyboard input provided by the user, to generate output to a display or we may want to write a program that never terminates, like an operating system or a server, or to implement a counter.

We conclude that type theories require various extra terms, called (computational) effects, in order to constitute a practical programming language. Since these extra terms do not correspond to logical proofs, this renders the type theory of a practical programming language inconsistent as a logic.

1.1.4 Logic Requires More Types: Dependent Types

At the same time, we may observe that while many real world programming languages implementing such **effectful type theories** have a type system implementing the equivalent of the logical connectives of propositional logic (so-called simple types)

[7] An example is given by matrix multiplication. Of course, one can give a purely functional implementation, representing matrices as lists of lists on which we recurse, but it would be complicated and inefficient compared to the obvious imperative definition.

and even the equivalents of some higher-order predicates and quantifiers (so-called parametric polymorphism or generics), the equivalents of first-order predicates and quantifiers (so-called dependent types) are missing. However, these first-order quantifiers are of crucial importance in logical frameworks that are sufficiently expressive to be useful to formalise mathematics. From a programming point of view, such dependent types allow us to assign more precise types to existing programs of the simple λ-calculus, expressing detailed and useful program properties which a program that type checks is guaranteed to satisfy.

This means we are faced with a choice, at the moment: either we choose a language with many programs (an effectful programming language) while accepting a type system missing dependent types or we choose to have many types (a dependently typed language) and accept the lack of effects. All practical programming languages are in the former camp (e.g. Java, C++, Python, OCaml, Go) while the languages in this camp are inconsistent as a logic. The languages in the latter camp (e.g. Coq, Agda) can be useful as a logic or proof-assistant, but the lack of effects usually renders them impractical as programming languages.

1.1.5 Effects as Proofs of Modal Propositions

A first important issue to address if one wants to close the gap between programming languages and logics is the logical inconsistency introduced by effects. Effects need to be excluded from proofs, in order to retain their logical consistency (otherwise, using for instance general recursion, we could trivially construct a proof of any proposition), but not from programs. One possible way of solving this issue is to introduce new types of which the possibly effectful programs will be inhabitants, while keeping the inhabitants of other types pure. In such a formalism, not all types are propositions, just the ones whose terms are pure computations, not involving effects. In addition to restoring the logical consistency of the type theory, such a typing discipline makes it easier to reason about programs, as it is immediately clear from the type system which effects may occur in terms.

A particularly pleasing such formalism is given by **(strong) monads**, which can be used to encapsulate effects [6, 7]. The idea is that code is by default pure, unless specified otherwise by the type system. For example, a program of type $\mathbb{N} \Rightarrow \mathbb{N}$ is a primitive recursive functional from natural numbers to natural numbers, but a program of type $\mathbb{N} \Rightarrow T_{\mathsf{rec}}\mathbb{N}$ may be a general recursive function, where T_{rec} is a (strong) monad that makes fixpoint combinators available.

Particularly pleasing is that such (strong) monads T can in fact be given a logical interpretation. Under the Curry-Howard correspondence, they correspond to certain **diamond modalities** $\Diamond$ on the level of logic (sometimes called lax modalities and written $\bigcirc$) [8]. This means that we can interpret the effectful programs of type $A \Rightarrow TB$ as all the extra proofs of $A \Rightarrow \Diamond B$ that do not arise from proofs of $A \Rightarrow B$, if you will all the derivations starting from A of "possibly B" that aren't also proofs of B.

A point that is often elaborated on is that many such modalities may already be definable in a pure type theory. For instance, global state can be emulated with a modality $S \Rightarrow ((-) \times S)$, errors with a modality $(-) + E$, non-determinism with a powerset modality $\mathcal{P}(-)$ (if our pure type theory is a higher-order logic) and printing with a modality $(-) \times M$ where M is some internal monoid in the type theory. Another concrete example would be the double negation modality $\neg\neg(-)$ or, more generally, continuation modalities $((-) \Rightarrow R) \Rightarrow R$, which make the classical principle of Peirce's law (equivalent to double negation elimination) available from a logical point of view and the (universal) control operator `call/cc` from the point of view of programming languages [9]. In this sense, (constructive[8]) classical propositional logic is at the same time a simply typed programming language as well: not a purely functional one, but one enriched with constructs for non-local control flow. Another interesting modality $\mathsf{Dist}(-)$ is that of probability distributions, which, when definable in a pure type theory, lets us emulate probabilistic choice. It is interesting to note that one would define $\mathsf{Dist}(X)$, for a discrete type X, as

[8]Constructive in the sense that double negation elimination is not an isomorphism of types. This means that we avoid equating all terms of the same type, which would otherwise happen according to Joyal's lemma [10].

the type $\Sigma_{f:X \Rightarrow \mathbb{R}^+} \mathsf{Id}_{\mathbb{R}^+}(\int f, 1)$ of pairs of a positive real valued function $f : X \Rightarrow \mathbb{R}^+$ and a proof $p : \mathsf{Id}_{\mathbb{R}^+}(\int f, 1)$ that f sums (or integrates) to 1. We see that in order to define such modalities, we need, in particular, dependent type formers Σ and Id corresponding to existential quantification and identity predicates.

Such definable modalities allows us to **emulate** certain computational effects in a pure language. We would like to stress, however, that to treat effects **natively** with their intended custom operational semantics, rather than the emulation inherited from the conversions of the pure type theory, modalities should explicitly be added to the type system as new type formers and effects as new term formers which inhabit these types.

1.1.6 CBV, CBN and Half-Modalities

Recall that in pure functional languages the choice of an **evaluation strategy** does not effect the result of computations, merely their efficiency. This is why we call these languages declarative: they specify what should be computed, not how it should be computed. The user does not need to know about the how; this is left to the discretion of the compiler.

By contrast, the same is not generally true for languages with effects. Effects tend to bring us into the realm of imperative languages: the evaluation strategy (the order in which we evaluate the various parts of the program) can have a significant impact on the result of computations, so we need to think about language constructs not only in terms of what they compute but also in terms of how they compute it in time. Here, it is important for the user to know which strategy is being used.

Two strategies are particularly studied from a theoretical point of view: **call-by-value (CBV)** and **call-by-name (CBN)** evaluation. An important distinction between the two is that in CBN function arguments are only evaluated when they are needed, while in CBV they are always evaluated eagerly whether they are needed or not. CBN evaluation can sometimes be preferable from a performance or correctness point of view. On the other hand, in the presence of some effects, it can be difficult

to reason about, which is why CBV evaluation is often preferred as the default option in software engineering practice, with CBN being reserved for special situations.

An idea that we believe to be underemphasized in literature is the perspective that it is instructive to further decompose a monad T into an **adjunction** $F \dashv U$ (or the corresponding modality $\Diamond$ into a pair of "**half-modalities**") between two type theories of **values** on the one hand and **computations** (and more generally **stacks**) on the other, whose types we shall write $A, A', \ldots$ and $\underline{B}, \underline{B}', \ldots$ respectively. This is the point of view taken by Levy's **call-by-push-value (CBPV)**. The advantage is that we obtain an elegant language (simpler than a monadic language, in many ways) for proofs and effectful computations with a single intuitive canonical evaluation strategy that is expressive enough to encode both CBV and CBN and many things in between.

In particular, if we define the monad $T := UF$ and **comonad**[9] $! := FU$, we recover (thunks of) CBV computations as the terms of type $x : A \vdash a : TA'$ and the CBN computations as those of type $x :!\underline{B} \vdash b : \underline{B}'$. We see that the type system now also provides guarantees about the evaluation strategy of programs. Meanwhile, proofs can still be interpreted as general terms $x : A \vdash a : A'$ (including the proofs of modal propositions which can also be read as thunked call-by-value computations).

1.1.7 The Relationship Between Proving and Programming

It is clear that a blunt statement of the Curry-Howard correspondence like "a programming language is the same as a logic" is far from the truth. In fact, even the weaker statement, which may be closer to the truth, that "every logic extends to a programming language" does not accurately reflect the reality of programming languages research at the moment, although it may be a possible future that the field is trying to realise.

The relationship between programming languages and logics may be more accurately summarised by figure 1.2, where we refer to pure languages without

[9] This corresponds to a certain **box modality** if we try to give the type theory for computations a logical interpretation. As we shall see, it can be understood to define a certain generalization of linear logic, hence the notation ! for the comonad.

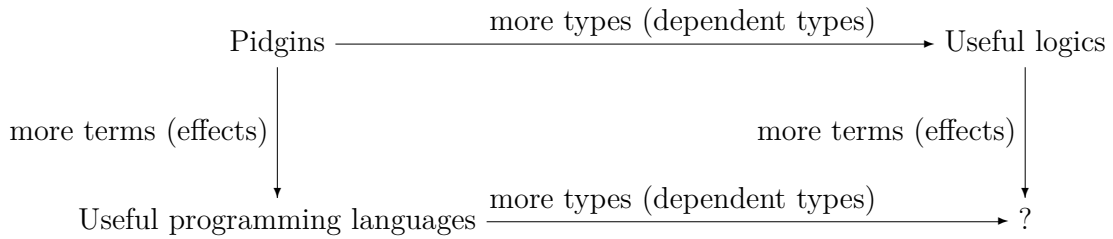


Figure 1.2: The present relationship between type theories that can serve as a logic and as a programming language: it is not clear what sort of type theory would be satisfactory as both.

dependent types as **pidgins** (e.g. the pure polymorphic λ-calculus) as they can be seen as simplified languages that can be interpreted both as a programming language and as a logic but are not entirely satisfactory as either. So far, however, it is not yet clear if there exists a Promised Land of genuine **programming logics**, languages that can serve as both a useful programming language and logic by combining dependent types and effects.

What is clear is that there is an inherent tension between the extensions of a pidgin with effects and with dependent types. The extra terms introduced by the former allow for wilder kinds of program behaviour while the extra types introduced by the latter serve to tame the behaviour of a program.

1.1.8 Why Unify Proving and Programming?

One might wonder why we should be looking for such a Promised Land at all. In fact, who promised such a land in the first place?

Firstly, it is of fundamental importance to both the disciplines of mathematical logic and programming language research to be very precise about the relationship between mathematical proofs and computer programs. The promise of a unification of proving and programming has been repeatedly made either implicitly or explicitly, given how intertwined both disciplines have been historically, how much cross fertilisation has taken place and how many parallels have been sketched (often under the name of a Curry-Howard correspondence). It is stunning that no precise result exists yet to either show how to unify the notions of logic (using dependent types) and practical programming (using effects) in a single language or to show

that this cannot be done satisfactorily. Such a marriage or the demonstration of its impossibility would provide conceptual clarity about the foundations about two important disciplines that have been flirting with each other for eighty years.

Secondly, a combined system with dependent types and effects could provide a very useful practical framework for writing verified software. It may give us a single language to both write real world programs (making use of effects) and prove their correctness (using the expressive logic embedded in the type system, using dependent types). Currently, this often needs to happen in two separate systems: a programming language and a proof assistant or another verification tool. We are required to build a model of the program in the verification tool in order to prove its properties. This transcription results in an overhead of work as well as in an extra source of potential bugs[10]. It is both safer and more efficient to directly prove properties of the production code.

1.2 Goals of This Thesis

The distal goal that this thesis can be understood to be pursuing is an understanding of the precise relationship between logic and programming. The main motivating questions for this line of work are the following three.

- How should we understand the relationship between logic and programming?
- Can we design languages that are simultaneously satisfactory as a programming language and as a logic and in which both aspects of the language interact in a meaningful way?
- Can we use such a language for certified real world programming?

The desire to answer these questions leads us to the more proximal goal of understanding how dependent types (from the realm of logic) can be combined with computational effects (which define real world programming languages):

[10] In fact, it turns out that the construction of such a model can in some cases be automated using game semantics [11].

- Can we combine dependent types and computational effects in an elegant and meaningful way?

We believe the goals and questions we are pursuing are of tremendous importance both from a fundamental academic point of view and from the concrete point of view of software engineering. If these hugely ambitious questions had a straightforward answer, the community would have found it a long time ago. This thesis only claims to make a small contribution to solving this difficult puzzle, while hoping to illustrate both its relevance and complexity.

Concretely, this thesis describes three closely related lines of work:

1. Studying a dependently typed version of **linear logic**, in the sense of a dependent type theory in which terms cannot be copied or discarded freely;

2. Providing a **game semantics** for dependent type theory, interpreting types as games and their terms as strategies on these games;

3. Studying a dependently typed version of Levy's **call-by-push-value** in the presence of various effects.

These are closely related to the goals of thesis. Indeed, firstly, CBPV is a very useful paradigm for understanding effectful languages and their relationship to logic as it gives us a fine-grained way of controlling where effects are allowed to occur (and in what order they should be evaluated) and what parts of a program should be pure.

Secondly, effectful computations and the stacks used to evaluate these behave linearly in some sense. To be precise, they cannot be discarded in the syntax of CBPV[11]. On a more conceptual level, we like to point out that effectful computations can be **dynamic**, in the sense that their reductions generally break equality, for instance for a non-deterministic choice we can have a reduction

[11]This corresponds to the structural rule of weakening not being valid. We generally, for non-commutative effects, only consider contexts of at most one identifier of computation/stack type $\underline{B}$, meaning that the rule of contraction does not have any meaning. We shall later see, in theorems 2.2.5 and 2.3.5, that we can conservatively extend the syntax for stacks with a multiplicative conjunction $\otimes$, or, equivalently, with linear contexts of longer length, if we are dealing with only commutative effects.

$\mathsf{choose}(\mathsf{return}\ \mathsf{tt}, \mathsf{return}\ \mathsf{ff}) \rightsquigarrow \mathsf{return}\ \mathsf{ff}$ where the result, after the choice has been made, should clearly not be considered 'equal' to the initial computation before the program makes a non-deterministic choice. This should be contrasted with the **static** nature of values or pure computations (whose normalization does not break equations). Dynamic objects, in particular, cannot be copied in the usual sense, as both copies might later cease to be equal, and are in that sense linear. In fact, we shall argue that linear logic can be seen as a type system for commutative effectful computations.

Thirdly, game semantics has been perhaps the most successful paradigm for providing a unified intuitive semantics for many effectful programming languages and pure logics. We can hope to gain useful semantic intuitions for the problem of how to relate effects to dependent types, here. Moreover, game semantics naturally arises from a model of linear logic. Recall that game semantics is naturally effectful in the sense that computational effects like state, non-termination and non-local control have to be explicitly excluded by putting conditions on the strategies we consider on games. Therefore, we believe, the current absence of a game semantics for dependent types reflects the same lack of fundamental understanding of how to relate logic to programming, particularly the question of how type dependency should interact with effectful computations.

We encounter similar possibilities and obstacles in all three lines of work – for instance, we need to decide if it makes sense to have types depend on dynamic or linear objects – and the simultaneous study of these three topics has hugely helped us to see a bigger picture emerge of what the Promised Land of genuine programming logics might look like. We hope it will do the same for the reader.

1.3 Key Contributions

This thesis makes the following key contributions:

- Explaining the difficulty of combining dependent types with linear types, game semantics and effects and presenting a way of still doing so in the following sense;

- Developing a syntax for dependently typed dual intuitionistic linear logic (dDILL);
- Developing a categorical semantics for dDILL and the dependently typed linear/non-linear (dLNL) calculus;
- Developing a range of concrete models for dDILL and dLNL, including a coherence space semantics;
- Explaining the relationship with commutative effects;
- Presenting a game semantics for dependent type theory;
- Showing it has strong (full and faithful) completeness properties with respect to CBN dependent type theory;
- Examining effectful game models of dependent types;
- Presenting a dependently typed call-by-push-value (dCBPV-) calculus;
- Developing its categorical semantics;
- Showing that dLNL models give models of dCBPV-, as do algebras for indexed monads on models of pure dependent type theory;
- Showing that the operational semantics of dCBPV- is well-behaved;
- Showing that we need to extend dCBPV- to dCBPV+ with dependent Kleisli extensions if we want CBV and CBN translations;
- Showing that dCBPV+ is less straightforward than dCBPV- from the point of view of operational semantics and concrete categorical models...;
- ... and that the same goes for dCBPV- extended with dependent projection products (additive Σ-types);
- As an alternative, presenting a dependently typed enriched effect calculus (dEEC) and showing it to be very well-behaved.

In the course of his doctoral studies, the author has communicated the majority of the material included in this thesis in [12–17] and in various oral presentations.

1.4 Thesis Outline

We have chosen to present this work in the order in which the research was conducted, to best convey to the reader the author's motivations for studying the various topics. Chapter 2 provides background material on CBPV, linear logic and game semantics, all in simply typed form, as well as on (cartesian) dependent type theory. Most material in these sections is not original, but we present it in a novel way, in order to ensure a smooth transition to the rest of this thesis. Our first pillar of original work is presented in chapter 3, which discusses a dependently typed version of linear logic. This naturally leads us to chapter 4, where we discuss our second line of work: a game semantics for dependent type theory. Our third and last topic, a discussion of dependently typed CBPV, can be found in chapter 5. We end on a discussion of our conclusions and future work in chapter 6.

We have tried to keep chapters 3, 4 and 5 as self-contained as possible (apart from their dependence on the appropriate sections of chapter 2). Historically, our three lines of work roughly relate to each other as follows. Following a question by Samson Abramsky, we set off to construct a game semantics for dependent type theory or to understand why none existed yet. As categories of games originate from models of linear logic (categories of cofree !-coalgebras), this pushed us to investigate the relationship between linearity and dependent types first. Later, we came to understand the tension between game semantics and dependent types as arising from the natural effectful character of unrestricted strategies. This understanding made clear to us our bias in studying type dependency only in CBN game semantics and generally focussing on Girard's first (CBN) translation into linear logic. This finally led to our study of dependently typed CBPV, which we now understand, after realising that linear logic can be read as a calculus for commutative effects, as giving a generalization of our work on linear dependent type theory to non-commutative effects, providing, additionally, an account of operational semantics.

16

Knowledge is knowing that a tomato is a fruit; wisdom is not putting it in a fruit salad.

— Miles Kington

2 Preliminaries

In this chapter, we present our views on simply typed linear logic, game semantics and call-by-push-value, as well as on their relation to each other, in order to easily be able to extend all three with a notion of type dependency in later chapters. We start with a discussion of cartesian type theory, however. The material in this chapter mostly consists of definitions and results published by other authors as well as folklore results. In most cases, however, it is reformulated in a non-trivial way in order to make developments in further chapters go through as smoothly as possible. We hope that this novel presentation of known results can be of value in its own right.

2.1 Cartesian Type Theory

We briefly recall the syntax and semantics of simple (STT) and dependent type theory (DTT). We describe a general syntactic and semantic framework for both, in the context of which we can consider many theories (in the case of syntax) or models (in the case of semantics). We discuss, in particular, two CBN type theories $\mathsf{STT}_{\mathsf{CBN}}$ and $\mathsf{DTT}_{\mathsf{CBN}}$, with respect to which the game theoretic models we discuss in section 2.4 and chapter 4 have full and faithful completeness properties[1].

[1]These are CBN type theories in the sense that we only demand a limited η-rule with some (but not all) commutative conversions for ground types. As we shall see in section 2.2, the full η-law is typically broken in effectful settings under CBN evaluation. If we were to demand the fully general η-rule (which would automatically imply all commutative conversions), we would be

2.1.1 Syntax of Type Theories

2.1.1.1 Dependently Typed Equational Logic

In this section, we briefly recall the framework of dependently typed equational logic (sometimes called generalised algebraic theories [18]), which will serve as the structural core type theory and on top of which we later consider two theories in particular: a flavour of simple type theory ($\mathsf{STT}_{\mathsf{CBN}}$) and a flavour of dependent type theory ($\mathsf{DTT}_{\mathsf{CBN}}$). This framework puts both flavours of type theory on an equal footing and allows us to better study their relationship. We go into this level of precision in our specification of the syntax we are modelling, in order to accurately state the appropriate completeness results in sections 2.4 and 4.5. Although much more informal, our treatment is close in spirit to those of [19] and [20], to which we refer the interested reader for more background and where the reader can find details on delicate topics like pre-syntax, α-conversion, identifier binding and capture-avoiding substitution.

The key feature of a dependent type system is that we allow types to refer to free identifiers from the context. The reader may want to keep in mind the analogy that dependent types are to predicates what non-dependent types are to propositions. One consequence is that order in the context becomes important as all free identifiers in a type need to be declared in the context to its left. As types can depend on terms in a dependently typed system and equations of terms can lead to equations of types which can lead to new typing judgements, we define all judgements together in one big inductive definition.

Judgements

Figure 2.1 presents the various kinds of **judgements** of dependently typed equational logic and their intended meaning. Here, $\Gamma, \Gamma', A, , A', a$ and a' are all symbolic expressions from a set Expr, built from an alphabet Sym, in which we have countably infinite designated subsets Idf of identifiers and Cons of constants. As usual, we

modelling pure type theory. We briefly note that the usual set theoretic semantics is fully and faithfully complete for this pure type theory.

Judgement	Intended meaning
$\vdash \Gamma$ ctxt	Γ is a valid context
$\Gamma \vdash A$ type	A is a type in context Γ
$\Gamma \vdash a : A$	a is a term of type A in context Γ
$\vdash \Gamma = \Gamma'$	Γ and Γ' are judgementally equal contexts
$\Gamma \vdash A = A'$	A and A' are judgementally equal types in context Γ
$\Gamma \vdash a = a' : A$	a and a' are judgementally equal terms of type A in context Γ

Figure 2.1: Judgements of dependently typed equational logic.

$$\frac{\Gamma, \Gamma' \vdash \mathcal{J} \qquad \vdash \Gamma, x : A, \Gamma' \text{ ctxt}}{\Gamma, x : A, \Gamma' \vdash \mathcal{J}}\ \textsf{Weak} \qquad \frac{\Gamma, x : A, \Gamma' \vdash \mathcal{J} \qquad \Gamma \vdash a : A}{\Gamma, \Gamma'[a/x] \vdash \mathcal{J}[a/x]}\ \textsf{Subst}$$

(a) Weakening and substitution rules. Here, $\mathcal{J}$ represents a statement of the form B type, $B = B'$, $b : B$, or $b = b' : B$. Note that these rules, additionally, make contraction and exchange rules derivable.

$$\frac{}{\vdash \cdot \text{ ctxt}}\ \textsf{C-Emp} \qquad \frac{\vdash \Gamma \text{ ctxt} \qquad \Gamma \vdash A \text{ type} \qquad x \text{ is fresh for } \Gamma \text{ and } A}{\vdash \Gamma, x : A \text{ ctxt}}\ \textsf{C-Ext} \qquad \frac{\vdash \Gamma, x : A, \Gamma' \text{ ctxt}}{\Gamma, x : A, \Gamma' \vdash x : A}\ \textsf{Idf}$$

$$\frac{\Gamma = \Gamma' \text{ ctxt} \qquad \Gamma \vdash A = B \qquad \vdash \Gamma, x : A \text{ ctxt} \qquad \vdash \Gamma', x : B \text{ ctxt}}{\vdash \Gamma, x : A = \Gamma', x : B}\ \textsf{C-Ext-Eq}$$

(b) Context formation and identifier declaration rules.

$$\frac{\vdash \Gamma \text{ ctxt}}{\vdash \Gamma = \Gamma}\ \textsf{C-Eq-R} \qquad \frac{\vdash \Gamma = \Gamma'}{\vdash \Gamma' = \Gamma}\ \textsf{C-Eq-S} \qquad \frac{\vdash \Gamma = \Gamma' \qquad \vdash \Gamma' = \Gamma'' \text{ ctxt}}{\vdash \Gamma = \Gamma''}\ \textsf{C-Eq-T}$$

$$\frac{\Gamma \vdash A \text{ type}}{\Gamma \vdash A = A}\ \textsf{Ty-Eq-R} \qquad \frac{\Gamma \vdash A = A'}{\Gamma \vdash A' = A}\ \textsf{Ty-Eq-S} \qquad \frac{\Gamma \vdash A = A' \qquad \Gamma \vdash A' = A''}{\Gamma \vdash A = A''}\ \textsf{Ty-Eq-T}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash a = a : A}\ \textsf{Tm-Eq-R} \qquad \frac{\Gamma \vdash a = a' : A}{\Gamma \vdash a' = a : A}\ \textsf{Tm-Eq-S} \qquad \frac{\Gamma \vdash a = a' : A \qquad \Gamma \vdash a' = a'' : A}{\Gamma \vdash a = a'' : A}\ \textsf{Tm-Eq-T}$$

$$\frac{\Gamma \vdash A \text{ type} \qquad \vdash \Gamma = \Gamma' \text{ ctxt}}{\Gamma' \vdash A \text{ type}}\ \textsf{Ty-Conv} \qquad \frac{\Gamma \vdash a : A \qquad \vdash \Gamma = \Gamma' \text{ ctxt} \qquad \Gamma; \cdot \vdash A = A' \text{ type}}{\Gamma' \vdash a : A'}\ \textsf{Tm-Conv}$$

$$\frac{\Gamma \vdash a = a' : A \qquad \Gamma, x : A, \Gamma' \vdash B \text{ type}}{\Gamma, \Gamma'[a/x] \vdash B[a/x] = B[a'/x] \text{ type}}\ \textsf{Ty-Cong} \qquad \frac{\Gamma \vdash a = a' : A \qquad \Gamma, x : A, \Gamma' \vdash b : B}{\Gamma, \Gamma'[a/x] \vdash b[a/x] = b[a'/x] : B}\ \textsf{Tm-Cong}$$

(c) Rules for judgemental equality, making it a congruence relation, compatible with typing.

Figure 2.2: The structural rules of dependently typed equational logic.

distinguish between the free and bound identifiers occurring in an expression $\mathcal{J}$ and we consider expressions $\mathcal{J}$ up to α-equivalence, or up to permutations of Idf fixing the free identifiers of $\mathcal{J}$. We denote the syntactic metaoperation of capture-avoiding substitution of an expression a for all occurrences of a free identifier x in an expression $\mathcal{J}$ by $\mathcal{J}[a/x]$.

Structural Rules and Theories

Dependently typed equational logic has the structural rules presented in figure 2.2, which will be shared in particular by $\mathsf{STT}_{\mathsf{CBN}}$ and $\mathsf{DTT}_{\mathsf{CBN}}$. We can use our

framework to talk about various type theories. By a **theory**, we mean a set $\mathbb{T}$ of judgements which is closed under the structural rules above, in the sense that their conclusions (written under the horizontal line) are in $\mathbb{T}$ if their hypotheses (written above) are. Usually, we specify a theory by a set of **axioms**, a set of judgements which can be inductively closed under the structural rules to obtain a theory. For our purposes, we only consider theories with no context symbols. That is, all our contexts consist of lists of type declarations for identifiers and context equalities consist of type equalities and identifier equalities.

2.1.1.2 A Simple Type Theory, $\mathsf{STT}_{\mathsf{CBN}}$

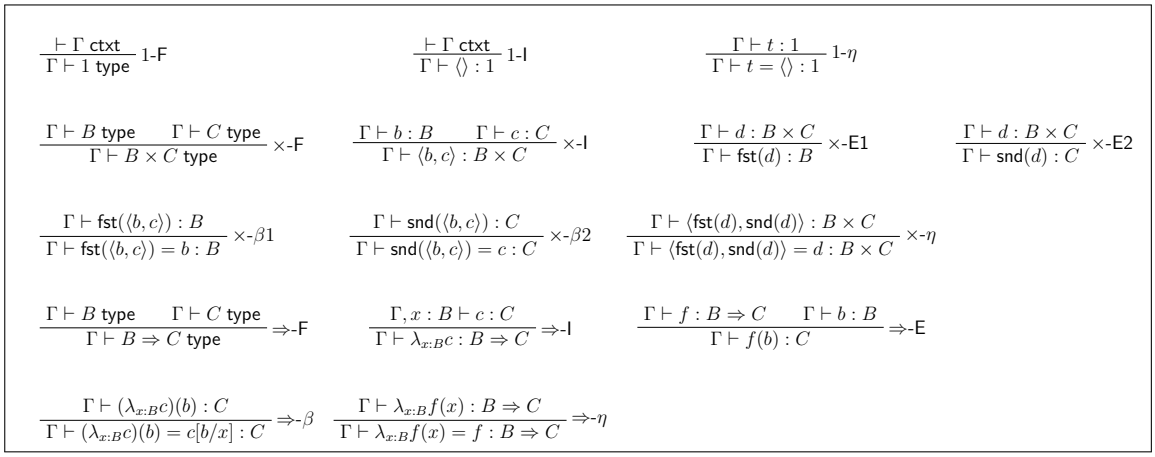
The simple type theory we use is a variant $\mathsf{STT}_{\mathsf{CBN}}$ of the simply typed λ-calculus with finite product types and finite inductive[2] types $\{a_i \mid i\}$ for any finite set of **distinct** constants $a_1, \ldots, a_n$, with β- and η'-rules[3] and certain commutative conversions for the corresponding case-constructs – essentially the PCF commutative conversions [22] (section 3.2). We are considering a total finitary PCF, if you will. Specifically, with $\mathsf{STT}_{\mathsf{CBN}}$, we are referring to the theory in dependently typed equational logic generated by the rules of figure 2.3. Note that the rule that $\Gamma \vdash t = u : A$ implies that $\Gamma \vdash t : A$ is admissible.

2.1.1.3 A Dependent Type Theory, $\mathsf{DTT}_{\mathsf{CBN}}-$

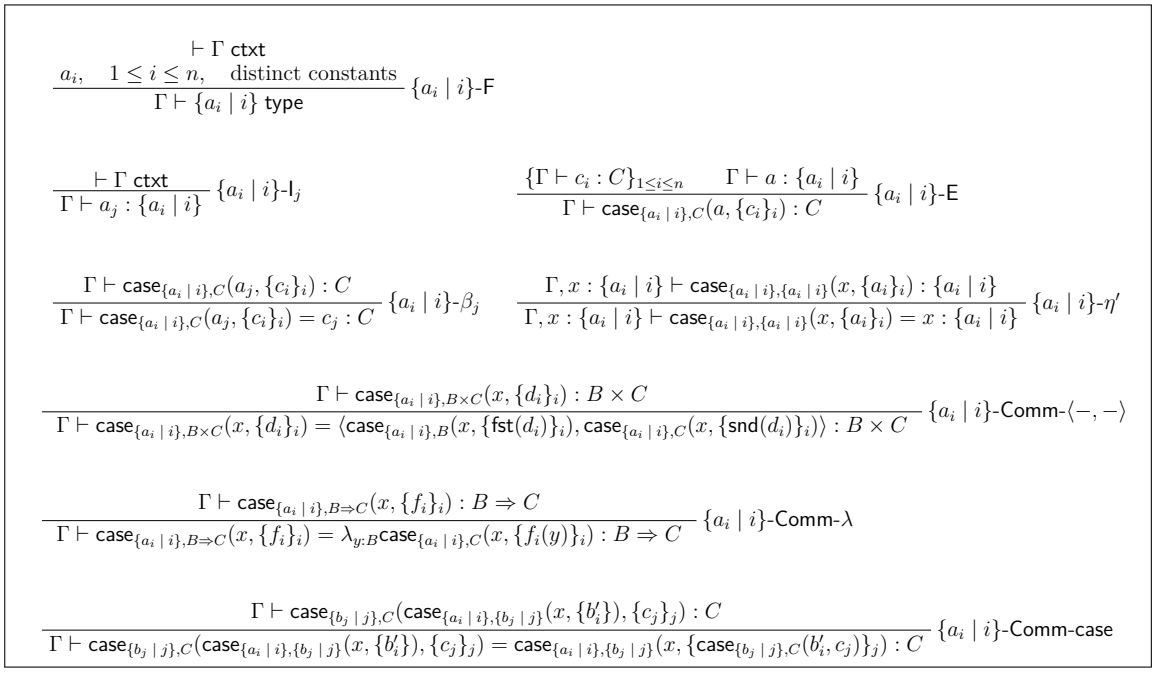
Similarly, we can present our preferred variant $\mathsf{DTT}_{\mathsf{CBN}}$ of dependent type theory as a theory in dependently typed equational logic. First, we present a smaller

[2]We use this terminology as we see them as a specific instance of general inductive types, to which one might want to generalise in future work.

[3]Note that we are using a restricted form of the η-rule for inductive types which we call η'. This is why we are left to impose certain commutative conversions, which (among other things) would be implied by the general η-rule $\mathsf{case}_{\{a_i \mid i\},\{a_i \mid i\}}(x, \{b[a_i/x]\}_i) = b$. More discussion of the matter of commutative conversions and η-rules can be found in [21]. Our equational theory is easily seen to precisely correspond precisely to observational equivalence if we extend the syntax with some sufficiently evil computational effect (in fact, it can be shown using an embedding into CBPV that no other effect can weaken the equational theory of pure type theory further) like printing or state and use CBN evaluation. We present this equational theory as it will precisely correspond to equality in CBN game semantics. As a rule of thumb, we would like to note that in the presence of effects (which can be modelled in game semantics) the general η-laws fail for positive connectives in CBN and for negative connectives in CBV. This is one of the mysteries that CBPV addresses.



(a) Formation (F), introduction (I), elimination (E) and β- and η-conversion rules for the usual connectives of simple type theory. For $\Rightarrow$ $-\eta$, we demand the usual side condition that x not free in f.



(b) The rules for a notion of ground types for simple type theory: finite inductive types.

Figure 2.3: The rules generating the axioms for $\mathsf{STT}_{\mathsf{CBN}}$.

theory $\mathsf{DTT}_{\mathsf{CBN}}-$, which does not yet include the β- and η-rules and commutative conversions of $\mathsf{DTT}_{\mathsf{CBN}}$, but rather only consists of its F-, I- and E-rules. Later, $\mathsf{DTT}_{\mathsf{CBN}}$ is obtained by adding to $\mathsf{DTT}_{\mathsf{CBN}}-$ the equational theory that results from that of $\mathsf{STT}_{\mathsf{CBN}}$, under a syntactic translation to $\mathsf{STT}_{\mathsf{CBN}}$.

$\mathsf{DTT}_{\mathsf{CBN}}-$ consists of the rules of figure 2.4. In addition to term and type formation rules for Σ-, Π- and Id-types, we have a mechanism for forming finite

$$\frac{\vdash \Gamma\ \mathsf{ctxt}}{\vdash 1\ \mathsf{type}}\ \text{1-F} \qquad \frac{\vdash \Gamma\ \mathsf{ctxt}}{\Gamma \vdash \langle\rangle : 1}\ \text{1-I}$$

$$\frac{\Gamma, x : A \vdash B\ \mathsf{type}}{\Gamma \vdash \Sigma_{x:A} B\ \mathsf{type}}\ \Sigma\text{-F} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B[a/x]}{\Gamma \vdash \langle a, b\rangle : \Sigma_{x:A} B}\ \Sigma\text{-I} \qquad \frac{\Gamma \vdash t : \Sigma_{x:A} B}{\Gamma \vdash \mathsf{fst}(t) : A}\ \Sigma\text{-E1} \qquad \frac{\Gamma \vdash t : \Sigma_{x:A} B}{\Gamma \vdash \mathsf{snd}(t) : B[\mathsf{fst}(t)/x]}\ \Sigma\text{-E2}$$

$$\frac{\Gamma, x : A \vdash B\ \mathsf{type}}{\Gamma \vdash \Pi_{x:A} B\ \mathsf{type}}\ \Pi\text{-F} \qquad \frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda_{x:A} b : \Pi_{x:A} B}\ \Pi\text{-I} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash f : \Pi_{x:A} B}{\Gamma \vdash f(a) : B[a/x]}\ \Pi\text{-E}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash a' : A}{\Gamma \vdash \mathsf{Id}_A(a, a')\ \mathsf{type}}\ \mathsf{Id}\text{-F} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl}(a) : \mathsf{Id}_A(a, a)}\ \mathsf{Id}\text{-I}$$

$$\frac{\begin{array}{l}\Gamma \vdash a : A \\ \Gamma \vdash a' : A \qquad\qquad \Gamma, x : A, x' : A, y : \mathsf{Id}_A(x, x') \vdash D\ \mathsf{type} \\ \Gamma \vdash p : \mathsf{Id}_A(a, a') \quad \Gamma, z : A \vdash d : D[z/x, z/x', \mathsf{refl}(z)/y]\end{array}}{\Gamma \vdash \mathsf{let}\ p\ \mathsf{be}\ \mathsf{refl}(z)\ \mathsf{in}\ d : D[a/x, a'/x', p/y]}\ \mathsf{Id}\text{-E}$$

(a) Rules for 1-, Σ-, Π-, and Id-types. In case x is not free in B, we sometimes write $A \Rightarrow B$ for $\Pi_{x:A} B$ and $A \times B$ for $\Sigma_{x:A} B$.

$$\frac{\vdash \Gamma\ \mathsf{ctxt} \qquad \vdash a_1 : A \qquad \ldots \qquad \vdash a_n : A \qquad b_{i,j},\ \ 1 \le i \le n,\ 1 \le j \le m_i,\ \text{distinct constants}}{\Gamma, x : A \vdash (a_i \mapsto_i \{b_{i,j} \mid j\})(x)\ \mathsf{type}}\ (a_i \mapsto_i \{b_{i,j} \mid j\})(x)\text{-F}$$

$$\frac{\vdash \Gamma\ \mathsf{ctxt}}{\Gamma \vdash b_{i,j} : (a_i \mapsto_i \{b_{i,j} \mid j\})(a_i)}\ (a_i \mapsto_i \{b_{i,j} \mid j\})(x)\text{-I}_{i,j}$$

$$\frac{\begin{array}{ll}\Gamma \vdash a : A & x : A, y : (a_i \mapsto_i \{b_{i,j} \mid j\})(x) \vdash C\ \mathsf{type} \\ \Gamma \vdash b : (a_i \mapsto_i \{b_{i,j} \mid j\})(a) & \{\Gamma \vdash c_{i,j} : C[a_i/x, b_{i,j}/y]\}_{i,j}\end{array}}{\Gamma \vdash \mathsf{case}_{(a_i \mapsto_i \{b_{i,j} \mid j\})(a), C}(b, \{c_{i,j}\}_{i,j}) : C[a/x, b/y]}\ (a_i \mapsto_i \{b_{i,j} \mid j\})(x)\text{-E}$$

$$\frac{\begin{array}{ll}\Gamma \vdash a : A & \Gamma, y : (a_i \mapsto_i \{b_{i,j} \mid j\})(a) \vdash C\ \mathsf{type} \\ \Gamma \vdash b : (a_i \mapsto_i \{b_{i,j} \mid j\})(a) & \{\Gamma, p_{i,j} : \mathsf{Id}_A(a_i, a), q_{i,j} : \mathsf{Id}_{(a_i \mapsto_i \{b_{i,j} \mid j\})(a)}(\mathsf{subst}(p, b_{i,j}), b) \vdash c_{ij} : C[b/y]\}_{i,j}\end{array}}{\Gamma \vdash \mathsf{case}^{p,q}_{(a_i \mapsto_i \{b_{i,j} \mid j\})(a), C}(b, \{c_{i,j}\}_{i,j}) : C[b/y]}\ (a_i \mapsto_i \{b_{i,j} \mid j\})(x)\text{-E'}$$

(b) Rules for a finite inductive type family $x : A \vdash (a_i \mapsto_i \{b_{i,j} \mid j\})(x)\ \mathsf{type}$, generated by $\vdash b_{i,1}, \ldots, b_{i,m_i} : (a_i \mapsto_i \{b_{i,j} \mid j\})(x)[a_i/x]$ for $\vdash a_1, \ldots, a_n : A$.

Figure 2.4: The rules generating the axioms for $\mathsf{DTT}_{\mathsf{CBN}}-$.

inductive type families, which play the rôle of ground types. Let $\vdash A\ \ \mathsf{type}$[4]. Then, we can give a finite inductive definition of a type family $x : A \vdash (a_i \mapsto_i \{b_{i,j} \mid j\})(x)\ \mathsf{type}$ by specifying finitely many closed terms $a_1, \ldots, a_n : A$ and distinct symbols $b_{i,j}$, $1 \le i \le n$, $1 \le j \le m_i$. The idea is that $B = (a_i \mapsto_i \{b_{i,j} \mid j\})(x)$ is a type family, such that $(a_i \mapsto_i \{b_{i,j} \mid j\})(a_i)$ contains precisely the distinct closed terms $b_{i,1}, \ldots, b_{i,m_i}$. These type families are more limited than general inductive definitions as they are freely generated by **closed** terms, while one would allow open terms in the general case [23]. This means that we precisely get the inductive type families with finitely many non-empty fibres which are all finite types. An example the reader may want to keep in mind is given by calendars in format dd-mm (for the year 1984, for instance): here $A = \mathsf{mm} := \{01, \ldots, 12\}$ and

[4]Perhaps, it would be more elegant to allow the specification of an inductive type family depending on an arbitrary context $\vdash x_1 : A_1, \ldots, x_n : A_n\ \ \mathsf{ctxt}$ rather than a single type. However, given that we consider a system with strong Σ-types, the two are equivalent and only letting inductive families depend on a single types allows us to keep notation more lightweight.

$B = \mathsf{dd} - \mathsf{mm} := (i \mapsto_i \{01\text{-}i, \ldots, N_i\text{-}i\})(x)$, where N_i is 29, 30, or 31, depending on the number of days the month in question has.

We interpret such a definition as specifying F-, I- and E-rules for $(a_i \mapsto_i \{b_{i,j} \mid j\})(x)$. In fact, instead of $(a_i \mapsto_i \{b_{i,j} \mid j\})(x)$-$E$, we may equivalently specify an alternative elimination rule $(a_i \mapsto_i \{b_{i,j} \mid j\})(x)$-$E'$. While the former is the usual elimination rule for finite inductive type families, the latter is closer, in a sense, to the intuition of our model and arises naturally in the completeness proofs in chapter 4. Here, we write subst for the following principle of indiscernability of identicals.

$$\frac{\dfrac{\Gamma, x : A \vdash B\ \mathsf{type}}{\Gamma, x : A \vdash \lambda_{y:B} y : \Pi_{y:B} B} \quad \dfrac{}{\Gamma, x, x' : A, p : \mathsf{Id}_A(x, x') \vdash x, x' : A} \quad \dfrac{}{\Gamma, x, x' : A, p : \mathsf{Id}_A(x, x') \vdash p : \mathsf{Id}_A(x, x')}}{\Gamma, x, x' : A, p : \mathsf{Id}_A(x, x') \vdash \mathsf{subst}(p, -) : \Pi_B B[x'/x]}\ \mathsf{Id}\text{-}E$$

More generally, for a context $\Gamma, x : A, y_1 : B_1, \ldots, y_n : B_n$, we can inductively define

$$\Gamma, x, x' : A, p : \mathsf{Id}_A(x, x'), y_1 : B_1, \ldots, y_{n-1} : B_{n-1} \vdash \mathsf{subst}(p, -) : \Pi_{y_n:B_n} B_n[x'/x, \ldots, \mathsf{subst}(p, y_i)/y_i, \ldots].$$

We note that $(a_i \mapsto_i \{b_{i,j} \mid j\})(x)$-$E$ and $(a_i \mapsto_i \{b_{i,j} \mid j\})(x)$-$E'$ really are equivalent in a precise sense.

Theorem 2.1.1. *We have translations between $(a_i \mapsto_i \{b_{i,j} \mid j\})(x)$-$E$ and $(a_i \mapsto_i \{b_{i,j} \mid j\})(x)$-$E'$. These become mutually inverse in the equational theory of $\mathsf{DTT}_{\mathsf{CBN}}$.*

Proof. Let us write B for $(a_i \mapsto_i \{b_{i,j} \mid j\})(x)$. In the presence of B-E', we can define B-E by noting that

$$\frac{\dfrac{\dfrac{}{x : A, y : B, z_{1,1} : C[a_1/x, b_{1,1}/y], \ldots, z_{n,m_n} : C[a_n/x, b_{n,m_n}/y] \vdash z_{i,j} : C[a_i/x, b_{i,j}/y]}}{x : A, y : B, z_{1,1} : C[a_1/x, b_{1,1}/y], \ldots, z_{n,m_n} : C[a_n/x, b_{n,m_n}/y], p_{i,j} : \mathsf{Id}_A(a_i, x), q : \mathsf{Id}_B(\mathsf{subst}(p_{i,j}, b_{i,j}), y) \vdash z_{i,j} : C[a_i/x, b_{i,j}/y]}}{x : A, y : B, z_{1,1} : C[a_1/x, b_{1,1}/y], \ldots, z_{n,m_n} : C[a_n/x, b_{n,m_n}/y], p_{i,j} : \mathsf{Id}_A(a_i, x), q_{i,j} : \mathsf{Id}_B(\mathsf{subst}(p_{i,j}, b_{i,j}), y) \vdash \mathsf{subst}(q_{i,j}, \mathsf{subst}(p_{i,j}, z_{i,j})) : C}$$

and applying B-E' with

$A' = \Sigma_{x:A}\Sigma_{y:B}\Sigma_{z_{1,1}:C[a_1/x,b_{1,1}/y]} \cdots \Sigma_{z_{n,m_n-1}:C[a_n/x,b_{n,m_n-1}/y]} C[a_n/x, b_{n,m_n}/y]$, $a = x$ (the projection to A), $b = y$ (the projection to B) and the from $z_{i,j}$ derived expression above for $c_{i,j}$ to derive B-E'.

Conversely, in the presence of B-E, we derive B-E':

$$\dfrac{\dfrac{\dfrac{\{x' : A', p_{i,j} : \mathsf{Id}_A(a_i, a), q_{i,j} : \mathsf{Id}_{B[a/x]}(\mathsf{subst}(p_{i,j}, b_{i,j}), b) \vdash c_{i,j} : C[b/y]\}_{i,j}}{\{\vdash \lambda_{x':A'}\lambda_{p_{i,j}:\mathsf{Id}_A(a_i,a)}\lambda_{q_{i,j}:\mathsf{Id}_{B[a/x]}(\mathsf{subst}(p_{i,j},b_{i,j}),b)}c_{i,j} : \Pi_{x':A'}\Pi_{p_{i,j}:\mathsf{Id}_A(a_i,a)}\Pi_{q_{i,j}:\mathsf{Id}_{B[a/x]}(\mathsf{subst}(p_{i,j},b_{i,j}),b)}C[b/y]\}_{i,j}}}{x : A, y' : B \vdash \mathsf{case}_{B,\Pi_{x':A'}\Pi_{p:\mathsf{Id}_A(x,a)}\Pi_{q:\mathsf{Id}_{B[a/x]}(\mathsf{subst}(p,y'),b)}C[b/y]}(y', \{\lambda_{x':A'}\lambda_{p_{i,j}:\mathsf{Id}_A(a_i,a)}\lambda_{q_{i,j}:\mathsf{Id}_{B[a/x]}(\mathsf{subst}(p_{i,j},b_{i,j}),b)}c_{i,j}\}) : \Pi_{x':A'}\Pi_{p:\mathsf{Id}_A(x,a)}\Pi_{q:\mathsf{Id}_{B[a/x]}(\mathsf{subst}(p,y'),b)}C[b/y]}}{x' : A' \vdash \mathsf{case}_{B,\Pi_{x':A'}\Pi_{p:\mathsf{Id}_A(x,a)}\Pi_{q:\mathsf{Id}_{B[a/x]}(\mathsf{subst}(p,y'),b)}C[b/y]}(y', \{\lambda_{x':A'}\lambda_{p_{i,j}:\mathsf{Id}_A(a_i,a)}\lambda_{q_{i,j}:\mathsf{Id}_{B[a/x]}(\mathsf{subst}(p_{i,j},b_{i,j}),b)}c_{i,j}\})[a/x, b/y'](x', \mathsf{refl}(a), \mathsf{refl}(b)) : C[b/y]}$$

These translations are easily seen to be mutually inverse in their translation to $\mathsf{STT}_{\mathsf{CBN}}$, which we define in the next section, due to the $\{b_{i,j} \mid i, j\}$-Comm-λ-rule. Therefore, they are mutually inverse in $\mathsf{DTT}_{\mathsf{CBN}}$. □

We conclude that case and $\mathsf{case}^{p,q}$ are equivalent. We prefer to use the latter as the default, as it naturally arises in the completeness proofs in chapter 4. For the purposes of proof theory, however, the former may be the preferred choice, as the metatheory of the resulting system is known to be well-behaved (at least in absence of the commutative conversions).

2.1.1.4 A Syntactic Translation from $\mathsf{DTT}_{\mathsf{CBN}}$ to $\mathsf{STT}_{\mathsf{CBN}}$

Morally, $\mathsf{DTT}_{\mathsf{CBN}}$ should describe the same algorithms as $\mathsf{STT}_{\mathsf{CBN}}$ (at least at the type hierarchy over finite types), possibly assigning them a more precise type. Formally, this idea is captured by the existence of a syntactic translation from $\mathsf{DTT}_{\mathsf{CBN}}-$ into $\mathsf{STT}_{\mathsf{CBN}}$. By noting that it is compositional and faithful on all term constructors, we note that we can add to $\mathsf{DTT}_{\mathsf{CBN}}-$ the equational theory of $\mathsf{STT}_{\mathsf{CBN}}$ under this translation. We refer to the theory we obtain as $\mathsf{DTT}_{\mathsf{CBN}}$. Some examples of equations this implies are β- and η-laws for 1-, Σ- and Π-types and finite inductive type families and commutative conversions for the case-constructs, analogous to those for their simply typed equivalents, as well as β-laws for Id-types which state that $\mathsf{let}\ \mathsf{refl}(z)\ \mathsf{be}\ \mathsf{refl}(z)\ \mathsf{in}\ d = d$. We note that we then have a faithful translation $(-)^T$ from $\mathsf{DTT}_{\mathsf{CBN}}$ to $\mathsf{STT}_{\mathsf{CBN}}$.

The translation $(-)^T$ is inductively defined on types and terms through the schema of figure 2.5. This translation will later suggest our game theoretic interpretation of dependent type theory, by demanding that it agrees with (a total, finitary equivalent of) the usual PCF game semantics [22] after translating the syntax. A semantically inclined reader may want to think about the translation we

$$
\begin{array}{ll}
\vdash b_{i,j} : (a_i \mapsto_i \{b_{i,j} \mid j\})(a_i) & \mapsto \ \vdash b_{i,j} : \{b_{i,j} \mid i, j\} \\
x : A, y : B, z_{1,1} : C[a_1/x, b_{1,1}/y], \ldots, & \mapsto \ x : A^T, y : B^T, z_{1,1} : C^T, \ldots, z_{n,m_n} : C^T \\
z_{n,m_n} : C[a_n/x, b_{n,m_n}/y] \vdash \mathsf{case}_{B,C}(y, \{z_{i,j}\}_{i,j}) : C & \quad \vdash \mathsf{case}_{B^T,C^T}(y, \{z_{i,j}\}_{i,j}) : C^T \\
x' : A' \vdash \mathsf{case}^{p,q}_{B[a/x],C}(b, \{c_{i,j}\}_{i,j}) : C[b/y] & \mapsto \ x' : (A')^T \vdash \mathsf{case}_{B^T,C^T}(b^T, \{c^T_{i,j}[\mathtt{refl}/p_{i,j}, \mathtt{refl}/q_{i,j}]\}_{i,j}) : C^T \\
x : A \vdash \langle\rangle : 1 & \mapsto \ x : A^T \vdash \langle\rangle : 1 \\
x : A \vdash \langle b, c\rangle : \Sigma_{y:B} C & \mapsto \ x : A^T \vdash \langle b^T, c^T\rangle : B^T \times C^T \\
x : A \vdash \mathsf{fst}(d) : B & \mapsto \ x : A^T \vdash \mathsf{fst}(d^T) : B^T \\
x : A \vdash \mathsf{snd}(d) : C[\mathsf{fst}(d)/y] & \mapsto \ x : A^T \vdash \mathsf{snd}(d^T) : C^T \\
x : A \vdash \lambda_{y:B} c : \Pi_{y:B} C & \mapsto \ x : A^T \vdash \lambda_{y:B^T} c^T : B^T \Rightarrow C^T \\
x : A \vdash f(b) : C[b/y] & \mapsto \ x : A^T \vdash f^T(b^T) : C^T \\
x : A \vdash \mathsf{refl}(b) : \mathsf{Id}_B(b, b) & \mapsto \ x : A^T \vdash \mathtt{refl} : \{\mathtt{refl}\} \\
x : A \vdash \mathsf{let}\ p\ \mathsf{be}\ \mathsf{refl}(z)\ \mathsf{in}\ d : D[b/y, b'/y', p/w] & \mapsto \ x : A^T \vdash \mathsf{case}_{\{\mathtt{refl}\},D^T}(p^T, \{d^T[b^T/z]\}) : D^T \\
\Gamma, x : A, \Delta \vdash x : A & \mapsto \ \Gamma^T, x : A^T, \Delta^T \vdash x : A^T
\end{array}
$$

Figure 2.5: A syntactic translation on terms and types from $\mathsf{DTT}_{\mathsf{CBN}}-$ into $\mathsf{STT}_{\mathsf{CBN}}$. Note that it is functorial in the sense that it respects identifiers and substitutions.

$$
\frac{\Gamma \vdash_{\mathsf{DTT}_{\mathsf{CBN}}} a : A \qquad \Gamma \vdash_{\mathsf{DTT}_{\mathsf{CBN}}} b : A \qquad \Gamma^T \vdash_{\mathsf{STT}_{\mathsf{CBN}}} a^T = b^T : A^T}{\Gamma \vdash_{\mathsf{DTT}_{\mathsf{CBN}}} a = b : A}\ \mathsf{DTT}_{\mathsf{CBN}}\text{-}\mathsf{Eq}
$$

$$
\frac{\begin{array}{l}\forall_{1 \le i \le 2}\, x_1 : A_1, \ldots, x_n : A_n \vdash B_i\ \ \mathsf{type} \\ \vdash B_1^T = B_2^T \\ \forall_{\vdash t_1 : A_1} \ldots \forall_{\vdash t_n : A_n[t_1/x_1, \ldots, t_{n-1}/x_{n-1}]} \vdash B_1[t_1/x_1, \ldots, t_n/x_n] = B_2[t_1/x_1, \ldots, t_n/x_n]\end{array}}{\vdash \Pi_{x_1:A_1} \cdots \Pi_{x_n:A_n} B_1 = \Pi_{x_1:A_1} \cdots \Pi_{x_n:A_n} B_2}\ \mathsf{Ty}\text{-}\mathsf{Ext}
$$

Figure 2.6: The final rule, $\mathsf{DTT}_{\mathsf{CBN}}$-Eq, which $\mathsf{DTT}_{\mathsf{CBN}}$ has on top of $\mathsf{DTT}_{\mathsf{CBN}}-$, letting it inherit the equational theory of $\mathsf{STT}_{\mathsf{CBN}}$, as well as the type extensionality rule Ty-Ext which we sometimes consider.

define as a faithful non-full functor $(-)^T$ from the syntactic category (or, category of contexts) of $\mathsf{DTT}_{\mathsf{CBN}}$ to the syntactic category of $\mathsf{STT}_{\mathsf{CBN}}$.

Finally, we define $\mathsf{DTT}_{\mathsf{CBN}}$ as the theory generated by the rules of $\mathsf{DTT}_{\mathsf{CBN}}-$ together with the final rule, $\mathsf{DTT}_{\mathsf{CBN}}$-Eq, of figure 2.6, which says that $\mathsf{DTT}_{\mathsf{CBN}}$ inherits the judgemental equalities of $\mathsf{STT}_{\mathsf{CBN}}$. We note that $\mathsf{DTT}_{\mathsf{CBN}}$-Eq gives us a concise way of equipping $\mathsf{DTT}_{\mathsf{CBN}}$ with the appropriate β- and η-rules for its type formers as well as all necessary commutative conversions. We sometimes also consider the rule Ty-Ext, which expresses that types are extensional from the point of view of their sections[5].

We note that, by induction, $(-)^T$ respects the judgemental equalities introduced by the rule above, meaning that $(-)^T$ defines a translation from $\mathsf{DTT}_{\mathsf{CBN}}$ to $\mathsf{STT}_{\mathsf{CBN}}$. This lets us conclude the following.

[5]It remains to be verified if type checking remains decidable in the presence of this rule.

Corollary 2.1.2. *The translation* $(-)^T$ *defined above defines a faithful translation from* $\mathsf{DTT}_{\mathsf{CBN}}$ *to* $\mathsf{STT}_{\mathsf{CBN}}$.

We observe that we have defined a flavour of intensional type theory.

Remark 2.1.3. *The reader might wonder how the equational theory of* $\mathsf{DTT}_{\mathsf{CBN}}$ *compares to the usual ones we use for dependent type theories. We note that it implies all the usual* β*- and* η*-rules (weak* η' *for inductive families) for the type formers we consider (as well as some PCF-like commutative conversions), with the exception of the* η*-rule* $\mathsf{let}\ z\ \mathsf{be}\ \mathsf{refl}(x)\ \mathsf{in}\ c[\mathsf{refl}(x)/z] = c$ *for* Id*-types.*

Indeed, we easily see that $\mathsf{DTT}_{\mathsf{CBN}}$ *refutes one of the notorious consequences of* $\mathsf{Id} - \eta$*, the principle of equality reflection,*

$$\frac{\Gamma \vdash p : \mathsf{Id}_A(f,g)}{\Gamma \vdash f = g : A}\ \mathsf{Reflection},$$

as, for instance, for $\Gamma = x : \{a\}$ *and* $A = \{a\}$*,* $f = \mathsf{case}_{\{a\},\{a\}}(x, a)$ *and* $g = a$*, we do not have that* $x : \{a\} \vdash f = g : \{a\}$*, while we do have* $x : \{a\} \vdash \mathsf{case}_{\{a\},\mathsf{Id}_{\{a\}}(f,g)}(x, \mathsf{refl}(a)) : \mathsf{Id}_{\{a\}}(f, g)$.

2.1.2 Categorical Semantics

In this section, we briefly discuss a notion of categorical semantics for dependently typed equational logic.

It is clear that a type theory with dependent types should be modelled by some indexed category $\mathcal{B}^{op} \xrightarrow{\mathcal{C}} \mathsf{Cat}$. (This would be even more obvious if we represented context morphisms as first class objects in the syntax, in the style of Pitts [19].) Indeed, we have the category $\mathcal{B}$ which interprets the contexts and the morphisms between them. We have a type theory in each context Γ, which is modelled by some category $\mathcal{C}(\llbracket\Gamma\rrbracket)$ with structure to interpret the appropriate connectives. And, whenever two contexts are related by some context morphism $\Gamma' \vdash \gamma : \Gamma$, we have substitution operations going from the type theory in context Γ to that in context Γ', which are modelled by structure preserving functors

$\mathcal{C}(\llbracket\Gamma\rrbracket) \xrightarrow{\mathcal{C}(\llbracket\gamma\rrbracket)} \mathcal{C}(\llbracket\Gamma'\rrbracket)$ (as substitution usually is compatible with all term and type formers). In this view, it is easily seen that existential quantifiers should get interpreted, à la Lawvere [24], as left adjoints to these substitution functors while universal quantifiers are their right adjoints.

The missing ingredient is that, in dependent type theory, quantification is not external but internal: the entities (in $\mathcal{B}$) we are quantifying over are of the same nature as the proofs of the predicates (in $\mathcal{C}$) that we quantify over. The idea is that objects in $\mathcal{B}$ can be built as lists of objects in the fibres of $\mathcal{C}$ and that the morphisms in $\mathcal{B}$ (the interpretation of context morphisms) then arise as corresponding lists of morphisms in the fibres of $\mathcal{C}$ (the interpretation of terms). This intuition is formalised by the so-called comprehension axiom.

Definition 2.1.4 (Comprehension Axiom)**.** *Let $\mathcal{B}^{op} \xrightarrow{\mathcal{C}} \mathsf{Cat}$ be a strict*[6] *indexed category (writing Cat for the category of small categories and functors). Given $B' \xrightarrow{f} B$ in $\mathcal{B}$, let us write $-\{f\}$ for the change of base functor $\mathcal{C}(f) : \mathcal{C}(B) \longrightarrow \mathcal{C}(B')$. Recall that $\mathcal{C}$ is said to satisfy the **comprehension axiom** if*

- *$\mathcal{B}$ has a terminal object $\cdot$;*
- *all fibres $\mathcal{C}(B)$ have terminal objects 1_B which are stable under change of base (for which we just write 1);*
- *the presheaves (writing Set for the category of small sets and functions)*

$$(\mathcal{B}/B)^{op} \longrightarrow \mathsf{Set}$$

$$(B' \xrightarrow{f} B) \longmapsto \mathcal{C}(B')(1, C\{f\})$$

are representable. That is, we have representing objects $B.C \xrightarrow{\mathbf{p}_{B,C}} B$ and natural bijections

$$\mathcal{C}(B')(1, C\{f\}) \xrightarrow{\cong} \mathcal{B}/B(f, \mathbf{p}_{B,C})$$

$$c \longmapsto \langle f, c\rangle.$$

[6] For brevity, from now on we shall often drop the modifier "strict" for indexed structures. For instance, if we mention an indexed honey badger, we shall really mean a strict indexed honey badger.

We write $\mathbf{v}_{B,C}$ for the element of $\mathcal{C}(B.C)(1, C\{\mathbf{p}_{B,C}\})$ corresponding to $\mathsf{id}_{\mathbf{p}_{B,C}}$ (the universal elements of the representation). We define the morphisms

$$B.C \xrightarrow{\mathsf{diag}_{B,C} := \langle \mathsf{id}_{B.C}, \mathbf{v}_{B,C} \rangle} B.C.C\{\mathbf{p}_{B,C}\};$$

$$B'.C\{f\} \xrightarrow{\mathbf{q}_{f,C} := \langle \mathbf{p}_{B',C\{f\}}; f, \mathbf{v}_{B',C\{f\}} \rangle} B.C.$$

We have maps (defining the **comprehension functor**)

$$\mathcal{C}(B)(C', C) \xrightarrow{\mathbf{p}_{B,-}} \mathcal{B}/B(\mathbf{p}_{B,C'}, \mathbf{p}_{B,C})$$

$$c \longmapsto \mathbf{p}_{B,c} := \langle \mathbf{p}_{B,C'}, \mathbf{v}_{B,C'}; c\{\mathbf{p}_{B,C'}\} \rangle.$$

When these are full and faithful, we call the comprehension **full and faithful**, respectively. When it induces an equivalence $\mathcal{C}(\cdot) \cong \mathcal{B}/\cdot \cong \mathcal{B}$, we call the comprehension **democratic**.

Note that the comprehension axiom says that we build as lists of closed terms the morphisms into objects that arise as lists of types in our category of contexts $\mathcal{B}$. Demanding the comprehension functor to be fully faithful means that also the terms in $\mathcal{C}(\Gamma)(A, B)$ correspond precisely with the terms in $\mathcal{C}(\Gamma.A)(1, B\{\mathbf{p}_{\Gamma,A}\})$. This is essential to get a precise fit with the syntax for cartesian dependent type theory. The notion of democracy corresponds to the syntactic condition that all contexts are formed from the empty context by adjoining types.

Remark 2.1.5 (Correspondence with Comprehension Categories)**.** *The definition of an indexed category with comprehension is easily seen to be equivalent to Jacobs' notion of a split comprehension category with unit [25]. We prefer this formulation in terms of indexed categories as strictness is important in computer science (syntactic substitution is strict), in which case the fibrational perspective is needlessly abstract. Jacobs' notion of fullness of a comprehension category corresponds – confusingly – to our demand of the comprehension both being full and faithful. We believe it is useful to use this more fine-grained terminology.*

Let us make the correspondence a bit more precise – the reader can find all details in [25]. There is a well-known correspondence between strict indexed categories and split fibrations:

- *given a strict indexed category* $\mathcal{B}^{op} \xrightarrow{\mathcal{C}} \mathsf{Cat}$*, we can define a split fibration* $\int \mathcal{C} \xrightarrow{\mathbf{p}_{\mathcal{C}}} \mathcal{B}$ *by using the Grothendieck construction: we take* $\int \mathcal{C}$ *to have objects* $\mathsf{ob}(\int \mathcal{C}) := \Sigma_{B \in \mathsf{ob}(\mathcal{B})} \mathsf{ob}(\mathcal{C}(B))$ *and morphisms*

$$\left(\int \mathcal{C}\right)(\langle B, C\rangle, \langle B', C'\rangle) := \Sigma_{b \in \mathcal{B}(B,B')} \mathcal{C}(C, C'\{b\})$$

and we take $\mathbf{p}_{\mathcal{C}}$ *to be the projection to the first component;*

- *given a split fibration* $\mathcal{E} \xrightarrow{\mathbf{p}} \mathcal{B}$*, we can define a strict indexed category* $\mathcal{B} \xrightarrow{\mathcal{C}} \mathsf{Cat}$ *as* $\mathcal{C}(B) := \mathbf{p}^{-1}(B)$ *where the functors* $-\{f\} : \mathcal{C}(B') \longrightarrow \mathcal{C}(B)$ *arise as the inverse image functors of the split fibration* $\mathcal{E}$ *along* $B \xrightarrow{f} B' \in \mathcal{B}$*.*

Now, our formulation of the comprehension axiom in definition 2.1.4 for a strict indexed category $\mathcal{B}^{op} \xrightarrow{\mathcal{C}} \mathsf{Cat}$ *then is easily seen to correspond to putting the following conditions on* $\int \mathcal{C} \xrightarrow{\mathbf{p}_{\mathcal{C}}} \mathcal{B}$*:*

- $\mathcal{B}$ *has a terminal object* $\cdot$*;*
- $\mathbf{p}_{\mathcal{C}} \xrightarrow{\mathbf{p}_{\mathcal{C}}} \mathsf{id}_{\mathcal{B}}$ *has a fibred right adjoint* $\mathsf{id}_{\mathcal{B}} \xrightarrow{1} \mathbf{p}_{\mathcal{C}}$ *(fibred terminal objects);*
- *this functor* $\mathcal{B} \xrightarrow{1} \int \mathcal{C}$ *has a further right adjoint* $\int \mathcal{C} \xrightarrow{-.-} \mathcal{B}; \langle B, C\rangle \mapsto B.C$*.*

Jacobs calls this structure a ***split comprehension category with unit****.*

Strict indexed categories with **full and faithful** comprehension admit a more minimalistic presentation in the form of Dybjer's notion of categories with families with unit. Recall that these categories are another standard notion of model of dependently typed equational logic [20, 26].

Definition 2.1.6 (Category with Families)**.** *A category with families (CwF) is a category* $\mathcal{B}$ *with a terminal object* $\cdot$*, for all objects* Γ *a set* $\mathsf{Ty}(\Gamma)$*, for all* $A \in \mathsf{Ty}(\Gamma)$ *a set* $\mathsf{Tm}(\Gamma, A)$*, for all* $\Gamma' \xrightarrow{f} \Gamma$ *in* $\mathcal{B}$ *functions* $\mathsf{Ty}(\Gamma) \xrightarrow{-\{f\}} \mathsf{Ty}(\Gamma')$ *and* $\mathsf{Tm}(\Gamma, A) \xrightarrow{-\{f\}} \mathsf{Tm}(\Gamma', A\{f\})$*, such that*

$$\begin{array}{llll} A\{\mathsf{id}_\Gamma\} = A & \text{(Ty-Id)} & A\{f;g\} = A\{g\}\{f\} & \text{(Ty-Comp)} \\ t\{\mathsf{id}_\Gamma\} = t & \text{(Tm-Id)} & t\{f;g\} = t\{g\}\{f\} & \text{(Tm-Comp),} \end{array}$$

for $A \in \mathsf{Ty}(\Gamma)$ *a morphism* $\Gamma.A \xrightarrow{\mathbf{p}_{\Gamma,A}} \Gamma$ *of* $\mathcal{B}$ *and* $\mathbf{v}_{\Gamma,A} \in \mathsf{Tm}(\Gamma.A, A\{\mathbf{p}_{\Gamma,A}\})$ *and, finally, for all* $t \in \mathsf{Tm}(\Gamma', A\{f\})$ *a morphism* $\Gamma' \xrightarrow{\langle f,t\rangle} \Gamma.A$ *such that*

$$\langle f,t\rangle;\mathbf{p}_{\Gamma,A} = f \quad \text{(Cons-L)} \qquad \mathbf{v}_{\Gamma,A}\{\langle f,t\rangle\} = t \quad \text{(Cons-R)}$$
$$\langle \mathbf{p}_{\Gamma,A}, \mathbf{v}_{\Gamma,A}\rangle = \mathsf{id}_{\Gamma.A} \quad \text{(Cons-Id)} \qquad g;\langle f,t\rangle = \langle g;f, t\{g\}\rangle \quad \text{(Cons-Nat).}$$

A CwF is said to have a ***unit*** *if we have* $1 \in \mathsf{Ty}(\Gamma)$*, for all* $\Gamma \in \mathsf{ob}(\mathcal{B})$*, such that* $\mathsf{Tm}(\Gamma, 1) \cong \{*\}$ *and* $1\{f\} = 1$ *for all* $f \in \mathcal{B}$*.*

The correspondence works as follows. Every strict indexed category with comprehension is easily seen to define a CwF with unit, if we define $\mathsf{Ty}(\Gamma)$ and $\mathsf{Tm}(\Gamma, A)$ as $\mathsf{ob}(\mathcal{C}(\Gamma))$ and $\mathcal{C}(\Gamma)(1, A)$, respectively. Conversely, we can define a strict indexed category with comprehension from a CwF with unit by defining $\mathsf{ob}(\mathcal{C}(\Gamma)) := \mathsf{Ty}(\Gamma)$ and $\mathcal{C}(\Gamma)(A, B) := \mathsf{Tm}(\Gamma.A, B\{\mathbf{p}_{\Gamma,A}\})$. We see that the resulting comprehension is full and faithful. Starting from a CwF with unit, defining the corresponding strict indexed category with comprehension and then defining the CwF from that again gives us back the CwF with unit we started with (up to equivalence) as $\mathsf{Tm}(\Gamma.1, B\{\mathbf{p}_{\Gamma,1}\}) \cong \mathsf{Tm}(\Gamma, B)$. Starting from a strict indexed category $\mathcal{C}$ with comprehension, defining the CwF with unit and from that again a strict indexed category $\mathcal{C}'$, gives us the strict indexed category with full and faithful comprehension where we have redefined $\mathcal{C}'(\Gamma)(A, B) := \mathcal{C}(\Gamma.A)(1, B\{\mathbf{p}_{\Gamma,A}\})$. We see that this restricts to a bijective correspondence between CwFs with unit and strict indexed categories with full and faithful comprehension (up to equivalence).

The advantage we see for formulating models as indexed categories is that various connectives get an elegant interpretation. We state some of these interpretations below, where we make use of the usual equational theory for extensional dependent type theory, using β- and η-rules for all type formers (including Id-types, unlike in section 2.1.1.4).

Theorem 2.1.7 (Pure DTT Semantics, [20, 25])**.** *We have a sound interpretation of pure dependent type theory with* 1*-types in any indexed category* $\mathcal{B}^{op} \xrightarrow{\mathcal{C}} \mathsf{Cat}$ *with full and faithful comprehension. We list necessary and sufficient conditions for the model to support various type formers:*

- *strong* Σ*-types*[7] *– objects* $\Sigma_C D$ *of* $\mathcal{C}(B)$ *such that* $\mathbf{p}_{B,\Sigma_C D} = \mathbf{p}_{B.C,D}; \mathbf{p}_{B,C}$*;*

- *weak* Σ*-types – left adjoint functors* $\Sigma_C \dashv -\{\mathbf{p}_{B,C}\}$ *satisfying the left Beck-Chevalley condition*[8] *for pullback squares in* $\mathcal{B}$ *of the following form, which we shall later refer to as* $\mathbf{p}$*-squares,*

$$\begin{array}{ccc} B'.C\{f\} & \xrightarrow{\mathbf{q}_{f,C}} & B.C \\ \downarrow{\scriptstyle \mathbf{p}_{B',C\{f\}}} & & \downarrow{\scriptstyle \mathbf{p}_{B,C}} \\ B' & \xrightarrow[f]{} & B; \end{array}$$

- *strong extensional*[9] Id*-types – objects* Id_C *of* $\mathcal{C}(B.C.C)$ *such that* $\mathbf{p}_{B.C.C,\mathsf{Id}_C} = \mathsf{diag}_{B,C}$*;*

- *weak extensional* Id*-types – left adjoints* $\mathsf{Id}_C \dashv -\{\mathsf{diag}_{B,C}\}$ *satisfying the left Beck-Chevalley condition for pullback squares in* $\mathcal{B}$ *of the following form,*

[7] That is, Σ-types with a dependent elimination rule. We call Σ-types $\Sigma_{x:A}B$ weak if the type we are eliminating into is not allowed to depend on $\Sigma_{x:A}B$ in the elimination rule. We use a similar terminology for other positive connectives. We note that as soon as we have strong Σ-types, the strong and weak elimination rules for other positive connectives become equivalent.

[8] Remember that the (left) Beck-Chevalley condition for a left adjoint functor $f_!$ to $f^* := \mathcal{C}(f)$ for a pullback square

$$\begin{array}{ccc} A & \xrightarrow{h} & B \\ \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle k} \\ C & \xrightarrow[g]{} & D \end{array}$$

corresponds to the statement that the obvious morphism (from commuting of pullback square, unit, and counit) $f_! h^* \longrightarrow f_! h^* k^* k_! \xrightarrow{\cong} f_! f^* g^* k_! \longrightarrow g^* k_!$ is an isomorphism. Similarly, by the (right) Beck-Chevalley condition for a right adjoint f_* to f^* we mean that the obvious morphism $g^* k_* \longrightarrow g^* k_* h_* h^* \xrightarrow{\cong} g^* g_* f_* h^* \longrightarrow f_* h^*$ is an iso. The reader is encouraged to think of this condition as the equivalent for Σ-, Π- and Id-types of the condition on the substitution functors preserving the appropriate categorical structure for other type formers. It says that, in a sense, Σ-, Π- and Id-types are preserved under substitution.

[9] These are identity types, which, in addition to the β-rule $\mathsf{let\ refl}(x)\ \mathsf{be\ refl}(x)\ \mathsf{in}\ d = d$ also satisfy the η-rule $d = \mathsf{let}\ x\ \mathsf{be\ refl}(x)\ \mathsf{in}\ d[x/x', \mathsf{refl}(x)/p]$. This η-rule is known to make type checking undecidable in the presence of the strong elimination rule, hence it is often omitted [27]. We include it to obtain a more elegant categorical semantics; we could also easily omit it.

which we shall later refer to as diag*-squares,*

$$\begin{array}{ccc} B'.C\{f\} & \xrightarrow{\quad\mathbf{q}_{f,C}\quad} & B.C \\ \Big\downarrow{\scriptstyle \mathsf{diag}_{B',C\{f\}}} & & \Big\downarrow{\scriptstyle \mathsf{diag}_{B,C}} \\ B'.C\{f\}.C\{f\}\{\mathbf{p}_{B',C\{f\}}\} & \xrightarrow[\mathbf{q}_{\mathbf{q}_{f,C},C\{\mathbf{p}_{B,C}\}}]{} & B.C.C\{\mathbf{p}_{B,C}\}; \end{array}$$

- *weak* $0,+$*-types*[10] *– finite indexed coproducts (i.e. finite coproducts in all fibres that are stable under change of base);*

- *strong* $0,+$*-types – if additionally the following canonical morphisms are bijections*

$$\mathcal{C}(C.\Sigma_{1\leq i\leq n}C_i)(C',C'') \longrightarrow \Pi_{1\leq i\leq n}\mathcal{C}(C.C_i)(C'\{\mathbf{p}_{C,\langle i,\mathsf{id}_{C_i}\rangle}\},C''\{\mathbf{p}_{C,\langle i,\mathsf{id}_{C_i}\rangle}\});$$

- Π*-types – right adjoint functors* $-\{\mathbf{p}_{B,C}\} \dashv \Pi_C$ *satisfying the right Beck-Chevalley condition for* $\mathbf{p}$*-squares.*

In fact, the interpretation in such categories is complete in the sense that an equality holds in all interpretations iff it is provable in the syntax of dependent type theory where we use both β*- and* η*-equality rules for all type formers.*

Remark 2.1.8. *Note that (weak)* Σ*-types and* Π*-types in particular allow us to interpret* $\times$*-types and* $\Rightarrow$*-types as their existence makes* $\mathcal{C}$ *into an indexed cartesian closed category (that is, equips the fibres of* $\mathcal{C}$ *with a cartesian closed structure that is stable under change of base).*

In particular, we can use such categories to model pure simple type theory with $0,+,1,\times,\Rightarrow$-types as a special case, rather than using the usual notion of model of a bicartesian closed category $\mathcal{C}$. Indeed, starting from such a bicartesian closed category $\mathcal{C}$, we can produce an indexed category $\mathcal{C}^{op} \xrightarrow{\mathsf{self}(\mathcal{C})} \mathsf{Cat}$ where $\mathsf{self}(\mathcal{C})(A)$ has the same objects as $\mathcal{C}$ and $\mathsf{self}(\mathcal{C})(A)(B,C) = \mathcal{C}(A\times B,C)$ with the obvious identities

[10] The syntactic rules for $0,+$-types can be found in [28]. We are mostly interested in stating the semantic condition here, as we shall need it to describe the semantics for sum types in linear and effectful settings, where we shall also treat the syntax.

and composition and with the change of base functors defined to be the identity on objects and to act on morphisms in the obvious way through precomposition. We see that every model of simple type theory gives, in particular, rise to a (rather degenerate) model of dependent type theory.

Theorem 2.1.9. *For a bicartesian closed category* $\mathcal{C}$*,* $\mathsf{self}(\mathcal{C})$ *is an indexed category with full and faithful democratic comprehension, which supports* 0*-,* $+$*-,* Σ*- and* Π*-types. In this case,* $\mathbf{p}_{A,B}$ *is the usual product projection from* $A \times B \longrightarrow A$*. It does not usually support extensional* Id*-types as these correspond to objects* $1/A$ *such that* $1/A \times A \cong 1$*.*

Inductive types and type families, in particular finite ones, can be given a pretty categorical semantics as initial algebras for certain endofunctors. We refer the interested reader to [29], as we shall not need those details in our development.

2.2 Call-By-Push-Value and Effectful Simple Type Theory

We believe Levy's call-by-push-value (CBPV) is an excellent setting for studying effectful type theories [30]. It unifies the CBV and CBN paradigms as follows.

Recall that one origin of the CBV-CBN-distinction is the fact that, in an effectful type theory, we cannot usually have both coproduct types and function types with their general η-laws: the η-law has to fail either for the former type formers, leading to CBN, or for the latter, leading to CBV. For a particular instantiation of this idea, we would like to remind the reader of the folklore theorem that a cartesian closed category with coproducts – just an initial object is enough, in fact – degenerates to the trivial category if it has fixpoints [31]. CBPV unifies CBV and CBN type theories by having two distinct classes of types: those for which we have connectives like coproduct types and those for which we have ones like function types. This allows us to retain the general η-laws for all connectives and lets us encode traditional CBV and CBN type theories.

In this section, we present a slight reformulation and simplification of CBPV's simply typed version, with the purpose of extending it with dependent types later. We start by discussing a syntax in section 2.2.1 which is almost identical to Levy's CBPV except that computations are treated as special stacks/homomorphisms. In section 2.2.2, we discuss a modified but equivalent (in the categorical sense) presentation of Levy's categorical semantics of simple CBPV that makes the transition to dependent types more natural, after which we give a few examples of models in section 2.2.3. Next, we briefly discuss the small-step operational semantics for CBPV in section 2.2.4. Finally, in section 2.2.5, we sketch how one proceeds to add effects to the pure CBPV calculus, which is, after all, the point of our endeavour. For this, we mostly take an operational point of view.

2.2.1 Syntax

We encourage the reader to look at the syntax of call-by-push-value (CBPV) in the following slightly simplified way: as providing an adjunction decomposition of Moggi's monadic metalanguage [32], similar (dual) to the one that Benton's linear/non-linear (LNL) calculus [33] gives of (the comonadic) dual intuitionistic linear logic (DILL) [34], but in the more general setting of possibly non-commutative effects. Roughly, CBPV consists of two type theories, related by an adjunction $F \dashv U$: one for defining **values and their types**, to be thought of as **static** objects which behave like a pure cartesian type theory, and one for defining effectful **computations/stacks and their types**, to be thought of as **dynamic** objects which behave linearly.

Therefore, CBPV distinguishes between two classes of types: **value types** and **computation types**. These can be similarly read as, respectively, positive and negative types or as types of data and codata. The linear types of the LNL calculus should be thought of as analogous to computation types, while its cartesian types correspond to value types. The idea is that in natural deduction, for some connectives, the positive/value connectives, the introduction rule involves a choice, while the elimination rule is invertible (works through pattern matching)

and for others, the negative/computation connectives, the opposite is true. As a rule of thumb, connectives that operate on value types arise as left adjoint functors in the categorical semantics, while connectives that operate on computation types are right adjoint functors.

Call-by-push-value (and polarised logic) chooses to keep the classes of types formed from both classes of connectives separate and adds two extra connectives F, which turns a value type into a type of computations that **return** a result of the original value type, and U, which turns a computation type into a value type of **thunks** of computations of the original computation type. This allows us to use the full $\beta\eta$-equational theory for all connectives, even in the presence of effects. Importantly, we have CBV and CBN embeddings of (effectful) type theory into (effectful) CBPV, that give rise to the usual equational theories.

CBPV has two classes of types (we sometimes underline types to emphasize that we mean a computation type):

$$\text{value types} \quad A \qquad\qquad \text{computation types} \quad \underline{B}.$$

In this thesis, simple value and computation types are formed using the connectives of figure 2.7, excluding general inductive and coinductive types. Here, $1, \times$ will denote pattern-matching products, while $\top, \&$ are projection products[11]. More generally, following Levy, we include primitives $\Pi_{1\leq i\leq n}\underline{B_i}$ for n-ary projection products and $\Sigma_{1\leq i\leq n}A_i$ for n-ary sum (we write nullary and binary sum as 0 and $A + A'$). We do this to emphasize their similarity to $\Pi^{-\circ}_{F(-)}$- and Σ-types in the dependently typed version of CBPV. We write $A \,{}_F\!\multimap \underline{B}$ for the type of computations that take an input of type A and return a computation of type $\underline{B}$. (These are conventionally written $A \Rightarrow \underline{B}$. We choose our notation to be reminiscent of the LNL calculus expression $F(A) \multimap \underline{B}$, which it should generalise.)

[11]Note that these correspond to the two ways of defining products in the categorical semantics: as left adjoints to the internal hom or as right adjoints to the diagonal functor, as positive and negative connectives, respectively.

value/positive types A	**computation/negative types** $\underline{B}$
0, $A + A'$, $\Sigma_{1 \leq i \leq n} A_i$	$A \ {}_F\!\multimap \underline{B}$
1, $A \times A'$	$\top$, $\underline{B} \& \underline{B'}$, $\Pi_{1 \leq i \leq n} \underline{B_i}$
$U\underline{B}$	FA
(inductive types)	(coinductive types)

Figure 2.7: An overview of the simple value and computation types we consider with exception of general inductive and coinductive types which we shall not attempt to incorporate.

Similarly, CBPV has separate typing judgements for terms representing values and computations, respectively,

$$\Gamma \vdash^v a : A \qquad\qquad \Gamma \vdash^c b : \underline{B}.$$

Here, Γ is a context, or list $x_1 : A_1, \ldots, x_n : A_n$ of declarations of distinct identifiers x_i of value type A_i. Additionally, Levy considers stacks (sometimes called homomorphisms, as many effects equip computation types with an algebraic structure which stacks preserve), which are represented as typed terms

$$\Gamma; \mathsf{nil} \ : \underline{B} \vdash^k c : \underline{C},$$

where Γ, as before, is a context of identifier declarations of value type and nil is an identifier of computation type $\underline{B}$. For notational convenience, and unlike Levy, we unify the computation and stack judgements as a single judgement

$$\Gamma; \Delta \vdash b : \underline{B},$$

where Γ is as before and Δ is a context of identifier declarations of computation type. For now, Δ will have at most length 1 and in that case is often referred to as a **stoup**. The case that Δ has length 0 corresponds to Levy's computation judgement and the case of length 1 to his stack judgement. To keep the notation light, we also omit the annotation v on the sequent in the value judgement and simply write

$$\Gamma \vdash a : A.$$

We encourage the reader to think of the dual context $\Gamma; \Delta$ to consist of cartesian region Γ in which the usual structural rules of weakening and contraction are valid

and of Δ as a linear region in which they are not. These typing judgements are defined through the rules of figure 2.8 and the obvious (admissible) two substitution rules and weakening rule for identifiers of value type.

As usual, we distinguish between free and bound (i.e. non-free) identifiers and consider terms up to α-equivalence, or permutation of their bound identifiers. The rules of the type theory force the free identifiers of a well-typed term to be declared in the context. For notational convenience, we treat indices i of (terms of) a sum $\Sigma_{1\leq i\leq n}A_i$ or product $\Pi_{1\leq i\leq n}\underline{B_i}$ similarly to bound identifiers. A proper formal treatment would involve including the indices and their range in the context, to distinguish between bound and free indices and to consider freshness of the appropriate indices in various η-rules. We prefer to avoid this extra formality and keep their treatment informal as we are convinced that the intended meaning will be clear to the reader and that anyone so inclined can fill in the technical details.

We can consider these terms up to α-equivalence and, as such, define an operational semantics for them in section 2.2.4. We frequently also consider the terms up to the additional equational theory of figure 2.9 together with the rules which state that all term formers respect equality and that equality is an equivalence relation, where we write $M[V/x]$ for the syntactic metaoperation of capture avoiding substitution of V for x in M. We shall see that this equational theory naturally arises from the categorical semantics of simple CBPV.

Recall that a call-by-value (CBV) and call-by-name (CBN) evaluation strategy on the λ-calculus generally give rise to different equational theories (in the presence of effects) [35]. For instance, the η-rule for function types typically fails in the former and that for sum types in the latter. CBPV gives rise to both of these equational theories by embedding an (impure) λ-calculus either with a CBV or with a CBN translation.

In the presence of effects, the usual pure connectives of products, coproducts and function types bifurcate into many variants due to the distinction of versions of different arities and the distinction between projection and pattern matching products. These are nicely and uniformly treated in Levy's Jumbo λ-calculus [36].

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A} \qquad \frac{\Gamma \vdash V : A \qquad \Gamma, x : A, \Gamma' \vdash W : A'}{\Gamma, \Gamma' \vdash \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ W : A'}$$

$$\frac{\Gamma \vdash V : A \qquad \Gamma, x : A, \Gamma'; \Delta \vdash K : \underline{B}}{\Gamma, \Gamma'; \Delta \vdash \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ K : \underline{B}}$$

$$\frac{}{\Gamma; \mathsf{nil}\ : \underline{B} \vdash \mathsf{nil}\ : \underline{B}} \qquad \frac{\Gamma; \Delta \vdash K : \underline{B} \qquad \Gamma; \mathsf{nil}\ : \underline{B} \vdash L : \underline{C}}{\Gamma; \Delta \vdash \mathsf{let\ nil\ be}\ K\ \mathsf{in}\ L : \underline{B}}$$

$$\frac{\Gamma \vdash V : A}{\Gamma; \cdot \vdash \mathsf{return}\ V : FA} \qquad \frac{\Gamma; \Delta \vdash K : FA \qquad \Gamma, x : A, \Gamma'; \cdot \vdash N : \underline{B}}{\Gamma, \Gamma'; \Delta \vdash K\ \mathsf{to}\ x\ \mathsf{in}\ N : \underline{B}}$$

$$\frac{\Gamma; \cdot \vdash M : \underline{B}}{\Gamma \vdash \mathsf{thunk}\ M : U\underline{B}} \qquad \frac{\Gamma \vdash V : U\underline{B}}{\Gamma; \cdot \vdash \mathsf{force}\ V : \underline{B}}$$

$$\frac{\Gamma \vdash V_i : A_i}{\Gamma \vdash \langle i, V_i \rangle : \Sigma_{1 \leq i \leq n} A_i} \qquad \frac{\Gamma \vdash V : \Sigma_{1 \leq i \leq n} A_i \qquad \{\Gamma, x : A_i \vdash W_i : A'\}_{1 \leq i \leq n}}{\Gamma \vdash \mathsf{pm}\ V\ \mathsf{as}\ \langle i, x \rangle\ \mathsf{in}\ W_i : A'}$$

$$\frac{\Gamma \vdash V : \Sigma_{1 \leq i \leq n} A_i \qquad \{\Gamma, x : A_i; \Delta \vdash K_i : \underline{B}\}_{1 \leq i \leq n}}{\Gamma; \Delta \vdash \mathsf{pm}\ V\ \mathsf{as}\ \langle i, x \rangle\ \mathsf{in}\ K_i : \underline{B}}$$

$$\frac{}{\Gamma \vdash \langle\rangle : 1} \qquad \frac{\Gamma \vdash V : 1 \qquad \Gamma \vdash W : A'}{\Gamma \vdash \mathsf{pm}\ V\ \mathsf{as}\ \langle\rangle\ \mathsf{in}\ W : A'}$$

$$\frac{\Gamma \vdash V : 1 \qquad \Gamma; \Delta \vdash K : \underline{B}}{\Gamma; \Delta \vdash \mathsf{pm}\ V\ \mathsf{as}\ \langle\rangle\ \mathsf{in}\ K : \underline{B}}$$

$$\frac{\Gamma \vdash V_1 : A_1 \qquad \Gamma \vdash V_2 : A_2}{\Gamma \vdash \langle V_1, V_2 \rangle : A_1 \times A_2} \qquad \frac{\Gamma \vdash V : A_1 \times A_2 \qquad \Gamma, x : A_1, y : A_2 \vdash W : A'}{\Gamma \vdash \mathsf{pm}\ V\ \mathsf{as}\ \langle x, y \rangle\ \mathsf{in}\ W : A'}$$

$$\frac{\Gamma \vdash V : A_1 \times A_2 \qquad \Gamma, x : A_1, y : A_2; \Delta \vdash K : \underline{B}}{\Gamma; \Delta \vdash \mathsf{pm}\ V\ \mathsf{as}\ \langle x, y \rangle\ \mathsf{in}\ K : \underline{B}}$$

$$\frac{\{\Gamma; \Delta \vdash K_i : \underline{B_i}\}_{1 \leq i \leq n}}{\Gamma; \Delta \vdash \lambda_i K_i : \Pi_{1 \leq i \leq n} \underline{B_i}} \qquad \frac{\Gamma; \Delta \vdash K : \Pi_{1 \leq i \leq n} \underline{B_i}}{\Gamma; \Delta \vdash i\text{`}K : \underline{B_i}}$$

$$\frac{\Gamma, x : A; \Delta \vdash K : \underline{B}}{\Gamma; \Delta \vdash \lambda_x K : A\ {}_F\!\multimap \underline{B}} \qquad \frac{\Gamma \vdash V : A \qquad \Gamma; \Delta \vdash K : A\ {}_F\!\multimap \underline{B}}{\Gamma; \Delta \vdash V\text{`}K : \underline{B}}$$

Figure 2.8: Values, computations and stacks of simple CBPV.

There are fully faithful translations $(-)^v$ and $(-)^n$, respectively, from CBV and CBN versions of this whole calculus into CBPV [30] and, in fact, the same is true if we consider arbitrary theories rather than the pure calculi. To convey the intuition without getting stuck on technicalities, we present some special cases of the translations in figures 2.10 and 2.11.

let w be S in $R = R[S/w]$	
(return V) to x in $M = M[V/x]$	$L[K/\mathsf{nil}\,] \overset{\#x}{=} K$ to x in $L[\mathsf{return}\ x/\mathsf{nil}\,]$
force thunk $M = M$	$V = \mathsf{thunk}\ \mathsf{force}\ V$
pm $\langle i, V\rangle$ as $\langle i, x\rangle$ in $R_i = R_i[V/x]$	$R[V/z] \overset{\#i,x}{=} \mathsf{pm}\ V$ as $\langle i, x\rangle$ in $R[\langle i, x\rangle/z]$
pm $\langle\rangle$ as $\langle\rangle$ in $R = R$	$R[V/z] = \mathsf{pm}\ V$ as $\langle\rangle$ in $R[\langle\rangle/z]$
pm $\langle V, V'\rangle$ as $\langle x, y\rangle$ in $R = R[V/x, V'/y]$	$R[V/z] \overset{\#x,y}{=} \mathsf{pm}\ V$ as $\langle x, y\rangle$ in $R[\langle x, y\rangle/z]$
$i\text{`}\lambda_j K_j = K_i$	$K \overset{\#i}{=} \lambda_i i\text{`}K$
$V\text{`}\lambda_x K = K[V/x]$	$K \overset{\#x}{=} \lambda_x x\text{`}K$

Figure 2.9: Equations of simple CBPV. These should be read as equations of typed terms: we impose them if we can derive that both sides of the equation are well-typed terms of the same type in the same context. We write $\overset{\#x_1,\dots,x_n}{=}$ to indicate that for the equation to hold, the identifiers or indices $x_1, \dots, x_n$ should, in both terms being equated, be replaced by fresh ones, in order to avoid unwanted identifier bindings. Note that in the first equation, w might either be an identifier of value type or of computation type.

CBV type	**CBPV type**	**CBV term**	**CBPV term**
A	A^v	$x_1 : A_1, \dots, x_m : A_m \vdash M : A$	$x_1 : A_1^v, \dots, x_m : A_m^v; \cdot \vdash M^v : F(A^v)$
		x	return x
		let x be M in N	M^v to x in N^v
$\Sigma_{1\le i\le n} A_i$	$\Sigma_{1\le i\le n} A_i^v$	$\langle i, M\rangle$	M^v to x in return $\langle i, x\rangle$
		pm M as $\langle i, x\rangle$ in N_i	M^v to z in (pm z as $\langle i, x\rangle$ in N_i^v)
$\Pi_{1\le i\le n} A_i$	$U\Pi_{1\le i\le n} F A_i^v$	$\lambda_i M_i$	return thunk $(\lambda_i M_i^v)$
		$i\text{`}N$	N^v to z in (i`force z)
$A \Rightarrow A'$	$U(A^v \ {}_F\!\multimap F A'^v)$	$\lambda_x M$	return thunk $\lambda_x M^v$
		$M\text{`}N$	M^v to x in (N^v to z in (x`force z))
1	1	$\langle\rangle$	return $\langle\rangle$
		pm M as $\langle\rangle$ in N	M^v to z in (pm z as $\langle\rangle$ in N^v)
$A \times A'$	$A^v \times A'^v$	$\langle M, N\rangle$	M^v to x in (N^v to y in return $\langle x, y\rangle$)
		pm M as $\langle x, y\rangle$ in N	M^v to z in (pm z as $\langle x, y\rangle$ in N^v)

Figure 2.10: A CBV translation of a simple λ-calculus into CBPV.

CBN type	**CBPV type**	**CBN term**	**CBPV term**
$\underline{B}$	$\underline{B}^n$	$x_1 : \underline{B}_1, \dots, x_m : \underline{B}_m \vdash M : \underline{B}$	$x_1 : U\underline{B}_1^n, \dots, x_m : U\underline{B}_m^n; \cdot \vdash M^n : \underline{B}^n$
		x	force x
		let x be M in N	let x be (thunk M^n) in N^n
$\Sigma_{1\le i\le n} \underline{B}_i$	$F\Sigma_{1\le i\le n} U\underline{B}_i^n$	$\langle i, M\rangle$	return $\langle i, \mathsf{thunk}\ M^n\rangle$
		pm M as $\langle i, x\rangle$ in N_i	M^n to z in (pm z as $\langle i, x\rangle$ in N_i^n)
$\Pi_{1\le i\le n} \underline{B}_i$	$\Pi_{1\le i\le n} \underline{B}_i^n$	$\lambda_i M_i$	$\lambda_i M_i^n$
		$i\text{`}M$	$i\text{`}M^n$
$\underline{B} \Rightarrow \underline{B'}$	$(U\underline{B}^n) \ {}_F\!\multimap \underline{B'}^n$	$\lambda_x M$	$\lambda_x M^n$
		$N\text{`}M$	(thunk N^n)`M^n
1	$F1$	$\langle\rangle$	return $\langle\rangle$
		pm M as $\langle\rangle$ in N	M^n to z in (pm z as $\langle\rangle$ in N^n)
$\underline{B} \times \underline{B'}$	$F(U\underline{B}^n \times U\underline{B'}^n)$	$\langle M, N\rangle$	return $\langle$thunk M^n, thunk $N^n\rangle$
		pm M as $\langle x, y\rangle$ in N	M^n to z in (pm z as $\langle x, y\rangle$ in N^n)

Figure 2.11: A CBN translation of a simple λ-calculus into CBPV.

2.2.2 Categorical Semantics

CBPV admits a simple notion of a categorical model. We present a variation of that of [37] to allow a smooth transition to dependent types. The philosophy is to add to a model $\mathsf{self}(\mathcal{C})$ of pure simple type theory an extra (locally) indexed category $\mathcal{D}$ to model computations and stacks separately from values and to demand all appropriate negative (right adjoint) connectives in $\mathcal{D}$ and all positive (left adjoint) ones in $\mathsf{self}(\mathcal{C})$. The idea will be that values $\Gamma \vdash V : A$ denote elements of $\mathsf{self}(\mathcal{C})(\llbracket\Gamma\rrbracket)(1, \llbracket A\rrbracket)$ and that computations and stacks $\Gamma; \Delta \vdash M : \underline{B}$ denote elements of $\mathcal{D}(\llbracket\Gamma\rrbracket)(\llbracket\Gamma; \Delta\rrbracket, \llbracket\underline{B}\rrbracket)$.

Definition 2.2.1 (Simple CBPV Model)**.** *By a categorical model of simple CBPV, we shall mean the following data.*

- *A cartesian category* $(\mathcal{C}, 1, \times)$ *of* ***values****;*
- *a locally indexed category* $\mathcal{C}^{op} \xrightarrow{\mathcal{D}} \mathsf{Cat}$ *of* ***stacks*** *(and* ***computations****, in particular), that is, an indexed category such that the change of base functors are identity on objects;*
- $0, +$*-types in* $\mathsf{self}(\mathcal{C})$[12] *such that, additionally, the following obvious induced maps are bijections:*

$$\mathcal{D}(C.\Sigma_{1\leq i\leq n}C_i)(\underline{D}, \underline{D'}) \longrightarrow \Pi_{1\leq i\leq n}\mathcal{D}(C.C_i)(\underline{D}, \underline{D'});$$

- *an indexed adjunction*[13] $\mathcal{D} \underset{U}{\overset{F}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathsf{self}(\mathcal{C})$;
- $\Pi^{\multimap}_{F(-)}$*-types in* $\mathcal{D}$ *in the sense of having right adjoint functors* $-\{\mathbf{p}_{A,B}\} \dashv \Pi^{\multimap}_{F(B)} : \mathcal{D}(A) \longrightarrow \mathcal{D}(A.B)$ *satisfying the right Beck-Chevalley condition for* $\mathbf{p}$*-squares;*
- *Finite indexed products* $(\top, \&)$ *in* $\mathcal{D}$ *(finite products, stable under change of base);*

[12] This amounts to having distributive finite coproducts in $\mathcal{C}$.

[13] As Plotkin pointed out at the time of Moggi's original work on the monadic metalanguage, this gives a strong monad $T = UF$ on $\mathcal{C}$ [6].

Note that $\mathsf{self}(\mathcal{C})$ *automatically has* 1- *and* Σ*-types.*

Theorem 2.2.2 (Simple CBPV Semantics)**.** *We have a sound interpretation of CBPV in a CBPV model:*

$$\begin{array}{ll}
[\![\cdot]\!] = 1 & [\![\Gamma;\cdot]\!] = F1 \\
[\![\Gamma, x:A]\!] = [\![\Gamma]\!].[\![A]\!] & [\![\Gamma;\mathsf{nil} : \underline{B}]\!] = [\![\underline{B}]\!] \\
[\![\Gamma \vdash A]\!] = \mathsf{self}(\mathcal{C})([\![\Gamma]\!])(1,[\![A]\!]) & [\![\Gamma;\Delta \vdash \underline{B}]\!] = \mathcal{D}([\![\Gamma]\!])([\![\Gamma;\Delta]\!],[\![\underline{B}]\!]) \\
[\![U\underline{B}]\!] = U[\![\underline{B}]\!] & [\![FA]\!] = F[\![A]\!] \\
[\![\Sigma_{1\leq i\leq n}A_i]\!] = (\cdot([\![A_1]\!]+[\![A_2]\!])+\cdots)+[\![A_n]\!]) & [\![\Pi_{1\leq i\leq n}\underline{B_i}]\!] = (\cdot([\![\underline{B}_1]\!]\&[\![\underline{B}_2]\!])\&\cdots)\&[\![\underline{B}_n]\!]) \\
[\![A\times A']\!] = [\![A]\!]\times[\![A']\!] \cong \Sigma_{[\![A]\!]}[\![A']\!]\{\mathbf{p}_{[\![\Gamma]\!],[\![A]\!]}\} & [\![A\ {}_F\!\multimap \underline{B}]\!] = \Pi^{\multimap}_{F([\![A]\!])}[\![\underline{B}]\!]\{\mathbf{p}_{[\![\Gamma]\!],[\![A]\!]}\} \\
[\![1]\!] = 1, &
\end{array}$$

together with the obvious interpretation of terms. The interpretation in such categories is complete in the sense that an equality of values or computations holds in all interpretations iff it is provable in the syntax of CBPV. In fact, we have a 1-1 relationship between models and theories which satisfy mutual soundness and completeness results.

Let us write T for the indexed monad UF on $\mathsf{self}(\mathcal{C})$ and ! for the indexed comonad FU on $\mathcal{D}$. We note that the translations from CBV and CBN into CBPV correspond to interpreting CBV and CBN in the Kleisli and co-Kleisli categories for T and ! respectively. More generally, we can note that the translations of figures 2.10 and 2.11 can be transformed into semantic translations which means that any CBPV model gives rise to models of the CBV and CBN λ-calculus.

Theorem 2.2.3 (Simple CBV Semantics)**.** *We obtain a sound interpretation of the CBV* λ*-calculus with* $1,\times,\Rightarrow,\Sigma_{1\leq i\leq n},\Pi_{1\leq i\leq n}$*-types in the Kleisli category for* T*:*

$$\begin{aligned}
[\![A_1,\cdots,A_n \vdash A]\!] = \mathcal{D}([\![A_1]\!].\cdots.[\![A_n]\!])(F1, F[\![A]\!]) &\cong \mathsf{self}(\mathcal{C})_T([\![A_1]\!].\cdots.[\![A_n]\!])(1,[\![A]\!]) \\
&\cong \mathsf{self}(\mathcal{C})(\cdot)_T([\![A_1]\!]\times\cdots\times[\![A_n]\!],[\![A]\!]).
\end{aligned}$$

The interpretation is complete with respect to this class of models.

Theorem 2.2.4 (Simple CBN Semantics)**.** *We obtain a sound interpretation of the CBN* λ*-calculus with* $1,\times,\Rightarrow,\Sigma_{1\leq i\leq n},\Pi_{1\leq i\leq n}$*-types in the co-Kleisli category for* !*:*

$$\begin{aligned}
[\![\underline{B}_1,\cdots,\underline{B}_n \vdash \underline{B}]\!] = \mathcal{D}(U[\![\underline{B}_1]\!].\cdots.U[\![\underline{B}_n]\!])(F1,[\![\underline{B}]\!]) &\cong \mathcal{D}_!(U[\![\underline{B}_1]\!].\cdots.U[\![\underline{B}_n]\!])(\top,[\![\underline{B}]\!]) \\
&\cong \mathcal{D}(\cdot)_!([\![\underline{B}_1]\!]\&\cdots\&[\![\underline{B}_n]\!],[\![\underline{B}]\!]).
\end{aligned}$$

The interpretation is complete with respect to this class of models.

Again, both of these results could be strengthened to the statement that we have a 1-1 relationship between models and theories which satisfy mutual soundness and completeness results.

2.2.3 A Few Words about Models

An extensive discussion of particular models as well as comparisons between CBPV models and other notions of categorical models of effects can be found in [30]. Here, we shall be very brief and just recall the following two results and provide some context for the relationship between effects and linear logic.

Theorem 2.2.5. *Let $\mathcal{D}'$ be a model of intuitionistic exponential additive multiplicative linear logic (see section 2.3) in the sense of a symmetric monoidal closed category $(\mathcal{D}', I, \otimes)$ with finite products $(\top, \&)$, finite coproducts $(0, \oplus)$ that distribute over $\otimes$ and a comonad $!$ that is induced by some adjunction*

$$\mathcal{C}' \underset{U'}{\overset{F'}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} \mathcal{D}'$$

to a cartesian monoidal category $(\mathcal{C}', 1, \times)$ with strong monoidal left adjoint F'. In that case, $\mathcal{D}'$ gives rise to a canonical model $F \dashv U : \mathsf{self}(\mathcal{C}) \leftrightarrows \mathcal{D}$ of simple CBPV where UF is a commutative monad [38].

Proof. First note that we can replace $\mathcal{C}'$ with its completion $\mathcal{C}$ under finite distributive coproducts. (Think of this as the completion under the notion of finite coproducts in the 2-category of symmetric monoidal categories and lax symmetric monoidal functors. Similarly, we should think of an adjunction with strong monoidal left adjoint as corresponding with an adjunction in this 2-category and a commutative monad as a monad in this 2-category.) Indeed, we have a full and faithful embedding $\iota : \mathcal{C}' \hookrightarrow \mathcal{C}$ and because $\mathcal{D}$ has finite distributive coproducts and F' is strong monoidal, we can extend $F' \dashv U'$ to an adjunction $F \dashv U : \mathcal{C} \leftrightarrows \mathcal{D}'$ with strong monoidal F,

where we define $F(\Sigma_{1\leq i\leq n}\iota(C_i)) := \Sigma_{1\leq i\leq n}F'C_i$ and $UD := \iota(U'D)$:

$$\begin{aligned}\mathcal{C}(\Sigma_{1\leq i\leq n}\iota(C_i), UD)) &:= \mathcal{C}(\Sigma_{1\leq i\leq n}\iota(C_i), \iota(U'(D)))\\ &\cong \Pi_{1\leq i\leq n}\mathcal{C}(\iota(C_i), \iota(U'(D)))\\ &\cong \Pi_{1\leq i\leq n}\mathcal{C}'(C_i, U'D)\\ &\cong \Pi_{1\leq i\leq n}\mathcal{D}'(F'C_i, D)\\ &\cong \mathcal{D}'(\Sigma_{1\leq i\leq n}F'C_i, D)\\ &=: \mathcal{D}'(F\Sigma_{1\leq i\leq n}\iota(C_i), D).\end{aligned}$$

We define the indexed category $\mathcal{C}^{op} \xrightarrow{\mathcal{D}} \mathsf{Cat}$ as having the same objects as $\mathcal{D}'$ in each fibre and morphisms $\mathcal{D}(A)(B, C) := \mathcal{D}'(B, FA \multimap C)$. To see that the monad is commutative, we note that a commutative monad is the same as a lax symmetric monoidal monad [39]. □

In this way, we can see that linear logic describes certain commutative effects. CBPV models for possibly non-commutative effects can be obtained from any monad model [32] of the monadic metalanguage [30].

Theorem 2.2.6. *Any bicartesian closed category $\mathcal{C}$ with a strong monad T gives rise to a CBPV model.*

Proof. It is well-known that the forgetful functor $\mathcal{C}^T \to \mathcal{C}$ creates finite products (limits). Recall that in this setting the Eilenberg-Moore category $\mathcal{C}^T$ has Kleisli exponentials, in the sense of algebras $A \multimap_F k$ of homomorphisms from free algebras μ_A to general algebras k ($A \Rightarrow Uk$ inherits a T-algebra structure from k) [32]. We define the indexed category $\mathcal{C}^{op} \xrightarrow{\mathcal{D}} \mathsf{Cat}$ to have the same objects as the Eilenberg-Moore category $\mathcal{C}^T$ in each fibre and morphisms $\mathcal{D}(A)(k, l) := \mathcal{C}^T(k, A \multimap_F l)$. $F \dashv U$ is interpreted by the usual Eilenberg-Moore adjunction. □

2.2.4 Operational Semantics

Importantly, CBPV admits a natural operational semantics that, for terms of ground type, reproduces the usual operational semantics of CBV and CBN under the specified translations into CBPV [30] and that can easily be extended to incorporate various effects that we may choose to add to pure CBPV. We very briefly discuss this.

First, we note that Levy chooses to only provide an operational semantics for computations without **complex values**. Complex values are defined to be values containing pm as in - and let be in -constructs. He does this as complex values introduce arbitrary choices into the operational semantics, as we need to decide when to evaluate them. As the normalization of values does not produce effects (in particular, values are equal to their normal form; they are static), all reasonable evaluation strategies for them are observationally indistinguishable and we could choose our favourite.

While excluding complex values from computations is not a terrible restriction (one can show that any computation is judgementally equal to one not having any complex values as subterms and the CBV and CBN translations do not produce any complex values), we do not see the need to introduce this restriction. Indeed, complex values will turn out to be useful in a dependently typed CBPV, when we want to substitute them in dependent types. For instance, we might want to define a dependent type through a case distinction.

For that purpose, let us point out that the β-reductions for complex values are directed versions (left-to-right) of their equations in the left hand column of figure 2.9. We use the parallel nested closure of β-reductions as our notion of reduction for values. Following the usual argument of logical relations [40], this gives us a strong normalization result for values. Let us write V_{nf} for the normal form of a value V. We write $V_{!\mathsf{nf}}$ to indicate a value which is not in normal form.

We present a small-step operational semantics for CBPV computations in terms of a simple abstract machine that Levy calls the CK-machine. The configuration of such a machine consists of a pair M, K where $\Gamma; \cdot \vdash M : \underline{B}$ is a computation and $\Gamma; \mathsf{nil} : \underline{B} \vdash K : \underline{C}$ is a compatible stack. We call $\underline{C}$ the type of the configuration.

The idea is that transitions are defined on a pair of a computation and a stack, rather than simply on computations, to be able to correctly model the operational behaviour of sequencing and function application: we push parts of a computation to the stack if other parts need to be executed first before we can pop the stack and resume.

The initial configurations, transitions (which embody left-to-right-directed versions of the β-rules of our equational theory) and terminal configurations in the evaluation of a computation $\Gamma; \cdot \vdash M : \underline{C}$ on the CK-machine are specified by figure 2.12 where we use the following abbreviations for stacks

$$V :: K := \mathsf{let\ nil\ be}\ V\text{`}\mathsf{nil\ in}\ K$$
$$j :: K := \mathsf{let\ nil\ be}\ j\text{`}\mathsf{nil\ in}\ K$$
$$[\cdot]\ \mathsf{to}\ x\ \mathsf{in}\ M :: K := \mathsf{let\ nil}_1\ \mathsf{be}\ (\mathsf{nil}_2\ \mathsf{to}\ x\ \mathsf{in}\ M)\ \mathsf{in}\ K.$$

We recall the following basic results about this operational semantics from [30, 41].

Theorem 2.2.7 (Determinism, Strong Normalization and Subject Reduction)**.** *For every configuration of the CK-machine, at most one transition applies. No transition applies precisely when the configuration is terminal. Every configuration of type $\underline{C}$ reduces, in a finite number of transitions, to a unique terminal configuration of type $\underline{C}$.*

Proof. The only real modification from [30, 41] is that our terms include complex values. It is well-known that in a pure simple type theory with projection products and coproducts, the reductions are strongly normalizing and satisfy subject reduction. This shows that the transitions for complex values do not break strong normalization or subject reduction. The distinction between values in normal form and those not in normal form ensures that determinism still applies. □

2.2.5 Adding Effects

So far, we have considered pure CBPV computations. Next, we add effects to them, making them into real dynamic objects in the sense that their reductions might not respect equality. We recall by example how one adds effects to CBPV. Figure 2.13 gives some examples of effects one could consider, from left to right, top to bottom:

Initial Configuration

M , nil

Transitions

$\mathsf{let}\ V_{!\mathsf{nf}}\ \mathsf{be}\ x\ \mathsf{in}\ M$	K	$\rightsquigarrow$	$\mathsf{let}\ V_{\mathsf{nf}}\ \mathsf{be}\ x\ \mathsf{in}\ M$	K
$\mathsf{let}\ V_{\mathsf{nf}}\ \mathsf{be}\ x\ \mathsf{in}\ M$	K	$\rightsquigarrow$	$M[V_{\mathsf{nf}}/x]$	K
$\mathsf{let}\ M\ \mathsf{be\ nil}\ \mathsf{in}\ L$	K	$\rightsquigarrow$	$L[M/\mathsf{nil}\,]$	K
$M\ \mathsf{to}\ x\ \mathsf{in}\ N$	K	$\rightsquigarrow$	M	$[\cdot]\ \mathsf{to}\ x\ \mathsf{in}\ N :: K$
$\mathsf{return}\ V_{!\mathsf{nf}}$	K	$\rightsquigarrow$	$\mathsf{return}\ V_{\mathsf{nf}}$	K
$\mathsf{return}\ V_{\mathsf{nf}}$	$[\cdot]\ \mathsf{to}\ x\ \mathsf{in}\ N :: K$	$\rightsquigarrow$	$N[V_{\mathsf{nf}}/x]$	K
$\mathsf{force}\ V_{!\mathsf{nf}}$	K	$\rightsquigarrow$	$\mathsf{force}\ V_{\mathsf{nf}}$	K
$\mathsf{force\ thunk}\ M$	K	$\rightsquigarrow$	M	K
$\mathsf{pm}\ V_{!\mathsf{nf}}\ \mathsf{as}\ \langle i, x\rangle\ \mathsf{in}\ M_i$	K	$\rightsquigarrow$	$\mathsf{pm}\ V_{\mathsf{nf}}\ \mathsf{as}\ \langle i, x\rangle\ \mathsf{in}\ M_i$	K
$\mathsf{pm}\ \langle j, V\rangle\ \mathsf{as}\ \langle i, x\rangle\ \mathsf{in}\ M_i$	K	$\rightsquigarrow$	$M_j[V_{\mathsf{nf}}/x]$	K
$\mathsf{pm}\ V_{!\mathsf{nf}}\ \mathsf{as}\ \langle\rangle\ \mathsf{in}\ M$	K	$\rightsquigarrow$	$\mathsf{pm}\ V_{\mathsf{nf}}\ \mathsf{as}\ \langle\rangle\ \mathsf{in}\ M$	K
$\mathsf{pm}\ \langle\rangle\ \mathsf{as}\ \langle\rangle\ \mathsf{in}\ M$	K	$\rightsquigarrow$	M	K
$\mathsf{pm}\ V_{!\mathsf{nf}}\ \mathsf{as}\ \langle x, y\rangle\ \mathsf{in}\ M$	K	$\rightsquigarrow$	$\mathsf{pm}\ V_{\mathsf{nf}}\ \mathsf{as}\ \langle x, y\rangle\ \mathsf{in}\ M$	K
$\mathsf{pm}\ \langle V, W\rangle\ \mathsf{as}\ \langle x, y\rangle\ \mathsf{in}\ M$	K	$\rightsquigarrow$	$M[V/x, W/y]$	K
$j`M$	K	$\rightsquigarrow$	M	$j :: K$
$\lambda_i M_i$	$j :: K$	$\rightsquigarrow$	M_j	K
$V_{!\mathsf{nf}}`M$	K	$\rightsquigarrow$	$V_{\mathsf{nf}}`M$	K
$V_{\mathsf{nf}}`M$	K	$\rightsquigarrow$	M	$V_{\mathsf{nf}} :: K$
$\lambda_x M$	$V :: K$	$\rightsquigarrow$	$M[V/x]$	K

Terminal Configurations

$\mathsf{return}\ V_{\mathsf{nf}}$	nil
$\lambda_i M_i$	nil
$\lambda_x M$	nil
$\mathsf{force}\ V_{\mathsf{nf}}^{x'}$	K
$\mathsf{pm}\ V_{\mathsf{nf}}^{x'}\ \mathsf{as}\ \langle i, x\rangle\ \mathsf{in}\ M_i$	K
$\mathsf{pm}\ V_{\mathsf{nf}}^{x'}\ \mathsf{as}\ \langle\rangle\ \mathsf{in}\ M$	K
$\mathsf{pm}\ V_{\mathsf{nf}}^{x'}\ \mathsf{as}\ \langle x, y\rangle\ \mathsf{in}\ M$	K

Figure 2.12: The behaviour of the CK-machine in the evaluation of a computation $\Gamma; \cdot \vdash M : \underline{C}$. We write $V_{\mathsf{nf}}^{x'}$ for a non-canonical normal form of a value which has at least one free identifier x'. Every time we encounter a computation term former taking a value as an argument, we first normalize the value before proceeding to the corresponding transition for the term former. We leave out type annotations.

divergence, recursion, printing an element m of some monoid $\mathcal{M}$, erratic choice from finitely many alternatives, errors e from some set E, writing a global state $s \in S$ and reading a global state to s. We note that the framework fits many more examples like probabilistic erratic choice, local references and control operators [30].

The small-step semantics of divergence, recursion, erratic choice and errors can easily be explained on our CK-machine as it is. This is summed up in figure 2.14. For the operational semantics of printing and state, we need to add some hardware to our machine. For that purpose, a configuration of our machine will now consist of a quadruple M, K, m, s where M, K are as before, m is an element of our printing

$$\frac{}{\Gamma;\cdot \vdash \mathsf{diverge} : \underline{B}} \qquad \frac{\Gamma, z : U\underline{B};\cdot \vdash M : \underline{B}}{\Gamma;\cdot \vdash \mu_z M : \underline{B}} \qquad \frac{\Gamma;\cdot \vdash M : \underline{B}}{\Gamma;\cdot \vdash \mathsf{print}\ m\ .\ M : \underline{B}} \qquad \frac{\{\Gamma;\cdot \vdash M_i : \underline{B}\}_{1\leq i\leq n}}{\Gamma;\cdot \vdash \mathsf{choose}_i(M_i) : \underline{B}}$$

$$\frac{}{\Gamma;\cdot \vdash \mathsf{error\ e} : \underline{B}} \qquad \frac{\Gamma;\cdot \vdash M : \underline{B}}{\Gamma;\cdot \vdash \mathsf{write}\ s\ .\ M : \underline{B}} \qquad \frac{\{\Gamma;\cdot \vdash M_s : \underline{B}\}_{s\in S}}{\Gamma;\cdot \vdash \mathsf{readto}_s(M_s) : \underline{B}}$$

Figure 2.13: Some examples of effects we could add to CBPV. μ_z is a name binding operation that binds the identifier z and $\mathsf{choose}_i()$ and $\mathsf{readto}_s()$ bind the indices i and s respectively.

Transitions

$\mathsf{diverge}$	, K	$\rightsquigarrow$	$\mathsf{diverge}$	, K		
$\mu_z M$	, K	$\rightsquigarrow$	$M[\mathsf{thunk}\ \mu_z M/z]$	, K		
$\mathsf{choose}_i(M_i)$	, K	$\rightsquigarrow$	M_j	, K		

Terminal Configurations

$\mathsf{error\ e}$, K

Figure 2.14: The operational semantics for divergence, recursion, erratic choice and errors.

Transitions

$\mathsf{print}\ n\ .\ M$	, K	, m	, s	$\rightsquigarrow$	M	, K	, $m * n$	, s				
$\mathsf{write}\ s'\ .\ M$	, K	, m	, s	$\rightsquigarrow$	M	, K	, m	, s'				
$\mathsf{readto}_{s'}(M_{s'})$	, K	, m	, s	$\rightsquigarrow$	M_s	, K	, m	, s				

Figure 2.15: The operational semantics for printing and writing and reading global state.

monoid $(\mathcal{M}, \epsilon, *)$ which models some channel for output and s is an element of our finite pointed set of states (S, s_0) which is the current value of our storage cell. We lift the operational semantics of all existing language constructs to this setting by specifying that they do not modify m and s, that terminal configurations can have any value of m and s and that initial configurations always have value $m = \epsilon$ and $s = s_0$ for the fixed initial state s_0. Printing and writing and reading the state can now be given the operational semantics of figure 2.15.

We can try to extend the results of the previous section to this effectful setting and indicate when they break [30].

Theorem 2.2.8 (Determinism, Strong Normalization and Subject Reduction)**.** *Every transition respects the type of the configuration. No transition occurs precisely if we are in a terminal configuration. In absence of erratic choice, at most one*

CBV Term M	**CBPV Term** M^v	**CBN Term** M	**CBPV Term** M^n
$\mathsf{op}(M)$	$\mathsf{op}(M^v)$	$\mathsf{op}(M)$	$\mathsf{op}(M^n)$
$\mu_x M$	$\mu_z(\mathsf{force}\ z\ \mathsf{to}\ x\ \mathsf{in}\ M^v)$	$\mu_z M$	$\mu_z M^n$

Figure 2.16: The CBV and CBN translations for effectful terms. z is assumed to be fresh in the CBV translation $\mu_x M$. For our examples, $\mathsf{op}(-)$ ranges over $\mathsf{diverge}$, $\mathsf{error}\ \mathsf{e}$, $\mathsf{choose}_i(-)$, $\mathsf{print}\ m\ .\ (-)$, $\mathsf{readto}_s(-)$ and $\mathsf{write}\ s\ .\ (-)$.

transition applies to each configuration. In absence of divergence and recursion, every configuration reduces to a terminal configuration in a finite number of steps.

We can again translate effectful CBV and CBNλ-calculi into CBPV with the appropriate effects as is indicated in figure 2.16.

Let us write $M \Downarrow N, m, s$ for a closed term $\cdot;\cdot \vdash M : \underline{B}$ if $M, \mathsf{nil}\ , \epsilon, s_0$ reduces to the terminal configuration $N, \mathsf{nil}\ , m, s$. We call this the **big-step semantics** of CBPV. Recall that, at least for terms of ground type, CBPV induces the usual operational semantics via the CBV and CBN translations [41].

Theorem 2.2.9. *The big-step semantics for CBPV induces the usual CBV and CBN big-step semantics for terms of ground type, via the respective translations.*

We list the basic equations we would typically demand for the effects we consider in figure 2.17. In addition to these general equations, we could include the usual specific equations from the algebraic theory for $\mathsf{op}(-)$ (like the lookup-update algebra equations for global state of Plotkin and Power [42]). In a dependently typed setting, we have to decide which effect specific equations to include as judgemental equalities, such that the type checker has to be able to decide them, and which to include as propositional equalities for manual reasoning by the user.

Although one could write down an equational theory for these effects and a corresponding categorical semantics, in which case one would obtain soundness and completeness properties for the CBV and CBN translations, we choose not to do so here for reasons of space. For this, we refer the reader, for instance, to [30, 42]. The important thing to note is that the CBV and CBN translations for effectful CBPV typically result in a broken η-law for function types and sum types respectively as is well-known from traditional CBV and CBN semantics of effectful type theories.

$K[\mathsf{op}(M)/\mathsf{nil}\,] = \mathsf{op}(K[M/\mathsf{nil}\,])$ $\qquad$ $\mu_z M = M[\mathsf{thunk}\ \mu_z M/z]$

Figure 2.17: For effects, we demand the basic equation defining the fixpoint combinator μ_z as well as algebraicity equations for all effects $\mathsf{op}(-)$ (in addition to the usual equational theory for the specific operations $\mathsf{op}(-)$, like the Plotkin-Power equations for global state). These algebraicity equations state that a stack K is a homomorphism of the algebra defined by the operations $\mathsf{op}(-)$. For our examples, $\mathsf{op}(-)$ ranges over $\mathsf{diverge}$, $\mathsf{error\ e}$, $\mathsf{choose}_i(-)$, $\mathsf{print}\ m\ .\ (-)$, $\mathsf{readto}_s(-)$ and $\mathsf{write}\ s\ .\ (-)$.

2.3 Linear Types

Linear logic was introduced by Girard in [43] as a resource sensitive refinement of intuitionistic logic, which was inspired by the structure present in certain models for system F. From a modern perspective, we can see the essence of linear logic, or rather that of its proof term calculus, the linear λ-calculus, to already be present in [44]. Put simply, the linear λ-calculus provides an internal language for symmetric monoidal closed categories in the same way that the ordinary (simply-typed) λ-calculus does for cartesian closed categories. The system is resource sensitive in the sense that a possibly non-cartesian monoidal structure does not generally admit copying and deleting morphisms. This means that, in the corresponding logic or λ-calculus, we lose the structural rules of contraction and weakening. This results in an exposure of the frequency with which assumptions are used in proofs in logic and gives us a better grip on complexity in the λ-calculus.

To be precise, the logic that is arises from this linear λ-calculus via a Curry-Howard correspondence is referred to as **(multiplicative) intuitionistic linear logic**. This system is strictly more general than the **(multiplicative-additive-exponential) classical linear logic** studied by Girard. This latter system differs from the former in three significant ways.

1. It admits a **classical** duality in the sense that there is a dualising object[14] $\bot$ for the implication $\multimap$. At the same time it still admits a non-trivial term calculus. This is one of the historically surprising aspects of the system, in the

[14]That is, an object $\bot$ such that the canonical evaluation morphism $A \longrightarrow (A \Rightarrow \bot) \Rightarrow \bot$ is an isomorphism for all objects A.

light of the Joyal lemma (see e.g. [45]), which states that a cartesian closed category with a dualising object is a preorder.

2. It comes equipped with a comodality (that is, a $\Box$-modality) !, called the **exponential**, which recovers the structural rules.

3. It comes equipped with an additional notion of conjunction, called the **additive conjunction**, written &, to be contrasted with the multiplicative conjunction $\otimes$ from multiplicative intuitionistic linear logic. It represents an internal choice, rather than a simultaneous occurence of resources. Classical duality also gives us an **additive disjunction** $\oplus$, which represents an external choice, and which, in absence of classical duality, we might choose to include in our linear logic as a primitive.

It will be the level of greater generality of (multiplicative) intuitionistic linear logic, including the more specific cases of systems à la Girard, that we think of when we refer to **linear logic**.

2.3.1 Categorical Semantics

Intuitionistic linear logic admits a relatively simple, though not historically uncontroversial, sound and complete categorical semantics which we describe here briefly. Our principal reference will be [34]. Some more background is provided in [46] and [47].

There are several notions of model in use that are equivalent, which only differ in their interpretation of !. It is clear that ! should be interpreted as a comonad, but the exact properties of those comonad can be stated in several equivalent ways. For our purposes, the notion of a linear/non-linear model of [33] is the best fit.

Definition 2.3.1 (Linear/Non-Linear Adjunction)**.** *By a linear/non-linear adjunction, we shall mean a lax symmetric monoidal adjunction (i.e. an adjunction in the 2-category of symmetric monoidal categories and lax symmetric monoidal functors)*

$$(\mathcal{C}, 1, \times) \underset{U}{\overset{F}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} (\mathcal{D}, I, \otimes)$$

from a symmetric monoidal category $\mathcal{D}$ to a cartesian monoidal category $\mathcal{C}$. An equivalent condition for the adjunction $F \dashv U$ to be lax symmetric monoidal is for the functor F to be strongly symmetric monoidal, in which case the symmetric oplax structure on F transfers along the adjunction to a symmetric lax structure on U.

Definition 2.3.2 (Model of Linear Logic)**.** *A model of intuitionistic linear logic with I- and $\otimes$-types consists of a symmetric monoidal category $\mathcal{D}$. The model supports...*

- *$\multimap$- types iff $\mathcal{D}$ is closed (as a multicategory or as a symmetric monoidal category in case we have I- and $\otimes$-types);*
- *$\top$- and $\&$-types iff $\mathcal{D}$ has finite products;*
- *0- and $\oplus$-types iff $\mathcal{D}$ has distributive coproducts (or coproducts in the multicategorical sense);*
- *!-types iff $\mathcal{D}$ is equipped with a comonad ! which arises as FU for a linear/non-linear adjunction $F \dashv U : (\mathcal{C}, 1, \times) \leftrightarrows (\mathcal{D}, I, \otimes)$.*

2.3.2 Syntax

As is the case for its categorical semantics, there are many different roughly equivalent syntactic proof calculi for linear logic. In order to allow a natural generalization to dependent types (which most naturally come in a natural deduction formulation), we choose a calculus in natural deduction style, rather than a sequent calculus. Of the natural deduction formalisms for linear logic, the two most mature options are Barber and Plotkin's **dual intuitionistic linear logic (DILL)** [34] and Benton's **linear/non-linear (LNL) calculus** [33].

DILL chooses to work with a single typing judgement $\Gamma; \Delta \vdash b : B$ and is closer to Girard's original formulations of linear logic. It uses a dual context $\Gamma; \Delta$, however, which consist of a cartesian region Γ, in which the structural rules of weakening and contraction are valid, and a linear region Δ, in which they are not. This separation of context should be seen as a metaoperation internalising !,

which was missing from Girard's formulations, just as $\otimes$ internalises the comma in the context and $\multimap$ internalises the turnstyle $\vdash$. We should see DILL as an internal language for $\mathcal{D}$ of definition 2.3.2.

The LNL calculus, by contrast, should be seen as providing an internal language for both $\mathcal{C}$ and $\mathcal{D}$ (including their relationship through $F \dashv U$) of definition 2.3.2. It adds on top of the linear typing judgement $\Gamma; \Delta \vdash b : B$ of DILL (which models the morphisms $\mathcal{D}$) a cartesian typing jugement $\Gamma \vdash a : A$ to model the morphisms of $\mathcal{C}$. This means we – in particular – have two kinds of types: linear types B and cartesian types A. Here, Γ consists of cartesian types and Δ of linear types[15]. In a model of the LNL calculus, $\mathcal{C}$ is considered part of the structure of the model, while, for a model of DILL, we only demand the existence of a linear/non-linear adjunction to some cartesian category $\mathcal{C}$.

We have chosen to work with DILL in this thesis and generalise it with a notion of type dependency (see chapter 3), mostly because it is closer to what we believe most people understand to be the essence of linear logic and because it seems to be more widely used. However, the LNL calculus can be useful to keep in mind in order to see better how CBPV generalises linear logic to non-commutative effects in a sense. We choose not to elaborate on the syntax of linear logic, here, as [33, 34] are excellent references for the syntax of DILL and the LNL calculus, respectively.

2.3.3 Girard Translations

An important aspect of linear type theory is that we have two translations of cartesian type theory (with commutative effects) into it, called the Girard translations. It turns out that, from the point of view of CBPV, these are simply the CBN and CBV translations, in disguise. Indeed, the point we like to stress in this thesis is that the LNL calculus is essentially the same as CBPV for commutative effects with the extra connectives of $\otimes$ and $\multimap$. It might seem as if DILL is a serious restriction in expressive power compared to the LNL calculus. In particular, from the point of view of CBPV, it seems as if DILL only allows the CBN translation

[15] We can read a DILL context $\Gamma; \Delta$ as the LNL context $U\Gamma; \Delta$, where we apply U to all types in Γ.

of cartesian type theory (known as the first Girard translation in the context of linear logic) as we are missing value types. However, the linear connectives $\otimes$ and $\multimap$ allow us to express the CBV translation purely in terms of computation types. Altough this was already known to Girard, he thought this second translation was "not of much interest" and stressed the importance of his first (CBN) translation [38]. It is precisely the absence of the connectives $\otimes$ and $\multimap$ for non-commutative effects (particularly the latter) that forces CBPV to consider two separate typing judgements, while linear logic can be formulated equally well using only one.

For completeness sake, figure 2.18 shows the CBN and CBV translations of cartesian type theory into DILL, at least at the level of types. Note that as we might be working with an effectful cartesian type theory, in general, we distinguish between product types with a pattern matching eliminator, which we denote $A_1 \times \cdots \times A_n$, and product types with a projection eliminator, which we denote $\Pi_{1 \leq i \leq n} A_i$. We trust that the reader can fill in the definitions at the level of terms, from the corresponding translations for CBPV. While $(-)^f$ corresponds precisely to the CBN translation $(-)^n$ of CBPV, $(-)^s$ can be read as the adjoint of $(-)^v$ in a sense. Indeed, while $(-)^v$ sends a term $x_1 : A_1, \ldots, x_n : A_n \vdash M : A$ to a term $x_1 : A_1^v, \ldots, x_n : A_n^v; \cdot \vdash M^v : FA^v$, $(-)^s$ sends it to the equivalent term $\cdot; x_1 : A_1^s, \ldots, x_n : A_n^s \vdash M^s : A^s$, where $A^s = FA^v$. This equivalence follows because $F(A_1 \times \cdots \times A_n) \cong FA_1 \otimes \cdots \otimes FA_n$.

We would like to point out that the conventional Girard translations choose to use projection products for the CBN translation and pattern matching products for the CBV translation, to make sure that η survives in either case. Note that there is no analogous way of salvaging the η-law for sum types in CBN. We also note that we only need additive conjunctions for the CBN translation of projection products.

Note that the usual practice of constructing models of cartesian type theory out of models of linear type theory $\mathcal{D}$ by taking the co-Kleisli category $\mathcal{D}_!$ for $!$ precisely is the semantic equivalent of Girard's first translation. As far as we are aware, there is no well-known categorical construction corresponding to Girard's

Cartesian Type A	**CBN Translation** A^f	**CBV Translation** A^s
1	I	I
$A_1 \times A_2$	$!A_1^f \otimes !A_2^f$	$A_1^s \otimes A_2^s$
$\Pi_{1 \leq i \leq n} A_i$	$A_1^f \& \ldots \& A_n^f$	$!A_1^s \otimes \ldots \otimes !A_n^s$
$A_1 \Rightarrow A_2$	$!A_1^f \multimap A_2^f$	$!(A_1^s \multimap A_2^s)$
0	0	0
$A_1 + A_2$	$!A_1^f \oplus !A_2^f$	$A_1^s \oplus A_2^s$

Figure 2.18: The definitions of the CBN and CBV translations of cartesian type theory into DILL, also known as the first and second Girard translation, respectively.

second translation, perhaps because it relies on the specific properties of ! as a comonad (its compatibility with the monoidal structure on $\mathcal{D}$).

2.3.4 Concrete Models

2.3.4.1 Commutative Computational Effects

In section 2.2.3, we saw that linear logic gives rise to certain models for CBPV for commutative effects. In fact, a following partial converse result can be obtained. A similar result was stated without proof or attribution[16] in [38], but we provide a construction here, in order to generalize it later.

Theorem 2.3.3. *Let $\mathcal{C}$ be a cartesian closed category with a commutative monad T, where $\mathcal{C}$ additionally has equalisers and the Eilenberg-Moore category $\mathcal{C}^T$ has reflexive coequalisers[17]. Then, $\mathcal{C}^T$ is symmetric monoidal closed and has finite products to interpret additive conjunctions and the Eilenberg-Moore adjunction $F \dashv U$ defines a linear/non-linear adjunction.*

Proof. The statement about additive conjunctions follows from the well-known result that the forget functor from the Eilenberg-Moore category creates limits.

For two algebras $k, l \in \mathcal{C}^T$, we define an object $k \overset{U}{\multimap} l$ of $\mathcal{C}$ as the equaliser (which represents the subobject of morphisms satisfying the homomorphism equations)

$$k \overset{U}{\multimap} l \overset{m}{\hookrightarrow} Uk \Rightarrow Ul \underset{\lambda_{f:Uk \Rightarrow Ul} k; f}{\overset{\lambda_{f:Uk \Rightarrow Ul} Tf; l}{\rightrightarrows}} TUk \Rightarrow Ul.$$

[16] We believe the result on closure should be attributed to [48] while the construction of the symmetric monoidal structure might be inspired by the results of [49].

[17] In fact, [50] provides an alternative construction for $\otimes$-types for which we demand instead all coequalisers in $\mathcal{C}$.

We note that we can equip $k \overset{U}{\multimap} l$ with a T-algebra structure if T is a commutative monad. Indeed, using the morphism $T(A \Rightarrow B) \overset{\phi_{A,B}}{\longrightarrow} A \Rightarrow TB$ which exists for every strong monad, we can define the map

$$T(k \overset{U}{\multimap} l) \xrightarrow{Tm} T(Uk \Rightarrow Ul) \xrightarrow{\phi_{Uk,Ul}} Uk \Rightarrow TUl \xrightarrow{\lambda_f f;l} Uk \Rightarrow Ul.$$

Commutativity of the monad gives us that this map is equalising, so we obtain a unique factorisation of the map over $k \overset{U}{\multimap} l$, which gives us our algebra structure $k \multimap l$.

For $\otimes$-types note that we have the following reflexive fork in $\mathcal{C}^T$, where we write $TA \times TB \overset{t_{A,B}}{\longrightarrow} T(A \times B)$ for the left or right pairing for the strong monad T (it doesn't matter which, as T is commutative):

$$F(Uk \times Ul) \xrightarrow{F\langle \eta_{Uk}, \eta_{Ul} \rangle} F(UFUk \times UFUl) \underset{F(t_{Uk,Ul});\epsilon_{F(Uk \times Ul)}}{\overset{F\langle k,l \rangle}{\rightrightarrows}} F(Uk \times Ul).$$

Taking the coequaliser of this fork gives us our interpretation of $k \otimes l$. Given morphisms $k \xrightarrow{\phi} k'$ and $l \xrightarrow{\psi} l'$, we easily see that we get natural transformations between the respective coequaliser diagrams (using the homomorphism laws and the naturality of ϵ and t), which, therefore, give us morphisms $k \otimes l \overset{\phi \otimes \psi}{\longrightarrow} k' \otimes l'$. This is easily seen to make $\otimes$ a functor in each argument. Using the commutativity of T, we can show that it is, in fact, a bifunctor. Using the strength and pairing, we can always define a cocone on the coequaliser diagram which gives rise to an associator $k \otimes (l \otimes m) \longrightarrow (k \otimes l) \otimes m$. Next, observe that $FA \otimes FB \cong F(A \times B)$. To see this, note that for $k = FA$ and $l = FB$, we have that $F(UFA \times UFB) \overset{F(t_{A,B});\epsilon_{F(A \times B)}}{\longrightarrow} F(A \times B)$ is a cocone for the diagram above. Moreover, it is easily seen to have a section $F\langle \eta_A, \eta_B \rangle$, making it into a split epi. Given another cocone ψ for the diagram, we can now define a factorisation over $F(t_{A,B});\epsilon_{F(A \times B)}$ by $F(\langle \eta_A, \eta_B \rangle);\psi$, which is unique as our cocone is an epi. Similarly, we can see that $F1 \otimes k \cong k \cong k \otimes F1$, showing that $I := F1$ makes $\otimes$ into a monoidal structure on $\mathcal{C}^T$. We can further note that commutativity of the monad means that the braiding of $\times$ gives us a cocone on the coequaliser diagram which, gives rise to a braiding for $\otimes$, which inherits to property of being involutive from the braiding of $\times$. The triangle, pentagon and

hexagon identities all follow from the universal property of the coequaliser defining $\otimes$. We conclude that $\otimes$ is a symmetric monoidal structure.

Using the universal property of the coequaliser defining $\otimes$ as well as the naturality of t and ϵ, the definition of a T-homomorphism and the universal property of the equaliser defining $\multimap$, it is a straightforward calculation to establish that $B \otimes (-) \dashv B \multimap (-)$. □

The condition that $\mathcal{C}^T$ has reflexive coequalisers can, of course, be reduced to $\mathcal{C}$ having such coequalisers and T preserving them. This happens, for instance, when T is induced by a finitary algebraic theory, as finite powers in a cartesian closed category preserve reflexive coequalisers [51] (section D5.3).

Remark 2.3.4 (A Linear Logic for Non-Commutative Effects?)**.** *In the light of theorem 2.3.3, it is tempting to wonder if we can define a similar, perhaps non-commutative, linear logic to describe non-commutative computational effects. It is clear that theorem 2.3.3 would not straightforwardly generalise to a non-commutative setting, however, as it has been shown in [52] that none of the categories of magmas, monoids, groups and rings admit a monoidal biclosed structure. At the same time, they arise as Eilenberg-Moore categories for a (strong) monad on a complete cocomplete cartesian closed category (*Set*). In fact, [53] shows that for a strong monad T, $k \overset{U}{\multimap} l$ above is a subalgebra of $Uk \Rightarrow Ul$ (which is the carrier of the Kleisli exponential $Uk \ {}_F\!\multimap l$, which is well-known to exist as a T-algebra) for all T-algebras k and l if and only if T is commutative. The construction above, inspired by [49], however, does yield a suitable (not necessarily symmetric, non-biclosed) premonoidal structure (see [54]) on on categories of algebras (with reflexive coequalisers) for arbitrary strong monads.*

In fact, if we do not have the appropriate limits and colimits, we can always extend our model to incorporate them.

Theorem 2.3.5. *Every cartesian closed category $\mathcal{C}$ with a commutative monad T embeds fully and faithfully into a model of intuitionistic exponential additive multiplicative linear logic inducing the monad T.*

Proof. Let $\widehat{\mathcal{C}}$ be the category of presheaves on $\mathcal{C}$ (its cocompletion). We note that the Yoneda embedding defines a strict 2-functor from the 2-category of categories to the 2-category of cocomplete categories (computing its action on morphisms by taking Yoneda extensions). In fact, using the Day convolution [55], it defines a strict 2-functor from the 2-category of symmetric monoidal categories (with lax symmetric monoidal functors and symmetric monoidal natural transformations) $\mathfrak{SMCat}$ to the (sub-) 2-category of cocomplete symmetric monoidal categories $\mathfrak{cCSMCat}$. Noting that a commutative monad is precisely the same as a monad in $\mathfrak{SMCat}$ (Proposition 20 in [33]) and that 2-functors preserve monads, we get a monad in $\mathfrak{cCSMCat}$ which is a cocontinuous commutative monad $\widehat{T}$ on $\widehat{\mathcal{C}}$, which restricts to T on $\mathcal{C}$. Note that $\widehat{\mathcal{C}}$ is bicartesian closed with equalisers and coequalisers (a topos even) and that $\widehat{T}$ preserves colimits (in particular, reflexive coequalisers), so $\widehat{\mathcal{C}}^{\widehat{T}}$ has reflexive coequalisers. Therefore, we can apply theorem 2.3.3 for the result that $\widehat{\mathcal{C}}^{\widehat{T}} \leftrightarrows \widehat{\mathcal{C}}$ defines a model of intuitionistic exponential additive multiplicative linear logic. □

Remark 2.3.6. *We see that intuitionistic linear logic almost precisely describes all commutative effects. Still, this point of view does not seem to be widely held. Perhaps this is due to the fact that in the (initial) syntactic model of linear logic, a so-called principle of uniformity of threads (called such because it implies the usual principle $\mathcal{D}(!A, B) \cong \mathcal{D}(!A, !B)$) holds [56]: the unit of the adjunction $F \dashv U$ is an isomorphism $\mathsf{id}_{\mathcal{C}} \cong UF =: T$. In this sense, the free linear logic model does not describe any effects from the monadic point of view. All its interesting information is contained in the comonad $! := FU$ of the adjunction.*

2.3.4.2 Scott Domains and Strict Functions

A simple model of linear type theory (with recursion, in the sense of a model of CBPV with recursion) can be built from Scott domains and strict functions. Here, $\mathcal{D}$ has as objects Scott domains (i.e. bounded complete, directed complete, algebraic cpos) and as morphisms strict (preserving the bottom element $\bot$) continuous (preserving directed colimits) functions between them. If we define $\mathcal{C}$ to be the category of Scott predomains (i.e. countable disjoint unions of Scott domains)

and continuous functions, we can note that the inclusion U of $\mathcal{D}$ into $\mathcal{C}$ has a left adjoint F which adjoins a new bottom element. $\mathcal{D}$ is easily seen to have a terminal object $\top$ (the one-element domain) and binary products $A\&B$ (the set-theoretic product, equipped with the product order $\langle a, b\rangle \leq \langle a', b\rangle := a \leq a \ \wedge \ b \leq b'$). The same is true for $\mathcal{C}$ where we write the cartesian structure $(1, \times)$. $\mathcal{D}$ also supports $\multimap$-types, where $A \multimap B$ is the set of strict continuous function from A to B under the pointwise order ($f \leq g := \forall_{x \in A} f(x) \leq g(x)$). We can note that $A \multimap -$ has a left adjoint $A \otimes -$ which gives rise to a symmetric monoidal closed structure on $\mathcal{D}$: $A \otimes B$ is defined as the smash product $\{\langle a, b\rangle \in UA\&UB \mid a \neq \bot \ \wedge \ b \neq \bot\} \cup \{\bot\}$. This has a unit $I := \{\bot \leq \top\}$. Note that this monoidal structure makes F into a strong symmetric monoidal functor. We see that we have a model of linear logic with $\top, \&, I, \otimes, \multimap$ and !-types.

2.3.4.3 Coherence Spaces

One of the most canonical kinds of denotational semantics of linear logic – and, in fact, the original motivation for Girard to introduce linear connectives – is found in stable domain theory and its linear decomposition through coherence spaces. Imposing the property of stability on top of continuity can be seen as taking a step closer to (what is definable in) the syntax of a functional language with recursion.

We briefly recall some of the definitions in Girard's coherence space model of (classical) linear logic. The model is given by the category Coh of coherence spaces and cliques. Its objects are coherence spaces $(A, \frown_A)$ (or, undirected graphs): a set A of tokens with a reflexive relation $\frown_A$, called the coherence relation. We write $\smile_A$ for the irreflexive part of $\frown_A$, $\not\frown_A$ for the complement of $\frown_A$, and $\asymp_A$ for the complement of $\smile_A$. Before we define the morphisms of the category, we describe a few operations on objects.

Given a coherence space A, we define its linear negation $A^\perp$ as the space with the same underlying set A of tokens and coherence relation $\asymp_A$:

$$a \frown_{A^\perp} a' := \neg(a \smile_A a').$$

Given coherence spaces A and B, we define their multiplicative conjunction $A \otimes B$ as having underlying set the product $A \times B$ of the underlying sets of A and B and coherence relation

$$(a, b) \frown_{A \otimes B} (a', b') := a \frown_A a' \wedge b \frown_B b'.$$

We can then define their multiplicative disjunction $A \parr B$, through De Morgan duality,

$$A \parr B := (A^\perp \otimes B^\perp)^\perp,$$

and their (multiplicative) linear implication $A \multimap B$,

$$A \multimap B := A^\perp \parr B.$$

(Explicitly, $(a, b) \frown_{A \parr B} (a', b') := a \frown_A a' \vee b \frown_B b'$ and $(a, b) \frown_{A \multimap B} (a', b') := a \frown_A a' \Rightarrow b \frown_B b'$.)

We can define their additive disjunction $A \oplus B$ as the disjoint union of coherence spaces, where never $a \frown_{A \oplus B} b$ if $a \in A$ and $b \in B$ and $\frown_{A \oplus B}$ restricts to $\frown_A$ and $\frown_B$, and we can define their additive conjunction $A \& B$, through De Morgan duality, as

$$A \& B := (A^\perp \oplus B^\perp)^\perp.$$

(Explicitly, always $a \frown_{A \& B} b$ for $a \in A$, $b \in B$ and $\frown_{A \& B}$ restricts to $\frown_A$ and $\frown_B$.)

The operations $\otimes$, $\parr$, $\oplus$, and $\&$ also have neutral elements which we shall denote by I, $\perp$, 0, and $\top$, respectively. Indeed, $I = \perp = \{*\}$ and $0 = \top = \emptyset$ are easily seen to do the trick. (These identities between the units can be seen as degeneracies of this model of linear type theory, as they do not follow from the syntax of classical linear logic.)

We now define the morphisms:

$$\mathsf{Coh}(A, B) := \mathsf{cliques}(A \multimap B),$$

where a clique σ in A is a subset $\sigma \subseteq A$ such that $a, a' \in \sigma \Rightarrow a \frown_A a'$. We compose cliques as relations, which gives us the identity relations (which are cliques!) as the identities of our category.

We note that $(I, \otimes, \multimap)$ make Coh into a symmetric monoidal closed category, that $\top$ and $\&$ are our nullary and binary products, and that 0 and $\oplus$ are our nullary and binary (distributive) coproducts. We note that we have obtained a model of linear type theory with I-, $\otimes$-, $\multimap$-, $\top$-, $\&$-, 0-, and $\oplus$-types. In fact, as $((-)^\perp)^\perp \cong \mathsf{id}_{\mathsf{Coh}}$, we even have a model of classical linear type theory. [46]

We have a linear/non-linear adjunction between the category Stable of Scott predomains with pullbacks and continuous stable functions[18] and the category Coh of coherence spaces, $F \dashv U$:

$$(\mathsf{Stable}, 1, \times) \underset{U}{\overset{F}{\underset{\longleftarrow}{\overset{\longrightarrow}{\perp}}}} (\mathsf{Coh}, I, \otimes).$$

U takes the domain of cliques on objects and sends a clique σ in $A \multimap B$ to the continuous stable function $d \mapsto \{b \mid \exists_{a \in d}(a, b) \in \sigma\}$. F sends a predomain D to the coherence space with set of tokens the compact elements of D and coherence relation $s \frown_{FD} t := \exists_{u \in D}(s \leq u) \wedge (t \leq u)$ and sends a continuous stable function $D' \xrightarrow{f} D$ to the clique $\{(x, y) \mid y \leq f(x) \wedge \forall_{x' \leq x} y \leq f(x') \Rightarrow x = x'\}$. (Note that F is a strong monoidal functor.) We have the following bijection of homsets, which demonstrates the adjunction,

$$\begin{array}{ccc} \sigma & \longmapsto & d \mapsto \{c \mid \exists_{d' \leq d}(d', c) \in \sigma\} \\ \mathsf{Coh}(FD, C) & \underset{\mathsf{trace}}{\overset{\mathsf{fun}}{\underset{\longleftarrow}{\overset{\longrightarrow}{\cong}}}} & \mathsf{Stable}(D, UC) \\ \{(d, c) \mid c \in f(d) \wedge \forall_{d' \leq d} c \in f(d') \Rightarrow d' = d\} & \longmapsto & f. \end{array}$$

This induces a comonad $! := FU$ on Coh. Explicitly, $!A$ has set of tokens $\mathsf{cliques}_{\mathsf{fin}}(A)$ and coherence relation

$$s \frown_{!A} s' := (s \cup s' \in !A).$$

This shows that that our model Coh of linear logic additionally supports !-types.

[18] Recall that a function $D' \xrightarrow{f} D$ is called continuous if it preserves directed suprema and that it is called stable if it preserves all pullbacks: $d_0, d_1 \leq d_\top$ and $d_0 \wedge d_1$ exists implies that $f(d_0 \wedge d_1) = f(d_0) \wedge f(d_1)$.

2.3.4.4 AJM-Games

In section 2.4, we introduce another important class of models of linear logic: categories of games and strategies. We encourage the reader to think of game semantics as giving a further decomposition of coherence space semantics, which itself gave a decomposition of domain semantics by interpreting domain elements as sets of tokens. Indeed, it replaces tokens with (even length) plays in a game which are built up as a sequence of moves. Strategies (certain sets of plays) will then play the rôle of cliques. This can be formalised as the statement that there is a (faithful) forgetful functor from the category of CBN AJM-games[19] to the category of coherence spaces, which sends a game to the coherence space with even length plays as tokens, where tokens are called coherent if they agree on P-moves. Strategies $\sigma : A \multimap B$ are then interpreted as the clique $\{(s \upharpoonright_A, s \upharpoonright_B) \mid s \in \sigma\}$. In [57], it is shown that this functor can be made full if we work with a suitable category of coherence spaces with a partial order on the tokens (representing the idea that some plays extend others in time). In that sense, game semantics is coherence space semantics extended in time. In turns out this more fine-grained description provided by game semantics is enough to precisely pin down the functions that are definable in a functional language with recursion (and with various other effects).

2.4 AJM Game Semantics

The idea behind game semantics is to model a computation by an alternating sequence of interactions (the play) between a program (Player) and its environment (Opponent), following some rules specified by its (data)type (the game). In this translation, programs become Player strategies, while termination corresponds to a strategy being winning or beating all Opponents. The charm of this interpretation is that it not only fully captures the intensional aspects of a program but that it

[19] In this thesis, we work with AJM-games because, in order to interpret dependent types, we need the extra option of explicitly restricting plays by defining the set P_A, rather than being forced to work with all legal positions, as we would be in HO-games. Indeed, [30] has shown that all HO-games can be defined from a simple type system with product and coproduct types as well as type level recursion. This shows that HO-games do not suffice to interpret more expressive types like dependent function types.

combines this with the structural clarity of a categorical model, thus interpolating between traditional operational and denotational semantics.

If we view a type theory as a logic rather than as a programming language, its game semantics formalises the idea of Socratic dialogues. The interpretation of a proposition can be thought of as the game of all formal debates about its validity, where Player argues in its favour and Opponent argues against it. In this view, a proof of a proposition gets interpreted as a winning strategy for Player. We see that proofs get interpreted by winning strategies, when giving a game semantics of a logic, while partial strategies are of interest too, for the game semantics of a programming language, as these model programs that do not always terminate.

We assume the reader has some familiarity with the basics of categories of AJM-games (contrasted with the other style of HO game semantics [58]) and ($\approx$-saturated[20]) strategies, as described in [59], and will only briefly recall the definitions. We define a category Game which has as objects AJM-games.

Let us fix some universal set of moves $\mathcal{M}$ with injective functions

$$\mathcal{M} + \mathcal{M} \xrightarrow{+} \mathcal{M}$$
$$\mathcal{M} \times \mathcal{M} \xrightarrow{\times} \mathcal{M}$$
$$\mathcal{M} \times \mathbb{N} \xrightarrow{\times} \mathcal{M},$$

say the set of ASCII strings with $[x \mapsto \text{“}\texttt{inl}\text{(”}++x++\text{“)”}, x \mapsto \text{“}\texttt{inr}\text{(”}++x++\text{“)”}]$, $\langle x, y\rangle \mapsto \text{“}\langle\text{”} + +x + +\text{“, ”} + +y + +\text{“}\rangle\text{”}$ and $\langle x, y\rangle \mapsto \text{“}\langle\text{”} + +x + +\text{“, ”} + +y + +\text{“}\rangle\text{”}$, to make sure that AJM-games (and later games with dependency) form a set.

Definition 2.4.1 (Game)**.** *A **game** A is a tuple $(M_A, \lambda_A, \mathsf{j}_A, P_A, \approx_A, W_A)$, where*

- *$M_A \subseteq \mathcal{M}$ is set of **moves**;*

[20]Note that this is a mild technical difference from the formalism of [22], where strategies are what we call skeletons, here, which are considered up to a partial equivalence relation induced by $\approx$. Both formalisms are equivalent as a class of skeletons up to this partial equivalence relation can precisely be identified with the unique strategy obtained by closing the plays of the skeleton under $\approx$.

- $M_A \xrightarrow{\lambda_A = \langle \lambda_A^{OP}, \lambda_A^{QA} \rangle} \{O, P\} \times \{Q, A\}$ *is a function which indicates if a move is made by* ***Opponent*** *(*O*)* ***or Player*** *(*P*) and if it is a* ***Question*** *(*Q*) or an* ***Answer*** *(*A*), for which we write* $\overline{O} = P$, $\overline{P} = O$ *and* $M_A^O := \lambda_A^{OP^{-1}}(O)$, $M_A^P := \lambda_A^{OP^{-1}}(P)$, $M_A^Q := \lambda_A^{QA^{-1}}(Q)$ *and* $M_A^A := \lambda_A^{QA^{-1}}(A)$*;*

- $M_A \xrightarrow{\mathsf{j}_A} M_A$ *is a partial function which indicates the* ***justifier*** *of a move, with the properties*

  (Well-Foundedness): j_A *defines a well-founded forest in the sense that for each move* $m \in M_A$ *there is some number* k *such that* $\mathsf{j}_A^k(m)$ *is undefined; such a move with an undefined justifier is called an* ***initial move****;*

  (Compatibility with λ_A*):* P*-moves are justified by* O*-moves and vice-versa; answers* m *are justified by questions* n *(but not necessarily vice-versa); in this case, we say that* m ***answers*** n*.*

  j_A *will be used to enforce* ***stack discipline*** *in strategies.*

- $P_A \subseteq M_A^{\circledast}$ *is a non-empty prefix-closed set of* ***plays****, where* $M_A^{\circledast}$ *is the set of finite sequences of moves, with the properties*

  (Initial Move): Opponent moves first;

  (Alternation): Player and Opponent alternate in making a move;

  (Linearity): Every move occurs at most once in a play;

  (Justification): A move can only be played after its justifier.

- $\approx_A$ *is an equivalence relation on* P_A*, satisfying*

  (Compatibility with λ_A*):* $s \approx_A t \Rightarrow \lambda_A^*(s) = \lambda_A^*(t)$*;*

  (Prefix-Closure): $s \approx_A t \ \wedge \ s' \leq s \ \wedge \ t' \leq t \ \wedge \ |s'| = |t'| \ \Rightarrow s' \approx_A t'$*;*

  (Completeness): $s \approx_A t \wedge sa \in P_A \Rightarrow \exists_b sa \approx_A tb$*.*

  Here, λ_A^* *is the extension of* λ_A *to sequences. The intuition is that* $\approx_A$*-equivalent plays represent the same computation performed using different threads.*

- $W_A \subseteq P_A^\infty$ *is a set of* ***winning plays****, where* P_A^∞ *is the set of infinite plays, i.e. infinite sequences of moves such that all their finite prefixes are in* P_A*, such that* W_A *is closed under* $\approx_A$ *in the sense that*

$$(s \in W_A \wedge t \notin W_A) \Rightarrow \exists_{s_0 \leq s, t_0 \leq t} |s_0| = |t_0| \wedge s_0 \not\approx_A t_0.$$

The intuition is that Opponent is the one who caused interactions in W_A *to be infinite.*

Our notion of morphism will be defined in terms of strategies on games.

Definition 2.4.2 (Strategy)**.** *A (Player)* ***strategy on*** A *is a non-empty subset* $\sigma \subseteq P_A^{\mathsf{even}}$ *satisfying*

(Causal Consistency): $sab \in \sigma \Rightarrow s \in \sigma$*;*

(Representation Independence): $s \in \sigma \ \wedge \ s \approx_A t \Rightarrow t \in \sigma$*.*

We sometimes identify σ with the subset of P_A that is obtained as its prefix closure. Generally, we impose some more conditions on strategies.

Definition 2.4.3 (Conditions on Strategies)**.** *We call a strategy* σ *on* A ***deterministic*** *if it satisfies*

(Determinacy): $sab, ta'b' \in \sigma \ \wedge \ sa \approx_A ta' \Rightarrow sab \approx_A ta'b'$*.*

We call it ***well-bracketed*** *if it satisfies*

(Well-Bracketing): If an answer is played, it is in response to (i.e. justified by) the pending question (i.e. the last unanswered question).

We call σ ***history-free****, if there exists a non-empty causally consistent subset* $\phi \subseteq \sigma$ *(called a* ***history-free skeleton****) such that*

(Uniformization): $\forall_{sab \in \sigma} s \in \phi \Rightarrow \exists!_{b'} sab' \in \phi$*;*

(History-Freeness 1): $sab, tac \in \phi \Rightarrow b = c$*;*

(History-Freeness 2): $(sab, t \in \phi \ \wedge \ ta \in P_A) \Rightarrow tab \in \phi$*.*

*We call σ **winning** if it satisfies*

(Finite Wins): If s is $\leq$-maximal in σ, then s is $\leq$-maximal in P_A.

(Infinite Wins): If $s_0 \leq s_1 \leq \ldots$ is an infinite chain in σ, then $\bigcup_i s_i \in W_A$.

The idea is that game semantics naturally models various effects (and indeed does so very precisely in the sense that full-abstraction results can be obtained): non-determinism [60], non-local control flow[21] [61, 62], local references of ground type[22] [64] and recursion/non-termination [22]. These four conditions on strategies respectively serve to exclude these four classes of effects. This idea has been dubbed the "semantic cube" by Abramsky [65], where the axes of the (hyper)cube correspond to various conditions one could impose on strategies.

We write $\mathsf{str}(A)$ for the cpo of strategies (satisfying our favourite selection of the four conditions above) on A ordered under inclusion and write $\bot_A$ or simply $\bot$ for the strategy $\{\epsilon\}$. In the rest of this thesis, we assume strategies to satisfy all four conditions, unless specified otherwise explicitly. (However, all constructions and results, with the exception of completeness results, go through for any of these classes of strategies.)

We note that a history-free skeleton ϕ for a strategy σ is induced by a partial function on moves and that it satisfies $\sigma = \{t \mid \exists_{s \in \phi} t \approx_A s\}$. A winning strategy is the semantic equivalent of a normalising or total term. It always has a response to any valid O-move. Furthermore, if the result of the interaction between a winning strategy and any (possibly history-sensitive) Opponent is an infinite play,

[21] If we drop the well-bracketing condition altogether, Laird showed that we allow for very wild kinds of control flow, which are customary in CBV but not CBN. To obtain a precise correspondence with a CBN language with the control operator call/cc we would still impose the weaker condition on strategy of being **weakly well-bracketed**: we are allowed to answer any open question, where by open question we mean a question for which no more recent question has been answered already. This corresponds to a stack discipline in which we can not just pop the top element, but we can pop an element that is deeper in the stack with the rule that we have to discard all elements on top of it.

[22] While in HO-games naturally model general references, it is not clear to the author if these can be modelled in AJM-style. Indeed, strategies on AJM-games (at least the simply typed hierarchy) automatically satisfy the so-called visibility condition as a consequence of the restrictions on valid plays (specifically the switching conditions). Visibility is known to be the semantic condition which corresponds to the exclusion of higher-order references. [63]

then this is a member of the set of winning plays, capturing the idea that the infinite interaction is Opponent's fault.

Next, we define some constructions on games, starting with their symmetric monoidal closed structure.

Definition 2.4.4 (Tensor Unit)**.** *We define the game* $I := (\emptyset, \emptyset, \emptyset, \{\epsilon\}, \{(\epsilon, \epsilon)\}, \emptyset)$.

Definition 2.4.5 (Tensor)**.** *Given games* A *and* B*, we define the game* $A \otimes B$ *by*

- $M_{A \otimes B} := M_A + M_B$;
- $\lambda_{A \otimes B} := [\lambda_A, \lambda_B]$;
- $\mathsf{j}_{A \otimes B} := \mathsf{j}_A + \mathsf{j}_B$
- $P_{A \otimes B} := \{s \in M^{\circledast}_{A \otimes B} \mid s \upharpoonright_A \in P_A \wedge s \upharpoonright_B \in P_B\}$;
- $s \approx_{A \otimes B} t := s \upharpoonright_A \approx_A t \upharpoonright_A \ \wedge s \upharpoonright_B \approx_B t \upharpoonright_B \ \wedge \forall_{1 \leq i \leq |s|}(s_i \in M_B \Leftrightarrow t_i \in M_B)$;
- $W_{A \otimes B} := \left\{s \in P^{\infty}_{A \otimes B} \mid \ (s \upharpoonright_A \in P^{\infty}_A \Rightarrow s \upharpoonright_A \in W_A) \wedge (s \upharpoonright_B \in P^{\infty}_B \Rightarrow s \upharpoonright_B \in W_B)\right\}$.

Definition 2.4.6 (Linear Implication)**.** *Given games* A *and* B*, and writing* $\mathsf{init}_B \subseteq M_B$ *for the set where* j_B *is undefined, we define the game* $A \multimap B$ *by*

- $M_{A \multimap B} := M_A \times \mathsf{init}_B + M_B$*; we write* $s \upharpoonright_A$ *for the subsequence of moves in* $M_A \times \mathsf{init}_B$ *of the play* s *where we further project to a sequence in* M_A*;*
- $\lambda_{A \multimap B} := [\overline{\lambda_A}, \lambda_B]$*, where* $\overline{\lambda_A}(m, n) := \overline{\lambda_A(m)}$;
- $\begin{array}{ll} \mathsf{j}_{A \multimap B}(m, n) := n & \text{if } m \in \mathsf{init}_A \\ \mathsf{j}_{A \multimap B}(m, n) := \mathsf{j}_A(m) & \text{if } m \in M_A \setminus \mathsf{init}_A \\ \mathsf{j}_{A \multimap B}(n) := \mathsf{j}_B(n) & \text{if } n \in M_B \end{array}$;
- $P_{A \multimap B} := \{s \in M^{\circledast}_{A \multimap B} \mid s \upharpoonright_A \in P_A \wedge s \upharpoonright_B \in P_B\}$;
- $s \approx_{A \multimap B} t := s \upharpoonright_A \approx_A t \upharpoonright_A \ \wedge s \upharpoonright_B \approx_B t \upharpoonright_B \ \wedge \forall_{1 \leq i \leq |s|}(s_i \in M_B \Leftrightarrow t_i \in M_B)$;
- $W_{A \multimap B} := \{s \in P^{\infty}_{A \multimap B} \mid s \upharpoonright_A \in W_A \Rightarrow s \upharpoonright_B \in W_B\}$.

I is the unique game whose only play has length 0. Both $A \otimes B$ and $A \multimap B$ are obtained by playing A and B in parallel by interleaving. Note that the definitions of $P_{\_}$ and $\lambda_{\_}$ imply that in $A \otimes B$ only Opponent can switch between A and B, while in $A \multimap B$ only Player can, and that in $A \otimes B$ Opponent can start the play in either A or B, while in $A \multimap B$ the play must commence in B. In both cases, a question is answered in the game where it was asked.

These definitions on objects extend to strategies, e.g. for strategies $\sigma \in \mathsf{str}(A), \tau \in \mathsf{str}(B)$, we can define a strategy $\sigma \otimes \tau = \{s \in P^{\mathsf{even}}_{A \otimes B} \mid s \upharpoonright_A \in \sigma \ \wedge \ s \upharpoonright_B \in \tau\} \in \mathsf{str}(A \otimes B)$. This gives us a model of multiplicative intuitionistic linear logic, with all structural morphisms consisting of appropriate variants of copycat strategies, which are introduced next.

Theorem 2.4.7 (Linear Category of Games)**.** *We define a category* Game *by*

- $\mathsf{ob}(\mathsf{Game}) := \{A \mid A \text{ is an AJM-game}\}$*;*
- $\mathsf{Game}(A, B) := \mathsf{str}(A \multimap B)$*;*
- $\mathsf{id}_A := \{s \in P^{\mathsf{even}}_{A \multimap A} \mid \forall_{s' \in P^{\mathsf{even}}_{A \multimap A}} s' \leq s \Rightarrow s' \upharpoonright_{A^{(1)}} \approx_A s' \upharpoonright_{A^{(2)}}\}$*, the* ***copycat strategy*** *on* A*;*
- *for* $A \xrightarrow{\sigma} B \xrightarrow{\tau} C$*, the composition (or* ***interaction****)* $A \xrightarrow{\sigma;\tau} C$ *is defined from parallel composition* $\sigma || \tau := \{s \in M^{\circledast}_{(A \multimap B) \multimap C} \mid s \upharpoonright_{A,B} \in \sigma \ \wedge \ s \upharpoonright_{B,C} \in \tau\}$ *plus hiding:* $\sigma; \tau := \{s \upharpoonright_{A,C} \mid s \in \sigma || \tau\}$*.*

Then, $(\mathsf{Game}, I, \otimes, \multimap)$ *is, in fact, a symmetric monoidal closed category.*

To make this into a model of intuitionistic logic, a cartesian closed category (ccc), through the first Girard translation (CBN translations), we need two more constructions on games, to interpret the additive conjunction and exponential, respectively.

Definition 2.4.8 (With)**.** *Given games* A *and* B*, we define the game* $A \& B$ *by*

- $M_{A \& B} := M_A + M_B$*;*
- $\lambda_{A \& B} := [\lambda_A, \lambda_B]$*;*

- $\mathsf{j}_{A\&B} := \mathsf{j}_A + \mathsf{j}_B$;
- $P_{A\&B} := P_A + P_B$;
- $\approx_{A\&B} := \approx_A + \approx_B$;
- $W_{A\&B} := W_A + W_B$.

Definition 2.4.9 (Bang)**.** *Given a game A, we define the game $!A$ by*

- $M_{!A} := \mathbb{N} \times M_A$;
- $\lambda_{!A}(i, a) := \lambda_A(a)$;
- $\mathsf{j}_{!A}(i, a) := (i, \mathsf{j}_A(a))$;
- $P_{!A} := \{s \in M_{!A}^{\circledast} \mid \forall_{i\in\mathbb{N}} s \upharpoonright_i \in P_A\}$;
- $s \approx_{!A} t := \exists_{\pi\in S(\mathbb{N})} \forall_{i\in\mathbb{N}} s \upharpoonright_i \approx_A t \upharpoonright_{\pi(i)} \ \wedge (\mathsf{fst}; \pi)^*(s) = \mathsf{fst}^*(t)$, *writing $S(\mathbb{N})$ for the set of permutations of $\mathbb{N}$;*
- $W_{!A} := \{s \in P_{!A}^{\infty} \mid \forall_i s \upharpoonright_i \in P_A^{\infty} \Rightarrow s \upharpoonright_i \in W_A\}$.

A play in $A\&B$ consists of either a play in A or in B, where Opponent chooses which as by our convention Opponent always makes the initial move. A play in $!A$ consists of any number of interleaved **threads** of plays in A. Because of the definition of $\approx_A$, $!A$ behaves as a countably infinite symmetric $\otimes$-product of A with itself. As before, the definition of $\lambda_{!A}$ assures only Opponent can switch games, while the definition of justification ensures that a question is answered in the thread in which it was asked.

Next, we note that ! can be made into a comonad by defining, for $A \xrightarrow{\sigma} B$,

$$!\sigma := \{s \in P^{\mathsf{even}}_{!A\multimap !B} \mid \exists_{\pi\in S(\mathbb{N})} \forall_{i\in\mathbb{N}} s \upharpoonright_{(\pi(i),A),(i,B)} \in \sigma\},$$

and natural transformations $!A \xrightarrow{\mathsf{der}_A} A$ and $!A \xrightarrow{\delta_A} !!A$ where

$$\mathsf{der}_A := \{s \in P^{\mathsf{even}}_{!A\multimap A} \mid \forall_{s'\in P^{\mathsf{even}}_{!A\multimap A}} s' \leq s \Rightarrow \exists_{i\in\mathbb{N}} s' \upharpoonright_{!A} \upharpoonright_i \approx_A s' \upharpoonright_A\} \quad \text{and}$$

$$\delta_A := \{s \in P^{\mathsf{even}}_{!A\multimap !!A} \mid \forall_{s'\in P^{\mathsf{even}}_{!A\multimap !!A}} s' \leq s \Rightarrow \exists_{p:\mathbb{N}\times\mathbb{N}\hookrightarrow\mathbb{N}} \forall_{i,j\in\mathbb{N}} s' \upharpoonright_{!A} \upharpoonright_{p(i,j)} \approx_A s' \upharpoonright_{!!A} \upharpoonright_i \upharpoonright_j\}.$$

This allows us to define the co-Kleisli category $\mathsf{Game}_!$ of CBN games, which has the same objects as Game, while $\mathsf{Game}_!(A,B) := \mathsf{Game}(!A,B)$. Let us write $\mathsf{dom}(f)$ for the domain of a morphism f. We have a composition $(f,g) \mapsto f^\dagger; g$, where we write $f^\dagger := \delta_{\mathsf{dom}(f)}; !(f)$, for which the strategies der_A serve as identities. We can define finite products in $\mathsf{Game}_!$ by I and & and write

$$\mathsf{diag}_A := \Big\{ s \in P^{\mathsf{even}}_{!A \multimap (A\&A)} \mid \forall_{s' \in P^{\mathsf{even}}_{!A \multimap (A\&A)}} s' \leq s \Rightarrow \exists_{i \in \mathbb{N}} (s' = \epsilon) \vee (s' \upharpoonright_{!A} \upharpoonright_i \approx_A s' \upharpoonright_{A^{(1)}} \neq \epsilon) \vee (s' \upharpoonright_{!A} \upharpoonright_i \approx_A s' \upharpoonright_{A^{(2)}} \neq \epsilon) \Big\}$$

for the diagonal $A \xrightarrow{\mathsf{diag}_A} A\&A$ in $\mathsf{Game}_!$. Moreover, we have Seely-isomophisms $!I \cong I$ and $!(A\&B) \cong !A \otimes !B$, so we obtain a linear/non-linear adjunction $\mathsf{Game} \leftrightarrows \mathsf{Game}_!$, hence a model of multiplicative exponential intuitionistic linear logic. In particular, by defining $A \Rightarrow B := !A \multimap B$, we make $\mathsf{Game}_!$ into a ccc. We write $\mathsf{comp}_{A,B,C}$ for the internal composition $((A \Rightarrow B) \,\&\, (B \Rightarrow C)) \longrightarrow A \Rightarrow C$ in $\mathsf{Game}_!$.

Theorem 2.4.10 (Intuitionist Category of Games)**.** $(\mathsf{Game}_!, I, \&, \Rightarrow)$ *is a ccc.*

Remark 2.4.11. *Note that for the hierarchy of cartesian types A that are formed by operations I, & and $\Rightarrow$ from finite games (games A with finite P_A), winning strategies are the total strategies – strategies which respond to any O-move – for which infinite plays can only occur because Opponent opens infinitely many threads of the same game.*

So far, we have shown that $\mathsf{Game}_!$ provides a model of $\mathsf{STT}_{\mathsf{CBN}}$ without finite inductive types. It is not clear that inductive types and (weak) coproducts are supported in our large world $\mathsf{Game}_!$ of all games if we work with history-free strategies[23]. To support these – to be precise, in order to define the appropriate eliminators –, we restrict the games we consider.

We can in fact construct a model of all of $\mathsf{STT}_{\mathsf{CBN}}$ in a full subcategory $\mathsf{Game}_!^{\mathrm{fin1}\times\Rightarrow}$ of $\mathsf{Game}_!$ by giving a suitable interpretation to finite inductive types,

[23] In fact, the condition of history-freeness is replaced with innocence in [66] which mends this defect. Note that history-free and innocent strategies coincide on simple types. This is no longer true in the presence of coproducts, as we can no longer encode the P-view in a thread index. In this case, innocent strategies give the right notion to obtain definability and full abstraction results [67].

which serve as the ground types for a type hierarchy built with 1, $\times$ and $\Rightarrow$. For a set X, let us define X_* to be a so-called **flat game** with Player moves X which are all justified by a single initial Opponent move $*$, $P_{X_*} = \{\epsilon, *\} \cup \{*x \mid x \in X\}$ and $\approx_X = \{(s,s) \in P_X \times P_X\}$ where the initial move $*$ is a question and the moves from X are answers. Let us interpret a finite inductive type $\{a_i \mid i\}$ as a finite flat game $\{a_i \mid i\}_*$. Let us write $\mathsf{Game}_!^{\mathrm{fin1}\times\Rightarrow}$ for the full subcategory of $\mathsf{Game}_!$ on the objects formed from finite flat games by 1, $\times$ and $\Rightarrow$. Then, $\mathsf{Game}_!^{\mathrm{fin1}\times\Rightarrow}$ is a model of $\mathsf{STT}_{\mathsf{CBN}}$. Indeed, the interpretation of the introduction rule for a_i is the strategy which answers a_i to $*$, while the case-eliminators for finite inductive types are interpreted inductively on the structure of the type C we are eliminating into: the cases where C is not a finite inductive type are defined through equations $1 - \eta$, $\{a_i \mid i\} - \mathsf{Comm} - \langle -, - \rangle$ and $\{a_i \mid i\} - \mathsf{Comm} - \lambda$ of figure 2.3. In case C is a finite inductive type $\{c_j \mid j\}$, we interpret $\mathsf{case}_{\{a_i \mid i\},C}$ as the winning history-free strategy on $\{a_i \mid i\}_* \Rightarrow \{c_j \mid j\}_*{}^{(1)} \Rightarrow \ldots \Rightarrow \{c_j \mid j\}_*{}^{(n)} \Rightarrow \{c_j \mid j\}_*$ which is given by the $\approx$-closure of the set of traces defined by the following partial function f on moves:

$$*^{(C)} \mapsto (0,*)^{(!\{a_i \mid i\}_*)} \quad (0,a_i) \mapsto (0,*)^{(!C^{(i)})} \quad (0,c_j)^{(!C^{(i)})} \mapsto c_j^{(C)}.$$

One of the interesting aspects of game semantics are the strong correspondences that can often be established with the syntax we are modelling. In this case, we have the following very strong completeness result. Note that fullness of the interpretation is strictly stronger than completeness: it is a notion of completeness with respect to proofs rather than mere provability.

Theorem 2.4.12. *The interpretation functor* $\mathsf{STT}_{\mathsf{CBN}} \xrightarrow{[\![-]\!]} \mathsf{Game}_!$ *is full and faithful and hence is an equivalence of categories to* $\mathsf{Game}_!^{\mathrm{fin1}\times\Rightarrow}$*, where we write* $\mathsf{STT}_{\mathsf{CBN}}$ *for the syntactic category corresponding to the eponymous theory of section 2.1.1.2.*

Proof. This is a straightforward finitary total variation on the results of [22] that can, in particular, be obtained as a special case of the results in [56]. We note that winning strategies are total and therefore the case of $\bot$ in the decomposition lemma does not occur. Moreover, we note that the iterated decomposition terminates

if we start with a winning strategy, for reasons outlined in the proof of lemma 4.5.3 (essentially because infinite plays are always Opponent's responsibility, we can assign a finite size to a strategy which shrinks under the decomposition). □

"Well! I've often seen a cat without a grin," thought Alice; "but a grin without a cat! It's the most curious thing I ever saw in all my life.".

— Lewis Carroll

3 Linear Dependent Type Theory

Starting from Church's simply typed λ-calculus (or cartesian propositional type theory), two extensions depart in perpendicular directions:

- following the Curry-Howard propositions-as-types interpretation, **dependent type theory** (DTT) [68] extends the simply typed λ-calculus from a proof calculus of intuitionistic propositional logic to one for predicate logic;

- **linear logic** [43] gives a more detailed resource sensitive analysis, exposing precisely how many times each assumption is used in proofs.

A combined **linear dependent type theory** is one of the interesting directions to explore to gain a more fine-grained understanding of dependent type theory from a computer science point of view, explaining its flow of information. Indeed, many of the usual settings for computational semantics are naturally linear in character, either because they arise from a model of linear logic as !-co-Kleisli categories (coherence space and game semantics) or for more fundamental reasons (quantum computation). Relatedly, as we have seen, linear types naturally arise in the semantics of commutative effects.

Combining dependent types and linear types is a non-trivial task, however, and despite some work by various authors that we shall discuss, the precise relationship

between the two systems remains poorly understood. The discrepancy between linear and dependent types is the following.

- The lack of structural rules in **linear type theory** forces us to refer to each identifier precisely once – for a sequent $x : A \vdash t : B$, x occurs uniquely in t.
- In **dependent type theory**, types can have free identifiers – $x : A \vdash B \quad \mathsf{type}$, where x is free in B. Crucially, if $x : A \vdash t : B$, x may also be free in t.

What does it mean for x to occur uniquely in t in a dependent setting? Do we not count its occurrence in B? This point of view seems incompatible with universes, which play an important rôle in dependent type theory. If we do, however, the language seems to lose much of its expressive power. In particular, it prevents us from talking about constant types, it seems.

The usual way out, which we shall follow too, is to restrict type dependency on cartesian terms, which can be copied and deleted freely. Although this seems very limiting – for instance, we do not obtain full equivalents of the Girard translations, embedding DTT in the resulting system –, it is not clear that there is a reasonable alternative. Moreover, as even this limited scenario has not been studied extensively, we hope that a semantic analysis, which was so far missing entirely, may shed new light on the old mystery of linear type dependency.

Historically, Girard's early work in linear logic already makes movements to extend a linear analysis to predicate logic. Although it talks about first-order quantifiers, the analysis appears to have stayed rather superficial, omitting the identity predicates which, in a way, are what make first-order logic tick. Closely related is that an account of internal quantification, or a linear variant of Martin-Löf's type theory, was missing, let alone a Curry-Howard correspondence.

Later, linear types and dependent types were first combined in a Linear Logical Framework [69], where a syntax was presented that extends a Logical Framework with linear types (that depend on terms of cartesian types). This has given rise to a line of work in the computer science community [70–72]. All the work seems to be syntactic in nature, however, and seems to be mostly restricted to the asynchronous

fragment in which we only have $\multimap$-, $\Pi^{\multimap}_{!}$-, $\top$-, and &-types. An exception is the Concurrent Logical Framework [73], which treats synchronous connectives resembling our I-, $\otimes$-, $\Sigma^{\otimes}_{!}$-, and !-types. An account of additive disjunctions and identity types is missing entirely.

On the other hand, similar ideas, this time at the level of categorical semantics and specific models (from homotopy theory, algebra, and physics), have emerged in the mathematical community [74–77]. In these models, as with Girard, a notion of comprehension was missing and, with that, a notion of identity type. Although in the last while some suggestions have been made on the nLab and nForum of possible connections between the syntactic and semantic work, no account of the correspondence was published.

The point of this chapter is to close this gap between syntax and semantics and to pave the way for a proper semantic analysis of linear type dependency, treating a range of type formers including $\mathsf{Id}^{\otimes}_{!}$-types. Firstly, in section 3.1, we present a syntax, dependently typed dual intuitionistic linear logic (dDILL), a natural blend of the dual intuitionistic linear logic (DILL) [34] and dependent type theory (DTT) [20] which generalises both. Secondly, in section 3.2, we present a complete categorical semantics, an obvious combination of linear/non-linear adjunctions [34] and comprehension categories [25]. Thirdly, we discuss how our semantics applies to the dependently typed LNL calculus [78] in section 3.3 and discuss dependently typed Girard translations in section 3.4. Finally, in sections 3.5.1, 3.5.2, 3.5.3, 3.5.4 and 3.5.5 we present various concrete models, including a class of models arising from commutative effects and a coherence space semantics.

Remark 3.0.1 (Related Publications)**.** *This chapter is largely based on [13, 16]. The material on many (the exception being the monoidal families) of the concrete models is new as is the material on dependent Girard translations and the dependent LNL calculus semantics. Around the same time that the author published his study [12, 13] of dDILL, [78] independently developed a syntax (but not a denotational semantics) and applications for dLNL.*

3.1 Syntax of dDILL

We next present the formal syntax of dDILL, c.f. section 2.1.1. We start with a presentation of its judgements and then discuss its rules of inference: first its structural core, then the logical rules for a series of optional type formers. We conclude this section with a few basic results about the syntax.

Judgements

We adopt a notation $\Gamma; \Delta$ for contexts, where Γ is 'a cartesian region' and Δ is 'a linear region', similarly to [34]. The idea will be that we have an empty context and can extend an existing context $\Gamma; \Delta$ with both cartesian and linear types that are allowed to depend on Γ. Our language will express judgements of the six forms of figure 3.1.

Structural Rules

We use the structural rules of figures 3.2, 3.3 and 3.4, which are essentially the structural rules of dependent type theory where some rules appear in both a cartesian and a linear form. We present the rules per group, with their names, from left-to-right, top-to-bottom.

Logical Rules

We introduce some basic (optional) type and term formers, for which we give type formation (denoted -F), term introduction (-I), term elimination (-E), term computation rules (-β), and (judgemental) term uniqueness principles (-η), in figure 3.5, 3.6 and 3.7. Moreover, $\Sigma^{\otimes}_{!(x:A)}$, $\Pi^{\multimap}_{!(x:A)}$, $\lambda_{!(x:A)}$, and $\lambda_{x:A}$ are name binding operators, binding free occurences of x within their scope. Preempting some theorems of the calculus, we overload some of the notation for -I and -E rules of various type formers, in order to avoid unnecessary syntactic clutter. Needless to say, uniqueness of typing can easily be restored by carrying around enough type information on the term formers corresponding to the various -I and -E rules.

dDILL judgement	Intended meaning
$\vdash \Gamma; \Delta \ \mathsf{ctxt}$	$\Gamma; \Delta$ is a valid context
$\Gamma; \cdot \vdash A \ \mathsf{type}$	A is a type in (cartesian) context Γ
$\Gamma; \Delta \vdash a : A$	a is a term of type A in context $\Gamma; \Delta$
$\vdash \Gamma; \Delta = \Gamma'; \Delta'$	$\Gamma; \Delta$ and $\Gamma'; \Delta'$ are judgementally equal contexts
$\Gamma; \cdot \vdash A = A'$	A and A' are judgementally equal types in (cartesian) context Γ
$\Gamma; \Delta \vdash a = a' : A$	a and a' are judgementally equal terms of type A in context $\Gamma; \Delta$

Figure 3.1: Judgements of dDILL.

$$\frac{}{\cdot; \cdot \ \mathsf{ctxt}}\ \mathsf{C\text{-}Emp}$$

$$\frac{\vdash \Gamma; \cdot \ \mathsf{ctxt} \qquad \Gamma; \cdot \vdash A \ \mathsf{type}}{\vdash \Gamma, x : A; \cdot \ \mathsf{ctxt}}\ \mathsf{Cart\text{-}C\text{-}Ext} \qquad \frac{\Gamma; \Delta = \Gamma'; \Delta' \qquad \Gamma; \cdot \vdash A = B}{\vdash \Gamma, x : A; \Delta = \Gamma', y : B; \Delta'}\ \mathsf{Cart\text{-}C\text{-}Ext\text{-}Eq}$$

$$\frac{\vdash \Gamma; \Delta \ \mathsf{ctxt} \qquad \Gamma; \cdot \vdash A \ \mathsf{type}}{\vdash \Gamma; \Delta, x : A \ \mathsf{ctxt}}\ \mathsf{Lin\text{-}C\text{-}Ext} \qquad \frac{\Gamma; \Delta = \Gamma'; \Delta' \qquad \Gamma; \cdot \vdash A = B}{\vdash \Gamma; \Delta, x : A = \Gamma'; \Delta', y : B}\ \mathsf{Lin\text{-}C\text{-}Ext\text{-}Eq}$$

$$\frac{\Gamma, x : A, \Gamma'; \cdot \ \mathsf{ctxt}}{\Gamma, x : A, \Gamma'; \cdot \vdash x : A}\ \mathsf{Cart\text{-}Idf} \qquad \frac{\Gamma; x : A \ \mathsf{ctxt}}{\Gamma; x : A \vdash x : A}\ \mathsf{Lin\text{-}Idf}$$

Figure 3.2: Context formation and identifier declaration rules.

$$\frac{\vdash \Gamma; \Delta \ \mathsf{ctxt}}{\vdash \Gamma; \Delta = \Gamma; \Delta}\ \mathsf{C\text{-}Eq\text{-}R} \qquad \frac{\vdash \Gamma; \Delta = \Gamma'; \Delta'}{\vdash \Gamma'; \Delta' = \Gamma; \Delta}\ \mathsf{C\text{-}Eq\text{-}S}$$

$$\frac{\vdash \Gamma; \Delta = \Gamma'; \Delta' \qquad \vdash \Gamma'; \Delta' = \Gamma''; \Delta''}{\vdash \Gamma; \Delta = \Gamma''; \Delta''}\ \mathsf{C\text{-}Eq\text{-}T}$$

$$\frac{\Gamma; \cdot \vdash A \ \mathsf{type}}{\Gamma; \cdot \vdash A = A}\ \mathsf{Ty\text{-}Eq\text{-}R} \qquad \frac{\Gamma; \cdot \vdash A = A'}{\Gamma; \cdot \vdash A' = A}\ \mathsf{Ty\text{-}Eq\text{-}S}$$

$$\frac{\Gamma; \cdot \vdash A = A' \qquad \Gamma; \cdot \vdash A' = A''}{\Gamma; \cdot \vdash A = A''}\ \mathsf{Ty\text{-}Eq\text{-}T}$$

$$\frac{\Gamma; \Delta \vdash a : A}{\Gamma; \Delta \vdash a = a : A}\ \mathsf{Tm\text{-}Eq\text{-}R} \qquad \frac{\Gamma; \Delta \vdash a = a' : A}{\Gamma; \Delta \vdash a' = a : A}\ \mathsf{Tm\text{-}Eq\text{-}S}$$

$$\frac{\Gamma; \Delta \vdash a = a' : A \qquad \Gamma; \Delta \vdash a' = a'' : A}{\Gamma; \Delta \vdash a = a'' : A}\ \mathsf{Tm\text{-}Eq\text{-}T}$$

$$\frac{\Gamma; \Delta \vdash a : A \qquad \vdash \Gamma; \Delta = \Gamma; \Delta' \qquad \Gamma; \cdot \vdash A = A'}{\Gamma'; \Delta' \vdash a : A'}\ \mathsf{Tm\text{-}Conv} \qquad \frac{\Gamma'; \cdot \vdash A \ \mathsf{type} \qquad \vdash \Gamma; \cdot = \Gamma'; \cdot}{\Gamma'; \cdot \vdash A \ \mathsf{type}}\ \mathsf{Ty\text{-}Conv}$$

Figure 3.3: A few standard rules for judgemental equality, saying that it is an equivalence relation and is compatible with typing.

Note that we are working with weak (non-dependent) elimination rules for positive connectives. This is forced on us by the requirement that types do not depend on linear assumptions. As an alternative, we could demand strong elimination rules, but only for terms without linear assumptions.

$$\frac{\Gamma, \Gamma'; \Delta \vdash \mathcal{J} \qquad \Gamma; \cdot \vdash A \ \mathsf{type}}{\Gamma, x : A, \Gamma'; \Delta \vdash \mathcal{J}} \ \textsf{Cart-Weak}$$

$$\frac{\Gamma, x : A, \Gamma'; \cdot \vdash B \ \mathsf{type} \qquad \Gamma; \cdot \vdash a : A}{\Gamma, \Gamma'[a/x]; \cdot \vdash B[a/x] \ \mathsf{type}} \ \textsf{Cart-Ty-Subst} \qquad \frac{\Gamma, x : A, \Gamma'; \cdot \vdash B = B' \qquad \Gamma; \cdot \vdash a : A}{\Gamma, \Gamma'[a/x]; \cdot \vdash B[a/x] = B'[a/x]} \ \textsf{Cart-Ty-Subst-Eq}$$

$$\frac{\Gamma, x : A, \Gamma'; \Delta \vdash b : B \qquad \Gamma; \cdot \vdash a : A}{\Gamma, \Gamma'[a/x]; \Delta[a/x] \vdash b[a/x] : B[a/x]} \ \textsf{Cart-Tm-Subst} \qquad \frac{\Gamma, x : A, \Gamma'; \Delta \vdash b = b' : B \qquad \Gamma; \cdot \vdash a : A}{\Gamma, \Gamma'[a/x]; \Delta \vdash b[a/x] = b'[a/x] : B[a/x]} \ \textsf{Cart-Tm-Subst-Eq}$$

$$\frac{\Gamma; \Delta, x : A, \Delta' \vdash b : B \qquad \Gamma; \Delta'' \vdash a : A}{\Gamma; \Delta, \Delta', \Delta'' \vdash b[a/x] : B} \ \textsf{Lin-Tm-Subst} \qquad \frac{\Gamma; \Delta, x : A, \Delta' \vdash b = b' : B \qquad \Gamma; \Delta'' \vdash a : A}{\Gamma; \Delta, \Delta', \Delta'' \vdash b[a/x] = b'[a/x] : B} \ \textsf{Lin-Tm-Subst-Eq}$$

$$\frac{\Gamma; \cdot \vdash a = a' : A \qquad \Gamma, x : A, \Gamma'; \Delta \vdash b : B}{\Gamma, \Gamma'[a/x]; \Delta[a/x] \vdash b[a/x] = b[a'/x] : B} \ \textsf{Cart-Tm-Cong} \qquad \frac{\Gamma; \cdot \vdash a = a' : A \qquad \Gamma, x : A, \Gamma'; \Delta \vdash B \ \mathsf{type}}{\Gamma, \Gamma'[a/x]; \Delta[a/x] \vdash B[a/x] = B[a'/x]} \ \textsf{Cart-Ty-Cong}$$

$$\frac{\Gamma; \Delta'' \vdash a = a' : A \qquad \Gamma; \Delta, x : A, \Delta' \vdash b : B}{\Gamma; \Delta, \Delta', \Delta'' \vdash b[a/x] = b[a'/x] : B} \ \textsf{Lin-Tm-Cong}$$

Figure 3.4: Weakening, substitution and congruence rules. Here, $\mathcal{J}$ represents a statement of the form $B \ \mathsf{type}$, $B = B'$, $b : B$, or $b = b' : B$, such that all judgements are well-formed. Note that these imply exchange rules for both linear and cartesian identifiers as well as a contraction rule for cartesian identifiers.

$$\frac{}{\Gamma; \cdot \vdash I \ \mathsf{type}} \ I\textsf{-F}$$

$$\frac{\Gamma; \cdot \vdash A \ \mathsf{type} \qquad \Gamma; \cdot \vdash B \ \mathsf{type}}{\Gamma; \cdot \vdash A \otimes B \ \mathsf{type}} \ \otimes\textsf{-F} \qquad \frac{\Gamma; \cdot \vdash A \ \mathsf{type} \qquad \Gamma; \cdot \vdash B \ \mathsf{type}}{\Gamma; \cdot \vdash A \multimap B \ \mathsf{type}} \ \multimap\textsf{-F}$$

$$\frac{\Gamma, x : A; \cdot \vdash B \ \mathsf{type}}{\Gamma; \cdot \vdash \Sigma^{\otimes}_{!(x:A)} B \ \mathsf{type}} \ \Sigma^{\otimes}_{!}\textsf{-F} \qquad \frac{\Gamma, x : A; \cdot \vdash B \ \mathsf{type}}{\Gamma; \cdot \vdash \Pi^{\multimap}_{!(x:A)} B \ \mathsf{type}} \ \Pi^{\multimap}_{!}\textsf{-F}$$

$$\frac{}{\Gamma; \cdot \vdash \top \ \mathsf{type}} \ \top\textsf{-F} \qquad \frac{\Gamma; \cdot \vdash A \ \mathsf{type} \qquad \Gamma; \cdot \vdash B \ \mathsf{type}}{\Gamma; \cdot \vdash A \& B \ \mathsf{type}} \ \&\textsf{-F}$$

$$\frac{}{\Gamma; \cdot \vdash 0 \ \mathsf{type}} \ 0\textsf{-F} \qquad \frac{\Gamma; \cdot \vdash A \ \mathsf{type} \qquad \Gamma; \cdot \vdash B \ \mathsf{type}}{\Gamma; \cdot \vdash A \oplus B \ \mathsf{type}} \ \oplus\textsf{-F}$$

$$\frac{\Gamma; \cdot \vdash A \ \mathsf{type}}{\Gamma; \cdot \vdash !A \ \mathsf{type}} \ !\textsf{-F} \qquad \frac{\Gamma; \cdot \vdash a : A \qquad \Gamma; \cdot \vdash a' : A}{\Gamma; \cdot \vdash \mathsf{Id}^{\otimes}_{!A}(a, a') \ \mathsf{type}} \ \mathsf{Id}^{\otimes}_{!}\textsf{-F}$$

Figure 3.5: Type formation rules for the various connectives.

Remark 3.1.1. *Note that all type formers that are defined context-wise (I, $\otimes$, $\multimap$, $\top$, $\&$, 0, $\oplus$, and $!$) are automatically preserved under the substitutions from Cart-Ty-Subst (up to canonical isomorphism[1]), in the sense that $F(A_1, \ldots, A_n)[a/x]$ is isomorphic to $F(A_1[a/x], \ldots, A_n[a/x])$ for an n-ary type former F. Similarly,*

[1]By an isomorphism of types $\Gamma; \cdot \vdash A \ \mathsf{type}$ and $\Gamma; \cdot \vdash B \ \mathsf{type}$ in context Γ, we here mean a pair of terms $\Gamma; x : A \vdash f : B$ and $\Gamma; y : B \vdash g : A$ together with a pair of judgemental equalities $\Gamma; x : A \vdash g[f/y] = x : A$ and $\Gamma; y : B \vdash f[g/x] = y : B$.

$$\frac{}{\Gamma;\cdot \vdash * : I}\ I\text{-I} \qquad \frac{\Gamma;\Delta' \vdash t : I \qquad \Gamma;\Delta \vdash a : A}{\Gamma;\Delta,\Delta' \vdash \mathsf{let}\ t\ \mathsf{be}\ *\ \mathsf{in}\ a : A}\ I\text{-E}$$

$$\frac{\Gamma;\Delta \vdash a : A \qquad \Gamma;\Delta' \vdash b : B}{\Gamma;\Delta,\Delta' \vdash a \otimes b : A \otimes B}\ \otimes\text{-I} \qquad \frac{\Gamma;\Delta \vdash t : A \otimes B \qquad \Gamma;\Delta', x : A, y : B \vdash c : C}{\Gamma;\Delta,\Delta' \vdash \mathsf{let}\ t\ \mathsf{be}\ x \otimes y\ \mathsf{in}\ c : C}\ \otimes\text{-E}$$

$$\frac{\Gamma;\Delta, x : A \vdash b : B}{\Gamma;\Delta \vdash \lambda_{x:A} b : A \multimap B}\ \multimap\text{-I} \qquad \frac{\Gamma;\Delta \vdash f : A \multimap B \qquad \Gamma;\Delta' \vdash a : A}{\Gamma;\Delta,\Delta' \vdash f(a) : B}\ \multimap\text{-E}$$

$$\frac{\Gamma;\cdot \vdash a : A \qquad \Gamma;\Delta \vdash b : B[a/x]}{\Gamma;\Delta \vdash !a \otimes b : \Sigma^{\otimes}_{!(x:A)} B}\ \Sigma^{\otimes}_{!}\text{-I} \qquad \frac{\begin{array}{l}\Gamma;\cdot \vdash C\ \ \mathsf{type}\\ \Gamma;\Delta \vdash t : \Sigma^{\otimes}_{!(x:A)} B\\ \Gamma, x : A;\Delta', y : B \vdash c : C\end{array}}{\Gamma;\Delta,\Delta' \vdash \mathsf{let}\ t\ \mathsf{be}\ !x \otimes y\ \mathsf{in}\ c : C}\ \Sigma^{\otimes}_{!}\text{-E}$$

$$\frac{\vdash \Gamma;\Delta\ \ \mathsf{ctxt} \qquad \Gamma, x : A;\Delta \vdash b : B}{\Gamma;\Delta \vdash \lambda_{!(x:A)} b : \Pi^{\multimap}_{!(x:A)} B}\ \Pi^{\multimap}_{!}\text{-I} \qquad \frac{\Gamma;\cdot \vdash a : A \qquad \Gamma;\Delta \vdash f : \Pi^{\multimap}_{!(x:A)} B}{\Gamma;\Delta \vdash f(!a) : B[a/x]}\ \Pi^{\multimap}_{!}\text{-E}$$

$$\frac{\vdash \Gamma;\Delta\ \ \mathsf{ctxt}}{\Gamma;\Delta \vdash \langle\rangle : \top}\ \top\text{-I}$$

$$\frac{\Gamma;\Delta \vdash a : A \qquad \Gamma;\Delta \vdash b : B}{\Gamma;\Delta \vdash \langle a, b\rangle : A \& B}\ \&\text{-I} \qquad \frac{\Gamma;\Delta \vdash t : A \& B}{\Gamma;\Delta \vdash \mathsf{fst}(t) : A}\ \&\text{-E1} \qquad \frac{\Gamma;\Delta \vdash t : A \& B}{\Gamma;\Delta \vdash \mathsf{snd}(t) : B}\ \&\text{-E2}$$

$$\frac{\Gamma;\Delta \vdash t : 0}{\Gamma;\Delta,\Delta' \vdash \mathsf{false}(t) : B}\ 0\text{-E}$$

$$\frac{\Gamma;\Delta \vdash a : A}{\Gamma;\Delta \vdash \mathsf{inl}(a) : A \oplus B}\ \oplus\text{-I1} \qquad \frac{\Gamma;\Delta, x : A \vdash c : C \qquad \Gamma;\Delta, y : B \vdash d : C \qquad \Gamma;\Delta' \vdash t : A \oplus B}{\Gamma;\Delta,\Delta' \vdash \mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}(x) \to c \mid\mid \mathsf{inr}(y) \to d : C}\ \oplus\text{-E}$$

$$\frac{\Gamma;\Delta \vdash b : B}{\Gamma;\Delta \vdash \mathsf{inr}(b) : A \oplus B}\ \oplus\text{-I2}$$

$$\frac{\Gamma;\cdot \vdash a : A}{\Gamma;\cdot \vdash !a : !A}\ !\text{-I} \qquad \frac{\Gamma;\Delta \vdash t : !A \qquad \Gamma, x : A;\Delta' \vdash b : B}{\Gamma;\Delta,\Delta' \vdash \mathsf{let}\ t\ \mathsf{be}\ !x\ \mathsf{in}\ b : B}\ !\text{-E}$$

$$\frac{\Gamma;\cdot \vdash a : A}{\Gamma;\cdot \vdash \mathsf{refl}(!a) : \mathsf{Id}^{\otimes}_{!A}(a, a)}\ \mathsf{Id}^{\otimes}_{!}\text{-I} \qquad \frac{\begin{array}{ll}\Gamma;\cdot \vdash a : A & \\ \Gamma;\cdot \vdash a' : A & \Gamma, x : A, x' : A;\cdot \vdash D\ \ \mathsf{type}\\ \Gamma;\Delta' \vdash p : \mathsf{Id}^{\otimes}_{!A}(a, a') & \Gamma, z : A;\Delta \vdash d : D[z/x, z/x']\end{array}}{\Gamma;\Delta[a/z],\Delta' \vdash \mathsf{let}\ (a, a', p)\ \mathsf{be}\ (z, z, \mathsf{refl}(!z))\ \mathsf{in}\ d : D[a/x, a'/x']}\ \mathsf{Id}^{\otimes}_{!}\text{-E}$$

Figure 3.6: Term introduction and elimination rules for the various connectives.

for $T = \Sigma^{\otimes}$ or $\Pi^{\multimap}$, we have that $(T_{!(y:B)} C)[a/x]$ is isomorphic to $T_{!(y:B[a/x])} C[a/x]$ and $(\mathsf{Id}_{!B}(b, b'))[a/x]$ is isomorphic to $\mathsf{Id}_{!B[a/x]}(b[a/x], b'[a/x])$. (This gives us Beck-Chevalley conditions in the categorical semantics.)

Remark 3.1.2. *The reader can note that the usual formulation of universes for DTT transfers very naturally to dDILL, giving us a notion of universes for linear types, where terms of the universes without linear assumptions code for types. This allows us to write rules for forming types as rules for forming terms, as usual. We do not choose this approach and define the various type formers in the setting without universes, as this will give a cleaner categorical semantics. As we shall*

$\mathsf{let}\ *\ \mathsf{be}\ *\ \mathsf{in}\ a = a$	$c[d/z] = \mathsf{let}\ d\ \mathsf{be}\ *\ \mathsf{in}\ c[*/z]$
$\mathsf{let}\ a \otimes b\ \mathsf{be}\ x \otimes y\ \mathsf{in}\ c = c[a/x, b/y]$	$c[d/z] \overset{\#x,y}{=} \mathsf{let}\ d\ \mathsf{be}\ x \otimes y\ \mathsf{in}\ c[x \otimes y/z]$
$(\lambda_{x:A} b)(a) = b[a/x]$	$f \overset{\#x}{=} \lambda_{x:A} f(x)$
$\mathsf{let}\ !a \otimes b\ \mathsf{be}\ !x \otimes y\ \mathsf{in}\ c = c[a/x, b/y]$	$c[d/z] \overset{\#x,y}{=} \mathsf{let}\ d\ \mathsf{be}\ !x \otimes y\ \mathsf{in}\ c[!x \otimes y/z]$
$(\lambda_{!(x:A)} b)(!a) = b[a/x]$	$f \overset{\#x}{=} \lambda_{!(x:A)} f(!x)$
$c = \langle\rangle$	
$\mathsf{fst}(\langle a, b\rangle) = a$ $\mathsf{snd}(\langle a, b\rangle) = b$	$c = \langle \mathsf{fst}(c), \mathsf{snd}(c)\rangle$
	$c[d/z] = \mathsf{false}(d)$
$\mathsf{case}\ \mathsf{inl}(a)\ \mathsf{of}\ \mathsf{inl}(x) \to c \mid\mid \mathsf{inr}(y) \to d = c[a/x]$ $\mathsf{case}\ \mathsf{inr}(b)\ \mathsf{of}\ \mathsf{inl}(x) \to c \mid\mid \mathsf{inr}(y) \to d = d[b/y]$	$c[d/z] \overset{\#x,y}{=} \mathsf{case}\ d\ \mathsf{of}\ \mathsf{inl}(x) \to c[\mathsf{inl}(x)/z] \mid\mid \mathsf{inr}(y) \to c[\mathsf{inr}(y)/z]$
$\mathsf{let}\ !a\ \mathsf{be}\ !x\ \mathsf{in}\ b = b[a/x]$	$c[d/z] \overset{\#x}{=} \mathsf{let}\ d\ \mathsf{be}\ !x\ \mathsf{in}\ c[!x/z]$
$\mathsf{let}\ (a, a, \mathsf{refl}(!a))\ \mathsf{be}\ (z, z, \mathsf{refl}(!z))\ \mathsf{in}\ d = d[a/z]$	$c[d/x, d'/y, e/z] \overset{\#w}{=} \mathsf{let}\ (d, d', e)\ \mathsf{be}\ (w, w, \mathsf{refl}(!w))\ \mathsf{in}\ c[w/x, w/y, \mathsf{refl}(!w)/z]$

Figure 3.7: β- and η-equations for the various connectives. These should be read as equations of typed terms in context: we impose them if we can derive that both terms being equated are well-typed of equal type in equal context. We write $\overset{\#x_1,\ldots,x_n}{=}$ to indicate that for the equation to hold, the identifiers $x_1, \ldots, x_n$ should, in both terms being equated, be replaced by fresh ones, in order to avoid unwanted identifier bindings.

argue in remark 5.6.6, it is more natural to consider a universe as a cartesian type.

Some Basic Results

As the focus of this chapter is the syntax-semantics correspondence, we only briefly mention some syntactic results. For some metatheoretic properties for the $\multimap, \Pi_!^{\multimap}, \top, \&$-fragment of our syntax, like confluence, Church-Rosser, subject reduction and strong normalisation for the (parallel nested, transitive closure of) β-reductions, we refer the reader to [69]. Standard techniques [68] and some small adaptations of the system should be enough to extend the results to all of dDILL. As we discuss a wide range of non-trivial models in section 3.5, consistency of dDILL follows immediately, both in the sense that not all terms are equated and in the sense that not all types are inhabited.

Theorem 3.1.3 (Consistency)**.** *dDILL with all its type formers is consistent.*

To give the reader some intuition for the novel connectives $\Pi_!^{\multimap}$- and $\Sigma_!^{\otimes}$, we suggest the following two interpretations.

Theorem 3.1.4 ($\Pi_!^{\multimap}$ and $\Sigma_!^{\otimes}$ as Dependent $!(-) \multimap (-)$ and $!(-) \otimes (-)$)**.** *Suppose we have* !*-types. Let* $\Gamma, x : A; \cdot \vdash B\ \mathsf{type}$*, where* x *does not occur freely in* B*. Then, for the purposes of the type theory,*

1. $\Pi^{\multimap}_{!(x:A)}B$ *is isomorphic to* $!A \multimap B$*, if we have* $\Pi^{\multimap}_{!}$*-types and* $\multimap$*-types;*

2. $\Sigma^{\otimes}_{!(x:A)}B$ *is isomorphic to* $!A \otimes B$*, if we have* $\Sigma^{\otimes}_{!}$*-types and* $\otimes$*-types.*

Proof. 1. We construct terms

$$\Gamma; y : \Pi^{\multimap}_{!(x:A)}B \vdash f :!A \multimap B \quad \text{and} \quad \Gamma; y' :!A \multimap B \vdash g : \Pi^{\multimap}_{!(x:A)}B$$

s.t.

$\Gamma; y : \Pi^{\multimap}_{!(x:A)}B \vdash g[f/y'] = y : \Pi^{\multimap}_{!(x:A)}B \quad \text{and} \quad \Gamma; y' :!A \multimap B \vdash f[g/y] = y' :!A \multimap B.$

First, we construct f.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x : A; \cdot \vdash x : A}\,\textsf{Cart-Idf} \quad \dfrac{}{\Gamma, x : A; y : \Pi^{\multimap}_{!(x:A)}B \vdash y : \Pi^{\multimap}_{!(x:A)}B}\,\textsf{Lin-Idf}}{\Gamma, x : A; y : \Pi^{\multimap}_{!(x:A)}B \vdash y(!x) : B}\,\Pi^{\multimap}_{!}\textsf{-E} \quad \dfrac{}{\Gamma; x' :!A \vdash x' :!A}\,\textsf{Lin-Idf}}{\Gamma; y : \Pi^{\multimap}_{!(x:A)}B, x' :!A \vdash \mathsf{let}\ x'\ \mathsf{be}\ !x\ \mathsf{in}\ y(!x) : B}\,!\textsf{-E}}{\Gamma; y : \Pi^{\multimap}_{!(x:A)}B \vdash f :!A \multimap B}\,\multimap\textsf{-I}$$

Then, we construct g.

$$\dfrac{\dfrac{\dfrac{}{\Gamma, x : A; \cdot \vdash x : A}\,\textsf{Cart-Idf}}{\Gamma, x : A; \cdot \vdash !x :!A}\,!\textsf{-I} \quad \dfrac{}{\Gamma, x : A; y' :!A \multimap B \vdash y' :!A \multimap B}\,\textsf{Lin-Idf}}{\Gamma, x : A; y' :!A \multimap B \vdash y'(!x) : B}\,\multimap\textsf{-E}}{\Gamma; y' :!A \multimap B \vdash g : \Pi^{\multimap}_{!(x:A)}B}\,\Pi^{\multimap}_{!}\textsf{-I}$$

It is easily verified that $\multimap$-β, $!$-β, and $\Pi^{\multimap}_{!}$-η imply the first judgemental equality:

$g[f/y'] = \lambda_{!(x:A)}(\lambda_{x':!A}\mathsf{let}\ x'\ \mathsf{be}\ !x\ \mathsf{in}\ y(!x))(!x) = \lambda_{!(x:A)}\mathsf{let}\ !x\ \mathsf{be}\ !x\ \mathsf{in}\ y(!x) = \lambda_{!(x:A)}y(!x) = y.$

Similarly, $\Pi^{\multimap}_{!}$-β, $!$-η, and $\multimap$-η imply the second judgemental equality:

$f[g/y] = \lambda_{x':!A}\mathsf{let}\ x'\ \mathsf{be}\ !x\ \mathsf{in}\ (\lambda_{!(x:A)}y'(!x))(!x) = \lambda_{x':!A}\mathsf{let}\ x'\ \mathsf{be}\ !x\ \mathsf{in}\ y'(!x) = \lambda_{x':!A}y'(\mathsf{let}\ x'\ \mathsf{be}\ !x\ \mathsf{in}\ !x) = \lambda_{x':!A}y'(x') = y'.$

2. We construct terms

$$\Gamma; y : \Sigma^{\otimes}_{!(x:A)}B \vdash f :!A \otimes B \quad \text{and} \quad \Gamma; y' :!A \otimes B \vdash g : \Sigma^{\otimes}_{!(x:A)}B$$

s.t.

$\Gamma; y : \Sigma^{\otimes}_{!(x:A)}B \vdash g[f/y'] = y : \Sigma^{\otimes}_{!(x:A)}B \quad \text{and} \quad \Gamma; y' :!A \otimes B \vdash f[g/y] = y' :!A \otimes B.$

First, we construct f.

$$\dfrac{\dfrac{}{\Gamma; y : \Sigma^{\otimes}_{!(x:A)} B \vdash y : \Sigma^{\otimes}_{!(x:A)} B}\,\textsf{Lin-Idf} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma; x' :!A \vdash x' :!A}\,\textsf{Lin-Idf} \quad \dfrac{}{\Gamma; z : B \vdash z : B}\,\textsf{Lin-Idf}}{\Gamma; x' :!A, z : B \vdash x' \otimes z :!A \otimes B}\,\otimes\textsf{-I}}{\Gamma, x : A; x' :!A, z : B \vdash x' \otimes z :!A \otimes B}\,\textsf{Cart-Weak} \qquad \dfrac{\dfrac{}{\Gamma, x : A; \cdot \vdash x : A}\,\textsf{Cart-Idf}}{\Gamma, x : A; \cdot \vdash !x :!A}\,\text{!-I}}{\Gamma, x : A; z : B \vdash !x \otimes z :!A \otimes B}\,\textsf{Lin-Subst}}{\Gamma; y : \Sigma^{\otimes}_{!(x:A)} B \vdash f :!A \otimes B}\,\Sigma^{\otimes}_{!}\textsf{-E}$$

Then, we construct g.

$$\dfrac{\dfrac{}{\Gamma; y' :!A \otimes B \vdash y' :!A \otimes B}\,\textsf{Lin-Idf} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma, x : A; \cdot \vdash x : A}\,\textsf{Cart-Idf} \quad \dfrac{}{\Gamma, x : A; y : B \vdash y : B}\,\textsf{Lin-Idf}}{\Gamma, x : A; y : B \vdash !x \otimes y : \Sigma^{\otimes}_{!(x:A)} B}\,\Sigma^{\otimes}_{!}\textsf{-I} \qquad \dfrac{}{\Gamma; x' :!A \vdash x' :!A}\,\textsf{Lin-Idf}}{\Gamma; x' :!A, y : B \vdash \textsf{let } x' \textsf{ be } !x \textsf{ in } !x \otimes y : \Sigma^{\otimes}_{!(x:A)} B}\,\text{!-E}}{\Gamma; y' :!A \otimes B \vdash g : \Sigma^{\otimes}_{!(x:A)} B}\,\otimes\textsf{-E}$$

Here, the first judgemental equality follows from $\otimes$-β, !-β, and $\Sigma^{\otimes}_{!}$-η:
$g[f/y'] = \textsf{let } (\textsf{let } y \textsf{ be } !x \otimes z \textsf{ in } !x \otimes z) \textsf{ be } x' \otimes y \textsf{ in } (\textsf{let } x' \textsf{ be } !x \textsf{ in } !x \otimes y) = \textsf{let } y \textsf{ be } !x \otimes z \textsf{ in let } !x \otimes z \textsf{ be } x' \otimes y \textsf{ in } (\textsf{let } x' \textsf{ be } !x \textsf{ in } !x \otimes y) = \textsf{let } y \textsf{ be } !x \otimes z \textsf{ in } (\textsf{let } x' \textsf{ be } !x \textsf{ in } !x \otimes y)[!x/x'][z/y] = \textsf{let } y \textsf{ be } !x \otimes z \textsf{ in } (\textsf{let } !x \textsf{ be } !x \textsf{ in } !x \otimes z) = \textsf{let } y \textsf{ be } !x \otimes z \textsf{ in } !x \otimes z = y.$

The second judgemental equality follows from $\Sigma^{\otimes}_{!}$-β, !-η, and $\otimes$-η:
$f[g/y] = \textsf{let } (\textsf{let } y' \textsf{ be } x' \otimes y \textsf{ in } (\textsf{let } x' \textsf{ be } !x \textsf{ in } !x \otimes y)) \textsf{ be } !x \otimes z \textsf{ in } !x \otimes z = \textsf{let } y' \textsf{ be } x' \otimes y \textsf{ in let } x' \textsf{ be } !x \textsf{ in let } !x \otimes y \textsf{ be } !x \otimes z \textsf{ in } !x \otimes z = \textsf{let } y' \textsf{ be } x' \otimes y \textsf{ in let } x' \textsf{ be } !x \textsf{ in } (!x \otimes y) = \textsf{let } y' \textsf{ be } x' \otimes y \textsf{ in } (\textsf{let } x' \textsf{ be } !x \textsf{ in } !x) \otimes y = \textsf{let } y' \textsf{ be } x' \otimes y \textsf{ in } x' \otimes y = y'.$

□

In particular, we have the following stronger version of a special case.

Theorem 3.1.5 (! as ΣI)**.** *Suppose we have $\Sigma^{\otimes}_{!}$- and I-types. Let $\Gamma; \cdot \vdash A$ type. Then, $\Sigma^{\otimes}_{!(x:A)} I$ satisfies the rules for $!A$. Conversely, if we have !- and I-types, then $!A$ satisfies the rules for $\Sigma^{\otimes}_{!(x:A)} I$.*

Proof. We obtain the !-I rule as follows.

$$\dfrac{\Gamma; \cdot \vdash a : A \qquad \dfrac{}{\Gamma, x : A; \cdot \vdash * : I}\,I\textsf{-I}}{\Gamma; \cdot \vdash !a \otimes * : \Sigma^{\otimes}_{!(x:A)} I}\,\Sigma^{\otimes}_{!}\textsf{-I}$$

We obtain the !-E rule as follows.

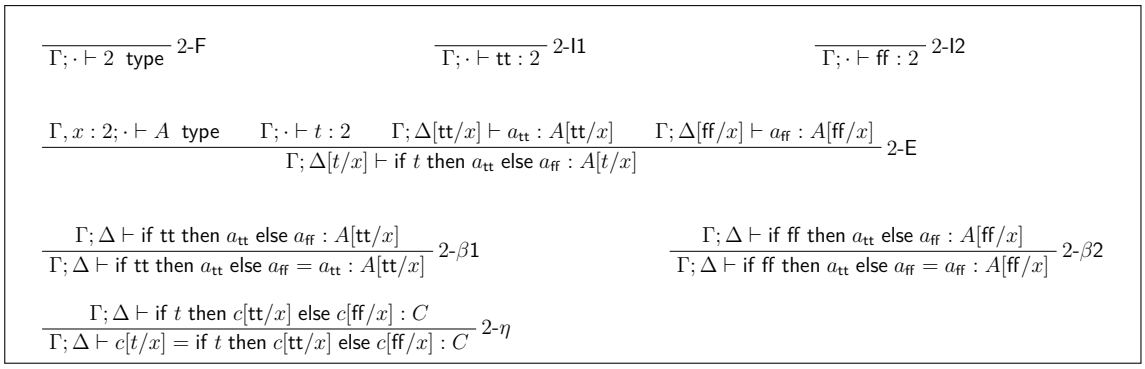


Figure 3.8: Rules for a discrete type 2.

$$\frac{\Gamma;\Delta \vdash t : \Sigma^{\otimes}_{!(x:A)} I \qquad \dfrac{\Gamma, x : A;\Delta' \vdash c : C \qquad \dfrac{}{\Gamma;y : I \vdash y : I}\ \text{Lin-Idf}}{\Gamma, x : A;\Delta', y : I \vdash \mathsf{let}\ y\ \mathsf{be}\ *\ \mathsf{in}\ c : C}\ I\text{-E}}{\Gamma;\Delta,\Delta' \vdash \mathsf{let}\ t\ \mathsf{be}\ !x \otimes y\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ *\ \mathsf{in}\ c : C.}\ \Sigma^{\otimes}_{!}\text{-E}$$

It is easily seen that $\Sigma^{\otimes}_{!}$-β and I-β imply !-β ($\mathsf{let}\ !a \otimes * \ \mathsf{be}\ !x \otimes y\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ *\ \mathsf{in}\ c = (\mathsf{let}\ y\ \mathsf{be}\ *\ \mathsf{in}\ c)[a/x][*/y] = \mathsf{let}\ *\ \mathsf{be}\ *\ \mathsf{in}\ c[a/x] = c[a/x]$) and that I-η and $\Sigma^{\otimes}_{!}$-η imply !-η ($\mathsf{let}\ t\ \mathsf{be}\ !x \otimes y\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ *\ \mathsf{in}\ c[!x \otimes */z] = \mathsf{let}\ t\ \mathsf{be}\ !x \otimes y\ \mathsf{in}\ c[!x \otimes \mathsf{let}\ y\ \mathsf{be}\ *\ \mathsf{in}\ */z] = \mathsf{let}\ t\ \mathsf{be}\ !x \otimes y\ \mathsf{in}\ c[!x \otimes y/z] = c[t/z]$).

The converse statement follows through a similarly trivial argument, noting that $I[a/x]$ is isomorphic to I. □

A second interpretation is that $\Pi^{-\circ}_{!}$ and $\Sigma^{\otimes}_{!}$ generalise & and $\oplus$. Indeed, the idea is that that (or their infinitary equivalents) is what they reduce to when taken over discrete types. The subtlety in this result will be the definition of a discrete type. The same phenomenon is observed in a different context in section 3.5.1.

For our purposes, a discrete type is a strong sum of I (a sum with a dependent -E-rule). Let us for simplicity limit ourselves to the binary case. For us, the discrete type with two elements will be $2 = I \oplus I$, where $\oplus$ has a strong/dependent -E-rule (note that this is not our $\oplus$-E). Explicitly, 2 is a type with the rules of figure 3.8.

Theorem 3.1.6 ($\Pi^{-\circ}_{!}$ and $\Sigma^{\otimes}_{!}$ as Infinitary Non-Discrete & and $\oplus$)**.** *If we have a discrete type* 2 *and a type family* $\Gamma, x : 2;\cdot \vdash A$*, then*

1. $\Pi^{-\circ}_{!(x:2)} A$ *satisfies the rules for* $A[\mathsf{tt}/x] \& A[\mathsf{ff}/x]$*;*

2. $\Sigma^{\otimes}_{!(x:2)}A$ *satisfies the rules for* $A[\mathsf{tt}/x] \oplus A[\mathsf{ff}/x]$.

Proof. 1. We obtain &-I as follows.

$$\dfrac{\dfrac{\Gamma, x:2; \Delta \vdash a : A[\mathsf{tt}/x] \qquad \Gamma, x:2; \Delta \vdash b : A[\mathsf{ff}/x] \qquad \dfrac{}{\Gamma, x:2; \cdot \vdash x : 2}\,\mathsf{Cart\text{-}Idf} \qquad \dfrac{}{\Gamma, x:2; \cdot \vdash A\ \ \mathsf{type}}\,\mathsf{Assumption}}{\Gamma, x:2; \Delta \vdash \mathsf{if}\ x\ \mathsf{then}\ a\ \mathsf{else}\ b : A}\,\mathsf{2\text{-}E\text{-}dep}}{\Gamma; \Delta \vdash \lambda_{!(x:2)}\mathsf{if}\ x\ \mathsf{then}\ a\ \mathsf{else}\ b : \Pi^{\multimap}_{!(x:2)}A}\,\Pi^{\multimap}_{!}\text{-}\mathsf{I}$$

Moreover, we obtain &-E1 as follows (similarly, we obtain &-E2).

$$\dfrac{\Gamma; \Delta \vdash t : \Pi^{\multimap}_{!(x:2)}A \qquad \dfrac{}{\Gamma; \cdot \vdash \mathsf{tt} : 2}\,\mathsf{2\text{-}I1}}{t(!\mathsf{tt})}\,\Pi^{\multimap}_{!}\text{-}\mathsf{E}$$

The &-β-rules follow from $\Pi^{\multimap}_{!}$-β and 2-β, e.g.

$\mathsf{fst}\langle a, b\rangle := (\lambda_{!(x:2)}\mathsf{if}\ x\ \mathsf{then}\ a\ \mathsf{else}\ b)(!\mathsf{tt}) = \mathsf{if}\ \mathsf{tt}\ \mathsf{then}\ a\ \mathsf{else}\ b = a.$

The &-η-rules follow from $\Pi^{\multimap}_{!}$-η and 2-η:

$\langle \mathsf{fst}(t), \mathsf{snd}(t)\rangle := \lambda_{!(x:2)}\mathsf{if}\ x\ \mathsf{then}\ t(!\mathsf{tt})\ \mathsf{else}\ t(!\mathsf{ff}) = \lambda_{!(x:2)}t(!x) = t.$

2. We obtain $\oplus$-I1 as follows (and similarly, we obtain $\oplus$-I2):

$$\dfrac{\dfrac{}{\Gamma; \cdot \vdash \mathsf{tt} : 2}\,\mathsf{2\text{-}I1} \qquad \Gamma; \Delta \vdash a : A[\mathsf{tt}/x]}{\Gamma; \Delta \vdash !\mathsf{tt} \otimes a : \Sigma^{\otimes}_{!(x:2)}A}\,\Sigma^{\otimes}_{!}\text{-}\mathsf{I}$$

Moreover, we obtain $\oplus$-E as follows.

$$\dfrac{\Gamma; \Delta' \vdash t : \Sigma^{\otimes}_{!(x:2)}A \qquad \dfrac{\Gamma; \Delta, z : A[\mathsf{tt}/x] \vdash c : C \qquad \Gamma; \Delta, w : A[\mathsf{ff}/x] \vdash d : C \qquad \dfrac{}{\Gamma, x:2; \cdot \vdash x : 2} \qquad \dfrac{}{\Gamma, x:2; \cdot \vdash A\ \ \mathsf{type}}}{\Gamma, x:2; \Delta, y : A \vdash \mathsf{if}\ x\ \mathsf{then}\ c[y/z]\ \mathsf{else}\ d[y/w] : C}\,\mathsf{2\text{-}E}}{\Gamma; \Delta, \Delta' \vdash \mathsf{let}\ t\ \mathsf{be}\ !x \otimes y\ \mathsf{in}\ \mathsf{if}\ x\ \mathsf{then}\ c[y/z]\ \mathsf{else}\ d[y/w] : C}\,\Sigma^{\otimes}_{!}\text{-}\mathsf{E}$$

The $\oplus$-β-rules follow from $\Sigma^{\otimes}_{!}$-β and 2-β, e.g.

$$\mathsf{case}\ \mathsf{inl}(a)\ \mathsf{of}\ \mathsf{inl}(z) \to c || \ \mathsf{inr}(w) \to d :=$$

$$\mathsf{let}\ !\mathsf{tt} \otimes a\ \mathsf{be}\ !x \otimes y\ \mathsf{in}\ \mathsf{if}\ x\ \mathsf{then}\ c[y/z]\ \mathsf{else}\ d[y/w] =$$

$$\mathsf{if}\ \mathsf{tt}\ \mathsf{then}\ c[a/z]\ \mathsf{else}\ d[a/w] = c[a/z].$$

The $\oplus$-η-rules follow from $\Sigma_!^{\otimes}$-η and 2-η:

$$\begin{aligned}
&\mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}(z) \to c[\mathsf{inl}(z)/u] || \ \mathsf{inr}(w) \to c[\mathsf{inr}(w)/u] := \\
&\mathsf{let}\ t\ \mathsf{be}\ !x \otimes y\ \mathsf{in}\ \mathsf{if}\ x\ \mathsf{then}\ c[\mathsf{inl}(z)/u][y/z]\ \mathsf{else}\ c[\mathsf{inr}(w)/u][y/w] = \\
&\mathsf{let}\ t\ \mathsf{be}\ !x \otimes y\ \mathsf{in}\ \mathsf{if}\ x\ \mathsf{then}\ c[!\mathsf{tt} \otimes z/u][y/z]\ \mathsf{else}\ c[!\mathsf{ff} \otimes w/u][y/w] = \\
&\mathsf{let}\ t\ \mathsf{be}\ !x \otimes y\ \mathsf{in}\ \mathsf{if}\ x\ \mathsf{then}\ c[!\mathsf{tt} \otimes y/u]\ \mathsf{else}\ c[!\mathsf{ff} \otimes y/u] = \\
&\mathsf{let}\ t\ \mathsf{be}\ !x \otimes y\ \mathsf{in}\ c[!(\mathsf{if}\ x\ \mathsf{then}\ \mathsf{tt}\ \mathsf{else}\ \mathsf{ff}) \otimes y/u] = \\
&\mathsf{let}\ t\ \mathsf{be}\ !x \otimes y\ \mathsf{in}\ [!x \otimes y/u] = c[t/u].
\end{aligned}$$

□

We see that we can also view $\Pi_!^{\multimap}$ and $\Sigma_!^{\otimes}$ as generalisations of & and $\oplus$, respectively.

3.2 Semantics of dDILL

The idea behind the categorical semantics we present for the structural core of our syntax (with I- and $\otimes$-types) will be to take our suggested categorical semantics for the structural core of DTT (with 1- and $\times$-types) and relax the assumption of the cartesian character of its fibres to them only being (possibly non-cartesian) symmetric monoidal. This entirely reflects the relation between the conventional semantics of non-dependent cartesian and linear type systems. The structure we obtain is that of a strict indexed symmetric monoidal[2] category with comprehension.

The $\Sigma_!^{\otimes}$- and $\Pi_!^{\multimap}$-types arise as left and right adjoints of substitution functors along projections in the base-category and the $\mathsf{Id}_!^{\otimes}$-types arise as left adjoints to substitution along diagonals, all satisfying Beck-Chevalley (and Frobenius) conditions, as is the case in the semantics for DTT. The !-types boil down to having a left adjoint to the comprehension (which can be made a functor), giving a linear/non-linear adjunction as in the conventional semantics for linear logic. Finally,

[2]It is plausible that we could obtain a sound and complete semantics for only the structural core, possibly without I- and $\otimes$-types, by considering strict indexed symmetric multicategories with comprehension.

additive connectives arise as compatible cartesian and distributive cocartesian structures on the fibres, as would be expected from the semantics of linear logic.

3.2.1 Models of dDILL (Tautologically)

First, we translate the structural core of our syntax to the tautological notion of model. We shall later prove this to be equivalent to the more intuitive notion of categorical model we referred to above.

Definition 3.2.1 (Model of dDILL)**.** *By a model $\widetilde{\mathbb{T}}$ of dDILL, we shall mean the following data.*

(Contexts)	*A set* CCtxt*;*
(Types)	*A map* $\mathsf{CCtxt} \xrightarrow{\mathsf{LType}} \mathsf{Set}$*;*
(Terms, C-Emp1, Lin-C-Ext)	*A map* $\Sigma_{\Gamma \in \mathsf{CCtxt}} \mathsf{LCtxt}(\Gamma) \times \mathsf{LType}(\Gamma) \xrightarrow{\mathsf{LTerm}} \mathsf{Set}$*, where we use the syntactic sugar* $\mathsf{LCtxt}(\Gamma)$ *for the free monoid on* $\mathsf{LType}(\Gamma)$ *whose unit and multiplication we shall write* $\cdot$ *and* $-.-$*;*
(C-Emp2)	*An element* $\cdot \in \mathsf{CCtxt}$*;*
(Cart-C-Ext)	*A map* $\Sigma_{\Gamma \in \mathsf{CCtxt}} \mathsf{LType}(\Gamma) \xrightarrow{\_.\_} \mathsf{CCtxt}$*;*
(Cart-Weak)	*Maps* $\mathsf{LType}(\Gamma.\Gamma') \xrightarrow{\mathsf{weak}} \mathsf{LType}(\Gamma.A.\mathsf{weak}(\Gamma'))$ *and* $\mathsf{LTerm}(\Gamma.\Gamma', \Delta, B) \xrightarrow{\mathsf{weak}} \mathsf{LTerm}(\Gamma.A.\mathsf{weak}(\Gamma'), \mathsf{weak}(\Delta), \mathsf{weak}(B))$ *(where we slightly abuse notation);*
(Cart-Idf)	*Elements* $\mathsf{der} \in \mathsf{LTerm}(\Gamma.A.\Gamma', \cdot, \mathsf{weak}(A))$*;*
(Lin-Idf)	*Elements* $\mathsf{id} \in \mathsf{LTerm}(\Gamma, A, A)$*;*
(Cart-Ty-Subst)	*For* $B \in \mathsf{LType}(\Gamma.A.\Gamma')$ *and* $a \in \mathsf{LTerm}(\Gamma, \cdot, A)$*, we have* $B\{\Gamma.a.\Gamma'\} \in \mathsf{LType}(\Gamma.\Gamma'\{\Gamma.a\})$*;*
(Cart-Tm-Subst)	*For* $b \in \mathsf{LTerm}(\Gamma.A.\Gamma', \Delta, B)$ *and* $a \in \mathsf{LTerm}(\Gamma, \cdot, A)$*, we have* $b\{\Gamma.a.\Gamma'\} \in \mathsf{LTerm}(\Gamma.\Gamma'\{\Gamma.a\}, \Delta\{\Gamma.a\}, B\{\Gamma.a\})$*;*
(Lin-Tm-Subst)	*For* $b \in \mathsf{LTerm}(\Gamma, \Delta.A.\Delta', B)$ *and* $a \in \mathsf{LTerm}(\Gamma, \Delta'', A)$*, we have* $(\Delta.a.\Delta'); b \in \mathsf{LTerm}(\Gamma, \Delta.\Delta'.\Delta'', B)$*,*

such that

- weak *preserves* id, der, $-;-$ *and* $-\{-\}$ *in the obvious sense;*
- $-\{-\}$ *commutes with* $-;-$ *in the obvious sense;*
- $-;-$ *is associative;*
- $-;-$*-substitutions in disjoint parts of the context commute: if* $j < i$ *(set* $N = 0$*) or* $j > i + m - 1$ *(set* $N = m$*) then*
  $(C_1.\ldots.C_{j-1}.a'.C_{j+1}.\ldots.C_{n+m-1}); (A_1.\ldots.A_{i-1}.a.A_{i+1}.\ldots.A_n); b$
  $= (C_1.\ldots.C_{j-1}.a.C_{j+1}.\ldots.C_{n+m-1}); (A_1.\ldots.A_{j+N-1}.a'.A_{j+N+1}.\ldots.A_n); b;$
- $-\{-\}$ *on terms is associative;*
- $-\{-\}$*-term substitutions in disjoint parts of the context commute (as for* $-;-$ *substitutions);*
- $(\Delta.\mathsf{id}.\Delta'); b = b$ *for all* $b \in \mathsf{LTerm}(\Gamma, \Delta.A.\Delta', B)$*;*
- *the actions on both* LType *and* LTerm *of* $-\{\Gamma.\mathsf{der}.\Gamma'\}$ *("substituting a diagonal") and* weak *("substituting a projection") satisfy all equations induced by the theory of cartesian products (see [79] for these precise equations).*

The equations we demand in this definition are all the standard equations that are implicit for syntactic substitution. The point of these laws is that we can form context morphisms as lists of compatible terms, which we can then substitute into terms (and types) in an associative way, using the operations $-\{-\}$ and weak in the case of cartesian contexts and using $-;-$ in the case of linear contexts. Note that commutativity of disjoint substitutions and the fact that weak preserves $-\{-\}$ imply that this parallel substitution is well-defined.

We interpret $[\![\vdash \mathsf{ctxt}]\!] := \mathsf{Ctxt}$, $[\![\Gamma \vdash \mathsf{type}]\!] := \mathsf{LType}(\Gamma)$ and $[\![\Gamma; \Delta \vdash A]\!] := \mathsf{LTerm}(\Gamma, \Delta, A)$. We interpret judgemental equality of contexts, types and terms as the equality on the sets Ctxt, $\mathsf{LType}(\Gamma)$ and $\mathsf{LTerm}(\Gamma, \Delta, A)$. Note that all rules for judgemental equality (the rules with Eq, Conv and Cong in their name) then automatically follow. It is tautological that there is a one to one correspondence between theories $\mathbb{T}$ in dDILL and models $\widetilde{\mathbb{T}}$ of this sort.

We now define what it means for the model to support various type formers.

Definition 3.2.2 (Semantic I- and $\otimes$-types)**.** *We say a model* $\widetilde{\mathbb{T}}$ *supports* I*-types, if for all* $\Gamma \in \mathsf{CCtxt}$*, we have an* $I \in \mathsf{LType}(\Gamma)$ *and* $* \in \mathsf{LTerm}(\Gamma, \cdot, I)$ *and whenever* $t \in \mathsf{LTerm}(\Gamma, \Delta, I)$ *and* $a \in \mathsf{LTerm}(\Gamma, \Delta', A)$*, we have* $\mathsf{let}\ t\ \mathsf{be}\ *\ \mathsf{in}\ a \in \mathsf{LTerm}(\Gamma, \Delta.\Delta', A)$*, such that* $\mathsf{let}\ *\ \mathsf{be}\ *\ \mathsf{in}\ a = a$ *and* $(\Delta'.a); t = \mathsf{let}\ a\ \mathsf{be}\ *\ \mathsf{in}\ ((\Delta'.*); t)$*.*

Similarly, we say it admits $\otimes$*-types, if for all* $A, B \in \mathsf{LType}(\Gamma)$*, we have a* $A \otimes B \in \mathsf{LType}(\Gamma)$*, for all* $a \in \mathsf{LTerm}(\Gamma, \Delta, A), b \in \mathsf{LTerm}(\Gamma, \Delta', B)$*, we have* $a \otimes b \in \mathsf{LType}(\Gamma, \Delta.\Delta', A \otimes B)$*, and if* $t \in \mathsf{LTerm}(\Gamma, \Delta, A \otimes B)$ *and* $c \in \mathsf{LTerm}(\Gamma, \Delta'.A.B, C)$*, we have* $\mathsf{let}\ t\ \mathsf{be}\ \mathsf{id}_A \otimes \mathsf{id}_B\ \mathsf{in}\ c \in \mathsf{LTerm}(\Gamma, \Delta.\Delta', C)$*, such that* $\mathsf{let}\ a \otimes b\ \mathsf{be}\ \mathsf{id}_A \otimes \mathsf{id}_B\ \mathsf{in}\ c = c$ *and* $(\Delta'.d); t = \mathsf{let}\ t\ \mathsf{be}\ \mathsf{id}_A \otimes \mathsf{id}_B\ \mathsf{in}\ (\Delta'.(\mathsf{id}_A \otimes \mathsf{id}_B); t)$*.*

Note that this defines a function $\mathsf{LCtxt}(\Gamma) \xrightarrow{\bigotimes} \mathsf{LType}$*. The* β*-rule precisely says that from the point of view of the (terms of the) type theory this map is an injection, while the* η*-rule says it is a surjection*[3]*. We conclude that in the presence of* I*- and* $\otimes$*-types, we can faithfully describe the type theory without mentioning linear contexts, replacing them by the linear type that is their* $\otimes$*-product.*

We shall henceforth assume that our type theory has I- and $\otimes$-types, as this simplifies the categorical semantics[4] and is appropriate for the examples we are interested in.

For the other type formers, one can give a similar, almost tautological, translation from the syntax into a model. We leave this to the reader when we discuss the semantic equivalent of various type formers in the categorical semantics we present next.

[3]The precise statement that we are alluding to here would be that the multicategory of linear contexts is equivalent to the (monoidal) multicategory of linear types. Really, $\bigotimes$ is only part of an equivalence of categories rather than an isomorphism, i.e. it is injective on objects up to isomorphism rather than on the nose.

[4]To be precise, it allows us to give a categorical semantics in terms of monoidal categories rather than multicategories.

3.2.2 Categorical Semantics of dDILL

Strict Indexed Symmetric Monoidal Categories with Comprehension

We now introduce a notion of categorical model for which soundness and completeness results hold with respect to the syntax of dDILL in the presence of I- and $\otimes$-types[5]. This notion of model will prove to be particularly useful when thinking about various type formers.

Definition 3.2.3. *By a* ***strict indexed symmetric monoidal category with comprehension****, we mean the following data.*

1. *A category $\mathcal{B}$ with a terminal object $\cdot$.*

2. *A strict indexed symmetric monoidal category $\mathcal{D}$ over $\mathcal{B}$, i.e. a contravariant functor $\mathcal{D}$ into the category* SMCat *of (small) symmetric monoidal categories and strict monoidal functors $\mathcal{B}^{op} \xrightarrow{\mathcal{D}} \mathsf{SMCat}$. We also write $-\{f\} := \mathcal{D}(f)$ for the action of $\mathcal{D}$ on a morphism f of $\mathcal{B}$.*

3. *A* ***comprehension schema****, i.e. for each $\Gamma \in \mathsf{ob}(\mathcal{B})$ and $A \in \mathsf{ob}(\mathcal{D}(\Gamma))$ a representation for the functor*

$$x \mapsto \mathcal{D}(\mathsf{dom}(x))(I, A\{x\}) : (\mathcal{B}/\Gamma)^{op} \longrightarrow \mathsf{Set}.$$

We write its representing object[6] $\Gamma.A \xrightarrow{\mathbf{p}_{\Gamma,A}} \Gamma \in \mathsf{ob}(\mathcal{B}/\Gamma)$ *and universal element* $\mathbf{v}_{\Gamma,A} \in \mathcal{D}(\Gamma.A)(I, A\{\mathbf{p}_{\Gamma,A}\})$. *We write $a \mapsto \langle f, a\rangle$ for the isomorphism $\mathcal{D}(\Gamma')(I, A\{f\}) \cong \mathcal{B}/\Gamma(f, \mathbf{p}_{\Gamma,A})$.*

Again, the comprehension schema means that the morphisms in our category of contexts $\mathcal{B}$, into a context built by adjoining types, arise as lists of closed linear terms. Here, there is the crucial identification with cartesian terms of linear terms without linear assumptions: they can be freely copied and discarded.

[5] In case we are interested in the case without I- and $\otimes$-types, the semantics easily generalises to strict indexed symmetric multicategories with comprehension.

[6] Really, $\Gamma.UA \xrightarrow{\mathbf{p}_{\Gamma,UA}} \Gamma$ would be a better notation, where we think of $F \dashv U$ as an adjunction inducing !, but it would be very verbose.

We note that the definition of comprehension for an indexed symmetric monoidal category is almost identical to that of definition 2.1.4 for an indexed cartesian monoidal category. The only difference is that the tensor unit now plays the rôle of the terminal object. We again use the same definitions for diag, $\mathbf{q}$ and the comprehension functors $\mathbf{p}_{\Gamma,-}$.

Theorem 3.2.4 (Comprehension functor)**.** *A comprehension schema* $(\mathbf{p}, \mathbf{v})$ *on a strict indexed symmetric monoidal category* $(\mathcal{B}, \mathcal{D})$ *defines a morphism* $\mathcal{D} \xrightarrow{U} \mathcal{C}$ *of indexed symmetric monoidal categories, which lax-ly sends the monoidal structure of* $\mathcal{D}$ *to products in* $\mathcal{C}$ *(where they exist), where* $\mathcal{C}$ *is the full subindexed*[7] *category of* $\mathcal{B}/-$ *on the objects of the form* $\mathbf{p}_{\Gamma,A}$.

Proof. First note that a morphism U of indexed symmetric monoidal categories consists of lax monoidal functors U_Γ in each context $\Gamma \in \mathcal{B}$ such that

$$\begin{array}{ccc} \mathcal{D}(\Gamma) & \xrightarrow{U_\Gamma} & \mathcal{C}(\Gamma) \\ \Big\downarrow{\scriptstyle \mathcal{D}(f)} & \cong & \Big\downarrow{\scriptstyle \mathcal{C}(f) = \text{“pullback along } f\text{”}} \\ \mathcal{D}(\Gamma') & \xrightarrow{U_{\Gamma'}} & \mathcal{C}(\Gamma'). \end{array}$$

We define

$$U_\Gamma(A \xrightarrow{a} B) := \mathbf{p}_{\Gamma,A} \xrightarrow{\langle \mathbf{p}_{\Gamma,A}, \mathbf{v}_{\Gamma,A}; a\{\mathbf{p}_{\Gamma,A}\}\rangle} \mathbf{p}_{\Gamma,B}.$$

Functoriality follows from the uniqueness property of $\langle \mathbf{p}_{\Gamma,A}, \mathbf{v}_{\Gamma,A}; a\{\mathbf{p}_{\Gamma,A}\}\rangle$.

We define the lax monoidal structure

$$\mathsf{id}_\Gamma \xrightarrow{m^I_\Gamma} U_\Gamma(I) = \mathbf{p}_{\Gamma,I}$$

$$\mathbf{p}_{\Gamma.A,B\{\mathbf{p}_{\Gamma,A}\}}; \mathbf{p}_{\Gamma,A} = U_\Gamma(A) \times U_\Gamma(B) \xrightarrow{m^{A,B}_\Gamma} U_\Gamma(A \otimes B) = \mathbf{p}_{\Gamma,A\otimes B},$$

where $m^{A,B}_\Gamma := \langle \mathbf{p}_{\Gamma.A,B\{\mathbf{p}_{\Gamma,A}\}}; \mathbf{p}_{\Gamma,A}, \mathbf{v}_{\Gamma,A}\{\mathbf{p}_{\Gamma.A,B\{\mathbf{p}_{\Gamma,A}\}}\} \otimes \mathbf{v}_{\Gamma.A,B\{\mathbf{p}_{\Gamma,A}\}}\rangle$ and $m^I_\Gamma := \langle \mathsf{id}_\Gamma, \mathsf{id}_I\rangle$.

[7] Here, we use the axiom of choice to make a choice of pullback and make $\mathcal{C}$ really into a (non-strict) indexed category (or cloven fibration). Alternatively, we can avoid the axiom of choice and treat it as a more general fibration.

Finally, we verify that $\mathcal{C}(f)U_\Gamma = U_{\Gamma'}\mathcal{D}(f)$. This follows directly from the fact that the following square is a pullback square:

$$\begin{array}{ccc} \Gamma'.A\{f\} & \xrightarrow{\mathbf{q}_{f,A}} & \Gamma.A \\ \downarrow{\scriptstyle \mathbf{p}_{\Gamma',A\{f\}}} & & \downarrow{\scriptstyle \mathbf{p}_{\Gamma,A}} \\ \Gamma' & \xrightarrow[f]{} & \Gamma, \end{array}$$

where $\mathbf{q}_{f,A} := \langle f\mathbf{p}_{\Gamma',A\{f\}}, \mathbf{v}_{\Gamma',A\{f\}}\rangle$. We leave this verification to the reader as an exercise. Alternatively, a proof for this fact in DTT, that will transfer to our setting in its entirety, can be found in [20]. □

Remark 3.2.5. *Note that $\mathcal{C}$ is a display map category (or, less specifically, a full comprehension category) and, using the axiom of choice to make a choice of pullbacks, can be viewed as a (non-strict) indexed category with full and faithful comprehension, another, slightly weaker, commonly used notion of model of dependent types. We shall see that, in many ways, we can regard $\mathcal{C}$ as the cartesian content of $\mathcal{D}$.*

Remark 3.2.6. *We shall see that this functor will give us a unique candidate for !-types: $! := FU$, where $F \dashv U$. We conclude that, in dDILL, the !-modality is uniquely determined by the indexing. This is worth noting, because, in propositional linear type theory, we might have many different candidates for !-types.*

Moreover, it explains why we do not demand U to be fully faithful in the case of linear types. Indeed, although we have a map $\mathcal{D}(\Gamma)(A, B) \xrightarrow{U_\Gamma} \mathcal{C}(\Gamma)(\mathbf{p}_{\Gamma,A}, \mathbf{p}_{\Gamma,B}) \cong \mathcal{D}(\Gamma.A)(I, B\{\mathbf{p}_{\Gamma,A}\})$, this is not generally an isomorphism. In fact, in the presence of !-types, we shall see that the right hand side is precisely isomorphic to $\mathcal{D}(\Gamma)(!A, B)$ and the map is precomposition with dereliction.

Next, we prove that we have a sound interpretation of dDILL in such categories.

Theorem 3.2.7 (Soundness)**.** *A strict indexed symmetric monoidal category with comprehension $(\mathcal{B}, \mathcal{D}, \mathbf{p}, \mathbf{v})$ defines a model $\widetilde{\mathbb{T}}^{(\mathcal{B},\mathcal{D},\mathbf{p},\mathbf{v})}$ of dDILL with I- and $\otimes$-types.*

Proof. We define

1. Contexts: $\mathsf{CCtxt} := \mathsf{ob}(\mathcal{B})$

2. Types: $\mathsf{LType}(\Gamma) := \mathsf{ob}(\mathcal{D}(\Gamma))$

3. $\mathsf{LCtxt}(\Gamma) := \mathsf{free-monoid}(\mathsf{LType}(\Gamma))$ (where we write $[]$ and $++$ for the monoid operations)
   C-Emp1: $\cdot_{\mathsf{LCtxt}(\Gamma)} := []_{\mathsf{LCtxt}(\Gamma)}$
   Lin-C-Ext: $\Delta._{\mathsf{LCtxt}} A := \Delta + +A$
   Terms: $\mathsf{LTerm}(\Gamma, \Delta, A) := \mathcal{D}(\Gamma)(\bigotimes \Delta, A)$

4. C-Emp2: $\cdot_{\mathsf{CCtxt}} := \cdot_{\mathcal{B}}$

5. Cart-C-Ext: $\Gamma._{\mathsf{CCtxt}} A := \Gamma._{\mathcal{B}} A$.

6. Cart-Weak: The required morphisms are interpreted as follows. Suppose we are given $A, \Gamma' \in \mathsf{ob}(\mathcal{D}(\Gamma))$. We define a weakening functor
$$\mathcal{D}(\Gamma.\Gamma') \xrightarrow{\mathcal{D}(\langle f, a\rangle)} \mathcal{D}(\Gamma.A.\Gamma'\{\mathbf{p}_{\Gamma,A}\}),$$
where f and a are defined as follows.
$$\Gamma.A.\Gamma'\{\mathbf{p}_{\Gamma,A}\} \xrightarrow{f := \mathbf{p}_{\Gamma.A,\Gamma'\{\mathbf{p}_{\Gamma,A}\}};\mathbf{p}_{\Gamma,A}} \Gamma$$
and
$$I \xrightarrow{a = \mathbf{v}_{\Gamma,A.\Gamma'\{\mathbf{p}_{\Gamma,A}\}}} \Gamma'\{f\} = \Gamma'\{\mathbf{p}_{\Gamma,A.\Gamma'\{\mathbf{p}_{\Gamma,A}\}}\} \ \in \mathcal{D}(\Gamma.A.\Gamma'\{\mathbf{p}_{\Gamma,A}\}).$$
Note that this interpretation of weakening preserves der (by definition) and id (as it is a functor) and commutes with the three substitution operations (by functoriality of $-\{\mathbf{p}\}$ and by functoriality of $-\{-\}$ in the second argument).

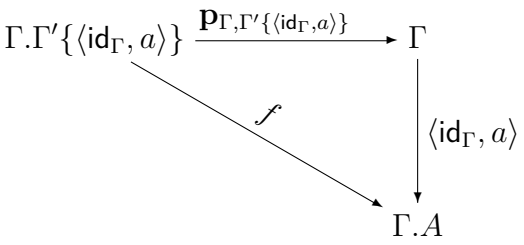
7. Cart-Idf: $\mathsf{der}_{\Gamma,A,\Gamma'} \in \mathsf{LTerm}(\Gamma.A.\Gamma', \cdot, A)$ is defined as
$$\mathbf{v}_{\Gamma,A}\{\mathbf{p}_{\Gamma.A,\Gamma'}\} : I \longrightarrow A\{\mathbf{p}_{\Gamma.A,\Gamma'};\mathbf{p}_{\Gamma,A}\} \in \mathcal{D}(\Gamma.A.\Gamma')$$
Note that $\mathsf{der}_{\Gamma,A,\Gamma'}$ defines a morphism
$\Gamma.A.\Gamma' \xrightarrow{\mathsf{diag}_{\Gamma,A,\Gamma'}} \Gamma.A.\Gamma'.A\{\mathbf{p}_{\Gamma.A,\Gamma'};\mathbf{p}_{\Gamma,A}\} := \langle \mathsf{id}_{\Gamma.A.\Gamma'}, \mathsf{der}_{\Gamma,A,\Gamma'}\rangle$.
We shall later show that this in fact behaves as a diagonal morphism on A.

8. Lin-Idf: $\mathsf{id}_A \in \mathsf{LTerm}(\Gamma, A, A)$ is taken to be $\mathsf{id}_{\mathsf{A}} \in \mathcal{D}(\Gamma)(A, A)$. Note that this is indeed the neutral element for our semantic linear term substitution operation that we shall define shortly.

9. Cart-Ty-Subst and Cart-Tm-Subst: substitution along a term $\Gamma; \cdot \vdash a : A$, are interpreted by the functors $\mathcal{D}(\langle \mathsf{id}_\Gamma, a\rangle) = -\{\langle \mathsf{id}_\Gamma, a\rangle\}$. Indeed, let $B \in \mathcal{D}(\Gamma.A.\Gamma')$ and $a \in \mathcal{D}(\Gamma)(I, A)$. Then, we define the context $\Gamma.\Gamma'\{\Gamma.a/x\}$ as $\Gamma.(\Gamma'\{\langle \mathsf{id}_\Gamma, a\rangle\})$ and the type $B\{\Gamma.a.\Gamma'\}$ as $B\{\langle f, a'\rangle\}$, where

$$\Gamma.\Gamma'\{\langle \mathsf{id}_\Gamma, a\rangle\}.I \xrightarrow{\langle f, a'\rangle} \Gamma.A.\Gamma'$$

is defined from



and

$$I \xrightarrow{a' := \mathbf{v}_{\Gamma,\Gamma'\{\langle \mathsf{id}_\Gamma, a\rangle\}}} \Gamma'\{f\} = (\Gamma'\{\langle \mathsf{id}_\Gamma, a\rangle\})\{\mathbf{p}_{\Gamma,\Gamma'\{\langle \mathsf{id}_\Gamma, a\rangle\}}\}.$$

10. Lin-Tm-Subst: interpreted by composition in $\mathcal{D}(\Gamma)$. To be precise, given $b \in \mathcal{D}(\Gamma)((\bigotimes \Delta) \otimes A \otimes (\bigotimes \Delta'), B)$ and $a \in \mathcal{D}(\Gamma)(\bigotimes \Delta'', A)$, we define $b[a/x] \in \mathcal{D}(\Gamma)((\bigotimes \Delta) \otimes (\bigotimes \Delta') \otimes (\bigotimes \Delta''), B)$ as $(\mathsf{id}_{\bigotimes \Delta} \otimes a \otimes \mathsf{id}_{\bigotimes \Delta'}); \mathsf{braid}_{\bigotimes \Delta', \bigotimes \Delta''}; b$.

Note that Cart-Ty-Subst and Cart-Tm-Subst are interpreted by functors and therefore preserve identities and compositions and are associative in their composition. Lin-Tm-Subst is interpreted by composition in the fibre categories, hence is also associative.

The fact that Cart-Idf and Cart-Weak define compatible diagonals and projections follows from the fact that $\Gamma.A.B\{\mathbf{p}_{\Gamma,A}\} \xrightarrow{\mathbf{p}_{\Gamma.A,B\{\mathbf{p}_{\Gamma,A}\}}} \Gamma.A \xrightarrow{\mathbf{p}_{\Gamma,A}} \Gamma$ defines the cartesian product of $\mathbf{p}_{\Gamma,A}$ and $\mathbf{p}_{\Gamma,B}$ in $\mathcal{B}/\Gamma$.

Finally, the model clearly supports I- and $\otimes$-types. We interpret $I \in \mathsf{LType}(\Gamma)$ as the unit object in $\mathcal{D}(\Gamma)$ while its term $*$ is interpreted as the identity morphism. Similarly, we interpret $\otimes$ by the monoidal product on the fibres: $* := \mathsf{id}_I \in \mathcal{D}(\Gamma)$,

$\mathsf{let}\ t\ \mathsf{be}\ *\ \mathsf{in}\ a := t \otimes a$, $a \otimes b$ is defined as the tensor product of morphisms in $\mathcal{D}(\Gamma)$, and $\mathsf{let}\ t\ \mathsf{be}\ \mathsf{id}_A \otimes \mathsf{id}_B\ \mathsf{in}\ c := (\mathsf{id}_{\Delta'} \otimes t); c$ (leaving out associatiators and unitors, here). The β- and η-rules are immediate. □

In fact, the converse is also true: we can build a category of this sort from the syntax of dDILL.

Theorem 3.2.8 (Co-Soundness)**.** *A model $\widetilde{\mathbb{T}}$ of dDILL with I and $\otimes$-types defines a strict indexed symmetric monoidal category with comprehension $(\mathcal{B}^{\mathbb{T}}, \mathcal{D}^{\mathbb{T}}, \mathbf{p}^{\mathbb{T}}, \mathbf{v}^{\mathbb{T}})$.*

Proof. The main technical difficulty in this proof will be that our syntactic category has context morphisms as morphisms (corresponding to lists of terms of the type theory) while the type theory only talks about individual terms. This exact difficulty is also encountered when proving completeness of the categories with families semantics for ordinary DTT. It is sometimes fixed by (conservatively) extending the the type theory to also talk about context morphisms explicitly. See e.g. [19].

1. We define $\mathsf{ob}(\mathcal{B}^{\mathbb{T}}) := \mathsf{CCtxt}$, modulo α-equivalence, and write $\Gamma.A$ for the equivalence class of $\Gamma, x : A$. The designated object $\cdot$ of $\mathcal{B}^{\mathbb{T}}$ will be the (equivalence class of) $\cdot$ (from $\mathsf{C\text{-}Emp}$), which will automatically become a terminal object because of our definition of a morphism of $\mathcal{B}^{\mathbb{T}}$ (context morphism). Indeed, we define morphisms in $\mathcal{B}^{\mathbb{T}}$, as follows, by induction.

   We start out by defining $\mathcal{B}^{\mathbb{T}}(\Gamma', \cdot) := \{\langle\rangle\}$ and for $\Gamma \in \mathsf{CCtxt}$ that are not of the form $\Gamma''.A$, define $\mathcal{B}^{\mathbb{T}}(\Gamma', \Gamma) = \{\mathsf{id}_\Gamma\}$ if $\Gamma' = \Gamma$ and $\mathcal{B}^{\mathbb{T}}(\Gamma', \Gamma) = \emptyset$ otherwise.

   Then, by induction on the length n of $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, we define

$$\mathcal{B}^{\mathbb{T}}(\Gamma', \Gamma.A_{n+1}) := \Sigma_{f \in \mathcal{B}^{\mathbb{T}}(\Gamma', \Gamma)} \mathsf{LTerm}(\Gamma', \cdot, A_{n+1}[f/x]),$$

   where $A_{n+1}[f/x]$ is defined, using $\mathsf{Cart\text{-}Ty\text{-}Subst}$, to be the (syntactic operation of) parallel substitution (see [20], section 2.4) of the list $f_1, \ldots, f_n$ of linear terms $\Gamma'; \cdot \vdash f_i : A_i[f_1/x_1, \ldots, f_{i-1}/x_{i-1}]$ that f is made up out of, for the identifiers $x_1, \ldots, x_n$ in Γ.

Note that, in particular, according to Cart-Idf, $\mathsf{LTerm}(A_1.\ldots.A_n.,\cdot,A_i)$ contains a term $\mathsf{der}_{A_1.....A_{i-1},A_i,A_{i+1}.....A_n}$, which allows us to define, inductively,

$$\mathbf{p}^n_{A_1.....A_n} := \langle\rangle \in \mathcal{B}^{\mathbb{T}}(A_1.\ldots.A_n,\cdot)$$

$$\mathbf{p}^{n-i}_{A_1.....A_n} :=$$

$$\mathbf{p}^{n-i+1}_{A_1.....A_n}, \mathsf{der}_{A_1.....A_{i-1},A_i,A_{i+1}.....A_n} \in \mathcal{B}^{\mathbb{T}}(A_1.\ldots.A_n, A_1.\ldots.A_i)$$

In particular, we define identities in $\mathcal{B}^{\mathbb{T}}$ from these: $\mathsf{id}_{A_1.....A_n} := \mathbf{p}^0_{A_1.....A_n}$. We shall also use these 'projections' in 3. to define the comprehension schema. In all cases, projections, identities and diagonals defined using der behave as such via substitutions because we have demanded that the actions of Cart-Idf and Cart-Weak interact via the laws induced from the theory of cartesian products.

We define composition in $\mathcal{B}^{\mathbb{T}}$ by induction. Let $B_1.\ldots.B_m = \Gamma' \overset{f=f_1,\ldots,f_n}{\longrightarrow} \Gamma = A_1.\ldots.A_n$ and $\Gamma'' \overset{g=g_1,\ldots,g_m}{\longrightarrow} \Gamma'$. Then, we define, by induction, $g;() := ()$ and $g;(f_1,\ldots,f_{n-1},f_n) := g;(f_1,\ldots,f_{n-1}),f_n[g/x]$, where $f_n[g/x]$ denotes the parallel substitution of $g = g_1,\ldots,g_m$ for the free identifiers $x_1,\ldots,x_m$ in f_n, using Cart-Tm-Subst. Note that associativity of composition comes from the associativity of substitution that is implicit in the syntax as well as the compatibility of substitution with weakening while the identity morphism we defined clearly acts as a neutral element for our composition.

2. Define $\mathsf{ob}(\mathcal{D}^{\mathbb{T}}(\Gamma)) := \mathsf{LCtxt}(\Gamma)$ and $\mathcal{D}^{\mathbb{T}}(\Gamma)(\Delta,\Delta') := \mathsf{LTerm}(\Gamma,\Delta,\bigotimes\Delta')$. Composition is defined through Lin-Tm-Subst and $\otimes$-E. Identities are given by Lin-Idf. The monoidal unit is given by $\cdot \in \mathsf{LCtxt}(\Gamma)$, while the monoidal product $\otimes$ on objects is given by context concatenation. The monoidal product $\otimes$ on morphisms is given by $\otimes$-I. Note that the associators and unitors follow from the associative and unital laws for the commutative monoid of contexts together with $\otimes$-β and $\otimes$-η and that the symmetry/braid comes from the commutativity of the monoid. (Note that the rules for $\otimes$ give

us an isomorphism between an arbitrary context Δ and the one-type-context $\bigotimes \Delta$, while the rules for I do the same for $\cdot$ and I.)

We define $\mathcal{D}^{\mathbb{T}}(f)$ on objects by parallel substitution and weakening in each type in a linear context, via Cart-Ty-Subst and Cart-Weak, and on morphisms by parallel substitution and weakening, via Cart-Tm-Subst and Cart-Weak. Note that functoriality is given by implicit properties of the syntax like associativity of substitution. Note that this defines a strict symmetric monoidal functor. We conclude that $\mathcal{D}^{\mathbb{T}}$ is a functor $\mathcal{B}^{\mathbb{T}op} \longrightarrow \mathsf{SMCat}$.

3. We define following comprehension schema on $\mathcal{D}^{\mathbb{T}}$. Suppose $\Gamma \in \mathcal{B}^{\mathbb{T}}$ and $A \in \mathcal{D}^{\mathbb{T}}(\Gamma)$.

   Define $\Gamma.A \overset{\mathbf{p}^{\mathbb{T}}{}_{\Gamma,A}}{\longrightarrow} \Gamma$ as $\mathbf{p}^1_{\Gamma.A}$ from 1. and $I \overset{\mathbf{v}^{\mathbb{T}}{}_{\Gamma,A}}{\longrightarrow} A\{\mathbf{p}^{\mathbb{T}}_{\Gamma,A}\}$ (through Cart-Idf) as $\mathsf{der}_A \in \mathsf{LTerm}(\Gamma.A, \cdot, A) = \mathcal{D}^{\mathbb{T}}(\Gamma.A)(I, A\{\mathbf{p}^{\mathbb{T}}{}_{\Gamma,A}\})$.

   Suppose we are given $\Gamma' \xrightarrow{f} \Gamma$ and $a \in \mathcal{D}^{\mathbb{T}}(\Gamma')(I, A\{f\}) = \mathsf{LTerm}(\Gamma', \cdot, A[f/c])$. Then, by definition of the morphisms in $\mathcal{B}^{\mathbb{T}}$, there is a unique morphism $\langle f, a\rangle := f, a \in \mathcal{B}^{\mathbb{T}}(\Gamma', \Gamma.A) := \Sigma_{f \in \mathcal{B}^{\mathbb{T}}(\Gamma',\Gamma)} \mathsf{LTerm}(\Gamma', \cdot, A[f/x])$ such that $\langle f, a\rangle; \mathbf{p}^{\mathbb{T}}_{\Gamma,A} = f$ and $\mathbf{v}^{\mathbb{T}}_{\Gamma,A}\{\langle f, a\rangle\} = a$. The uniqueness follows from the fact that $-; \mathbf{p}^{\mathbb{T}}_{\Gamma,A}$ and $\mathbf{v}^{\mathbb{T}}{}_{\Gamma,A}\{-\}$ are the two (dependent) projections of the Σ-type (in Set) that defines this homset. We can note this bijection is natural in the sense that $g; \langle f, a\rangle = \langle g; f, a\{g\}\rangle$ because of the associativity of the substitution Cart-Tm-Subst in the syntax.

□

Theorem 3.2.9 (Completeness)**.** *The construction described in 'Co-Soundness' followed by the one described in 'Soundness' is the identity (up to categorical equivalence): i.e. strict indexed symmetric monoidal categories with comprehension provide a complete semantics for dDILL with I- and $\otimes$-types*[8]*.*

Proof. This is a trivial exercise. □

[8]It is easy to see that, similarly, indexed symmetric multicategories with comprehension form a complete semantics for dDILL, possibly without I- and $\otimes$-types.

Theorem 3.2.10 (Failure of Co-Completeness)**.** *The construction described in 'Soundness' followed by the one described in 'Co-Soundness' may not be equivalent to the identity: i.e. Co-Completeness can fail (as for the categories with families semantics for DTT). Its fixed-points (up to equivalence) are precisely the models for which the comprehension is democratic.*

Proof. Indeed, if we start with a strict indexed symmetric monoidal category with comprehension, construct the corresponding model $\widetilde{\mathbb{T}}$ and then construct its syntactic category, we effectively have thrown away all the non-trivial morphisms into objects that are not of the form $\Gamma.A$ or $\cdot$. The definition of a democratic comprehension is precisely that every object is of that form.

Of course, we can easily obtain a co-complete model theory by putting this extra restriction on our models. Alternatively – this may be nicer from a categorical point of view –, we can take the obvious (see e.g. [19]) conservative extension of our syntax by also talking about context morphisms (corresponding to morphisms in our base category). In that case, we would obtain an actual internal language for strict indexed symmetric monoidal categories with comprehension. This also has the advantage that we can easily obtain an internal language for strict indexed monoidal categories by dropping the axioms Cart-C-Ext, Cart-C-Ext-Eq, Cart-Idf and Cart-Weak, which correspond to the comprehension schema. We have not chosen this route as it would mean that the syntax would not fit as well with what has been considered so far in the syntactic tradition. □

Corollary 3.2.11 (Relation to DTT and ILTT)**.** *As we have seen, a model* $(\mathcal{B}, \mathcal{D}, \mathbf{p}, \mathbf{v})$ *of dDILL with* I*- and* $\otimes$*-types defines a model* $\mathcal{C}$ *of DTT, that should be thought of the cartesian content of the linear type theory. This will become even more clear through our treatment of* $!$*-types and in the examples we treat.*

Moreover, it clearly defines a model of ILTT with I*- and* $\otimes$*-types (i.e. a symmetric monoidal category) in every context.*

Conversely, it is easily seen that every model of DTT can be obtained this way (up to equivalence), by noting that it is in particular a model of dDILL and that

every model of ILTT can be embedded in a model of dDILL. (As we shall see in section 3.5.1, we can cofreely add type dependency on Set*.)*

Semantic Type Formers

Next, we discuss the interpretation of various type formers in models of dDILL.

Theorem 3.2.12 (Semantic type formers)**.** *For the other type formers, we have the following. A model of dDILL with I- and $\otimes$-types (a strict indexed symmetric monoidal category with comprehension)...*

1. *...supports $\Sigma_!^\otimes$-types iff all the change of base functors $\mathcal{D}(\mathbf{p}_{\Gamma,A})$ have left adjoints $\Sigma_{!A}^\otimes$ that satisfy the left Beck-Chevalley condition for $\mathbf{p}$-squares and that satisfy Frobenius reciprocity*[9] *in the sense that the canonical morphism*

$$\Sigma_{!A}^\otimes(\Delta'\{\mathbf{p}_{\Gamma,A}\} \otimes B) \longrightarrow \Delta' \otimes \Sigma_{!A}^\otimes B$$

   is an isomorphism , for all $\Delta' \in \mathcal{D}(\Gamma)$, $B \in \mathcal{D}(\Gamma.A)$.

2. *...supports $\Pi_!^{-\circ}$-types iff all the change of base functors $\mathcal{D}(\mathbf{p}_{\Gamma,A})$ have right adjoints $\Pi_{!A}^{-\circ}$ that satisfy the right Beck-Chevalley condition for $\mathbf{p}$-squares.*

3. *...supports $\multimap$-types iff $\mathcal{D}$ factors over the category* SMCCat *of symmetric monoidal categories and (strict) symmetric monoidal functors.*

4. *...supports $\top$-types and $\&$-types iff $\mathcal{D}$ factors over the category* SMcCat *of cartesian categories with a symmetric monoidal structure and their (strict) homomorphisms.*

5. *...supports 0-types and $\oplus$-types iff $\mathcal{D}$ factors over the category* dSMcCCat *of cocartesian categories with a distributive*[10] *symmetric monoidal structure and their (strict) homomorphisms.*

[9] Frobenius reciprocity expresses compatibility of $\Sigma_!^\otimes$ and $\otimes$, which is reasonable if we want a reading of $\Sigma_!^\otimes$ as a generalisation of $\otimes$. If one wants to drop Frobenius reciprocity in the semantics, it is easy to see that the equivalent in the syntax is setting $\Delta' = \cdot$ in the $\Sigma_!^\otimes$-E-rule. Therefore, Frobenius reciprocity automatically follows if we have $\multimap$-types.

[10] Note that in the light of theorem 3.1.6, the demand of distributivity here is essentially the same phenomenon as the demand of Frobenius reciprocity for $\Sigma_!^\otimes$-types.

6. *...that supports $\multimap$-types*[11]*, supports* !*-types iff all the comprehension functors* $\mathcal{D}(\Gamma) \xrightarrow{U_\Gamma} \mathcal{C}(\Gamma)$ *have a strong monoidal left adjoint* $\mathcal{C}(\Gamma) \xrightarrow{F_\Gamma} \mathcal{D}(\Gamma)$ *in the 2-category* SMCat *of symmetric monoidal categories, lax symmetric monoidal functors, and monoidal natural transformations*[12] *and (compatibility with substitution) for all* $\Gamma' \xrightarrow{f} \Gamma \in \mathcal{B}$ *we have that* $F_\Gamma; \mathcal{D}(f) = \mathcal{C}(f); F_{\Gamma'}$ *(which makes* F_- *into a morphism of indexed categories). Then the linear exponential comonad* $!_\Gamma := U_\Gamma; F_\Gamma : \mathcal{D}(\Gamma) \longrightarrow \mathcal{D}(\Gamma)$ *will be our interpretation of the comodality* ! *in the context* Γ*.*

7. *... supports* $\mathsf{Id}^{\otimes}_{!}$*-types iff for all* $A \in \mathsf{ob}\ \mathcal{D}(\Gamma)$*, we have left adjoints* $\mathsf{Id}^{\otimes}_{!A} \dashv -\{\mathsf{diag}_{\Gamma,A}\}$ *that satisfy the left Beck-Chevalley condition for* diag*-squares and Frobenius reciprocity in the sense that the canonical morphisms*

$$\mathsf{Id}^{\otimes}_{!A}(B) \longrightarrow \mathsf{Id}^{\otimes}_{!A}(I) \otimes B\{\mathbf{p}_{\Gamma.A,A\{\mathbf{p}_{\Gamma,A}\}}\}$$

are isomorphisms.

Proof. 1. Assume our model supports $\Sigma^{\otimes}_{!}$-types. We exhibit the claimed adjunction. The morphism from left to right is provided by $\Sigma^{\otimes}_{!}$-I. The morphism from right to left is provided by $\Sigma^{\otimes}_{!}$-E. $\Sigma^{\otimes}_{!}$-β and $\Sigma^{\otimes}_{!}$-η say exactly that these are mutually inverse. Naturality corresponds to the compatibility of $\Sigma^{\otimes}_{!}$-I and $\Sigma^{\otimes}_{!}$-E with substitution.

$$c' \longmapsto (!\mathbf{v}_{\Gamma,A,\cdot} \otimes \mathsf{id}_B); (c'\{\mathbf{p}_{\Gamma,A}\})$$

$$\mathcal{D}(\Gamma)(\Sigma^{\otimes}_{!A}B, C) \underset{\cong}{\rightleftarrows} \mathcal{D}(\Gamma.A)(B, C\{\mathbf{p}_{\Gamma,A}\})$$

$$\mathsf{let}\ z\ \mathsf{be}\ !x \otimes y\ \mathsf{in}\ c \longmapsfrom c$$

[11] Actually, we only need this for the 'if'. The 'only if' always holds. To make the 'if' work, as well, in absence of $\multimap$-types, we have to restrict !-E to the case where $\Delta' = \cdot$. Alternatively, we could note that the semantic condition that precisely corresponds to having !-types (even in absence of $\multimap$-types) is to have a natural isomorphism $\mathcal{D}(\Gamma.A)(\Delta\{\mathbf{p}_{\Gamma,A}\}, B\{\mathbf{p}_{\Gamma,A}\}) \cong \mathcal{D}(\Gamma)(!A \otimes \Delta, B)$ (which we immediately recognise as a specific case of $\Sigma^{\otimes}_{!}$-types).

[12] i.e. a symmetric lax monoidal left adjoint functor F_Γ such that an inverse for its lax structure is given by the oplax structure on F_Γ coming from the lax structure on U_Γ. Put differently, F_Γ is a left adjoint functor to U_Γ and is a strong monoidal functor in a way that is compatible with the lax structure on U_Γ.

We show how the morphism from left to right arises from $\Sigma^{\otimes}_{!}$-I.

$$\dfrac{\dfrac{\dfrac{}{\Gamma, x:A; \cdot \vdash x : A}\,\textsf{Cart-Idf} \quad \dfrac{}{\Gamma, x:A; w:B \vdash w : B}\,\textsf{Lin-Idf}}{\Gamma, x:A; w:B \vdash\, !x \otimes w : \Sigma^{\otimes}_{!(x:A)} B}\,\Sigma^{\otimes}_{!}\textsf{-I} \quad \dfrac{\Gamma; z : \Sigma^{\otimes}_{!(x:A)} B \vdash c' : C}{\Gamma, x:A; z : \Sigma^{\otimes}_{!(x:A)} B \vdash c' : C}\,\textsf{Cart-Weak}}{\Gamma, x:A; w:B \vdash c'[!x \otimes w/z] : C}\,\textsf{Lin-Tm-Subst}$$

We show how the morphism from right to left is exactly $\Sigma^{\otimes}_{!}$-E (with $\Delta' = \cdot$, $\Delta = z : \Sigma^{\otimes}_{!(x:A)} B$, $t = z$).

$$\dfrac{\Gamma; \cdot \vdash C\ \ \textsf{type} \qquad \dfrac{}{\Gamma; z : \Sigma^{\otimes}_{!(x:A)} B \vdash z : \Sigma^{\otimes}_{!(x:A)} B}\,\textsf{Lin-Idf} \qquad \Gamma, x:A; y:B \vdash c : C}{\Gamma; z : \Sigma^{\otimes}_{!(x:A)} B \vdash \textsf{let}\ z\ \textsf{be}\ !x \otimes y\ \textsf{in}\ c : C}\,\Sigma^{\otimes}_{!}\textsf{-E}$$

We show how Frobenius reciprocity can be proved in our type system (particularly relying on the form of the $\Sigma^{\otimes}_{!}$-E-rule[13]).

Claim (Frobenius reciprocity)**.** *The canonical morphism*

$$\Sigma^{\otimes}_{!A}(\Delta'\{\mathbf{p}_{\Gamma,A}\} \otimes B) \xrightarrow{f} \Delta' \otimes \Sigma^{\otimes}_{!A} B$$

is an isomorphism, for all $\Delta' \in \mathcal{D}(\Gamma)$, $B \in \mathcal{D}(\Gamma.A)$.

Proof. We first show how to construct the morphism f we mean.

$$\dfrac{\dfrac{}{\Gamma; x' : \Sigma^{\otimes}_{!(x:A)}(\Delta' \otimes B) \vdash x' : \Sigma^{\otimes}_{!(x:A)}(\Delta' \otimes B)}\,\textsf{Lin-Idf} \qquad \dfrac{\dfrac{}{\Gamma, x:A; z:\Delta' \vdash z : \Delta'}\,\textsf{Lin-Idf} \quad \dfrac{\dfrac{}{\Gamma, x:A; \cdot \vdash x : A}\,\textsf{Cart-Idf} \quad \dfrac{}{\Gamma; y : B \vdash y : B}\,\textsf{Lin-Idf}}{\Gamma, x:A; y:B \vdash\, !x \otimes y : \Sigma^{\otimes}_{!(x:A)} B}\,\Sigma^{\otimes}_{!}\textsf{-I}}{\Gamma, x:A; z:\Delta', y:B \vdash z \otimes\, !x \otimes y : \Delta' \otimes \Sigma^{\otimes}_{!(x:A)} B}\,\otimes\textsf{-I}}{\Gamma, x:A; w:\Delta' \otimes B \vdash \textsf{let}\ w\ \textsf{be}\ z \otimes y\ \textsf{in}\ z \otimes\, !x \otimes y : \Delta' \otimes \Sigma^{\otimes}_{!(x:A)} B}\,\otimes\textsf{-E}}{\Gamma; x' : \Sigma^{\otimes}_{!(x:A)}(\Delta' \otimes B) \vdash f : \Delta' \otimes \Sigma^{\otimes}_{!(x:A)} B}\,\Sigma^{\otimes}_{!}\textsf{-E}$$

We now construct its inverse. Call it g[14].

$$\dfrac{\dfrac{}{\Gamma; y_2 : \Sigma^{\otimes}_{!(x:A)} B \vdash y_2 : \Sigma^{\otimes}_{!(x:A)} B}\,\textsf{Lin-Idf} \qquad \dfrac{\dfrac{}{\Gamma, x:A; \cdot \vdash x : A}\,\textsf{Cart-Idf} \quad \dfrac{\dfrac{}{\Gamma; y_1 : \Delta' \vdash y_1 : \Delta'}\,\textsf{Lin-Idf} \quad \dfrac{}{\Gamma; y : B \vdash y : B}\,\textsf{Lin-Idf}}{\Gamma, x:A; y_1 : \Delta', y : B \vdash y_1 \otimes y : \Delta' \otimes B}\,\otimes\textsf{-I}}{\Gamma, x:A; y_1 : \Delta', y:B \vdash\, !x \otimes y_1 \otimes y : \Sigma^{\otimes}_{!(x:A)}(\Delta' \otimes B)}\,\Sigma^{\otimes}_{!}\textsf{-I}}{\Gamma; y_1 : \Delta', y_2 : \Sigma^{\otimes}_{!(x:A)} B \vdash \textsf{let}\ y_2\ \textsf{be}\ !x \otimes y\ \textsf{in}\ !x \otimes y_1 \otimes y : \Sigma^{\otimes}_{!(x:A)}(\Delta' \otimes B)}\,\Sigma^{\otimes}_{!}\textsf{-E}}{\Gamma; y' : \Delta' \otimes \Sigma^{\otimes}_{!(x:A)} B \vdash g : \Sigma^{\otimes}_{!(x:A)}(\Delta' \otimes B)}\,\otimes\textsf{-E}$$

[13]To be precise, we shall see Frobenius reciprocity is validated because we allow dependency on Δ' in the $\Sigma^{\otimes}_{!}$-E-rule. Conversely, it is easy to see we can prove Frobenius reciprocity holds in our model if we have (semantic) $\multimap$-types, as this allows us to remove the dependency on Δ' in $\Sigma^{\otimes}_{!}$-E.

[14]Frobenius reciprocity really comes in where $\Sigma^{\otimes}_{!}$-E is used, because of the factor Δ' in the $\Sigma^{\otimes}_{!}$-E-rule.

We leave it to the reader to verify that these morphisms are mutually inverse in the sense that

$$\Gamma; x' : \Sigma^{\otimes}_{!(x:A)}(\Delta' \otimes B) \vdash g[f/y'] = x' : \Sigma^{\otimes}_{!(x:A)}(\Delta' \otimes B)$$

and

$$\Gamma; y' : \Delta' \otimes \Sigma^{\otimes}_{!(x:A)} B \vdash f[g/x'] = y' : \Delta' \otimes \Sigma^{\otimes}_{!(x:A)} B.$$

□

For the converse, we show how to obtain $\Sigma^{\otimes}_{!}$-I from our morphism from left to right:

$$\dfrac{\dfrac{\dfrac{}{\Gamma; z : \Sigma^{\otimes}_{!(x:A)}B \vdash z : \Sigma^{\otimes}_{!(x:A)}B}\,\textsf{Lin-Idf}}{\Gamma, x : A; w : B \vdash !x \otimes w : \Sigma^{\otimes}_{!(x:A)}B}\,\text{“left to right”} \qquad \Gamma; \cdot \vdash a : A}{\Gamma; w : B \vdash !a \otimes w : \Sigma^{\otimes}_{!(x:A)}B}\,\textsf{Cart-Tm-Subst} \qquad \Gamma; \Delta \vdash b : B[a/x]}{\Gamma; \Delta \vdash !a \otimes b : \Sigma^{\otimes}_{!(x:A)}B}\,\textsf{Lin-Tm-Subst}$$

We show how to obtain $\Sigma^{\otimes}_{!}$-E from our morphism from right to left, using Frobenius reciprocity.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma; \cdot \vdash C\ \textsf{type} \qquad \dfrac{\Gamma, x : A; y : \Delta', B \vdash c : C}{\Gamma, x : A; y : \Delta' \otimes B \vdash c : C}\,\otimes\textsf{-E}}{\Gamma; z : \Sigma^{\otimes}_{!(x:A)}(\Delta' \otimes B) \vdash \textsf{let } z \textsf{ be } !x \otimes y \textsf{ in } c : C}\,\text{“right to left”}}{\Gamma; z : (\Delta' \otimes \Sigma^{\otimes}_{!(x:A)}B) \vdash \textsf{let frob}(z) \textsf{ be } !x \otimes y \textsf{ in } c : C}\,\textsf{Frobenius reciprocity}}{\Gamma; z_1 : \Delta', z_2 : \Sigma^{\otimes}_{!(x:A)}B \vdash \textsf{let frob}(z_1 \otimes z_2) \textsf{ be } !x \otimes y \textsf{ in } c : C}\,\textsf{Lin-Tm-Subst,}\otimes\textsf{-I,2}\times\textsf{Lin-Idf} \qquad \Gamma; \Delta \vdash t : \Sigma^{\otimes}_{!(x:A)}B}{\Gamma; z_1 : \Delta', \Delta \vdash (\textsf{let frob}(z_1 \otimes z_2) \textsf{ be } !x \otimes y \textsf{ in } c)[t/z_2] : C}\,\textsf{Lin-Tm-Subst}$$

As usual, the left Beck-Chevalley condition says precisely that $\Sigma^{\otimes}_{!}$-types commute with substitution, as dictated by the type theory.

2. Assume our model supports $\Pi^{\multimap}_{!}$-types. We exhibit the claimed adjunction. The morphism from left to right is provided by $\Pi^{\multimap}_{!}$-I – in fact, it is exactly the I-rule – and the one from right to left by $\Pi^{\multimap}_{!}$-E. $\Pi^{\multimap}_{!}$-β and $\Pi^{\multimap}_{!}$-η say exactly that these are mutually inverse. Naturality corresponds to the compatibility of $\Pi^{\multimap}_{!}$-I and $\Pi^{\multimap}_{!}$-E with substitution.

$$b \longmapsto \lambda_{!(x:A)} b$$

$$\mathcal{D}(\Gamma.A)(\Delta\{\mathbf{p}_{\Gamma.A}\}, B) \underset{\longleftarrow}{\overrightarrow{\cong}} \mathcal{D}(\Gamma)(\Delta, \Pi^{\multimap}_{!(x:A)}B)$$

$$f(!x) \longmapsfrom f.$$

We show how we obtain the definition of $f(!x)$ from $\Pi^{\multimap}_{!}$-E.

$$\dfrac{\dfrac{}{\Gamma, x : A; \cdot \vdash x : A}\,\textsf{Cart-Idf} \quad \dfrac{\Gamma; \Delta \vdash f : \Pi^{\multimap}_{!(x:A)} B}{\Gamma, x : A; \Delta \vdash f : (\Pi^{\multimap}_{!(x:A)} B)}\,\textsf{Cart-Weak}}{\Gamma, x : A; \Delta \vdash f(!x) : B}\,\Pi^{\multimap}_{!}\textsf{-E}$$

For the converse, we have to show that we can recover $\Pi^{\multimap}_{!}$-E from the definition of $f(!x)$.

$$\dfrac{\Gamma; \cdot \vdash a : A \quad \dfrac{\Gamma; \Delta \vdash f : \Pi^{\multimap}_{!(x:A)} B}{\Gamma, x : A; \Delta \vdash f(!x) : B}\,\textsf{Definition } f(!x)}{\dfrac{\Gamma; \Delta \vdash f(!x)[a/x] : B[a/x]}{\Gamma; \Delta \vdash f(!a) : B[a/x]}}\,\textsf{Cart-Tm-Subst}$$

This shows that individual $\Pi^{\multimap}_{!}$-types correspond to right adjoint functors to substitution along projections. The type theory dictates that $\Pi^{\multimap}_{!}$-types interact well with substitution. This corresponds to the right Beck-Chevalley condition, as usual.

3. From the categorical semantics of (non-dependent) linear type theory (see e.g. [47] for a very complete account) we know that $\multimap$-types correspond to monoidal closure of the category of contexts. The extra feature in dependent linear type theory is that the syntax dictates that the type formers are compatible with substitution. This means that we also have to restrict the functors $\mathcal{D}(f)$ to preserve the relevant categorical structure.

4. Idem.

5. Idem.

6. Assume that we have !-types. We define a left adjoint $F_\Gamma \dashv U_\Gamma$ as $F_\Gamma \mathbf{p}_{\Gamma,A} :=\!A$ (this is easily seen to be well-defined up to isomorphism, so we can use AC for a definition on the nose) and, noting that every morphism $\mathbf{p}_{\Gamma,A} \longrightarrow \mathbf{p}_{\Gamma,B}$ in $\mathcal{B}/\Gamma$ is of the form $\langle \mathbf{p}_{\Gamma,A}, b\rangle$ for some unique $I \xrightarrow{b} B\{\mathbf{p}_{\Gamma,A}\} \in \mathcal{D}(\Gamma.A)$, we define F_Γ as acting on b as the map obtained from

$$\dfrac{\dfrac{\dfrac{\Gamma, x : A; \cdot \vdash b : B}{\Gamma, x : A; \cdot \vdash !b :!B}\,!\textsf{-I} \quad \dfrac{}{\Gamma; y :!A \vdash y :!A}\,\textsf{Lin-Idf}}{\Gamma; y :!A \vdash \textsf{let } y \textsf{ be } !x \textsf{ in } !b :!B}\,!\textsf{-E}$$

which indeed gives us $F_\Gamma(\langle \mathbf{p}_{\Gamma,A}, b\rangle) \in \mathcal{D}(\Gamma)(!A, !B)$.

We exhibit the adjunction by the following isomorphism of hom-sets, where the morphism from left to right comes from !-I and the one from right to left comes from !-E.

$$b \longmapsto b[!x/x']$$

$$\mathcal{D}(\Gamma)(F_\Gamma \mathbf{p}_{\Gamma,A}, B) = \mathcal{D}(\Gamma)(!A, B) \underset{\cong}{\overrightarrow{\leftarrow}} \mathcal{D}(\Gamma.A)(I, B\{\mathbf{p}_{\Gamma,A}\}) \cong \mathcal{B}/\Gamma(\mathbf{p}_{\Gamma,A}, \mathbf{p}_{\Gamma,B}) = \mathcal{C}(\Gamma)(\mathbf{p}_{\Gamma,A}, U_\Gamma B)$$

$$\mathsf{let}\ y\ \mathsf{be}\ !x\ \mathsf{in}\ b' \longmapsto b'$$

We show how to construct the morphism from left to right, using !-I.

$$\dfrac{\dfrac{\Gamma; x' :!A \vdash b : B}{\Gamma, x : A; x' :!A \vdash b : B}\,\mathsf{Cart\text{-}Weak} \qquad \dfrac{\dfrac{}{\Gamma, x : A; \cdot \vdash x : A}\,\mathsf{Cart\text{-}Idf}}{\Gamma, x : A; \cdot \vdash !x :!A}\,!\text{-I}}{\Gamma, x : A; \cdot \vdash b[!x/x'] : B}\,\mathsf{Lin\text{-}Tm\text{-}Subst}$$

We show to construct the morphism from right to left, using !-E. Suppose we're given $b' \in \mathcal{D}(\Gamma.A)(I, B\{\mathbf{p}_{\Gamma,A}\})$. From this, we produce a morphism in $\mathcal{D}(\Gamma)(!A, B)$ as follows.

$$\dfrac{\dfrac{}{\Gamma; y :!A \vdash y :!A}\,\mathsf{Lin\text{-}Idf} \qquad \Gamma, x : A; \cdot \vdash b' : B}{\Gamma; y :!A \vdash \mathsf{let}\ y\ \mathsf{be}\ !x\ \mathsf{in}\ b' : B}\,!\mathsf{E}$$

We leave it up to the reader to verify that these morphisms are mutually inverse, according to !-β and !-η.

Note that F_Γ is strong monoidal, as the rules for ! define a natural bijection between terms $\Gamma; x' :!A, y' :!B \vdash t' : C$ and $\Gamma, x : A, y : B; \cdot \vdash t : C$ if $\Gamma \vdash C\ \mathsf{type}$. In semantic terms, this gives a natural bijection

$$\begin{aligned}
\mathcal{D}(\Gamma)(!A \otimes !B, C) &\cong \mathcal{D}(\Gamma.A.B)(1, C\{\mathbf{p}_{\Gamma.A,B}; \mathbf{p}_{\Gamma,A}\}) \\
&\cong \mathcal{B}/\Gamma(\mathbf{p}_{\Gamma.A,B}; \mathbf{p}_{\Gamma,A}, \mathbf{p}_{\Gamma,C}) \\
&= \mathcal{B}/\Gamma(U_\Gamma A \times U_\Gamma B, U_\Gamma C) \\
&\cong \mathcal{D}(\Gamma)(F_\Gamma(U_\Gamma A \times U_\Gamma B), C),
\end{aligned}$$

so strong monoidality follows by the Yoneda lemma. (A keen reader can verify that the oplax structure on F_Γ corresponds with the lax structure on U_Γ.)

Conversely, suppose we have a strong monoidal left adjoint $F_\Gamma \dashv U_\Gamma$. We define, for $A \in \mathsf{ob}(\mathcal{D}(\Gamma))$, $!A := F_\Gamma U_\Gamma(A)$.

We verify that !-I can be derived from the homset morphism from left to right:

$$\dfrac{\dfrac{\dfrac{}{\Gamma; x' :!A \vdash x' :!A}\,\textsf{Lin-Idf}}{\Gamma, x : A; \cdot \vdash !x :!A}\,\text{“left to right”} \qquad \Gamma; \cdot \vdash a : A}{\Gamma; \cdot \vdash !x[a/x] :!A}\,\textsf{Cart-Tm-Subst}$$

We verify that, in the presence of $\multimap$-types, !-E can be derived from the homset morphism from right to left:

$$\dfrac{\Gamma; \Delta \vdash t :!A \qquad \dfrac{\dfrac{}{\Gamma; w : \Delta' \vdash w : \Delta'}\,\textsf{Lin-Idf} \qquad \dfrac{\dfrac{\Gamma, x : A; y : \Delta' \vdash b : B}{\Gamma, x : A; \cdot \vdash \lambda_{y:\Delta'} b : \Delta' \multimap B}\,\multimap\textsf{-I}}{\Gamma; z :!A \vdash \mathsf{let}\ z\ \mathsf{be}\ !x\ \mathsf{in}\ \lambda_{y:\Delta'} b : \Delta' \multimap B}\,\text{“right to left”}}{\Gamma; z :!A, \Delta' \vdash \mathsf{let}\ z\ \mathsf{be}\ !x\ \mathsf{in}\ b[w/y] : B}\,\multimap\textsf{-E}}{\Gamma; \Delta, \Delta' \vdash \mathsf{let}\ t\ \mathsf{be}\ !x\ \mathsf{in}\ b[w/y] : B}\,\textsf{Lin-Tm-Subst}$$

Note that the !-β- and !-η-rules correspond precisely to the fact that our morphisms from left to right and from right to left define a homset isomorphism.

Finally, it is easily verified that the condition that $F_\Gamma; \mathcal{D}(f) \cong \mathcal{D}(f); F_{\Gamma'}$ corresponds exactly to the compatibility of ! with substitution.

7. Suppose we have $\mathsf{Id}^{\otimes}_{!A} \dashv -\{\mathsf{diag}_{\Gamma,A}\}$ (satisfying the appropriate Frobenius and Beck-Chevalley conditions). Then, we have a (natural) homset isomorphism

$$\mathcal{D}(\Gamma.A.A\{\mathbf{p}_{\Gamma,A}\})(\mathsf{Id}^{\otimes}_{!A}(B), C) \underset{\longleftarrow}{\overrightarrow{\cong}} \mathcal{D}(\Gamma.A)(B, C\{\mathsf{diag}_{\Gamma,A}\}).$$

The claim is that $\mathsf{Id}^{\otimes}_{!A}(I)$ satisfies the rules for the $\mathsf{Id}^{\otimes}_{!}$-type of A. Indeed, we have $\mathsf{Id}^{\otimes}_{!}$-I as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x : A, x' : A; w : \mathsf{Id}^{\otimes}_{!A}(I)(x,x') \vdash w : \mathsf{Id}^{\otimes}_{!A}(I)(x,x')}\,\textsf{Lin-Idf}}{\Gamma, x : A; y : I \vdash \mathsf{refl}(!x)^y : \mathsf{Id}^{\otimes}_{!A}(I)(x,x)}\,\text{“left to right”} \qquad \dfrac{}{\Gamma, x : A; \cdot \vdash * : I}\,I\textsf{-I}}{\Gamma, x : A; \cdot \vdash \mathsf{refl}(!x) : \mathsf{Id}^{\otimes}_{!A}(I)(x,x)}\,\textsf{Lin-Tm-Subst} \qquad \Gamma; \cdot \vdash a : A}{\Gamma; \cdot \vdash \mathsf{refl}(!x) : \mathsf{Id}^{\otimes}_{!A}(I)(a,a)}\,\textsf{Cart-Tm-Subst}$$

We obtain $\mathsf{Id}^{\otimes}_{!}$-E as follows. Let $\Gamma, x : A, x' : A; \cdot \vdash C \ \mathsf{type}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, x : A; B \vdash c : C[x/x']}{\Gamma, x : A, x' : A; \mathsf{Id}^{\otimes}_{!A}(B) \vdash c' : C}\ \text{“right to left”} \qquad \Gamma; \cdot \vdash a : A \qquad \Gamma; \cdot \vdash a' : A}{\Gamma; \mathsf{Id}^{\otimes}_{!A}(B)[a/x, a'/x'] \vdash c'[a/x, a'/x'] : C[a/x, a'/x']}\ \textsf{Cart-Tm-Subst} \qquad \textsf{Frobenius}\ \dfrac{\Gamma; B' \vdash p : \mathsf{Id}^{\otimes}_{!A}(I)[a/x, a'/x']}{\Gamma; B[a/x], B' \vdash p' : \mathsf{Id}^{\otimes}_{!A}(B)[a/x, a'/x']}}{\Gamma; B[a/x], B' \vdash \mathsf{let}\ (a, a', p)\ \mathsf{be}\ (z, z, \mathsf{refl}(!z))\ \mathsf{in}\ c : C[a/x, a'/x]}\ \textsf{Lin-Tm-Subst}$$

Conversely, suppose we have $\mathsf{Id}^{\otimes}_{!}$-types. Then, define $\mathsf{Id}^{\otimes}_{!A}(B) := \mathsf{Id}^{\otimes}_{!A} \otimes B\{\mathbf{p}_{\Gamma.A, A\{\mathbf{p}_{\Gamma,A}\}}\}$, with the obvious extension on morphisms. (This immediately implies Frobenius reciprocity, clearly.) Then, we obtain the morphism “left to right” as follows.

$$\dfrac{\dfrac{\Gamma, x : A, x' : A; z : \mathsf{Id}^{\otimes}_{!A}, y : B \vdash c : C \qquad \dfrac{}{\Gamma, x : A; \cdot \vdash x : A}\ \textsf{Cart-Idf}}{\Gamma, x : A; z : \mathsf{Id}^{\otimes}_{!A}[x/x'], y : B \vdash c[x/x'] : C[x/x']}\ \textsf{Cart-Tm-Subst} \qquad \dfrac{\dfrac{}{\Gamma, x : A; \cdot \vdash x : A}\ \textsf{Cart-Idf}}{\Gamma, x : A; \cdot \vdash \mathsf{refl}(!x) : \mathsf{Id}^{\otimes}_{!A}(x, x)}\ \mathsf{Id}^{\otimes}_{!}\text{-I}}{\Gamma, x : A; y : B \vdash c' : C[x/x']}\ \textsf{Lin-Tm-Subst}$$

The morphism “right to left” is obtained as follows.

$$\dfrac{\Gamma, x_0 : A; y : B \vdash c : C[x_0/x_1] \qquad \dfrac{}{\Gamma, x_0 : A, x_1 : A; w : \mathsf{Id}^{\otimes}_{!A} \vdash w : \mathsf{Id}^{\otimes}_{!A}}\ \textsf{Lin-Idf} \qquad \dfrac{}{\Gamma, x_0 : A, x_1 : A; \cdot \vdash x_i : A}\ \textsf{Cart-Idf}}{\Gamma, x_0 : A, x_1 : A; w : \mathsf{Id}^{\otimes}_{!A}, y : B \vdash c' : C}\ \mathsf{Id}^{\otimes}_{!}\text{-E}$$

We leave it to the reader to verify that the $\mathsf{Id}^{\otimes}_{!}$-$\beta$- and $\mathsf{Id}^{\otimes}_{!}$-$\eta$-rules translate precisely into the “right to left” and “left to right” morphisms being inverse. As usual, the Beck-Chevalley condition corresponds to the compatibility of $\mathsf{Id}^{\otimes}_{!}$-types with substitution, while the Frobenius condition says that $\mathsf{Id}^{\otimes}_{!A}$-functors are entirely determined by the object $\mathsf{Id}^{\otimes}_{!A}(I)$.

□

The semantics of ! suggests an alternative definition for the notion of a comprehension: if we have $\Sigma^{\otimes}_{!}$-types in a strong sense, it is a derived notion!

Theorem 3.2.13 (Lawvere Comprehension)**.** *Given a strict indexed monoidal category $(\mathcal{B}, \mathcal{D})$ with left adjoints $\Sigma^{\otimes}_{F(f)}$ to $\mathcal{D}(f)$ for arbitrary $\Gamma' \xrightarrow{f} \Gamma \in \mathcal{B}$, satisfying*

the left Beck-Chevalley condition for all pullback squares, then we can define $\mathcal{B}/\Gamma \xrightarrow{F_\Gamma} \mathcal{D}(\Gamma)$ *by*

$$F_\Gamma(-) := \Sigma^\otimes_{F(-)} I.$$

In that case, $(\mathcal{B}, \mathcal{D})$ *has a comprehension schema iff* F_Γ *has a right adjoint* U_Γ *(which then automatically satisfies* $\mathcal{D}(f); U_{\Gamma'} = U_\Gamma; \mathcal{D}(f)$ *for all* $\Gamma' \xrightarrow{f} \Gamma \in \mathcal{B}$*). That is, our notion of comprehension generalises that of [24].*

In particular, if either condition is satisfied, it supports !*-types iff* $\Sigma^\otimes_!$ *satisfies Frobenius reciprocity.*

Proof. Suppose that we have said right adjoints U_Γ. We construct a comprehension schema.

This allows us to define $\mathbf{p}_{\Gamma,A} := U_\Gamma(A)$ and note that we have natural isomorphisms

$$\mathcal{D}(\Gamma')(I, A\{f\}) \xrightarrow{\cong} \mathcal{D}(\Gamma)(\Sigma^\otimes_{F(f)} I_{\Gamma'}, A) = \mathcal{D}(\Gamma)(F_\Gamma f, A) \xrightarrow{\cong} \mathcal{B}/\Gamma(f, U_\Gamma A)$$

$$a \longmapsto a_f \longmapsto \langle f, a \rangle,$$

where the first natural isomorphism comes from the adjunction $\Sigma^\otimes_{F(f)} \dashv -\{f\}$ and the second one comes from the adjunction $F_\Gamma \dashv U_\Gamma$. This defines a comprehension for $\mathcal{D}$.

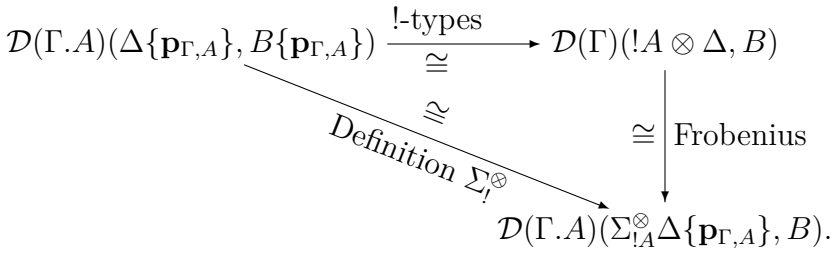
Conversely, suppose $\mathcal{D}$ satisfies the comprehension schema. Then, we know, by theorem 3.2.4, that we can define a comprehension functor U_Γ such that $\mathcal{D}(f); U_{\Gamma'} = U_\Gamma; \mathcal{D}(f)$. Then we have the following natural isomorphisms:

$$\mathcal{B}/\Gamma(f, U_\Gamma A) \xrightarrow{\cong} \mathcal{D}(\Gamma')(I, A\{f\}) \xrightarrow{\cong} \mathcal{D}(\Gamma)(\Sigma^\otimes_{F(f)} I_{\Gamma'}, A) = \mathcal{D}(\Gamma)(F_\Gamma f, A)$$

$$\langle f, a \rangle \longmapsto a \longmapsto a_f,$$

where the first isomorphism is precisely the representation defined by our comprehension and the second isomorphism comes from the fact that $\Sigma^\otimes_{F(f)} \dashv -\{f\}$. We see that $F_\Gamma \vdash U_\Gamma$.

Finally, note that we have the following commutative triangle of natural isomorphisms



Note that the Beck-Chevalley condition for $\Sigma_F^\otimes$ takes care of the substitution condition for !-types. Therefore, the existence of !-types boils down to the top isomorphism. Meanwhile, the Frobenius condition is by the Yoneda lemma equivalent to the right isomorphism. Noting that the diagonal always holds if we have $\Sigma_!^\otimes$-types, it follows that we have !-types iff we have Frobenius reciprocity. □

Theorem 3.2.14 (Type Formers in $\mathcal{C}$)**.** *$\mathcal{C}$ supports Σ-types iff $\mathsf{ob}(\mathcal{C})$ is closed under compositions (as morphisms in $\mathcal{B}$). It supports Id-types iff $\mathsf{ob}(\mathcal{C})$ is closed under postcomposition with maps $\mathsf{diag}_{\Gamma,A}$. If $\mathcal{D}$ supports !- and $\Pi_!^{-\circ}$-types, then $\mathcal{C}$ supports Π-types. Moreover, we have that*

$$\Sigma_{!A}^\otimes !B \cong F(\Sigma_{UA} UB) \qquad \mathsf{Id}_{!A}^\otimes(!B) \cong F\mathsf{Id}_{UA}(UB) \qquad U\Pi_{!B}^{-\circ} C \cong \Pi_{UB} UC.$$

Proof. We write out the adjointness condition

$$\begin{aligned}
\mathcal{C}(\Gamma)(\Sigma_{\mathbf{p}_{\Gamma,B}} f, \mathbf{p}_{\Gamma,D}) &\overset{!}{\cong} \mathcal{C}(\Gamma.B)(f, \mathbf{p}_{\Gamma,D}\{\mathbf{p}_{\Gamma,B}\}) \\
&\cong \mathcal{C}(\Gamma.B)(f, \mathbf{p}_{\Gamma,D\{\mathbf{p}_{\Gamma,B}\}}) \\
&\cong \mathcal{D}(\Gamma.B.C)(I, D\{\mathbf{p}_{\Gamma,B}\}\{f\}) \\
&\cong \mathcal{D}(\Gamma.B.C)(I, D\{f; \mathbf{p}_{\Gamma,B}\}) \\
&\cong \mathcal{C}(\Gamma)(f; \mathbf{p}_{\Gamma,B}, \mathbf{p}_{\Gamma,D}).
\end{aligned}$$

Now, the Yoneda lemma gives us that $\Sigma_{\mathbf{p}_{\Gamma,B}} f = f; \mathbf{p}_{\Gamma,B}$.

Similarly,

$$\begin{aligned}\mathcal{C}(\Gamma.A.A)(\mathsf{Id}_{\mathbf{p}_{\Gamma,A}}(f), \mathbf{p}_{\Gamma.A.A,C}) &\overset{!}{\cong} \mathcal{C}(\Gamma.A)(f, \mathbf{p}_{\Gamma.A.A,C}\{\mathsf{diag}_{\Gamma,A}\})\\ &\cong \mathcal{D}(\Gamma.A.B)(I, C\{\mathsf{diag}_{\Gamma,A}\}\{f\})\\ &\cong \mathcal{D}(\Gamma.A.B)(I, C\{f; \mathsf{diag}_{\Gamma,A}\})\\ &\cong \mathcal{C}(\Gamma.A.A)(f; \mathsf{diag}_{\Gamma,A}, \mathbf{p}_{\Gamma.A.A,C}),\end{aligned}$$

so $f; \mathsf{diag}_{\Gamma,A}$ models $\mathsf{Id}_{\mathbf{p}_{\Gamma,A}}(f)$.

Finally,

$$\begin{aligned}\mathcal{C}(\Gamma)(U_\Gamma D, \Pi_{\mathbf{p}_{\Gamma,B}}\mathbf{p}_{\Gamma.B,C}) &\overset{!}{\cong} \mathcal{C}(\Gamma.B)((U_\Gamma D)\{\mathbf{p}_{\Gamma,B}\}, \mathbf{p}_{\Gamma.B,C})\\ &\cong \mathcal{C}(\Gamma.B)((U_\Gamma D)\{\mathbf{p}_{\Gamma,B}\}, U_{\Gamma.B}C)\\ &\cong \mathcal{D}(\Gamma.B)(F_{\Gamma.B}((U_\Gamma D)\{\mathbf{p}_{\Gamma,B}\}), C)\\ &\cong \mathcal{D}(\Gamma.B)((F_\Gamma U_\Gamma D)\{\mathbf{p}_{\Gamma,B}\}, C)\\ &\cong \mathcal{D}(\Gamma)(F_\Gamma U_\Gamma D, \Pi^{-\circ}_{!B}C)\\ &\cong \mathcal{C}(\Gamma)(U_\Gamma D, U_\Gamma \Pi^{-\circ}_{!B}C).\end{aligned}$$

Again, using the Yoneda lemma, we conclude that $U_\Gamma \Pi^{-\circ}_{!B}C$ models $\Pi_{U_\Gamma B}U_{\Gamma.B}C$.

In all cases, we have not worried about Beck-Chevalley (and Frobenius reciprocity for $\Sigma^{\otimes}_{!}$-types) as they are trivially seen to hold.

Note that if $\mathcal{D}$ has $\Sigma^{\otimes}_{!}$-types (and, therefore, !-types), then

$$\begin{aligned}\mathcal{D}(\Gamma)(F_\Gamma(\Sigma_{U_\Gamma A}U_{\Gamma.A}B), C) &\cong \mathcal{C}(\Gamma)(\Sigma_{U_\Gamma A}U_{\Gamma.A}B, U_\Gamma C)\\ &\cong \mathcal{C}(\Gamma.A)(U_{\Gamma.A}B, (U_\Gamma C)\{\mathbf{p}_{\Gamma,A}\})\\ &\cong \mathcal{C}(\Gamma.A)(U_{\Gamma.A}B, U_{\Gamma.A}(C\{\mathbf{p}_{\Gamma,A}\}))\\ &\cong \mathcal{D}(\Gamma.A)(!B, C\{\mathbf{p}_{\Gamma,A}\})\\ &\cong \mathcal{D}(\Gamma)(\Sigma^{\otimes}_{!A}!B, C).\end{aligned}$$

By the Yoneda lemma, conclude that $\Sigma^{\otimes}_{!A}!B \cong F_\Gamma(\Sigma_{U_\Gamma A} U_{\Gamma.A} B)$.

Note that, in case $\mathcal{D}$ admits !- and $\mathsf{Id}^{\otimes}_{!}$-types,

$$\begin{aligned}
\mathcal{D}(\Gamma.A.A)(\mathsf{Id}^{\otimes}_{!A}(!B), C) &\cong \mathcal{D}(\Gamma.A)(!B, C\{\mathsf{diag}_{\Gamma,A}\}) \\
&\cong \mathcal{C}(\Gamma.A)(U_{\Gamma.A}B, U_{\Gamma.A}(C\{\mathsf{diag}_{\Gamma,A}\})) \\
&\cong \mathcal{C}(\Gamma.A)(U_{\Gamma.A}B, U_{\Gamma.A.A}(C)\{\mathsf{diag}_{\Gamma,A}\}) \\
&\cong \mathcal{C}(\Gamma.A.A)((U_{\Gamma.A}B); \mathsf{diag}_{\Gamma,A}, U_{\Gamma.A}(C)) \\
&\cong \mathcal{C}(\Gamma.A.A)(\mathsf{Id}_{U_\Gamma A}(U_{\Gamma.A}B), U_{\Gamma.A}(C)) \\
&\cong \mathcal{D}(\Gamma.A.A)(F_{\Gamma.A.A}\mathsf{Id}_{U_\Gamma A}(U_{\Gamma.A}B), C).
\end{aligned}$$

We conclude that $\mathsf{Id}^{\otimes}_{!A}(!B) \cong F_{\Gamma.A.A}\mathsf{Id}_{U_\Gamma A}(U_{\Gamma.A}B)$ and in particular $\mathsf{Id}^{\otimes}_{!A}(I) \cong F_{\Gamma.A.A}\mathsf{Id}_{U_\Gamma A}(\mathsf{id}_{\Gamma.A})$. (The last statement is easily seen to also be valid in absence of $\top$-types.) □

Remark 3.2.15 (Dependent Seely Isomorphisms?)**.** *Note that, in our setup, we have a version of the simply typed Seely isomorphisms in each fibre. Indeed, suppose $\mathcal{D}$ supports $\top$-, &-, and !-types. Then, $U_\Gamma(\top) = \mathsf{id}_\Gamma$ and $U_\Gamma(A\&B) = U_\Gamma(A) \times U_\Gamma(B)$, as U_Γ has a left adjoint and therefore preserves products. Now, F_Γ is strong monoidal and $!_\Gamma = F_\Gamma U_\Gamma$, so it follows that $!_\Gamma\top = I$ and $!_\Gamma(A\&B) = !_\Gamma A \otimes !_\Gamma B$.*

Now, theorem 3.2.14 suggests the possibility of similar Seely isomorphisms for $\Sigma^{\otimes}_{!}$-types and $\mathsf{Id}^{\otimes}_{!}$-types. Indeed, $\mathcal{C}$ supports Σ-types iff we have additive Σ-types in $\mathcal{D}$ in the sense of objects $\Sigma^{\&}_A B$ such that

$$U\Sigma^{\&}_A B \cong \Sigma_{UA} UB \quad \text{and hence} \quad !\Sigma^{\&}_A B \cong \Sigma^{\otimes}_{!A}!B.$$

In an ideal world, one would hope that $\Sigma^{\&}_A B$ generalises $A\&B$ in a similar way as how $\Sigma^{\otimes}_{!A}B$ is a dependent generalisation of $!A \otimes B$. In fact, it is easily seen that such categorical $\Sigma^{\&}$-types precisely (soundly and completely) correspond with the syntactic rules of figure 3.9, where we see a slight mismatch with &-types in the sense that the introduction and elimination rules only apply for cartesian contexts (without linear assumptions), here.

$$\frac{\vdash \Gamma, x : A, y : B; \cdot \ \mathsf{ctxt}}{\Gamma \vdash \Sigma^{\&}_{x:A} B \ \ \mathsf{type}}\ \Sigma^{\&}\text{-F} \qquad \frac{\Gamma; \cdot \vdash a : A \qquad \Gamma; \cdot \vdash b : B[a/x]}{\Gamma; \cdot \vdash \langle a, b\rangle : \Sigma^{\&}_{x:A} B}\ \Sigma^{\&}\text{-I}$$

$$\frac{\Gamma; \cdot \vdash t : \Sigma^{\&}_{x:A} B}{\Gamma; \cdot \vdash \mathsf{fst}\ (t) : A}\ \Sigma^{\&}\text{-E1} \qquad \frac{\Gamma; \cdot \vdash t : \Sigma^{\&}_{x:A} B}{\Gamma; \cdot \vdash \mathsf{snd}\ (t) : B[\mathsf{fst}\ (t)/x]}\ \Sigma^{\&}\text{-E2}$$

Figure 3.9: Rules for additive Σ-types. We also demand the obvious β- and η-equations.

Similarly, we get a notion of additive Id*-types:* $\mathcal{C}$ *supports* Id*-types iff we have objects* $\mathsf{Id}^{\&}_A(B)$ *in* $\mathcal{D}$ *such that*

$$U\mathsf{Id}^{\&}_A(B) \cong \mathsf{Id}_{UA}(UB) \quad \text{and hence} \quad !\mathsf{Id}^{\&}_A(B) \cong \mathsf{Id}^{\otimes}_{!A}(!B).$$

Note that this suggests that, in the same way that $\mathsf{Id}^{\otimes}_{!A}(B) \cong \mathsf{Id}^{\otimes}_{!A}(I) \otimes B$ *(a sense in which usual* $\mathsf{Id}^{\otimes}_{!}$*-types are multiplicative connectives),* $\mathsf{Id}^{\&}_A(B) \cong \mathsf{Id}^{\&}_A(\top)\&B$*. In fact, if we have* $\top$*- and* $\&$*-types, we only have to give* $\mathsf{Id}^{\&}_A(\top)$ *and can then define* $\mathsf{Id}^{\&}_A(B) := \mathsf{Id}^{\&}_A(\top)\&B$ *to obtain additive* Id*-types in generality.*

In the light of theorem 3.2.14, we obtain such additive Σ*- and* Id*-types in the fibre over* Γ *if some* U_Γ *is essentially surjective. In particular, we are in this situation if* $F_! \dashv U_!$ *is the usual co-Kleisli adjunction of* $!$*., where* $\mathcal{C}(\cdot) \cong \mathcal{B}$*. This shows that if we are hoping to obtain a model of dDILL indexed over the co-Kleisli category, in the natural way, we need to support these additive connectives.*

From experience, it seems like the natural models of dDILL do not generally support them, meaning that co-Kleisli categories often fail to give models of dependent types. Similarly, it is difficult to come up with an intuitive interpretation of the meaning of such connectives, in the sense of a resource interpretation.

To get some intuition of why such objects may be problematic, note that the usual resource interpretation $A\&B$ *is as follows: we either have (a resource of type)* A *or* B*. This means that we would expect a* $\Sigma^{\&}_A B$*-type, which should be a dependent generalisation of the ordinary* $\&$*-type, to have an additive reading too. However,* B *represents a predicate on* A*, so, if we have an object* c *of type* $\Sigma^{\&}_A B$*, we are in the situation that we can either produce an object* $\mathsf{fst}\ c$ *of type* A *or an object* $\mathsf{snd}\ c$ *embodying a property* B *of* $\mathsf{fst}\ c$*.*

Figure 3.10: We encourage the reader to compare the idea of additive Σ-types with Lewis Carroll's invention of the Cheshire cat.
"Well! I've often seen a cat without a grin," thought Alice; "but a grin without a cat! It's the most curious thing I ever saw in all my life.". [80]

This is like the Cheshire cat of Alice in Wonderland of figure 3.10: let A be the type of cats and let B be the predicate "is grinning". Then, fst *c corresponds to the cat and* snd *c may be thought to embody having a grin (of a cat) without having the cat.*

In section 5.4, we shall see a similar problem in the operational semantics of terms of such types. Terms of linear types generally represent dynamic objects. We shall see that the term fst *c, like the Cheshire cat, can lose information (e.g. the cat disappears; the term* fst *c, for instance, could be a computation which proceeds to make a non-deterministic choice or print to console) after which properties* snd *c which held true of* fst *c before the change no longer make sense (e.g. the cat is grinning; the program* fst *c is going to make a non-deterministic choice or prints* `hello world` *to console).*

3.3 dLNL Calculus

Independently from the author, Krishnaswami, Pradic and Benton developed a syntax for a dependently typed version of the LNL calculus in [78], which we refer to

as the dLNL calculus. It is a system with both cartesian types and linear types both of which are allowed to depend on terms of cartesian types, but not linear types. The cartesian type formers they consider are 1-, (strong) Σ-, Π- and extensional Id-types as well as universes (as cartesian types) that code for both linear and cartesian types and, finally, U-types which map linear types to cartesian types. The linear type formers they consider are I-, $\otimes$-, $\multimap$-, $\top$-, $\&$-, $\Sigma^{\otimes}_{F(-)}$- and $\Pi^{\multimap}_{F(-)}$-types and, finally, F-types which map cartesian types to linear types. In their work, they discuss an operational semantics but do not provide a denotational semantics. Therefore, we believe it might be useful to point out that our categorical framework can easily be adapted to model their dependent LNL calculus (minus universes, which can be given their usual awkward categorical semantics [81]).

Theorem 3.3.1 (Dependent LNL Calculus Semantics)**.** *A sound and complete categorical semantics for the universe-free fragment of the dependent LNL calculus of [78] is given by the following:*

- *a model* $\mathcal{B}^{op} \xrightarrow{\mathcal{C}} \mathsf{Cat}$ *of pure cartesian dependent type theory in the sense of an indexed category (with an indexed terminal object) with full and faithful comprehension* $(\mathbf{p}, \mathbf{v}, \langle --, - \rangle)$*;*
- *strong* Σ*-types in* $\mathcal{C}$*;*
- *strong (extensional)* Id*-types in* $\mathcal{C}$*;*
- Π*-types in* $\mathcal{C}$*;*
- *an indexed symmetric monoidal closed category* $\mathcal{B}^{op} \xrightarrow{\mathcal{D}} \mathsf{SMCCat}$*;*
- *indexed finite products* $(\top, \&)$ *in* $\mathcal{D}$*;*
- *a linear/non-linear indexed adjunction*[15] $F \dashv U : \mathcal{C} \leftrightarrows \mathcal{D}$*;*

[15]Note that it is, in fact, enough to merely ask for an indexed functor $\mathcal{D} \xrightarrow{U} \mathcal{C}$, as we then automatically obtain a linear/non-linear adjunction by defining $FA := \Sigma^{\otimes}_{F(A)} I$.

- $\Sigma^{\otimes}_{F(-)}$*-types in* $\mathcal{D}$ *in the sense of left adjoints* $\Sigma^{\otimes}_{F(A)} \dashv \mathcal{D}(\mathbf{p}_{\Gamma,A})$*, for all display maps* $\Gamma.A \xrightarrow{\mathbf{p}_{\Gamma,A}} \Gamma$*, satisfying the left Beck-Chevalley condition for all* $\mathbf{p}$*-squares and Frobenius reciprocity in the sense that the canonical maps*

$$\Sigma^{\otimes}_{F(A)}(\Delta'\{\mathbf{p}_{\Gamma,A}\} \otimes B) \longrightarrow \Delta' \otimes \Sigma^{\otimes}_{F(A)}B$$

  are isomorphisms;

- $\Pi^{\multimap}_{F(-)}$*-types in* $\mathcal{D}$ *in the sense of right adjoints* $\mathcal{D}(\mathbf{p}_{\Gamma,A}) \dashv \Pi^{\multimap}_{F(A)}$ *satisfying the right Beck-Chevalley condition for all* $\mathbf{p}$*-squares.*

Proof. The proof is entirely analogous to that for the categorical semantics of dependently typed DILL. □

Again, we see that the main distinction between DILL and the LNL calculus is that the latter considers the cartesian category $\mathcal{C}$ part of the structure while the DILL only assumes its existence. Note, in particular, that a model of the dependent LNL calculus in the sense described above defines a model of dependently typed DILL with $I, \otimes, \multimap, \top, \&, !, \Sigma^{\otimes}_{!}$ and $\Pi^{\otimes}_{!}$-types. Indeed, the comprehension on $\mathcal{C}$ together with the adjunction $F \dashv U$ is easily seen to give a comprehension for $\mathcal{D}$. However, we see that as U may not be full and faithful, a full and faithful comprehension for $\mathcal{C}$ may not give a full and faithful comprehension for $\mathcal{D}$ (just like in our semantics for dependently typed DILL).

3.4 Girard Translations

By contrast with the simply typed situation, I believe there are reasons to prefer an LNL calculus over a DILL-style system when working with dependent types. This is because the Girard translations fail for dDILL, meaning that it does not suffice to encode cartesian dependent type theory. Meanwhile, the LNL calculus is a proper extension of cartesian type theory, so, in particular, has more expressive power. Later, in chapter 5, we shall see another reason to prefer a LNL-style calculus, motivated by separating proving and programming. The idea is that

cartesian types are for pure proofs while linear types are assigned to (commutative) effectful programs.

Let us explain why the Girard translations of section 2.3.3 do not generalise well to dependent type theory in their conventional form. While we can define without any problems $(\Pi_A B)^f := \Pi^{\multimap}_{!A^f} B^f$, we run into problems with the first Girard translation of Σ-types. One would expect the first Girard translation of $\Sigma_A B$ to take the form of Σ-types $\Sigma^{\&}_{A^f} B^f$. We can indeed define this, but we know such connectives are often problematic from a denotational and operational point of view. Similarly, we would expect $(\mathsf{Id}_A)^f := \mathsf{Id}^{\&}_{A^f}$.

The second Girard translation is even more problematic. We would expect to define $(\Pi_A B)^s :=\!\Pi^{\multimap}_{A^s} B^s$ for some linear dependent function type $\Pi^{\multimap}_A B$ which generalises $A \multimap B$. We encounter a similar problem when defining $(\Sigma_A B)^s$ and $(\mathsf{Id}_A)^s$ which we would expect to be $\Sigma^{\otimes}_{A^s} B^s$ and $\mathsf{Id}^{\otimes}_{A^s}$ for some dependent connectives $\Sigma^{\otimes}$ generalising $\otimes$ and a connective $\mathsf{Id}^{\otimes}$ which takes the multiplicative identity type of linear terms. Such $\Pi^{\multimap}$, $\Sigma^{\otimes}$ and $\mathsf{Id}^{\otimes}$ types are even more problematic, in a sense, than additive Σ-types, as we simply cannot formulate natural deduction rules for such connectives in a system with types depending only on cartesian assumptions and not linear ones.

One can wonder if a satisfactory translation $(-)^t$ can be obtained by mixing the first and second Girard translations: using $(\Pi_A B)^t := \Pi_{!A^t} B^t$, $(\Sigma_A B)^t := \Sigma^{\otimes}_{!A^t} !B^t$ and $(\mathsf{Id}_A)^t := \mathsf{Id}^{\otimes}_{!A}$. It is easily seen that this leads to violation of the η-rules for Σ-types (as we are effectively modelling pattern matching Σ-types in CBN) and Id-types. It is at present unclear to us if such a translation still has value.

At the level of categorical semantics, the first Girard translation takes the form of the idea of modelling cartesian type theory in the co-Kleisli category $\mathcal{D}_!$ for ! of a model $\mathcal{D}$ of linear type theory. If we start with a model $\mathcal{B}^{op} \xrightarrow{\mathcal{D}} \mathsf{Cat}$ of dDILL with !-types and $\top$-types, it can easily be seen that the fibrewise co-Kleisli category $\mathcal{D}_!$ is a model of cartesian dependent type theory (with full and faithful comprehension).

Indeed, $\mathcal{D}_!(\Gamma')(\top, A\{f\}) \cong \mathcal{D}(I, A\{f\}) \cong \mathcal{B}/\Gamma(f, \mathbf{p}_{\Gamma,A})$ and

$$\begin{aligned}\mathcal{D}_!(\Gamma')(A, B) &\cong \mathcal{D}(\Gamma)(!A, B)\\ &\cong \mathcal{D}(\Gamma.A)(I, B\{\mathbf{p}_{\Gamma,A}\})\\ &\cong \mathcal{B}/\Gamma(\mathbf{p}_{\Gamma,A}, \mathbf{p}_{\Gamma,B}).\end{aligned}$$

Σ-types in $\mathcal{D}_!$ are easily seen to correspond precisely to $\Sigma^{\&}$-types in $\mathcal{D}$, which do not generally exist.

We can gain more insight into what is going on, by embedding the co-Kleisli category in the co-Eilenberg-Moore category, effectively closing it under certain equalisers (as every coalgebra has a presentation in terms of cofree coalgebras). Under that embedding, $A\&B$ is mapped to the cofree coalgebra $!A\otimes!B \cong !(A\&B) \xrightarrow{\delta_{A\&B}} !!(A\&B) \cong !(!A\otimes!B)$. Hence, we would expect $\Sigma^{\&}_A B$ to embed as a coalgebra $\Sigma^{\otimes}_{!A}!B \longrightarrow !\Sigma^{\otimes}_{!A}!B$. On closer inspection, it turns out that while (as in the simply typed case [33]) we can always define a canonical coalgebra structure[16] on $\Sigma^{\otimes}_{!A}!B$, this coalgebra structure is not always a cofree one $\delta_{\Sigma^{\&}_A B}$.

However, while we can always define Σ-types in the co-Eilenberg-Moore category, we have no guarantee that Π-types exist, by contrast with the co-Kleisli category where $\Pi^{\multimap}_{!A}B$ is the Π-type of A and B. Recalling that the co-Kleisli adjunction is the initial adjunction giving rise to ! and the co-Eilenberg-Moore one the terminal, we can hope to find a sweet spot $\mathcal{C}$ in between $\mathcal{D}_!$ and $\mathcal{D}^!$ which is closed under both Σ- and Π-types. If no natural candidate for $\mathcal{C}$ is available, we could, for instance, try to inductively close $\mathcal{D}_!$ under Σ-types in $\mathcal{D}^!$ (we work with formal Σ-types of cofree coalgebras) or coinductively close $\mathcal{D}^!$ under the Π-types of $\mathcal{D}_!$ (we work with a category of exponentiable coalgebras). We choose to employ the

[16] Indeed, given a coalgebra $B \xrightarrow{k} !B$ in $\mathcal{D}(\Gamma.A)$, we can define the coalgebra

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x:A, y:B; \cdot \vdash x : A} \quad \dfrac{}{\Gamma, x:A, y:B; \cdot \vdash y : B}}{\Gamma, x:A, y:B; \cdot \vdash !x \otimes y : \Sigma^{\otimes}_{!A} B}}{\Gamma, x:A, y:B; \cdot \vdash !(!x \otimes y) : !\Sigma^{\otimes}_{!A} B}}{\Gamma, x:A; z:!B \vdash \mathsf{let}\ z\ \mathsf{be}\ !y\ \mathsf{in}\ !(!x \otimes y) : !\Sigma^{\otimes}_{!A} B} \quad \dfrac{}{\Gamma, x:A; w:B \vdash k : !B}}{\Gamma, x:A; w:B \vdash \mathsf{let}\ k\ \mathsf{be}\ !y\ \mathsf{in}\ !(!x \otimes y) : !\Sigma^{\otimes}_{!A} B}}{\Gamma; v : \Sigma^{\otimes}_{!A} B \vdash \mathsf{let}\ v\ \mathsf{be}\ !x \otimes w\ \mathsf{in}\ \mathsf{let}\ k\ \mathsf{be}\ !y\ \mathsf{in}\ !(!x \otimes y) : !\Sigma^{\otimes}_{!A} B.}$$

We can, in particular, do this for the case that $k = \delta_C$.

former technique in the setting of game semantics. As we have an isomorphism of types $\Pi_{x:A}\Sigma_{y:B}C \cong \Sigma_{f:\Pi_{x:A}B}\Pi_{x:A}C[f(x)/y]$, we would expect these (co)inductively constructed categories to be closed under both Σ- and Π-types.

3.5 Concrete Models

3.5.1 Some Discrete Models: Monoidal Families

We discuss a simple class of models in terms of families with values in a symmetric monoidal category. On a logical level, what the construction boils down to is starting with a model $\mathcal{V}$ of a linear propositional logic and taking the cofree linear predicate logic on Set with values in this propositional logic. This important example illustrates how $\Sigma_!^{\otimes}$- and $\Pi_!^{\multimap}$-types can represent infinitary additive disjunctions and conjunctions. The model is discrete in nature, however, and in that respect not representative for the type theory.

Suppose $\mathcal{V}$ is a symmetric monoidal category. We can then consider a strict Set-indexed category, defined through the following enriched Yoneda embedding $\mathsf{Fam}(\mathcal{V}) := \mathcal{V}^- := \mathsf{SMCat}(-, \mathcal{V})$:

$$\mathsf{Set}^{op} \xrightarrow{\mathsf{Fam}(\mathcal{V})} \mathsf{SMCat} \qquad\qquad S \xrightarrow{f} S' \longmapsto \mathcal{V}^S \xleftarrow{f;-} \mathcal{V}^{S'}.$$

Note that this definition naturally extends to a functor Fam.

Theorem 3.5.1 (Families Model dDILL)**.** *The construction* Fam *adds type dependency on* Set *cofreely in the sense that it is right adjoint to the forgetful functor* ev_1 *that evaluates a model of dDILL at the empty context to obtain a model of linear propositional type theory (where* $\mathsf{SMCat}^{\mathsf{Set}^{op}}_{\mathsf{compr}}$ *is the full subcategory of* $\mathsf{SMCat}^{\mathsf{Set}^{op}}$ *on the objects with comprehension):*

$$\mathsf{SMCat} \underset{\mathsf{Fam}}{\overset{\mathsf{ev}_1}{\underset{\subset}{\overset{\longleftarrow}{\bot}\longrightarrow}}} \mathsf{SMCat}^{\mathsf{Set}^{op}}_{\mathsf{compr}}.$$

Proof. $\mathsf{Fam}(\mathcal{V})$ admits a comprehension, by the following isomorphism

$$\begin{aligned}\mathsf{Fam}(\mathcal{V})(S)(I, B\{f\}) &= \mathcal{V}^S(I, f; B)\\ &= \Pi_{s\in S}\mathcal{V}(I, B(f(s)))\\ &\cong \mathsf{Set}/S(S \xrightarrow{\mathsf{id}_S} S, \Sigma_{s\in S}\mathcal{V}(I, B(f(s))) \xrightarrow{\mathsf{fst}} S)\\ &\cong \mathsf{Set}/S'(S \xrightarrow{f} S', \Sigma_{s'\in S'}\mathcal{V}(I, B(s')) \xrightarrow{\mathsf{fst}} S')\\ &= \mathsf{Set}/S'(f, \mathbf{p}_{S',B}),\end{aligned}$$

where $\mathbf{p}_{S',B} := \Sigma_{s'\in S'}\mathcal{V}(I, B(s')) \xrightarrow{\mathsf{fst}} S'$. ($\mathbf{v}_{S',B}$ is obtained as the image of $\mathsf{id}_{S'} \in \mathsf{Set}/S'$ under this isomorphism.) To see that $\mathsf{ev}_1 \dashv \mathsf{Fam}$, note that we have the following naturality diagrams for elements $1 \xrightarrow{s} S$

$$\begin{array}{ccccc} 1 & \mathsf{ev}_1(\mathcal{D}) = \mathcal{D}(1) & \xrightarrow{\phi_1} & \mathcal{V} = \mathsf{Fam}(\mathcal{V})(1) \\ {\scriptstyle s}\downarrow & \uparrow{\scriptstyle -\{s\}} & & \uparrow{\scriptstyle s;-} \\ S & \mathcal{D}(S) & \xrightarrow[\phi_S]{} & \mathcal{V}^S = \mathsf{Fam}(\mathcal{V})(S) \end{array}$$

and that all $1 \xrightarrow{s} S$ are jointly surjective and therefore all $s;-$ are jointly injective, meaning that a natural transformation $\phi \in \mathsf{SMCat}^{\mathsf{Set}^{op}}(\mathcal{D}, \mathsf{Fam}(\mathcal{V}))$ is uniquely determined by $\phi_1 \in \mathsf{SMCat}(\mathsf{ev}_1(\mathcal{D}), \mathcal{V})$. □

We have the following results for type formers.

Theorem 3.5.2 (Type Formers for Families)**.** *$\mathcal{V}$ has small coproducts that distribute over $\otimes$ iff $\mathsf{Fam}(\mathcal{V})$ supports $\Sigma^{\otimes}_{!}$-types. In that case, $\mathsf{Fam}(\mathcal{V})$ also supports 0- and $\oplus$-types (which correspond precisely to finite distributive coproducts).*

$\mathcal{V}$ has small products iff $\mathsf{Fam}(\mathcal{V})$ supports $\Pi^{\multimap}_{!}$-types. In that case, $\mathsf{Fam}(\mathcal{V})$ also supports $\top$- and $\&$-types (which correspond precisely to finite products).

$\mathsf{Fam}(\mathcal{V})$ supports $\multimap$-types iff $\mathcal{V}$ is monoidal closed.

$\mathsf{Fam}(\mathcal{V})$ supports !-types iff $\mathcal{V}$ has small coproducts of I that are preserved by $\otimes$ in the sense that the canonical morphism $\bigoplus_S(\Delta' \otimes I) \longrightarrow \Delta' \otimes \bigoplus_S I$ is an isomorphism for any $\Delta' \in \mathsf{ob}\ \mathcal{V}$ and $S \in \mathsf{ob}\ \mathsf{Set}$. In particular, if $\mathsf{Fam}(\mathcal{V})$ supports $\Sigma^{\otimes}_{!}$-types, then it also supports !-types.

$\mathsf{Fam}(\mathcal{V})$ *supports* $\mathsf{Id}^{\otimes}_{!}$*-types if* $\mathcal{V}$ *has a distributive initial object. Supposing that* $\mathcal{V}$ *has a terminal object, the only if also holds.*

Proof. The statement about 0-, $\oplus$-, $\top$-, and &-types should be clear from the previous sections, as products and coproducts in $\mathcal{V}^S$ are pointwise (and hence automatically preserved under substitution).

We denote coproducts in $\mathcal{V}$ with $\bigoplus$. Then,

$$\begin{aligned}\Pi_{s'\in S'}\mathcal{V}(\bigoplus_{s\in f^{-1}(s')} A(s), B(s')) &\cong \Pi_{s'\in S'}\Pi_{s\in f^{-1}(s')}\mathcal{V}(A(s), B(s'))\\ &\cong \Pi_{s\in \Sigma_{s'\in S'} f^{-1}(s')}\mathcal{V}(A(s), B(f(s)))\\ &\cong \Pi_{s\in S}\mathcal{V}(A(s), B(f(s))\\ &= \mathcal{V}^S(A, f; B).\end{aligned}$$

So, we see that we can define $\Sigma^{\otimes}_{F(f)}(A)(s') := \bigoplus_{s\in f^{-1}(s')} A(s)$ to get a left adjoint $\Sigma^{\otimes}_{F(f)} \dashv -\{f\}$, if we have coproducts. (With the obvious definition on morphisms coming from the cocartesian monoidal structure on $\mathcal{V}$.) Conversely, we can clearly use $\Sigma^{\otimes}_{F(f)}$ to define any coproduct by using, for instance, an identity function for f on the set we want to take a coproduct over and a family A that denotes the objects we want to sum. The Beck-Chevalley condition is taken care of by the fact that our substitution morphisms are given by precomposition. Frobenius reciprocity precisely corresponds to distributivity of the coproducts over $\otimes$.

Similarly, if $\mathcal{V}$ has products, we denote them with $\mathbin{\&}$ to suggest the connections with linear type theory. In that case, we can define $\Pi^{\multimap}_{F(f)}(A)(s') := \mathbin{\&}_{s\in f^{-1}(s')} A(s)$ to get a right adjoint $-\{f\} \dashv \Pi^{\multimap}_{F(f)}$. (With the obvious definition on morphisms coming from the cartesian monoidal structure on $\mathcal{V}$.) Indeed,

$$\begin{aligned}\Pi_{s'\in S'}\mathcal{V}(B(s'), \mathop{\&}_{s\in f^{-1}(s')} A(s)) &\cong \Pi_{s'\in S'}\Pi_{s\in f^{-1}(s')}\mathcal{V}(B(s'), A(s))\\ &\cong \Pi_{s\in \Sigma_{s'\in S'} f^{-1}(s')}\mathcal{V}(B(f(s)), A(s))\\ &\cong \Pi_{s\in S}\mathcal{V}(B(f(s)), A(s))\\ &= \mathcal{V}^S(f; B, A).\end{aligned}$$

Again, in the same way as before, we can construct any product using $\Pi^{\multimap}_{F(f)}$. The right Beck-Chevalley condition comes for free as our substitution morphisms are precomposition.

The claim about $\multimap$-types follows immediately from the previous section: $\mathsf{Fam}(\mathcal{V})$ supports $\multimap$-types iff all its fibres have a monoidal closed structure that is preserved by the substitution functors. Seeing that our monoidal structure is pointwise, the same will hold for any monoidal closed structure. Seeing that substitution is given by precomposition, the preservation requirement comes for free.

The characterisation of !-types is given by theorem 3.1.5, which tells us we can define $!A := \Sigma^{\otimes}_{F(\mathbf{p}_{S',A})} I = s' \mapsto \bigoplus_{\mathcal{V}(I,A(s'))} I$ and conversely.

Finally, for $\mathsf{Id}^{\otimes}_{!}$-types, note that the adjointness condition $\mathsf{Id}^{\otimes}_{!A} \dashv -\{\mathsf{diag}_{\Gamma,A}\}$ boils down to the requirement (*)

$$\begin{aligned}\Pi_{s\in S}\Pi_{a\in A(s)}\mathcal{V}(B(s,a),C(s,a,a)) &\cong \mathcal{V}^{\Sigma_{s\in S}A(s)}(B, C\{\mathsf{diag}_{S,A}\}) \\ &\overset{!}{\cong} \mathcal{V}^{\Sigma_{s\in S}A(s)\times A(s)}(\mathsf{Id}^{\otimes}_{!A}(B), C) \\ &\cong \Pi_{s\in S}\Pi_{a\in A(s)}\Pi_{a'\in A(s)}\mathcal{V}(\mathsf{Id}^{\otimes}_{!A}(B)(s,a,a'), C(s,a,a')).\end{aligned}$$

We see that if we have an initial object $0 \in \mathsf{ob}(\mathcal{V})$, we can define

$$\mathsf{Id}^{\otimes}_{!A}(B)(s,a,a') := \begin{cases} B(s,a) \text{ if } a = a' \\ 0 \text{ else} \end{cases}$$

Distributivity of the initial object then gives us Frobenius reciprocity. For a partial converse, suppose we have a terminal object $\top \in \mathcal{V}$. Let $V \in \mathsf{ob}(\mathcal{V})$. Let $S := \{*\}$, $A := \{0,1\}$ and C s.t. $C(0,0) = C(1,1) = C(0,1) = \top$ and $C(1,0) = V$. Then, (*) becomes the condition that $\{*\} \cong \mathcal{V}(\mathsf{Id}^{\otimes}_{!A}(B)(1,0), V)$. We conclude that $\mathsf{Id}^{\otimes}_{!A}(B)(1,0)$ is initial in $\mathcal{V}$. □

Remark 3.5.3. *Note that an obvious way to guarantee distributivity of coproducts over $\otimes$ is by demanding that $\mathcal{V}$ is monoidal closed.*

Remark 3.5.4. *It is easily seen that Σ-types in $\mathcal{C}$, or additive Σ-types in $\mathcal{D} = \mathsf{Fam}(\mathcal{V})$, boil down to having an object $\mathsf{or}_{s\in S}C(s) \in \mathsf{ob}(\mathcal{V})$ for a family $(C(s) \in \mathsf{ob}(\mathcal{V}))_{s\in S}$ such that $\Sigma_{s\in S}\mathcal{V}(I, C(s)) \cong \mathcal{V}(I, \mathsf{or}_{s\in S}C(s))$. Similarly, Id-types in $\mathcal{C}$, or additive Id-types in $\mathcal{D}$, boil down to having objects $\mathsf{one}, \mathsf{zero} \in \mathsf{ob}(\mathcal{V})$ such that $\mathcal{V}(I, \mathsf{one}) \cong 1$ and $\mathcal{V}(I, \mathsf{zero}) = 0$.*

Two particularly simple concrete examples of $\mathcal{V}$ come to mind that can accommodate all type formers (except additive Σ- and Id-types, which are easily seen not to be supported) and form a nice illustration: a category $\mathcal{V} = \mathsf{Vect}_F$ of vector spaces over a field F, with the tensor product, and the category $\mathcal{V} = \mathsf{Set}_*$ of pointed sets, with the smash product. All type formers get their obvious interpretation, but let us stop to think about ! for a second as it is a novelty of dDILL that it gets uniquely determined by the indexing, while in propositional linear type theory we might have several different choices. In the first example, ! boils down to the following: $(!B)(s') = \bigoplus_{\mathsf{Vect}_F(F,B(s'))} F \cong \bigoplus_{B(s')} F$, i.e. taking the vector space freely spanned by all the vectors. In the second example, $(!B)(s') = \bigoplus_{\mathsf{Set}_*(2_*,B(s'))} 2_* = \bigvee_{B(s')} 2_* = B(s') + \{*\}$, i.e. ! freely adds a new basepoint.

We note the following consequence of theorem 3.5.1.

Theorem 3.5.5 (DTT, DILL$\subsetneq$dDILL)**.** *dDILL is a proper generalisation of DTT and DILL: we have inclusions of the classes of models DTT,DILL$\subsetneq$dDILL.*

Proof. Models of DTT with 1- and $\times$-types, i.e. indexed cartesian monoidal categories with full and faithful comprehension, clearly, are a special case of our notion of model of dDILL. Moreover, in such cases, we easily see that $!A \cong A$. From their categorical descriptions, it is also clear that the other connectives of dDILL reduce to those of DTT. This proves the inclusion DDT$\subseteq$dDILL.

The dDILL models described above based on symmetric monoidal families are clearly more general than those of DTT, as we are dealing with a non-cartesian symmetric monoidal structure on the fibre categories. This proves that the inclusion is proper.

We have seen that the Fam-construction realises the category of models of DILL as a reflective subcategory of the category of models of dDILL. Moreover, from various non-trivial models of DTT indexed over other categories than Set it is clear that this inclusion is proper as well.

Finally, we note that these inclusions still remain valid in the sub-algebraic setting where we do not have I- and $\otimes$-types. A simple variation of the argument using multicategories rather than monoidal categories does the trick. $\square$

Although this class of families models is important, it is clear that it only represents a very limited part of the generality of dDILL: not every model of dDILL is either a model of DTT or of DILL. Hence, we are in need of models that are less discrete in nature but still linear, if we are hoping to observe interesting new phenomena arising from the connectives of dDILL.

3.5.2 Commutative Effects

As in the simply typed situation, commutative effects in dependent type theory give rise to linear types (under mild completeness and cocompleteness assumptions).

Theorem 3.5.6. *Suppose we are given a model $\mathcal{C}$ of pure dependent type theory with $1, \Sigma, 0, +, \Pi$-types which is equipped with an indexed commutative monad T, where $\mathcal{C}$ additionally has equalisers and $\mathcal{C}^T$ has reflexive coequalisers. Then, $\mathcal{C}^T$ is a model of dDILL with $I, \otimes, \multimap, \top, \&, !, \Sigma_!^{\otimes}, \Pi_!^{\multimap}$-types.*

Proof. The interpretation of $\top, \&, \Pi_!^{\multimap}, !$-types follows from theorem 5.2.9. Meanwhile, an indexed variation of theorem 2.3.3 gives us the interpretation of $I, \otimes, \multimap$-types. Finally, theorem 5.6.3 lets us interpret $\Sigma_!^{\otimes}$-types. □

3.5.3 A Double Glueing Construction

Of course, any model $\mathcal{C}$ of cartesian type theory is a degenerate model $\mathcal{D} = \mathcal{C}$ of linear type theory in which the additive and multiplicative connectives coincide and where we can define ! to be the identity to obtain $\mathcal{C} = \mathcal{D}_!$. This shows us that every model of dependent type theory is trivially obtained through a co-Kleisli construction on a model of dDILL. This shows, in particular, in a rather boring way, that $\Sigma^{\&}$- and $\mathsf{Id}^{\&}$-types are consistent.

A more interesting construction is the following, which arises as a simple case of the double gluing construction of [82], saying that every model of propositional intuitionistic logic arises from a model of classical linear logic, as a category of cofree !-coalgebras. Note that this is a properly linear model in the sense that its symmetric monoidal structure is not cartesian, i.e. there is a real difference between

additive and multiplicative connectives. This follows from Joyal's lemma which says that cartesian $*$-autonomous categories are preorders [10].

Theorem 3.5.7. *Let $(\mathcal{C}, 1, \times, \Rightarrow)$ be a cartesian closed category. Then, $\mathcal{D} := \mathcal{C} \times \mathcal{C}^{op}$ can be given the structure of a $*$-autonomous category equipped with a linear exponential comonad $!$, such that $\mathcal{C} \cong \mathcal{D}_!$.*

Proof. We see $\mathcal{D}$ as a special case of the Chu-construction, where the pairing takes values in the terminal object 1: we define the duality $(a, x)^* := (x, a)$. We obtain the usual formula for the symmetric monoidal structure on $\mathcal{D}$:

$$I := (1, 1)$$
$$(a, x) \otimes (b, y) := (a \times b, (a \Rightarrow y) \times (b \Rightarrow x)).$$

This allows us to define

$$(a, x) \multimap (b, y) := ((a, x) \otimes (b, y)^*)^* = ((a \Rightarrow b) \times (y \Rightarrow x), a \times y).$$

Note that we have a linear/non-linear adjunction

$$\begin{array}{ccc} (a, 1) & \longmapsto & a \\ & & \\ \mathcal{D} & \underset{U}{\overset{F}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} & \mathcal{C} \\ & & \\ (a, x) & \longmapsto & a \end{array}$$

meaning that we obtain a linear exponential comonad $! := FU$ on $\mathcal{D}$. Finally, note that we have an equivalence of categories $\mathcal{D}_! \longrightarrow \mathcal{C}$, $(a, x) \mapsto a$, being a full and faithful and essentially surjective functor. □

Remark 3.5.8. *Note that $\mathcal{D}$ in the previous theorem supports finite products (or, equivalently, finite coproducts), if and only if $\mathcal{C}$ supports finite coproducts. Indeed, $\top := (1, 0)$ and $(a, x)\&(b, y) := (a \times b, x + y)$. $\mathcal{D}$ having finite products (additive conjunctions) is a sufficient but not necessary condition for $\mathcal{D}_!$ to have finite products. Indeed, in the above example, $\mathcal{D}_! \cong \mathcal{C}$ always has finite products. It is not clear what additive versions of the dependent connectives Σ and Id should be, except in*

the weaker sense of objects in $\mathcal{D}$ that in $\mathcal{D}_!$ give a sound interpretation of ordinary cartesian Σ-types and Id-types. In particular, it is not clear what a dependently typed generalisation should be of the binary coproduct, in the same sense that Σ- and Π-types, respectively, provide dependently typed generalisations of the binary product and the internal hom: the idea of "having one of two types of objects, where the type of the second depends on the first" sounds puzzling at best.

This result extends to the dependently typed setting, as follows.

Theorem 3.5.9. *Let $\mathcal{B}^{op} \xrightarrow{\mathcal{C}} \mathsf{Cat}$ be a strict indexed cartesian closed category with full and faithful comprehension (i.e. a model of cartesian dependent type theory). Write $\mathcal{D} := \mathcal{C} \times \mathcal{C}^{op}$, where we take the cartesian product of the fibre categories. Then, $\mathcal{D}$ is a strict indexed $*$-autonomous category with comprehension (i.e. a model of classical linear dependent type theory). Moreover,*

- *$\mathcal{D}$ supports $!$-types and we have an (indexed) equivalence $\mathcal{D}_! \cong \mathcal{C}$.*
- *Therefore, $\mathcal{D}$ supports $\Sigma^{\&}$- and $\mathsf{Id}^{\&}$-types, respectively, iff $\mathcal{C}$ supports (strong) Σ- and Id-types.*
- *$\mathcal{D}$ supports $\Sigma_!^{\otimes}$ and $\Pi_!^{\multimap}$-types iff $\mathcal{C}$ supports both (weak) Σ- and Π-types.*
- *$\mathcal{D}$ supports (extensional, resp. intensional) $\mathsf{Id}_!^{\otimes}$-types iff $\mathcal{C}$ supports (weak) (extensional, resp. intensional) Id-types.*

Proof. These are straightforward verifications. By analogy with $!(-) \otimes (-)$ and $!(-) \multimap (-)$, we can define $\Sigma^{\otimes}_{!(a,x)}(b,y) := (\Sigma_a b, \Pi_a y)$, $\Pi^{\multimap}_{!(a,x)}(b,y) := (\Pi_a b, \Sigma_a y)$. Note they are dual, in the sense that $(\Sigma_{!(a,x)}(b,y))^* = \Pi_{!(a,x)}(b,y)^*$. $\mathsf{Id}_!^{\otimes}$-types, we can interpret by $\mathsf{Id}^{\otimes}_{!(a,x)} := (\mathsf{Id}_a, 1)$. (More generally, we define the functor $\mathsf{Id}^{\otimes}_{!(a,x)}(b,y) := \mathsf{Id}^{\otimes}_{!(a,x)} \otimes (b,y) = (\mathsf{Id}_a \times b, (\mathsf{Id}_a) \Rightarrow y)$.) Indeed, this definition of $\mathsf{Id}^{\otimes}_{!(a,x)}$ follows from applying the Yoneda lemma to the following sequence of natural

isomorphisms:

$$\begin{aligned}\mathcal{C} \times \mathcal{C}^{op}(\Gamma.a.a)(\mathsf{Id}^{\otimes}_{!(a,x)}, (b,y)) &\cong \mathcal{C} \times \mathcal{C}^{op}(\Gamma.a)((1,1),(b,y)\{\mathsf{diag}_{\Gamma,a}\}) \\ &\cong \mathcal{C}(\Gamma.a)(1, b\{\mathsf{diag}_{\Gamma,a}\}) \\ &\cong \mathcal{C}(\Gamma.a.a)(\mathsf{Id}_a, b) \\ &\cong \mathcal{C} \times \mathcal{C}^{op}(\Gamma.a.a)((\mathsf{Id}_a, 1),(b,y)).\end{aligned}$$

□

3.5.4 Scott Domains and Strict Functions

We can extend the model of section 2.3.4.2 to a model of dDILL, following [83]. All constructions and proofs are exactly as in [83] with the only difference that in some cases we have to replace the word domain with predomain.

We take $\mathcal{B}$ to be the category of Scott predomains and continuous functions. For a preorder-enriched category like $\mathcal{B}$, we call a pair of morphisms $e : A \leftrightarrows B : p$ an embedding-projection pair if $e; p = \mathsf{id}_A$ and $p; e \leq \mathsf{id}_B$. We write $\mathcal{B}_{ep}$ for the lluf subcategory of $\mathcal{B}$ of the embedding-projection pairs. We can make this into a model $\mathcal{C}$ of DTT in the same way as we can for the category of Scott domains and continuous functions [83]: we define $\mathcal{C}(A)$ to be the category of Scott predomains parametrised over A (continuous families of predomains), which we define to be directed colimit preserving functors from A into $\mathcal{B}_{ep}$. Change of base is given by precomposition. This supports 1-, Σ-, $\Sigma_{1\leq i\leq n}$- and (intensional) Id-types. Briefly, 1 is the one-point predomain, Σ-types are just the set-theoretic Σ-types equipped with the product order, $\Sigma_{1\leq i\leq n}$-types are given by disjoint unions and $\mathsf{Id}_A(x,y) := \{z \in A \mid z \leq x \ \wedge z \leq y\}$ with the induced order from A. For predomains A and a predomain B parametrised over A, we can define a poset (which generally will not be a predomain, for size reasons) $\Pi_{x\in A}B$ as the set of continuous[17] dependent functions from A to B under the pointwise order. This

[17]Here, I am referring to the appropriate generalisation of continuity to dependent functions (called p-continuity in [83]): functions $f : \Pi_{x:A}B$ such that, for all $x \in A$, for all compact elements $b \leq_{B(x)} f(x)$, there exists a compact element $a \leq_A x$ such that $b \leq_{B(x)} f(a)$.

allows us to define $\mathcal{C}(A)(B,C) := \Pi_{x\in A}\Pi_{y\in B(x)}C(x)$ on which we have the obvious identities, composition and change of base functions.

We can define $\mathcal{D}(A)$ to be the category of Scott domains parametrised over A (continuous families of Scott domains) with strict continuous (families of) functions as morphisms. We do this by extending the operations $\top, \&, I, \otimes$ and $\multimap$ to $\mathcal{D}(A)$ in a pointwise way and defining $\mathcal{D}(A)(B,C) := \Pi_{x\in UA}U(B(x) \multimap C(x))$. This extends to give an indexed category of parametrised Scott domains indexed over Scott predomains. We note that we have an indexed adjunction $F \dashv U$ to $\mathcal{C}$ (pointwise) where F is strong monoidal. We see that we have a model of the dependently typed LNL calculus.

We can note that $-\mathcal{D}\{\mathbf{p}_{A,B}\}$ have both left and right adjoints $\Sigma^{\otimes}_{F(B)}$ and $\Pi^{\multimap}_{F(B)}$ satisfying (Frobenius and) Beck-Chevalley conditions. Here, $\Sigma^{\otimes}_{F(B)}C(x) := \{\langle b, c\rangle \in \Sigma_{y\in B(x)}UC(x,y) \mid c \neq \bot\} \cup \{\bot\}$ and $\Pi^{\multimap}_{F(B)}C(x) := \Pi_{y\in B(x)}UC(x,y)$. In fact, we have additive Σ- and Id-types as well by noting that $\Sigma_{y\in UB}UC(y)$ and Id_{UB} are (parametrised) Scott domains. Moreover, we can define $\mathsf{Id}^{\otimes}_{FA}$ as $F\mathsf{Id}_A$.

3.5.5 Coherence Spaces

The usual coherence space model of linear type theory can be extended with a notion of dependent types, which gives us a non-trivial model of dDILL (of classical linear dependent type theory). We define it as a strict indexed symmetric monoidal cloed category with comprehension

$$\mathsf{Stable}^{op} \xrightarrow{\mathcal{D}} \mathsf{SMCCat}.$$

For our category of cartesian contexts we take the category Stable of Scott predomains with pullbacks and continuous stable functions. Note that we have a large Scott (pre)domain $\mathcal{U}$ with pullbacks (given by intersection) of small coherence spaces, using the following ordering on coherence spaces:

$$X \trianglelefteq Y := X \subseteq Y \wedge \ \frown_X = \frown_Y \Big|_{X\times X}.$$

$\mathcal{U}$ will play the rôle of a cartesian universe of linear types. For a predomain $D \in \mathsf{ob}(\mathsf{Stable})$, we define $\mathcal{D}(D)$ to be a category with set of objects stable functions

from D to $\mathcal{U}$: $\mathsf{ob}(\mathcal{D}(D)) := \mathsf{Stable}(D,\mathcal{U})$. We shall define its morphisms shortly, but, first, we define a few operations on the objects: $I(=\bot), \otimes, \multimap, \&, 0(=\top)$, and $\oplus$ are defined pointwise, as on coherence spaces. Given $G' \in \mathsf{Stable}(D,\mathcal{U})$, we define two coherence spaces with the same underlying set

$$\Sigma^{\otimes}_{F(x:D)}G'(x) := \Pi^{\multimap}_{F(x:D)}G'(x) := \{(x,u) \mid x \in FD, u \in G'(x)\}$$

but different coherence relations

$$(s,u) \frown_{\Sigma^{\otimes}_{F(x:D)}G'(x)} (t,v) := s \frown_{FD} t \ \wedge \ u \frown_{G'(s\vee t)} v,$$

and

$$(s,u) \frown_{\Pi^{\multimap}_{F(x:D)}G'(x)} (t,v) := s \frown_{FD} t \ \Rightarrow \ u \frown_{G'(s\vee t)} v.$$

This defines two coherence spaces $\Sigma^{\otimes}_{F(x:D)}G'(x)$ and $\Pi^{\multimap}_{F(x:D)}G'(x)$.

We define (as the type theory dictates[18] , if $\Pi^{\multimap}_{F(x:D)}$ is to give the $\Pi^{\multimap}_{F(-)}$-type) the morphisms in the fibres of $\mathcal{D}$ as cliques in the appropriate $\Pi^{\multimap}_{F(-)}$-type:

$$\mathcal{D}(D)(G',G) := \mathsf{cliques}(\Pi^{\multimap}_{F(x:D)}(G' \multimap G)(x)).$$

Composition and identities are defined pointwise (where it is left to the reader to verify that these are indeed cliques):

$$G' \xrightarrow{\sigma} G \xrightarrow{\tau} H := \{(x,f,h) \mid \left(\exists_{y,z\leq x}\exists_{g\in G(x)}(y,f,g) \in \sigma \wedge (z,g,h) \in \tau\right) \ \wedge$$

$$\forall_{x'\leq x}\left(\exists_{y',z'\leq x'}\exists_{g\in G(x')}(y',f,g) \in \sigma \wedge (z',g,h) \in \tau\right) \Rightarrow x' = x\}$$

$$G' \xrightarrow{\mathsf{id}_{G'}} G' := \{(x,f,f) \mid f \in G'(x) \wedge \forall_{x'\leq x} f \in G'(x') \Rightarrow x' = x\}.$$

The reader can check that the pointwise operations I and $\otimes$ make $\mathcal{D}(D)$ into a symmetric monoidal category.

Theorem 3.5.10. *Stable continuous families of coherence spaces, indexed over Scott predomains with pullbacks and stable continuous functions define a strict indexed symmetric monoidal category with comprehension, hence a model of dDILL.*

[18] Indeed, $\mathsf{cliques}(\Pi^{\multimap}_{F(x:D)}(G' \multimap G)(x)) = \mathcal{D}(\cdot)(I, \Pi^{\multimap}_{F(x:D)}(G' \multimap G)(x)) \cong \mathcal{D}(D)(I, G' \multimap G) \cong \mathcal{D}(D)(G',G)$.

Proof. Note that Stable is a category with terminal object the one point domain and that $\mathcal{D}(D)$ is a symmetric monoidal category. We define change of base in $\mathcal{D}$ for morphisms $D' \xrightarrow{f} D$ in Stable: $\mathcal{D}(f)(G') := f; G'$ and, for $\sigma \in \mathcal{D}(D)(G', G)$, $\mathcal{D}(f)(\sigma) := F(f); \sigma$, where we see $F(f)$ as a clique in $FD' \multimap FD$. This gives a clique in $\Pi^{\multimap}_{F(x:D')}(G' \multimap G)(f(x))$. $\mathcal{D}(f)$ is easily seen to strictly preserve I and $\otimes$ as precomposition is compatible with the pointwise defined connectives.

What remains to be done is define the comprehension. We define this as $U_D(G') := \mathbf{p}_{D,UG'} := \Sigma_{x:D} UG'(x) \xrightarrow{\mathsf{fst}} D$, where the Σ-type is taken in Stable. That is, we take the set theoretic Σ-type, equip it with the product order and note that it gives a Scott predomain with pullbacks. The fact that it is a Scott predomain follows from section 3.5.4 as we are taking the Σ-type of a continuous family of Scott domains. To see that it has pullbacks, note that $UG'(x)$ and $UG'(y)$ are downward closed subsets of $UG'(z)$ if $x, y \leq z$, such that $UG'(x \wedge y) = UG'(x) \cap UG'(y)$ (because of stability of G'). That means that we can take component-wise meets.

We define

$$\mathbf{v}_{D,UG'} \in \mathcal{D}(\Sigma_{x:D} UG'(x))(I, G'\{\mathbf{p}_{D,UG'}\}) \cong \mathsf{cliques}(\Pi^{\multimap}_{F((z,s):\Sigma_{x:D} UG'(x))} G'(z))$$

as the "trace" of the (dependent) projection onto the second component:

$$\mathbf{v}_{D,UG'} := \{((x,s), v) \mid (x,s) \in F(\Sigma_{x:D} UG'(x)), v \in s, \forall_{(x',s') \leq (x,s)} v \in s' \Rightarrow (x',s') = (x,s)\},$$

which is a clique.

Claim. $\mathbf{v}_{D,UG'}$ *is a clique in* $\Pi^{\multimap}_{F((z,s):\Sigma_{x:D} UG'(x))} G'(z)$.

Proof. Assume $((x,s),v) \neq ((x',s'),v')$, then

$$\begin{aligned}
&((x,s),v) \frown_{\Pi^{\multimap}_{F((z,s):\Sigma_{x:D} UG'(x))} G'(z)} ((x',s'),v') \\
&\equiv (x,s) \supset_{F(\Sigma_{x:D} UG'(x))} (x',s') \Rightarrow v \frown_{G'(x \vee x')} v' \\
&\equiv \exists_{(x'',s'') \in \Sigma_{x:D} UG'(x)} ((x,s),(x',s') \leq_{\Sigma_{x:D} UG'(x)} (x'',s'')) \Rightarrow v \frown_{G'(x'')} v'.
\end{aligned}$$

Assume $v = v'$. Then, the maximality condition on (x, s) gives that $(x, s) = (x, s')$, contradicting our assumption that $((x, s), v) \neq ((x', s'), v')$. Therefore, $v \neq v'$.

Then, $(x, s), (x', s') \leq (x'', s'')$ implies that $v \in s \subseteq s'' \supseteq s' \ni v'$, which in turn implies that $v \supset_{G'(x'')} v'$ and, as $v \neq v'$, we conclude that $v \supset_{G'(x'')} v'$. □

Given $f \in \mathsf{Stable}(D', D)$ and $\sigma \in \mathcal{D}(D')(I, f; G') = \mathsf{cliques}(\Pi^{\multimap}_{F(x:D')} G'(f(x)))$, we define $\langle f, \sigma \rangle \in \mathsf{Stable}(D', \Sigma_{x:D} UG'(x))$ as the function $(f, \mathsf{fun}(\sigma))$ with first component f and second component $\mathsf{fun}(\sigma)$, where (writing $\Pi_{x:D'} UG'(f(x))$ for the set of dependent continuous stable functions from D' to $f'; UG$)

$$\mathsf{fun}(\sigma) := \{(x, \bigvee \{a \in UG'(f(x)) | \exists y \leq x, (y, a) \in \sigma\}) \mid x \in D'\} \in \Pi_{x:D'} UG'(f(x)).$$

We verify that $(\mathbf{p}, \mathbf{v}, \langle -, - \rangle)$ gives a representation, demonstrating the comprehension axiom. Clearly, $\langle f, \sigma \rangle; \mathbf{p}_{D,UG'} = \langle f, \sigma \rangle; \mathsf{fst} = f$ and $\mathbf{v}_{D,UG'}\{\langle f, \sigma \rangle\} = F(\langle f, \sigma \rangle); \mathsf{trace}(\mathsf{snd}) = F((f, \mathsf{fun}(\sigma))); \mathsf{trace}(\mathsf{snd}) = \mathsf{trace}((f, \mathsf{fun}(\sigma)); \mathsf{snd}) = \mathsf{trace}(\mathsf{fun}(\sigma)) = \sigma$. Conversely, it is easily seen that $\langle f, \sigma \rangle$ is uniquely determined by these two equations. Indeed, suppose $t \in \mathsf{Stable}(D', \Sigma_{x:D} UG'(x))$ such that $f = t; \mathbf{p}_{D,UG'} = t; \mathsf{fst}$ and $\sigma = \mathbf{v}_{D,UG'}\{t\} := Ft; \mathbf{v}_{D,UG'} = Ft; \mathsf{trace}(\mathsf{snd}) = \mathsf{trace}(t; \mathsf{snd})$. Then, $\langle f, \sigma \rangle = \langle t; \mathsf{fst}, \mathsf{trace}(t; \mathsf{snd}) \rangle = (t; \mathsf{fst}, \mathsf{fun}(\mathsf{trace}(t; \mathsf{snd}))) = (t; \mathsf{fst}, t; \mathsf{snd}) = t$. □

Theorem 3.5.11. *The model supports $I-, \otimes-, \multimap -, \top-, \&-, 0-, \oplus-, \Sigma^{\otimes}_{!}-, \Pi^{\otimes}_{!}-$, $!-$, and $\mathsf{Id}^{\otimes}_{!}$-types. Moreover, it is a model of classical linear dependent type theory.*

Proof. For $I-, \otimes-, \multimap -, \top-, \&-, 0-$ and $\oplus-$-types the verifications are trivial as the type formers are defined pointwise. It is clear that I and $\otimes$ give a symmetric monoidal structure on $\mathcal{D}(D)$. It then follows that $\multimap$ gives internal homs, from the facts that our operations are defined pointwise and that $\multimap$ gives internal homs in Coh: $\mathcal{D}(D)(G' \otimes G, H) \cong \mathsf{cliques}(\Pi_{x:D}((G' \otimes G) \multimap H))(x)) = \mathsf{cliques}(\Pi_{x:D}((G'(x) \otimes G(x)) \multimap H(x)) \cong \mathsf{cliques}(\Pi_{x:D}(G'(x) \multimap (G(x) \multimap H(x))) = \mathsf{cliques}(\Pi_{x:D}((G' \multimap (G \multimap H))(x)) = \mathcal{D}(D)(G', G \multimap H)$.

We have to show that $\top$ and $\&$ give finite products on $\mathcal{D}(D)$. Let $G' \in \mathsf{ob}(\mathcal{D}(D))$. Then, we have a unique $!_{G'} \in \mathcal{D}(D)(G', \top) = \mathsf{cliques}(\Pi_{x:D}(G' \multimap$

$\top)(x)) = \mathsf{cliques}(\Pi_{x:D}\emptyset) = \mathsf{cliques}(\emptyset) = \{\emptyset\}$. Let $G', G \in \mathsf{ob}(\mathcal{D}(D))$. Then, we have projections $(G'\&G \xrightarrow{\mathsf{fst}} G') = \{(x, f, f) \mid x \text{ is minimal such that } f \in G'(x) \subseteq G'\&G(x)\}$ and $(G'\&G \xrightarrow{\mathsf{snd}} G) = \{(x, g, g) \mid x \text{is minimal such that } \ g \in G(x) \subseteq G'\&G(x)\}$. Given $H \xrightarrow{f} G'$ and $H \xrightarrow{g} G$, we define $(H \xrightarrow{\langle f,g\rangle} G'\&G) := f \cup g$. This is a clique in $\Pi_{x:D}(H \multimap G'\&G)(x)$, as $(x, h, e) \frown (x', h', e') = x \supset\!\subset_{FD} x' \Rightarrow (h \supset\!\subset_{H(x)} h' \Rightarrow e \frown_{G'(x)\&G(x)} e')$. Now, we have three cases: if both e and e' are in $G'(x)$, the fact that f is a clique takes care of the coherence and, similarly, if both e and e' are in $G(x)$, g does this. Finally, if $e \in G'(x)$ and $e' \in G(x)$ (or vice versa), the definition of coherence in $G'(x)\&G(x)$ makes sure that $e \supset\!\subset_{G'(x)\&G(x)} e'$, so $(x, h, e) \supset\!\subset (x', h', e')$.

We verify the rules for $\Sigma^{\otimes}_{!}$-types.

Claim. *We have a left adjoint*

$$\mathcal{D}(\Sigma_{x:D} UG'(x)) \xrightarrow{\Sigma^{\otimes}_{F(UG')}} \mathcal{D}(D)$$

to the change of base functor

$$\mathcal{D}(D) \xrightarrow{-\{\mathbf{p}_{D,UG'}\}} \mathcal{D}(\Sigma_{x:D} UG'(x)).$$

Moreover, this satisfies Frobenius reciprocity,

$$\Sigma^{\otimes}_{F(UG')}(\mathbf{p}_{D,UG'}; G \otimes H) \cong G \otimes \Sigma^{\otimes}_{F(UG')} H,$$

and the left Beck-Chevalley condition.

Proof. We define, on objects,

$$\Sigma^{\otimes}_{F(UG')}(G)(x) := \Sigma^{\otimes}_{F(s:UG'(x))} G(x, s)$$

and, on morphisms,

$$\begin{aligned}
&\Sigma^{\otimes}_{F(UG')}(G \xrightarrow{\sigma} H) := \\
&\{(x, (s, g), (s, h)) \in \Pi^{\multimap}_{F(x\in D)} \left(\Sigma^{\otimes}_{F(s\in UG'(x))} G(x, s)\right) \multimap \left(\Sigma^{\otimes}_{F(s'\in UG'(x))} H(x, s')\right) \\
&\mid ((x, s), (g, h)) \in \sigma\}.
\end{aligned}$$

We verify that this, indeed, defines a clique in

$$\Pi^{\multimap}_{F(x:D)}(\Sigma^{\otimes}_{F(s:UG'(x))}G(x,s)) \multimap (\Sigma^{\otimes}_{F(s:UG'(x))}H(x,s)) :$$

$$\begin{aligned}
&(x,(s,g),(s,h)) \frown (x',(s',g'),(s',h')) \\
&\equiv x \subset x' \Rightarrow ((s,g),(s,h)) \frown ((s',g'),(s',h')) \\
&\equiv x \subset x' \Rightarrow ((s,g) \subset (s',g') \Rightarrow (s,h) \frown (s',h')) \\
&\equiv x \subset x' \Rightarrow ((s \subset s' \wedge g \subset g') \Rightarrow (s,h) \frown (s',h')) \\
&\equiv x \smile x' \vee s \smile s' \vee g \smile g' \vee (s,h) \frown (s',h').
\end{aligned}$$

Now, as σ is a clique in $\Pi^{\multimap}_{F((x,s):\Sigma_{x:D}UG'(x))}G(x,s) \multimap H(x,s)$, we have that

$$x \smile x' \vee s \smile s' \vee g \smile g' \vee h \frown h'.$$

Suppose that not $x \smile x' \vee s \smile s' \vee g \smile g'$ (then, in particular, $s \subset s'$). We have to show that $h \frown h' \Rightarrow (s,h) \frown (s',h')$. This clearly holds as $s \subset s'$. We conclude that $\Sigma^{\otimes}_{F(UG')}(\sigma)$ is a clique.

$\Sigma^{\otimes}_{F(UG')}$ clearly respects identities and composition so we conclude it is a well-defined functor.

We verify that adjointness condition

$$\mathcal{D}(\Sigma_{x:D}UG'(x))(G, \mathbf{p}_{D,UG'}; H) \cong \mathcal{D}(D)(\Sigma^{\otimes}_{F(UG')}G, H).$$

The LHS is equal to

$$\mathsf{cliques}(\Pi^{\multimap}_{F((x,s):\Sigma_{x:D}UG'(x))}(G(x,s) \multimap H(x))),$$

while RHS is equal to

$$\mathsf{cliques}(\Pi^{\multimap}_{F(x:D)}((\Sigma^{\otimes}_{F(s:UG'(x))}(G(x,s))) \multimap H(x))).$$

Now,

$$\begin{aligned}
&\Pi^{\multimap}_{F((x,s):\Sigma_{x:D}UG'(x))}(G(x,s) \multimap H(x)) \\
&= \{((x,s),(g,h)) \mid x \in FD, s \in UG'(x), g \in G(x,s), h \in H(x)\} \\
&\cong \{(x,((s,g),h)) \mid x \in FD, s \in UG'(x), g \in G(x,s), h \in H(x)\} \\
&= \Pi^{\multimap}_{F(x:D)}((\Sigma^{\otimes}_{F(s:UG'(x))}G(x,s)) \multimap H(x)).
\end{aligned}$$

Moreover, $((x,s),(g,h)),((x',s'),(g',h')) \in \Pi^{\multimap}_{F((x,s):\Sigma_{x:D}UG'(x))}(G(x,s) \multimap H(x)))$ are related via $\frown$ iff any of the following equivalent conditions hold

$$\begin{aligned}
(x \subset x' \wedge s \subset s') \Rightarrow (g,h) \frown (g',h') &\equiv x \smile x' \vee s \smile s' \vee (g \subset g' \Rightarrow h \frown h') \\
&\equiv x \smile x' \vee s \smile s' \vee g \smile g' \vee h \frown h',
\end{aligned}$$

while $(x,((s,g),h) \frown (x',((s',g'),h'))$ in $\Pi^{\multimap}_{F(x:D)}(\Sigma^{\otimes}_{F(s:UG'(x))}(G(x,s)) \multimap H(x))$ iff any of the following equivalent conditions hold

$$\begin{aligned}
x \subset x' \Rightarrow ((s,g),h) \frown ((s',g'),h') &\equiv x \subset x' \Rightarrow ((s,g) \subset (s',g') \Rightarrow h \frown h') \\
&\equiv x \subset x' \Rightarrow ((s \subset s' \wedge g \subset g') \Rightarrow h \frown h') \\
&\equiv x \smile x' \vee s \smile s' \vee g \smile g' \vee h \frown h'.
\end{aligned}$$

We see that the conditions on both sides coincide.

We can therefore take the canonical bijection between both sets of vertices to induce an isomorphism of coherence spaces, hence a bijection of cliques.

Furthermore, it is immediately obvious from the definitions that Frobenius reciprocity holds:

$$\begin{aligned}
\Sigma^{\otimes}_{F(UG')}(G\{\mathbf{p}_{D,UG'}\} \otimes H) &= x \mapsto \Sigma^{\otimes}_{F(s:UG'(x))}G(x) \otimes H(x,s) \\
&\cong x \mapsto G(x) \otimes \Sigma^{\otimes}_{F(s:UG'(x))}H(x,s) \\
&= G \otimes \Sigma^{\otimes}_{F(UG')}H.
\end{aligned}$$

Finally, the Beck-Chevalley condition trivially holds, as the change of base functors act by precomposition. □

We verify the rules for Π-types.

Claim. *We have a right adjoint*

$$\mathcal{D}(\Sigma_{x:D} UG'(x)) \overset{\Pi^{\multimap}_{F(UG')}}{\longrightarrow} \mathcal{D}(D)$$

to the change of base functor

$$\mathcal{D}(D) \overset{-\{\mathbf{p}_{D,UG'}\}}{\longrightarrow} \mathcal{D}(\Sigma_{x:D} UG'(x)),$$

satisfying the right Beck-Chevalley condition.

Proof. We define, on objects,

$$\Pi^{\multimap}_{F(UG')}(G)(x) := \Pi^{\multimap}_{F(s:UG'(x))} G(x,s)$$

and, on morphisms,

$$\begin{aligned}
&\Pi^{\multimap}_{F(UG')}(G \overset{\sigma}{\longrightarrow} H) := \\
&\{(x,(s,g),(s,h)) \in \Pi^{\multimap}_{F(x\in D)} \left(\Pi^{\multimap}_{F(s\in UG'(x))} UG(x,s)\right) \multimap \left(\Pi^{\multimap}_{F(s'\in UG'(x))} UH(x,s')\right) \\
&\mid ((x,s),(g,h)) \in \sigma\}.
\end{aligned}$$

We verify that this, indeed, defines a clique in

$$\Pi^{\multimap}_{F(x:D)}(\Pi^{\multimap}_{F(s:UG'(x))} G(x,s)) \multimap (\Pi^{\multimap}_{F(s:UG'(x))} H(x,s)) :$$

$$\begin{aligned}
&(x,(s,g),(s,h)) \frown (x',(s',g'),(s',h')) \\
&\equiv x \subset x' \Rightarrow ((s,g),(s,h)) \frown ((s',g'),(s',h')) \\
&\equiv x \subset x' \Rightarrow ((s,g) \subset (s',g') \Rightarrow (s,h) \frown (s',h')) \\
&\equiv x \subset x' \Rightarrow ((s,g) \subset (s',g') \Rightarrow (s \subset s' \Rightarrow h \frown h')) \\
&\equiv x \smile x' \vee (s,g) \smile (s',g') \vee s \smile s' \vee h \frown h'.
\end{aligned}$$

Now, as σ is a clique in $\Pi^{\multimap}_{F((x,s):\Sigma_{x:D} UG'(x))} G(x,s) \multimap H(x,s)$, we have that

$$x \smile x' \vee s \smile s' \vee g \smile g' \vee h \frown h'.$$

Suppose that not $x \smile x' \vee s \smile s' \vee h \frown h'$ (then, in particular, $s \subset s'$). We have to show that $(s \subset s' \wedge g \smile g') \Rightarrow (s,g) \smile (s',g')$. For this, note that $g \smile g'$ implies

that $g \neq g'$ hence $(s, g) \neq (s', g')$, so an equivalent thing to prove would be that $(s \subset s' \wedge g \supset g') \Rightarrow \neg((s, g) \frown (s', g'))$, which is $(s \subset s' \wedge \neg(g \subset g')) \Rightarrow \neg(s \subset s' \Rightarrow g \frown g')$ by definition of $\frown$ on $\Pi^{\multimap}_{F(-)}$-types, which clearly holds. We conclude that $\Pi^{\multimap}_{F(UG')}(\sigma)$ is a clique.

$\Pi^{\multimap}_{F(UG')}$ clearly respects identities and composition so we conclude it is a well-defined functor.

We verify the adjointness condition

$$\mathcal{D}(\Sigma_{x:D} UG'(x))(\mathbf{p}_{D,UG'}; G, H) \cong \mathcal{D}(D)(G, \Pi^{\multimap}_{F(UG')} H).$$

The LHS is equal to

$$\mathsf{cliques}(\Pi^{\multimap}_{F((x,s):\Sigma_{x:D} UG'(x))}(G(x) \multimap H(x, s))),$$

while RHS is equal to

$$\mathsf{cliques}(\Pi^{\multimap}_{F(x:D)}((G(x)) \multimap \Pi^{\multimap}_{F(s:UG'(x))} H(x, s))).$$

Now,

$$\begin{aligned}
&\Pi^{\multimap}_{F((x,s):\Sigma_{x:D} UG'(x))}(G(x) \multimap H(x, s)) \\
&= \{((x, s), (g, h)) \mid x \in FD, s \in UG'(x), g \in G(x), h \in H(x, s)\} \\
&\cong \{(x, (g, (s, h))) \mid x \in FD, s \in UG'(x), g \in G(x), h \in H(x, s)\} \\
&= \Pi^{\multimap}_{F(x:D)}((G(x)) \multimap \Pi^{\multimap}_{F(s:UG'(x))} H(x, s)).
\end{aligned}$$

Moreover, $((x, s), (g, h)),((x', s'), (g', h')) \in \Pi^{\multimap}_{F((x,s):\Sigma_{x:D} UG'(x))}(G(x) \multimap H(x, s)))$ are related via $\frown$ iff any of the following equivalent conditions hold

$$\begin{aligned}
(x \subset x' \wedge s \subset s') \Rightarrow (g, h) \frown (g', h') &\equiv x \supset x' \vee s \supset s' \vee (g \subset g' \Rightarrow h \frown h') \\
&\equiv x \supset x' \vee s \supset s' \vee g \supset g' \vee h \frown h',
\end{aligned}$$

while in $\Pi^{\multimap}_{F((x,s):\Sigma_{x:D} UG'(x))}(G(x) \multimap \Pi^{\multimap}_{F(s:UG'(x))} H(x, s))$ we have that $(x, (g, (s, h)) \frown (x', (g', (s', h'))))$ iff any of the following equivalent conditions hold

$$\begin{aligned}
x \subset x' \Rightarrow (g, (s, h)) \frown (g', (s', h')) &\equiv x \subset x' \Rightarrow (g \subset, g') \Rightarrow (s, h) \frown (s', h')) \\
&\equiv x \subset x' \Rightarrow (g \subset, g') \Rightarrow (s \subset s' \Rightarrow h \frown h') \\
&\equiv x \supset x' \vee s \supset s' \vee g \supset g' \vee h \frown h'.
\end{aligned}$$

We see that the conditions on both sides coincide. We can therefore take the canonical bijection between both sets of vertices to induce an isomorphism of coherence spaces, hence a bijection of cliques.

Finally, the Beck-Chevalley condition trivially holds, as the change of base functors act by precomposition. □

We verify the rules for !-types. Seeing that we already have $\Sigma^{\otimes}_{F(-)}$-types and I-types, we know we can construct !-types as $\Sigma^{\otimes}_{F(-)}I$. We also give a direct proof, however, as it may provide more insight in the definition of the exponential.

Claim. *The comprehension functors U_D have a strong monoidal left adjoint F_D.*

Proof. We define $F_D(\mathbf{p}_{D,UG'}) := G'; U; F = G'; !$. Note that is well-defined as we can construct $G'; U$ from $\mathbf{p}_{D,UG'}$ as $x \mapsto \mathbf{p}^{-1}_{D,UG'}(x)$. Moreover, $G'; !$ is a type family, as, obviously, $\mathsf{cliques}_{\mathsf{fin}}(\bigcup_i A_i) = \bigcup_i \mathsf{cliques}_{\mathsf{fin}}(A_i)$, for a directed family $(A_i)_i$, and $\mathsf{cliques}_{\mathsf{fin}}(A \cap B) = \mathsf{cliques}_{\mathsf{fin}}(A) \cap \mathsf{cliques}_{\mathsf{fin}}(B)$, where we write $\mathsf{cliques}_{\mathsf{fin}}(A) :=\!A$ to emphasise that we are taking the coherence space of finite cliques.

This definition extends to morphisms between objects of the form $\mathbf{p}_{D,UG'}$. Indeed, we note that a morphism $\mathbf{p}_{D,UG'} \xrightarrow{f} \mathbf{p}_{D,UG}$ restricts to stable functions $UG'(x) \xrightarrow{f_x} UG(x)$ for all $x \in D$. We define $F_D(f) \in \mathsf{cliques}(\Pi^{\multimap}_{F(x:D)}(!G'(x) \multimap !G(x)))$ as $\{(x,(s,t)) \mid x \in D, (s,t) \in F(f_x)\ \forall_{x' \leq x}(s,t) \in f_{x'} \Rightarrow x' = x\}$. This is a clique as

$$\begin{aligned}(x,(s,t)) \frown (x',(s',t')) &= x \subset x' \Rightarrow (s,t) \frown (s',t') \\ &= x \subset x' \Rightarrow (s,t) \neq (s',t') \quad (\text{as } f_x \text{ is a clique}) \quad .\end{aligned}$$

We finally take the unique strong monoidal extension to obtain a (strong monoidal) functor from $\mathcal{C}(D)$.

The condition that F_D is compatible with change of base (precomposition) follows immediately because of the pointwise definition of F_D. □

Finally, we verify that we have $\mathsf{Id}^{\otimes}_{!}$-types.

Claim. *Our model supports (intensional) $\mathsf{Id}^{\otimes}_{!}$-types.*

Proof. We verify the formation, introduction, elimination and β-rules for $\mathsf{Id}^{\otimes}_{!}$-types.

$\mathsf{Id}^{\otimes}_{!}$-F Let $a, a' : I \longrightarrow G' \in \mathcal{D}(D)$. Then, we define a type family over D:

$$\mathsf{Id}^{\otimes}_{!G'}(a,a')(x) := \{b \in !G'(x) \mid b \leq \mathsf{fun}(a)(x), \mathsf{fun}(a')(x)\} \subseteq !G'(x),$$

with the induced coherence relation.

The fact that this is a continuous function $D \overset{\mathsf{Id}^{\otimes}_{!G'}(a,a')}{\longrightarrow} \mathcal{U}$ is a direct consequence of its definition as a subfamily of a continuous family $!G'$ with a continuous bound $(\mathsf{fun}(a)(x) \wedge \mathsf{fun}(a)(x'))$. The exact same argument gives stability:

$$\begin{aligned}
&\mathsf{Id}^{\otimes}_{!G'}(a,a')(x_1) \wedge \mathsf{Id}^{\otimes}_{!G'}(a,a')(x_2) \\
&= \{b \in !G'(x_1) \mid b \leq \mathsf{fun}(a)(x_1), \mathsf{fun}(a')(x_1)\} \cap \\
&\quad \{b \in !G'(x_2) \mid b \leq \mathsf{fun}(a)(x_2), \mathsf{fun}(a')(x_2)\} \\
&= \{b \in !G'(x_1) \cap !G'(x_2) \mid b \leq \mathsf{fun}(a)(x_1) \wedge \mathsf{fun}(a)(x_2) \wedge \\
&\qquad\qquad \mathsf{fun}(a')(x_1) \wedge \mathsf{fun}(a')(x_2)\} \\
&= \{b \in !G'(x_1 \wedge x_2) \mid b \leq \mathsf{fun}(a)(x_1 \wedge x_2) \wedge \mathsf{fun}(a')(x_1 \wedge x_2)\} \\
&= \mathsf{Id}^{\otimes}_{!G'}(a,a')(x_1 \wedge x_2).
\end{aligned}$$

$\mathsf{Id}^{\otimes}_{!}$-I For $a : I \longrightarrow G' \in \mathcal{D}(D)$, we define, for $x \in D$,

$$\mathsf{refl}(a) := \{(x,b) \mid x \in D, b \in \mathsf{Id}^{\otimes}_{!G'}(a,a)(x), \forall_{x' \leq x} b \in \mathsf{Id}^{\otimes}_{!G'}(a,a)(x') \Rightarrow x' = x\}.$$

This is easily verified to be a clique in $\Pi^{\multimap}_{F(x:D)} \mathsf{Id}^{\otimes}_{G'}(a,a)(x)$, as $a(x)$ is a clique in $G'(x)$, and hence a morphism $I \overset{\mathsf{refl}(a)}{\longrightarrow} \mathsf{Id}^{\otimes}_{!G'}(a,a) \in \mathcal{D}(D)$.

$\mathsf{Id}^{\otimes}_{!}$-E Suppose we're given

- $G' \in \mathsf{ob}\mathcal{D}(D))$
- $C \in \mathsf{ob}\mathcal{D}(\Sigma_{(y,x,x'):\Sigma_{y:D} UG'(y) \times UG'(y)} U \mathsf{Id}^{\otimes}_{!G'}(x,x')(y)$
- $c \in \mathcal{D}(\Sigma_{y:D} UG')(\Xi, C\{\langle \mathsf{id}_D, \mathsf{id}_{G'}, \mathsf{id}_{G'}, \mathsf{refl}(\mathsf{id}_{G'})\rangle\})$
- $a, a' \in \mathcal{D}(D)(I, G')$
- $p \in \mathcal{D}(D)(I, \mathsf{Id}^{\otimes}_{G'}(a,a'))$.

We construct

$$(\text{let } (a, a', p) \text{ be } (\mathsf{id}_{G'}, \mathsf{id}_{G'}, \mathsf{refl}(\mathsf{id}_{G'})) \text{ in } c) \in \mathcal{D}(D)(\Xi, C\{\langle \mathsf{id}_C, a, a', p\rangle\}).$$

(as a dependent stable function $\in \Pi_{x:D} U(\Xi \multimap C\{a, a', p\})$) by defining

$$\mathsf{fun}(\text{let } (a, a', p) \text{ be } (\mathsf{id}_{G'}, \mathsf{id}_{G'}, \mathsf{refl}(\mathsf{id}_{G'})) \text{ in } c)(y)(\xi) := \mathsf{fun}(d)(y, \bigcup \mathsf{fun}(p)(y))(\xi).$$

$\mathsf{Id}^{\otimes}_{!}$-$\beta$ We calculate

$$\begin{aligned}
&\mathsf{fun}(\text{let } (a, a, \mathsf{refl}(a)) \text{ be } (\mathsf{id}_{G'}, \mathsf{id}_{G'}, \mathsf{refl}(\mathsf{id}_{G'})) \text{ in } c)(y)(\xi) \\
&= \mathsf{fun}(c)(y, \bigcup \mathsf{fun}(\mathsf{refl}(a))(y))(\xi) \\
&= \mathsf{fun}(c)(y, \mathsf{fun}(a)(y))(\xi) \\
&= \mathsf{fun}(c\{\langle \mathsf{id}_D, a\rangle\})(y)(\xi).
\end{aligned}$$

□

Finally, we note that $1 = \bot$ is a dualising object: $(-) \multimap \bot = (-)^{\bot}$ is an involution, as this is the case pointwise. This means we have a model of classical linear dependent type theory. □

While we can, in fact, define additive Id-types: $\mathsf{Id}^{\&}_{G'}(a, a')(x) := \mathsf{fun}(a)(x) \cap \mathsf{fun}(a')(x) \subseteq G'(x)$, the interpretation of $\Sigma^{\&}$ in the model turns out to be problematic.

Theorem 3.5.12 (Absence of $\Sigma^{\&}$-Types)**.** *The model does not support $\Sigma^{\&}$-types.*

Proof. Let A be the coherence space I. Then, $UA = \{\emptyset \leq \{0\}\}$. Let B be the stable continuous family of coherence spaces indexed by UA where $B(\emptyset) := \top$ and $B(\{0\}) := I$. In that case, we note that $\Sigma_{UA} UB = \{\langle \emptyset, \emptyset\rangle \leq \langle \{0\}, \emptyset\rangle \leq \langle \{0\}, \{0\}\rangle\}$. Now, we claim that there is no coherence space $\Sigma^{\&}_A B$ such that $U\Sigma^{\&}_A B \cong \Sigma_{UA} UB$. To see this, note that UC always has strictly more elements than the sum of the number of edges and vertices in C. Seeing that $\Sigma_{UA} UB$ has 3 elements, that would leave only three possibilities for $\Sigma^{\&}_A B$: $\top$, I and $I \oplus I$. However, we have $U\top = \{\emptyset\}$, $UI = \{\emptyset \leq \{0\}\}$ and $U(I \oplus I) = \{\emptyset \leq \{0\}, \{1\}\}$, none of which is isomorphic to $\Sigma_{UA} UB$. We conclude that no suitable $\Sigma^{\&}_A B$ exists. □

We see that the category $\mathsf{Coh}_!$ of coherent qualitative domains and stable functions is too restrictive to admit the interpretation of Σ-types. To interpret those, we have to pass to a larger category of domains, like the category Stable of all Scott predomains with pullbacks. There, however, we face the usual problem that we cannot interpret Π-types (or even simple function types; this was the raison d'être for dI-domains). We ask the reader to compare this to our discussion in section 3.4 about finding a sweet spot between the co-Kleisli category and co-Eilenberg-Moore category where we can interpret both Σ- and Π-types. As is shown in [84], dI-domains are such a sweet spot. In future work, we plan to demonstrate how these arise as the co-Kleisli category of another model of linear logic, a certain finitary variation on the linear information systems of [85].

It may be that all games are silly. But then, so are humans.

— Roibéard Ó Floinn

4 Games for Dependent Types

DTT can be seen as the extension of the simple λ-calculus along the Curry-Howard correspondence from a proof calculus for (intuitionistic) propositional logic to one for predicate logic. It forms the basis of many proof assistants, like NuPRL, LEGO and Coq, and is increasingly being considered as an expressive type system for programming, as implemented in e.g. ATS, Cayenne, Epigram, Agda and Idris [86] and with even Haskell approaching its expressive power with the addition of GADTs [87].

A recent source of enthusiasm in this field is homotopy type theory (HoTT), which refers to an interpretation of DTT into abstract homotopy theory [88] or, conversely, an extension of DTT that is sufficient to reproduce significant results of homotopy theory [89]. In practice, the latter means DTT with Σ-, Π-, Id-types (corresponding to existential and universal quantifiers and identity predicates, respectively, through the Curry-Howard correspondence), a universe (roughly, a type of types) satisfying the **univalence axiom**, and certain higher inductive types (playing the rôle of ground types whose towers of iterated identity types behave like the homotopy types of certain spaces). The univalence axiom is an extensionality principle which implies the axiom of function extensionality [89].

Game semantics provides a unified framework for intensional, computational semantics of various type theories, ranging from pure logics [90] to programming

languages [22, 58, 91, 92] with a variety of effects (e.g. non-local control [61], state [63, 64, 93], non-determinism [60], probability [94], dynamically generated local names [95]) and evaluation strategies [96].

A game semantics for DTT has, surprisingly, so far been absent, perhaps because of the naturally effectful character of game semantics. We hope to fill this gap in the present chapter. Our hope is that such a semantics will provide an alternative analysis of the implications of the subtle shades of intensionality that arise in the analysis of DTT [20, 27]. Moreover, the game semantics of DTT is based on very different, one might say orthogonal intuitions to those of the homotopical models: temporal rather than spatial, and directly reflecting the structure of computational processes. One goal, to which we hope this work will be a stepping stone, is a game semantics of HoTT doing justice to both the spatial and temporal aspects of identity types. Indeed, such an investigation might even lead to a computational interpretation of the univalence axiom which has long been missing, although a significant step in this direction was recently taken by the constructive cubical sets model of HoTT [97]. Finally, a game semantics for DTT should hopefully shed light on how dependent types can interact with effects.

We interpret dependent types as families of games indexed by strategies. We adapt the viewpoint of the game semantics of system F of [92] to describe the Π-type, capturing the intuitive idea that the specialisation of a term at type $\Pi_{x:A}B$ to a specific instance $B[a/x]$ is the responsibility solely of the context that provides the argument a of type A; in contrast, any valid term of $\Pi_{x:A}B$ has to operate within the constraints enforced by the context. Our definition draws its power from the fact that in a game semantics, these constraints are enforced not only on completed computations, but also on incomplete ones that arise when a term interacts with its context. The temporal character of game semantics results in a model with strikingly different properties from existing models like the domain semantics [83].

In this chapter, we describe a game theoretic model of DTT with 1-, Σ-, Π- and intensional Id-types, where (lists of dependent) (call-by-name) AJM-games interpret types and (lists of) deterministic history-free well-bracketed winning strategies on

games of dependent functions interpret terms. We next specialize to the semantic type hierarchy formed by the 1-, Σ-, Π-, and Id-constructions and substitution over finite dependent games. This gives a model of DTT which additionally supports finite inductive type families. Our model has the following key properties.

- The place of the Id-types in the intensionality spectrum (in either model) compares as follows with the domain semantics with totality and with HoTT.

	Domains	HoTT	Games
Failure of Equality Reflection	✓	✓	✓
Streicher [27] Intensionality Criteria ($I1$) and ($I2$)	✓	✓	✓
Streicher Intensionality Criterion ($I3$)	✗	✗	✓
Failure of Function Extensionality (FunExt)	✗	✗	✓
Failure of Uniqueness of Identity Proofs (UIP)	✗	✓	✗

- We show that the smaller model faithfully interprets $\mathsf{DTT}_{\mathsf{CBN}}$. Moreover, it is fully complete at the types A which do not involve Id in their construction or which involve one strictly positive Id-type as a subformula, if we add the $\mathsf{Ty\text{-}Ext}$ rule for types $x : A \vdash B \ \ \mathsf{type}$. Full completeness for the full type hierarchy remains to be investigated but seems plausible. In contrast, the domain theoretic model of [83] is not (fully) complete or faithful.

- It can be extended from a model of pure type theory to, additionally, interpreting various effects when we drop some of the conditions on strategies.

In section 4.1, we introduce a notion of dependent game and dependently typed strategy, together with a semantic equivalent $☺(-)$ of $(-)^T$, the syntactic translation of section 2.1.1.4 from $\mathsf{DTT}_{\mathsf{CBN}}$ to $\mathsf{STT}_{\mathsf{CBN}}$: a translation to simply typed game semantics. Although this almost gives a model of dependent type theory, we show that we cannot interpret Σ-types (or comprehension). Adding Σ-types formally, we next construct an interpretation of DTT in sections 4.2, 4.3 and 4.4, in the form of a category with families with Σ-, Π- and Id-types and finite inductive type families. Section 4.3 further characterises various intensionality properties of the Id-types. Soundness and faithfulness of the interpretation of DTT are finally proved in section 4.5, as the interpretation factors faithfully over the

faithful sound games interpretation of STT, as well as full completeness results which are obtained by a dependently typed modification of the definability proofs of [22, 56]. Finally, in section 4.6 we lift the various conditions on strategies which ensure purity of the computations they model and we draw lessons on the interaction between dependent types and effects.

Remark 4.0.1 (Related Publications)**.** *This chapter is based on [14, 15]. We have changed the interpretation of* Id*-types to make them compatible with effects. To give a uniform treatment for all classes of strategies, we have chosen to define a dependent game in the pure setting only on winning strategies. We have also slightly changed the equational theory* $\mathsf{DTT}_{\mathsf{CBN}}$ *which lets us simplify the completeness proof considerably. We believe the current presentation to be both simpler and more robust with respect to extensions to broader classes of types and terms. After we presented our game semantics for dependent types in [14], [98] provided an alternative game semantics for dependent type theory, while with very different motivations. Where our work is motivated by precisely characterising effectful (CBN) type theory (e.g. through completeness results) with the purpose of understanding dependently typed effectful programming, [98] seems to be interested exclusively in modelling pure type theory and providing a constructive foundation of mathematics.*

4.1 An Indexed Category of Dependent Games

Section 2.4 sketched how $\mathsf{Game}_!$ models simple cartesian type theory. In this chapter, we extend this to a model of dependent type theory. In this section, we first show how to equip $\mathsf{Game}_!$ with a notion of dependent type and we show how this leads to an indexed ccc $\mathsf{DGame}_!$ of dependent games and dependently typed strategies.

We define a poset $\mathsf{Game}_{\trianglelefteq}$ of games with

$$A \trianglelefteq B := (M_A = M_B) \ \wedge \ (\lambda_A = \lambda_B) \ \wedge \ (\mathsf{j}_A = \mathsf{j}_B) \ \wedge \ (P_A \subseteq P_B) \wedge$$
$$(W_A = W_B \cap P_A^{\infty}) \ \wedge \ \forall_{s,t \in P_B}(s \approx_A t \quad \Leftrightarrow \quad s \in P_A \ \wedge \ s \approx_B t).$$

Given a game C, we define the complete lattice $\mathsf{Sub}(C)$ as the poset of its $\trianglelefteq$-subgames. We note that, for $A, B \in \mathsf{Sub}(C)$, $A \trianglelefteq B \Leftrightarrow P_A \subseteq P_B$. We make the following simple observation that we shall refer to later.

Theorem 4.1.1. *We have a functor* $\mathsf{Game}_! \xrightarrow{\mathsf{Sub}} \mathsf{CjsLat}$ *to the category* CjsLat *of complete lattices and join-preserving functions.*

Proof. An element of $\mathsf{Sub}(C)$ is precisely specified by a $\approx_C$-closed prefix-closed subset of P_C, so we can compute joins and meets simply by unions and intersections. Given $A \xrightarrow{f} B \in \mathsf{Game}_!$ and $A' \trianglelefteq A$, we define

$$\mathsf{Sub}(f)(A') := \{s \in P_B \mid \exists_{t \in f} s \leq t \restriction_B \wedge \forall_i t \restriction_{!A} \restriction_i \in A'\}.$$

The result is clearly prefix-closed and closed under $\approx_B$, as f is closed under $\approx_{A \Rightarrow B}$. $\mathsf{Sub}(f)$ clearly preserves unions. □

This allows us to define a dependent game as follows, where $☺(B)$ can be seen as the semantic counterpart to the syntactic translation B^T of section 2.1.1.4.

Definition 4.1.2 (Dependent game)**.** *For a game A, we define the set* $\mathsf{ob}(\mathsf{DGame}_!(A))$ *of* ***games with dependency on*** *A as the set of pairs of a game $☺(B)$ (without dependency) and a function* $\mathsf{str}(A) \xrightarrow{B} \mathsf{Sub}(☺(B))$.

We note that $\mathsf{ob}(\mathsf{DGame}_!(I))$ is the set of pairs $(A(\bot), ☺(A))$ where $A(\bot) \trianglelefteq ☺(A)$, in which $\mathsf{ob}(\mathsf{Game}_!)$ embeds as the proper subset of diagonal elements (A, A). As the definability results of section 4.5 illustrate, we need the generality of $\mathsf{ob}(\mathsf{DGame}_!(I))$ to properly capture the notion of closed typed in $\mathsf{DTT}_{\mathsf{CBN}}$. Therefore, we define, more generally, for a pair $(A, ☺(A)) \in \mathsf{ob}(\mathsf{DGame}_!(I))$, $\mathsf{ob}(\mathsf{DGame}_!(A(\bot), ☺(A))) := \mathsf{ob}(\mathsf{DGame}_!(☺(A)))$. As an example, let us write $x : \mathsf{mm} \vdash \mathsf{dd} - \mathsf{mm}(x)$ for the (finite inductive) type family encoding the calendar of the year 1984 in dd-mm format. For instance, $\mathsf{dd} - \mathsf{mm}(02)$ has constructors 01-02,...,29-02. In this case, we note that for the purposes of the type theory the closed type $\mathsf{dd} - \mathsf{mm}(02)$ will behave differently from the closed (inductive) type $\{01\text{-}02, \ldots, 29\text{-}02\}$. Indeed, when eliminating from the former, our case analysis

contains (redundant) additional information on how to handle the all other days of the year as well. This example shows that for a substituted type like $\mathsf{dd}-\mathsf{mm}(02)$ the type theory still remembers information about the whole type family $\mathsf{dd}-\mathsf{mm}$ (like the constructors outside the particular fibre under consideration), hence our interpretation of closed types as pairs $A(\bot) \trianglelefteq \☺(A)$ of games rather than as single games. From now on, we write A for the pair $(A(\bot), \☺(A)) \in \mathsf{ob}(\mathsf{DGame}_!(I))$ and, more generally and slightly ambiguously, B for the pair $(B, \☺(B)) \in \mathsf{ob}(\mathsf{DGame}_!(A))$.

Writing $s \mapsto \overline{s}$ for the function from $P_{!\☺(A)}$ to the power set $\mathcal{P}P_{\☺(A)}$, inductively defined on the empty play, Opponent moves and Player moves, respectively, as $\epsilon \mapsto \emptyset, \quad s(i,a) \mapsto \overline{s}, \quad s(i,a)(i,b) \mapsto \overline{s(i,a)} \cup \{t \mid \exists_{s' \in \overline{s}} t \approx_{\☺(A)} s'ab\}$, we define the Π-game as follows.

Definition 4.1.3 (Π-Game)**.** *Given* $A \in \mathsf{ob}(\mathsf{DGame}_!(I))$, $B \in \mathsf{ob}(\mathsf{DGame}_!(A))$, *we define* $\Pi_A B \in \mathsf{ob}(\mathsf{DGame}_!(I))$ *with* $\☺(\Pi_A B) := \☺(A) \Rightarrow \☺(B)$ *and* $(\Pi_A B)(\bot)$ *carved out in* $\☺(\Pi_A B)$ *as follows*

$$
\begin{aligned}
P_{(\Pi_A B)(\bot)} := \{\epsilon\} \bigcup & \\
& \{sa \in P^{\mathsf{odd}}_{\☺(A)\Rightarrow\☺(B)} \mid s \in P^{\mathsf{even}}_{(\Pi_A B)(\bot)} \ \wedge \ \exists_{\overline{sa\restriction_{!\☺(A)}} \subseteq \tau \in \mathsf{str}(A(\bot))} sa \in P_{A(\bot)\Rightarrow B(\tau)} \} \bigcup \\
& \{sab \in P^{\mathsf{even}}_{\☺(A)\Rightarrow\☺(B)} \mid sa \in P^{\mathsf{odd}}_{(\Pi_A B)(\bot)} \wedge \\
& \qquad \forall_{\overline{sab\restriction_{!\☺(A)}} \subseteq \tau \in \mathsf{str}(A(\bot))} sa \in P_{A(\bot)\Rightarrow B(\tau)} \Rightarrow sab \in P_{A(\bot)\Rightarrow B(\tau)} \}.
\end{aligned}
$$

We note that we can make $\mathsf{DGame}_!(A)$ into a ccc[1] by defining I and $\&$ pointwise on dependent games B, while also performing the operation on $\☺(B)$, and by defining $P_{(B\Rightarrow C)(\sigma)} := \{s \in P_{B(\sigma)\Rightarrow C(\sigma)} \mid \exists_{\tau \in \mathsf{str}(B(\sigma))} \overline{s \restriction_{B(\sigma)}} \subseteq \tau \}$ and $\☺(B \Rightarrow C) := \☺(B) \Rightarrow \☺(C)$. This lets us define $\mathsf{DGame}_!(A)(B,C) := \mathsf{str}(\mathsf{O\text{-}sat}(\Pi_A(B \Rightarrow C)))$ with the obvious identity morphisms and composition, which we discuss later. Here, $\mathsf{ob}(\mathsf{DGame}_!(I)) \xrightarrow{\mathsf{O\text{-}sat}} \mathsf{ob}(\mathsf{Game}_!)$, sends $(A(\bot), \☺(A))$ to the game in which Opponent can play freely in $\☺(A)$ and Player has to respect the rules of the more restrictive

[1]Perhaps a more insightful way to think of this is as $\mathsf{DGame}_!(A)$ being obtained as a co-Kleisli category for a linear exponential comonad ! on a symmetric monoidal closed category $\mathsf{DGame}(A)$. Here, $\mathsf{DGame}(A)$ has the same objects as $\mathsf{DGame}_!(A)$ on which we define operations I, $\otimes$, $\multimap$ pointwise, while also performing the operation on $\☺(B)$, and $\☺(!B) := !\☺(B)$ while $(!B)(\sigma) := \{s \in P_{!(B(\sigma))} \mid \exists_{\tau \in \mathsf{str}(B(\sigma))} \overline{s} \subseteq \tau \}$. We define $\mathsf{DGame}(A)(B,C) := \mathsf{str}(\mathsf{O\text{-}sat}(\Pi_A(B \multimap C))$ with the obvious identity morphisms and composition. In fact, along similar lines, the games model of DTT that we present in this chapter can easily be modified to give a model of dDILL.

game $A(\bot)$ as long as Opponent does:

$$\begin{aligned} P_{\mathsf{O\text{-}sat}(A(\bot),☺(A))} :=& \{\epsilon\} \bigcup \\ & \{sa \in P^{\mathsf{odd}}_{☺(A)} \mid s \in P^{\mathsf{even}}_{\mathsf{O\text{-}sat}(A(\bot),☺(A))} \} \bigcup \\ & \{sab \in P^{\mathsf{even}}_{☺(A)} \mid sa \in P^{\mathsf{odd}}_{\mathsf{O\text{-}sat}(A(\bot),☺(A))} \wedge (sa \in P_{A(\bot)} \Rightarrow sab \in P_{A(\bot)})\}. \end{aligned}$$

Remark 4.1.4. *Note that, explicitly, the* ***game of dependent functions from*** A ***to*** B, $\mathsf{O\text{-}sat}(\Pi_A B)$, *is carved out in* $☺(A) \Rightarrow ☺(B)$, *as*

$$\begin{aligned} P_{\mathsf{O\text{-}sat}(\Pi_A B)} :=& \{\epsilon\} \bigcup \\ & \{sa \in P^{\mathsf{odd}}_{☺(A)\Rightarrow ☺(B)} \mid s \in P^{\mathsf{even}}_{\mathsf{O\text{-}sat}(\Pi_A B)} \} \bigcup \\ & \{sab \in P^{\mathsf{even}}_{☺(A)\Rightarrow ☺(B)} \mid sa \in P^{\mathsf{odd}}_{\mathsf{O\text{-}sat}(\Pi_A B)} \wedge \\ & \qquad \forall_{\overline{sab\restriction_{!☺(A)}} \subseteq \tau \in \mathsf{str}(A(\bot))} sa \in P_{A(\bot)\Rightarrow B(\tau)} \Rightarrow sab \in P_{A(\bot)\Rightarrow B(\tau)} \}. \end{aligned}$$

Indeed, this follows as $\overline{sab\restriction_{!☺(A)}} = \overline{sa\restriction_{!☺(A)}}$. *An explicit proof is given for the more general claim of theorem 4.2.3.*

Recall that we would like $☺(-)$ *to define a faithful functor to the world of simply typed games, being the semantic equivalent of* $(-)^T$. *It is for this reason that the game of dependent functions from* A *to* B *is saturated under all* O*-moves in* $☺(A) \Rightarrow ☺(B)$. *We present* $\mathsf{O\text{-}sat}$ *as a separate operation as this presentation will simplify the treatment of higher-order dependent functions in section 4.2.*

Following the mantra of game semantics for quantifiers [92], in $\mathsf{O\text{-}sat}(\Pi_A B)$, Opponent can choose a strategy τ on $A(\bot)$ while Player has to play in a way that is compatible with all choices of τ that have not yet been excluded. Similarly to the approach taken in the game semantics for polymorphism [92], we do not specify all of τ in one go, as this would violate "Scott's axiom" of continuity of computation. Instead, τ is gradually revealed, explicitly so by playing in $!☺(A)$ and implicitly by playing in $☺(B)$. That is, unless Opponent behaves naughtily, in the sense that there is no strategy τ on $A(\bot)$ which is consistent with her behaviour such that $s\restriction_{☺(B)}$ obeys the rules of $B(\tau)$. In case of such a naughty Opponent, any further play in $☺(A) \Rightarrow ☺(B)$ is permitted.

Remark 4.1.5. *In particular,* $\mathsf{DGame}_!(I)$ *is a ccc which has* $\mathsf{Game}_!$ *as a proper full subcategory. Note that the morphisms from* A *to* B *consist of the strategies on*

$\mathbb{N}_*$	days_*
	$*$
	364

$!\mathbb{N}_*$	days_*
	$*$
$(i,*)$	
$(i,1984)$	
	365

$!\mathbb{N}_*$	days_*
	$*$
$(i,*)$	
$(i,1985)$	
$(i+1,*)$	
$(i+1,1986)$	
	365

$!\mathbb{N}_*$	$!\mathsf{days}_*$	days_*	
		$*$	O
	$(i,*)$		P
	(i,m)		O
		m	P
			O
			P

Figure 4.1: Three plays in $\mathsf{O\text{-}sat}(\Pi_{\mathbb{N}_*}\mathsf{days}_*)$ and one in $\mathsf{O\text{-}sat}(\Pi_{\mathbb{N}_*}(\mathsf{days}_* \Rightarrow \mathsf{days}_*))$. The first as all years have > 364 days, the second as 1984 was a leap year, the third as Player can play any move in $☺(\mathsf{days}_*) = \mathbb{N}_{<366*}$ after Opponent has not played along a (history-free) strategy on $\mathbb{N}_*$ and the fourth as Opponent makes the move m first, after which Player can safely copy it. In the paired moves, Player chooses an (irrelevant) index i. For an interpretation of the plays in $\mathsf{O\text{-}sat}(\Pi_{\mathbb{N}_*}\mathsf{days}_*)$, imagine them as a dialogue between a departmental education manager (Opponent) and an academic (Player) where Player gets to choose for every year the date that she promises to have marked the students' end-of-year exam. A cheeky academic might try to suggest that she'll return the marked exams every year on the 366th day of the year without asking the manager for which year he wants to know the date. Clearly the manager should not accept this. This would correspond to a play $*365$, which is illegal as, by making the move 365, Player would exclude certain fibres (the non-leap years), which is a privilege only Opponent has.

$☺(A) \Rightarrow ☺(B)$ *for which Player plays along the rules of* $A(\bot) \Rightarrow B(\bot)$ *as long as Opponent does so and as long as there is a strategy on* $A(\bot)$ *which is consistent with her play.*

For a function $Y \xrightarrow{X} \mathsf{Set}$ to the class Set of sets, we define $☺(X_*) := (\bigcup_{y\in Y} X(y))_*$ and $X_*(y) := X(y)_*$. For an example of non-constant type dependency, write $\mathsf{days}(n) := \{m \mid \text{there are} > m \text{ days in the year } n\}$. Then $\mathsf{days}_*(n) := \mathsf{days}(n)_*$ is a game depending on $\mathbb{N}_*$ (with $\mathsf{days}_*(n) = \mathbb{N}_{<365*}$ or $\mathbb{N}_{<366*}$). Note that this will not correspond to a finite inductive type family as the fibres of the type are not disjoint. Then, figure 4.1 gives four examples of valid dependently typed strategies. The fourth example is especially important, as it generalises to a (derelicted) B-copycat on $\mathsf{O\text{-}sat}(\Pi_A(B \Rightarrow B))$ for arbitrary B, denoted $\mathbf{v}_{[A],[B]}$ in section 4.2. This motivates why Opponent can narrow down the fibre of B freely, while Player can only play without narrowing down the fibre further. To see that Player should not be able to narrow down the fibre of B, note that we do not want $f := \{\epsilon, *365\}$ to define a strategy on $\mathsf{O\text{-}sat}(\Pi_{\mathbb{N}_*}\mathsf{days}_*)$, as $1983; f = \{\epsilon, *365\} \notin \mathsf{str}(\mathsf{days}_*(1983))$.

We now obtain the following result, whose proof we omit, as we shall prove a more general result in theorem 4.2.5.

Theorem 4.1.6. *We obtain a strict indexed ccc*[2]

$$\mathsf{DGame}_!(I)^{op} \xrightarrow{(\mathsf{DGame}_!, -\{-\})} \mathsf{CCCat}$$

of dependent games, if we define

- *fibrewise objects* $\mathsf{ob}(\mathsf{DGame}_!(A)) := \{\mathsf{str}(☺(A)) \xrightarrow{B} \mathsf{Sub}(☺(B)) \mid ☺(B) \in \mathsf{ob}(\mathsf{Game}_!)\}$*;*
- *fibrewise hom-sets* $\mathsf{DGame}_!(A)(B,C) := \mathsf{str}(\mathsf{O\text{-}sat}(\Pi_A(B \Rightarrow C)))$*;*
- *fibrewise identities* $\mathsf{der}_B := \{s \in P^{\mathsf{even}}_{\mathsf{O\text{-}sat}(\Pi_A(B \Rightarrow B))} \mid \forall_{s' \in P^{\mathsf{even}}_{\mathsf{O\text{-}sat}(\Pi_A(B \Rightarrow B))}} s' \leq s \Rightarrow \exists_i s' \restriction_{!☺(B)} \restriction_i \approx_{☺(B)} s' \restriction_{☺(B)}\}$*;*
- *for* $B \xrightarrow{\tau} C \xrightarrow{\tau'} D \in \mathsf{DGame}_!(A)$*, we define the fibrewise composition* $B \xrightarrow{\tau^\dagger ;_A \tau'} D \in \mathsf{DGame}_!(A)$ *as* $\tau^\dagger ;_A \tau' := \mathsf{diag}^\dagger_A ; (\tau^\dagger \otimes \tau') ; \mathsf{comp}_{☺(B),☺(C),☺(D)}$*;*
- *given* $f \in \mathsf{Game}_!(A', A)$*, we define the change of base functor* $-\{f\}$*:* $B\{f\} \in \mathsf{ob}(\mathsf{DGame}_!(A'))$ *where* $B\{f\}(\sigma) := B(!(\sigma); f)$ *and* $☺(B\{f\}) := ☺(B)$ *and* $\tau\{f\} := f^\dagger ; \tau$*.*

Seeing that $\mathsf{DGame}_!(I)$ additionally has a terminal object I to interpret the empty context, we are well on our way to producing a model of dependent type theory: we only need to interpret context extension. This takes the form of the full and faithful comprehension axiom for $\mathsf{DGame}_!$, which states that for each $A \in \mathsf{ob}(\mathsf{DGame}_!(I))$ and $B \in \mathsf{ob}(\mathsf{DGame}_!(A))$ the following presheaf is representable

$$x \mapsto \mathsf{DGame}_!(\mathsf{dom}(x))(I, B\{x\}) : (\mathsf{DGame}_!(I)/A)^{op} \longrightarrow \mathsf{Set}$$

and that this induces a bijection $\mathsf{DGame}_!(A)(B,C) \cong \mathsf{DGame}_!(I)/A(\mathbf{p}_{A,B}, \mathbf{p}_{A,C})$. Unfortunately, this fails, as $\mathsf{DGame}_!(I)$ does not yield a sound interpretation of dependent contexts. Essentially, the problem is that we do not have **additive Σ-types**, appropriate generalisations $\Sigma^{\&}_A B$ of $\&$ to interpret dependent context extension in $\mathsf{DGame}_!(I)$ (c.f theorem 3.5.12).

[2]That is, a functor from $\mathsf{DGame}_!(I)^{op}$ to the 1-category CCCat of cartesian closed categories and strict cartesian closed functors.

Theorem 4.1.7. $\mathsf{DGame}_!$ *does not satisfy the full and faithful comprehension axiom.*

Proof. Let us write $\mathbb{B} := \{\mathsf{tt}, \mathsf{ff}\}$. Then, $\mathbb{B}_*$ is the usual flat game of Booleans. We can define a dependent game just_* over $\mathbb{B}_*$, where $☺(\mathsf{just}_*) := \mathbb{B}_*$, $\mathsf{just}_*(\mathsf{ff}) = \{\mathsf{ff}\}_*$ and $\mathsf{just}_*(\mathsf{tt}) = \{\mathsf{tt}\}_*$.

Then, note that the comprehension axiom (supposing that it holds) implies that, for any $C \in \mathsf{ob}(\mathsf{DGame}_!(\mathbb{B}_*))$,

$$\begin{aligned}
\mathsf{str}(\mathsf{O\text{-}sat}(\Pi_{\mathbb{B}_*}(\mathsf{just}_* \Rightarrow C))) &= \mathsf{DGame}_!(\mathbb{B}_*)(\mathsf{just}_*, C) \\
&\cong \mathsf{DGame}_!(I)/\mathbb{B}_*(\mathbf{p}_{\mathbb{B}_*,\mathsf{just}_*}, \mathbf{p}_{\mathbb{B}_*,C}) \\
&\cong \mathsf{DGame}_!(\Sigma^{\&}_{\mathbb{B}_*}\mathsf{just}_*)(I, C\{\mathbf{p}_{\mathbb{B}_*,\mathsf{just}_*}\}) \\
&= \mathsf{str}(\mathsf{O\text{-}sat}(\Pi_{\Sigma^{\&}_{\mathbb{B}_*}\mathsf{just}_*} C)),
\end{aligned}$$

where the second isomorphism is the full and faithfulness of the comprehension functor and the third isomorphism is the comprehension axiom (representability condition) and where $\Sigma^{\&}_{\mathbb{B}_*}\mathsf{just}_* \overset{\mathbf{p}_{\mathbb{B}_*,\mathsf{just}_*}}{\longrightarrow} \mathbb{B}_*$ is the representing object above for $A = \mathbb{B}_*$ and $B = \mathsf{just}_*$.

Now, taking $C(\tau) = I$ for all τ and $☺(C) = D$ for some game D, implies that $☺(\Sigma^{\&}_{\mathbb{B}_*}\mathsf{just}_*) \cong \mathbb{B}_*\&\mathbb{B}_*$. Indeed, we have a natural bijection $\mathsf{Game}_!(☺(\Sigma^{\&}_{\mathbb{B}_*}\mathsf{just}_*), D) = \mathsf{str}(☺(\Sigma^{\&}_{\mathbb{B}_*}\mathsf{just}_*) \Rightarrow D) = \mathsf{str}(\mathsf{O\text{-}sat}(\Pi_{\Sigma^{\&}_{\mathbb{B}_*}\mathsf{just}_*} C)) = \mathsf{str}(\mathsf{O\text{-}sat}(\Pi_{\mathbb{B}_*}\mathsf{just}_* \Rightarrow C)) = \mathsf{str}(\mathbb{B}_* \Rightarrow \mathbb{B}_* \Rightarrow D) = \mathsf{Game}_!(\mathbb{B}_*, \mathbb{B}_* \Rightarrow D) \cong \mathsf{Game}_!(\mathbb{B}_*\&\mathbb{B}_*, D)$, which according to the Yoneda lemma is induced by an isomorphism $☺(\Sigma^{\&}_{\mathbb{B}_*}\mathsf{just}_*) \cong \mathbb{B}_*\&\mathbb{B}_*$ in $\mathsf{Game}_!$.

According to theorem 4.1.1 this induces an isomorphism $\mathsf{Sub}(☺(\Sigma^{\&}_{\mathbb{B}_*}\mathsf{just}_*)) \cong \mathsf{Sub}(\mathbb{B}_*\&\mathbb{B}_*)$. Therefore, symmetry of just in tt and ff implies that there are only nine options for $(\Sigma^{\&}_{\mathbb{B}_*}\mathsf{just}_*)(\bot)$: $I\&I$, $I\&\emptyset_*$, $\emptyset_*\&I$, $\emptyset_*\&\emptyset_*$, $I\&\mathbb{B}_*$, $\mathbb{B}_*\&I$, $\emptyset_*\&\mathbb{B}_*$, $\mathbb{B}_*\&\emptyset_*$ and $\mathbb{B}_*\&\mathbb{B}_*$.

We take $C = \mathsf{just}_*$ in the bijection implied by the comprehension axiom above, to obtain $\mathsf{str}(\mathsf{O\text{-}sat}(\Pi_{\mathbb{B}_*}(\mathsf{just}_* \Rightarrow \mathsf{just}_*))) \cong \mathsf{str}(\mathsf{O\text{-}sat}(\Pi_{\Sigma^{\&}_{\mathbb{B}_*}\mathsf{just}_*}\mathsf{just}_*))$. We see that none of the nine options is satisfactory. Indeed, $I\&I$, $I\&\emptyset_*$, $\emptyset_*\&I$, $\emptyset_*\&\emptyset_*$, $\mathbb{B}_*\&I$ and $\mathbb{B}_*\&\emptyset_*$ would imply that the negation between the two copies of just_* is a member of the right hand side, but not the left hand side, which is a contradiction. Similarly,

$I\&I$, $I\&\emptyset_*$, $\emptyset_*\&I$, $\emptyset_*\&\emptyset_*$, $I\&\mathbb{B}_*$ and $\emptyset_*\&\mathbb{B}_*$ would imply that the negation between $\mathbb{B}_*$ and the second copy of just_* is a member of the right hand side, but not the left hand side, which is a contradiction. The last case of $\mathbb{B}_*\&\mathbb{B}_*$ also leads to a contradiction as it would restrict members of the right hand side to output tt in response to having been supplied with arguments tt and ff to the function upon request, while members of the left hand side would also be free to answer ff. □

This is a common problem we discussed in section 3.4. It also occurred for coherence space semantics, which is not surprising if we view game semantics as coherence space semantics extended in time. While we had a good candidate category Stable of !-coalgebras to extend $\mathsf{Coh}_!$, such an obvious candidate is not available for games. Section 3.4 suggested that such a suitable category of !-coalgebras may be **constructed** either by inductively closing the co-Kleisli category under Σ-types or by coinductively restricting the co-Eilenberg-Moore category to be closed under Π-types. As it is easier to get an explicit description of the former category, we construct a category of **context games** by formally closing $\mathsf{Game}_!$ under a notion of Σ-type. It is on this category that we base our model of dependent type theory.

4.2 A Category with Families of Context Games

All is not lost, however. In fact, we have almost translated the syntax of dependently typed equational logic into the world of games and strategies. The remaining generalisation, necessitated by the lack of additive Σ-types, is to dependent games depending on multiple (mutually dependent) games. We can produce a categorical model of $\mathsf{DTT}_{\mathsf{CBN}}$ out of the resulting structure by applying a so-called **category of contexts (Ctxt) construction**, which is precisely how one builds a categorical model from the syntax of dependent type theory [19, 20]. This construction can be seen as a way of making our indexed category satisfy the comprehension axiom, extending its base category by (inductively) adjoining (strong) Σ-types formally, analogous to the Fam-construction of [96] which adds formal coproducts. We

encourage the reader to view this closure in the light of section 3.4: as inductively closing the co-Kleisli category under Σ-types.

The problem which needs to be addressed is how to interpret dependent types and dependent functions of more identifiers. This is done through a notion of context game and a generalisation of the Π-game construction from the previous section.

Definition 4.2.1 (Context Game)**.** *We inductively define a* ***context game*** *to be a (finite) list* $[X_i]_{1\leq i\leq n}$ *where* X_i *is a* ***game with dependency on*** $[X_j]_{j<i}$*, i.e. a function* $\mathsf{str}(☺(X_1)) \times \cdots \times \mathsf{str}(☺(X_{i-1})) \cong \mathsf{str}(☺(X_1)\& \cdots \& ☺(X_{i-1})) \xrightarrow{X_i} \mathsf{Sub}(☺(X_i))$ *for some game* $☺(X_i)$*.*

To keep notation light, we sometimes abuse notation and write A for the context game $[A]$ of length 1.

Definition 4.2.2 (Dependent Π-game)**.** *For a game* X_{n+1} *depending on* $[X_i]_{i\leq n}$*, we define the game* $\Pi_{X_n} X_{n+1}$ *depending on* $[X_i]_{i\leq n-1}$ *by* $☺(\Pi_{X_n} X_{n+1}) := ☺(X_n) \Rightarrow ☺(X_{n+1})$ *from which* $(\Pi_{X_n} X_{n+1})(\sigma_1, \ldots, \sigma_{n-1})$ *is carved out as*

$$
\begin{aligned}
P_{(\Pi_{X_n} X_{n+1})(\sigma_1,\ldots,\sigma_{n-1})} = &\{\epsilon\} \bigcup \\
&\{sa \mid s \in P^{\mathsf{even}}_{(\Pi_{X_n} X_{n+1})(\sigma_1,\ldots,\sigma_{n-1})} \wedge \\
&\qquad \exists_{\overline{sa\restriction_{!☺(X_n)}} \subseteq \tau \in \mathsf{str}(X_n(\sigma_1,\ldots,\sigma_{n-1}))} sa \in P_{X_n(\sigma_1,\ldots,\sigma_{n-1}) \Rightarrow X_{n+1}(\sigma_1,\ldots,\sigma_{n-1},\tau)} \} \bigcup \\
&\{sab \mid sa \in P^{\mathsf{odd}}_{(\Pi_{X_n} X_{n+1})(\sigma_1,\ldots,\sigma_{n-1})} \wedge \\
&\qquad \forall_{\overline{sab\restriction_{!☺(X_n)}} \subseteq \tau \in \mathsf{str}(X_n(\sigma_1,\ldots,\sigma_{n-1}))} sa \in P_{X_n(\sigma_1,\ldots,\sigma_{n-1}) \Rightarrow X_{n+1}(\sigma_1,\ldots,\sigma_{n-1},\tau)} \Rightarrow \\
&\qquad sab \in P_{X_n(\sigma_1,\ldots,\sigma_{n-1}) \Rightarrow X_{n+1}(\sigma_1,\ldots,\sigma_{n-1},\tau)} \}.
\end{aligned}
$$

The following explicit characterisation of **the game** $\mathsf{O\text{-}sat}(\Pi_{X_1} \cdots \Pi_{X_n} X_{n+1})$ **of dependent functions of multiple arguments** will be useful later. Indeed, its strategies will represent dependent functions from $[X_i]_{1\leq i\leq n}$ to X_{n+1}.

Theorem 4.2.3. *Explicitly,* $(\Pi_{X_k} \cdots \Pi_{X_n} X_{n+1})(\sigma_1, \ldots, \sigma_{k-1})$ *can be inductively defined as the following subset of the plays of* $☺(X_k) \Rightarrow \cdots \Rightarrow ☺(X_{n+1})$*:*

$$
\begin{array}{l}
\{\epsilon\} \bigcup \\
\{sa \mid s \in P^{\mathsf{even}}_{(\Pi_{X_k} \cdots \Pi_{X_n} X_{n+1})(\sigma_1,\ldots,\sigma_{k-1})} \wedge \\
\qquad \exists_{\overline{sa\restriction_{!\mathsf{O}(X_k)}} \subseteq \sigma_k \in \mathsf{str}(X_k(\sigma_1,\ldots,\sigma_{k-1}))} \cdots \exists_{\overline{sa\restriction_{!\mathsf{O}(X_n)}} \subseteq \sigma_n \in \mathsf{str}(X_n(\sigma_1,\ldots,\sigma_{n-1}))} \\
\qquad sa \in P_{X_k(\sigma_1,\ldots,\sigma_{k-1}) \Rightarrow \cdots \Rightarrow X_{n+1}(\sigma_1,\ldots,\sigma_n)} \} \bigcup \\
\{sab \mid sa \in P^{\mathsf{odd}}_{(\Pi_{X_k} \cdots \Pi_{X_n} X_{n+1})(\sigma_1,\ldots,\sigma_{k-1})} \wedge \\
\qquad \forall_{\overline{sab\restriction_{!\mathsf{O}(X_k)}} \subseteq \sigma_k \in \mathsf{str}(X_k(\sigma_1,\ldots,\sigma_{k-1}))} \cdots \forall_{\overline{sab\restriction_{!\mathsf{O}(X_n)}} \subseteq \sigma_n \in \mathsf{str}(X_n(\sigma_1,\ldots,\sigma_{n-1}))} \\
\qquad sa \in P_{X_k(\sigma_1,\ldots,\sigma_{k-1}) \Rightarrow \cdots \Rightarrow X_{n+1}(\sigma_1,\ldots,\sigma_n)} \Rightarrow sab \in P_{X_k(\sigma_1,\ldots,\sigma_{k-1}) \Rightarrow \cdots \Rightarrow X_{n+1}(\sigma_1,\ldots,\sigma_n)} \}.
\end{array}
$$

As a consequence, the game of dependent functions $\mathsf{O\text{-}sat}(\Pi_{X_1} \cdots \Pi_{X_n} X_{n+1})$ *is carved out in* $\mathsf{O}(X_1) \Rightarrow \cdots \Rightarrow \mathsf{O}(X_n) \Rightarrow \mathsf{O}(X_{n+1})$ *as the set of plays*

$$
\begin{array}{l}
\{\epsilon\} \bigcup \\
\{sa \mid s \in P^{\mathsf{even}}_{\mathsf{O\text{-}sat}(\Pi_{X_1} \cdots \Pi_{X_n} X_{n+1})} \} \bigcup \\
\{sab \mid sa \in P^{\mathsf{odd}}_{\mathsf{O\text{-}sat}(\Pi_{X_1} \cdots \Pi_{X_n} X_{n+1})} \wedge \\
\qquad \forall_{\overline{sab\restriction_{!\mathsf{O}(X_1)}} \subseteq \tau_1 \in \mathsf{str}(X_1())} \cdots \forall_{\overline{sab\restriction_{!\mathsf{O}(X_n)}} \subseteq \tau_n \in \mathsf{str}(X_n(\tau_1,\ldots,\tau_{n-1}))} \\
\qquad sa \in P_{X_1() \Rightarrow \cdots \Rightarrow X_n(\tau_1,\ldots,\tau_{n-1}) \Rightarrow X_{n+1}(\tau_1,\ldots,\tau_n)} \Rightarrow sab \in P_{X_1() \Rightarrow \cdots \Rightarrow X_n(\tau_1,\ldots,\tau_{n-1}) \Rightarrow X_{n+1}(\tau_1,\ldots,\tau_n)} \}.
\end{array}
$$

That is, the set of plays where Opponent can do whatever she pleases, while Player can only move without further determining the fibre of any of $X_1, \ldots, X_{n+1}$ *as long as Opponent plays along compatible strategies* $\sigma_1, \ldots, \sigma_n$ *on* $\mathsf{O}(X_1), \ldots, \mathsf{O}(X_n)$*, in the sense that they extend to*

$$\langle \tau_1, \ldots, \tau_n \rangle \in \Sigma(\mathsf{str}(X_1), \ldots, \mathsf{str}(X_n)) := \{\langle \tau_1, \ldots, \tau_n \rangle \mid \tau_1 \in \mathsf{str}(X_1()) \wedge \cdots \wedge \tau_n \in \mathsf{str}(X_n(\tau_1, \ldots, \tau_{n-1}))\}$$

such that the current play obeys the rules of $X_1() \Rightarrow \cdots \Rightarrow X_n(\tau_1, \ldots, \tau_{n-1}) \Rightarrow X_{n+1}(\tau_1, \ldots, \tau_n)$.

Proof. We first note that the second claim follows straightforwardly from the first. Clearly, the proposed description of Opponent moves in the second claim is correct as, by definition of $\mathsf{O\text{-}sat}$, Opponent is free to move in $\mathsf{O}(X_1) \Rightarrow \cdots \Rightarrow \mathsf{O}(X_n) \Rightarrow \mathsf{O}(X_{n+1})$ in $\mathsf{O\text{-}sat}(\Pi_{X_1} \cdots \Pi_{X_n} X_{n+1})$. For Player moves, note that $\overline{sab\restriction_{!\mathsf{O}(X_i)}} = \overline{sa\restriction_{!\mathsf{O}(X_i)}}$ for all $1 \leq i \leq n$. Therefore, assuming the first claim holds, it follows that we are in one of two cases:

- Opponent has been naughty and has broken the rules of $\Pi_{X_1} \cdots \Pi_{X_n} X_{n+1}$. In this case, there are no $\overline{sab\restriction_{!\mathsf{O}(X_1)}} \subseteq \tau_1 \in \mathsf{str}(X_1()), \ldots, \overline{sab\restriction_{!\mathsf{O}(X_n)}} \subseteq \tau_n \in \mathsf{str}(X_n(\tau_1, \ldots, \tau_{n-1}))$ such that $sa \in X_1() \Rightarrow \cdots \Rightarrow X_{n+1}(\tau_1, \ldots, \tau_n)$.

In this case, Player is allowed to do whatever she wants according to our proposed description of the second claim as the hypotheses of the implication defining the incremental condition on Player moves are false. This matches, of course, the definition of $\mathsf{O\text{-}sat}(\Pi_{X_1} \cdots \Pi_{X_n} X_{n+1})$ from the description of $\Pi_{X_1} \cdots \Pi_{X_n} X_{n+1}$ of the first claim.

- Opponent has been nice and has followed the rules of $\Pi_{X_1} \cdots \Pi_{X_n} X_{n+1}$ (i.e. there are in fact such $\tau_1, \ldots, \tau_n$). In this case, Player has to keep obeying the rules of $\Pi_{X_1} \cdots \Pi_{X_n} X_{n+1}$ as well according to the definition of $\mathsf{O\text{-}sat}(\Pi_{X_1} \cdots \Pi_{X_n} X_{n+1})$. This matches our proposed description of the second claim.

For the first claim, the idea is that Opponent has to play precisely such that there is **some** compatible assignment of strategies $\sigma_k, \ldots, \sigma_n$ on $X_k, \ldots, X_n$ while Player has to play such that she does not exclude **any** such compatible assignment of strategies. Formally, we prove by induction that the proposed description of plays in $(\Pi_{X_k} \cdots \Pi_{X_n} X_{n+1})(\sigma_1, \ldots, \sigma_{k-1})$ coincides with its definition

$$
\begin{aligned}
&\{\epsilon\} \bigcup \\
&\{sa \mid s \in P^{\mathsf{even}}_{(\Pi_{X_k} \cdots \Pi_{X_n} X_{n+1})(\sigma_1,\ldots,\sigma_{k-1})} \ \wedge \ \exists_{\overline{sa\restriction_{!\odot(X_k)}} \subseteq \sigma_k \in \mathsf{str}(X_k(\sigma_1,\ldots,\sigma_{k-1}))} sa \restriction_{!\odot(X_{k+1}),\ldots,\odot(X_{n+1})} \in P_{(\Pi_{X_{k+1}} \cdots \Pi_{X_n} X_{n+1})(\sigma_1,\ldots,\sigma_k)} \\
&\wedge \ \exists_{\overline{sa\restriction_{!\odot(X_{k+1})}} \subseteq \sigma_{k+1} \in \mathsf{str}(X_{k+1}(\sigma_1,\ldots,\sigma_k))} sa \restriction_{!\odot(X_{k+2}),\ldots,\odot(X_{n+1})} \in P_{(\Pi_{X_{k+2}} \cdots \Pi_{X_n} X_{n+1})(\sigma_1,\ldots,\sigma_{k+1})} \ \wedge \ \cdots \\
&\wedge \ \exists_{\overline{sa\restriction_{!\odot(X_n)}} \subseteq \sigma_n \in \mathsf{str}(X_{k+1}(\sigma_1,\ldots,\sigma_{n-1}))} sa \restriction_{\odot(X_{n+1})} \in P_{X_{n+1}(\sigma_1,\ldots,\sigma_n)} \ \} \bigcup \\
&\{sab \mid sa \in P^{\mathsf{odd}}_{(\Pi_{X_k} \cdots \Pi_{X_n} X_{n+1})(\sigma_1,\ldots,\sigma_{k-1})} \wedge (\forall_{\overline{sab\restriction_{!\odot(X_k)}} \subseteq \sigma_k \in \mathsf{str}(X_k(\sigma_1,\ldots,\sigma_{k-1}))} sa \in P_{X_k(\sigma_1,\ldots,\sigma_{k-1}) \Rightarrow (\Pi_{X_{k+1}} \cdots \Pi_{X_n} X_{n+1})(\sigma_1,\ldots,\sigma_k)} \\
&\Rightarrow \ sab \restriction_{!\odot(X_k)} \in P_{X_k(\sigma_1,\ldots,\sigma_{k-1})}) \ \wedge \ (\forall_{\overline{sab\restriction_{!\odot(X_{k+1})}} \subseteq \sigma_{k+1} \in \mathsf{str}(X_{k+1}(\sigma_1,\ldots,\sigma_k))} sa \restriction_{!\odot(X_{k+1}),\ldots,\odot(X_{n+1})} \in P_{X_{k+1}(\sigma_1,\ldots,\sigma_k) \Rightarrow (\Pi_{X_{k+2}} \cdots \Pi_{X_n} X_{n+1})(\sigma_1,\ldots,\sigma_{k+1})} \\
&\Rightarrow \ sab \restriction_{!\odot(X_{k+1})} \in P_{X_{k+1}(\sigma_1,\ldots,\sigma_k)}) \ \wedge \ \cdots \ \wedge \ (\forall_{\overline{sab\restriction_{!\odot(X_n)}} \subseteq \sigma_n \in \mathsf{str}(X_n(\sigma_1,\ldots,\sigma_{n-1}))} \\
&sa \restriction_{!\odot(X_n),\odot(X_{n+1})} \in P_{(\Pi_{X_n} X_{n+1})(\sigma_1,\ldots,\sigma_{n-1})} \Rightarrow sab \restriction_{!\odot(X_n)} \in P_{X_n(\sigma_1,\ldots,\sigma_{n-1})} \ \wedge \ sab \restriction_{\odot(X_{n+1})} \in P_{X_{n+1}(\sigma_1,\ldots,\sigma_n)}) \ \}.
\end{aligned}
$$

We note that the proposed description is valid for ϵ. Let us suppose it is valid for s. Note that all conjuncts involving s (rather than sa) in the incremental condition on Opponent moves then hold by induction. Rearranging the incremental condition on Opponent moves now gives us a description in which we have obtained the required incremental condition on Opponent moves, but not yet on Player moves – in particular, our proposed description is now valid for sa:

$\{\epsilon\} \bigcup$
$\{sa \mid s \in P^{\mathsf{even}}_{(\Pi_{X_k}\cdots\Pi_{X_n}X_{n+1})(\sigma_1,\ldots,\sigma_{k-1})} \wedge \exists_{\overline{sa\restriction_{!\odot(X_k)}}\subseteq\sigma_k\in\mathsf{str}(X_k(\sigma_1,\ldots,\sigma_{k-1}))}\cdots\exists_{\overline{sa\restriction_{!\odot(X_n)}}\subseteq\sigma_n\in\mathsf{str}(X_n(\sigma_1,\ldots,\sigma_{n-1}))}$
$sa \in P_{X_k(\sigma_1,\ldots,\sigma_{k-1})\Rightarrow\cdots\Rightarrow X_{n+1}(\sigma_1,\ldots,\sigma_n)}\} \bigcup$
$\{sab \mid sa \in P^{\mathsf{odd}}_{(\Pi_{X_k}\cdots\Pi_{X_n}X_{n+1})(\sigma_1,\ldots,\sigma_{k-1})} \wedge (\forall_{\overline{sab\restriction_{!\odot(X_k)}}\subseteq\sigma_k\in\mathsf{str}(X_k(\sigma_1,\ldots,\sigma_{k-1}))} sa \in P_{X_k(\sigma_1,\ldots,\sigma_{k-1})\Rightarrow(\Pi_{X_{k+1}}\cdots\Pi_{X_n}X_{n+1})(\sigma_1,\ldots,\sigma_k)}$
$\Rightarrow\ sab\restriction_{!\odot(X_k)} \in P_{X_k(\sigma_1,\ldots,\sigma_{k-1})}) \wedge (\forall_{\overline{sab\restriction_{!\odot(X_{k+1})}}\subseteq\sigma_{k+1}\in\mathsf{str}(X_{k+1}(\sigma_1,\ldots,\sigma_k))} sa\restriction_{!\odot(X_{k+1}),\ldots,\odot(X_{n+1})} \in P_{X_{k+1}(\sigma_1,\ldots,\sigma_k)\Rightarrow(\Pi_{X_{k+2}}\cdots\Pi_{X_n}X_{n+1})(\sigma_1,\ldots,\sigma_{k+1})}$
$\Rightarrow\ sab\restriction_{!\odot(X_{k+1})} \in P_{X_{k+1}(\sigma_1,\ldots,\sigma_k)}) \wedge \cdots \wedge (\forall_{\overline{sab\restriction_{!\odot(X_n)}}\subseteq\sigma_n\in\mathsf{str}(X_n(\sigma_1,\ldots,\sigma_{n-1}))}$
$sa\restriction_{!\odot(X_n),\odot(X_{n+1})} \in P_{(\Pi_{X_n}X_{n+1})(\sigma_1,\ldots,\sigma_{n-1})} \Rightarrow sab\restriction_{!\odot(X_n)} \in P_{X_n(\sigma_1,\ldots,\sigma_{n-1})} \wedge sab\restriction_{\odot(X_{n+1})} \in P_{X_{n+1}(\sigma_1,\ldots,\sigma_n)})\}.$

Next, noting that our proposed description holds for sa, we note that the conjuncts

$$sa\restriction_{!\odot(X_m),\ldots,\odot(X_{n+1})} \in P_{X_m(\sigma_1,\ldots,\sigma_{m-1})\Rightarrow(\Pi_{X_{m+1}}\cdots\Pi_{X_n}X_{n+1})(\sigma_1,\ldots,\sigma_m)}$$

in the incremental condition on P-moves can be replaced by the conditions

$$\exists_{\overline{sa\restriction_{!\odot(X_{m+1})}}\subseteq\sigma_{m+1}\in\mathsf{str}(X_{m+1}(\sigma_1,\ldots,\sigma_m))}\cdots\exists_{\overline{sa\restriction_{!\odot(X_n)}}\subseteq\sigma_n\in\mathsf{str}(X_n(\sigma_1,\ldots,\sigma_{n-1}))} sa\restriction_{!\odot(X_m),\ldots,\odot(X_{n+1})} \in P_{X_m(\sigma_1,\ldots,\sigma_{m-1})\Rightarrow\cdots\Rightarrow X_{n+1}(\sigma_1,\ldots,\sigma_n)}.$$

(Seeing that Opponent chooses the fibre, provided that our description holds.)

Now, again noting that $\overline{sa\restriction_{!\odot(X_l)}} = \overline{sab\restriction_{!\odot(X_l)}}$, this means that all universal quantifiers in the incremental condition on P-moves range over a non-empty domain, so we might as well move them to the front of our formula, seeing that they do not bind any more identifiers:

$\{\epsilon\} \bigcup$
$\{sa \mid s \in P^{\mathsf{even}}_{(\Pi_{X_k}\cdots\Pi_{X_n}X_{n+1})(\sigma_1,\ldots,\sigma_{k-1})} \wedge \exists_{\overline{sa\restriction_{!\odot(X_k)}}\subseteq\sigma_k\in\mathsf{str}(X_k(\sigma_1,\ldots,\sigma_{k-1}))}\cdots\exists_{\overline{sa\restriction_{!\odot(X_n)}}\subseteq\sigma_n\in\mathsf{str}(X_n(\sigma_1,\ldots,\sigma_{n-1}))}$
$sa \in P_{X_k(\sigma_1,\ldots,\sigma_{k-1})\Rightarrow\cdots\Rightarrow X_{n+1}(\sigma_1,\ldots,\sigma_n)}\} \bigcup$
$\{sab \mid sa \in P^{\mathsf{odd}}_{(\Pi_{X_k}\cdots\Pi_{X_n}X_{n+1})(\sigma_1,\ldots,\sigma_{k-1})} \wedge \forall_{\overline{sab\restriction_{!\odot(X_k)}}\subseteq\sigma_k\in\mathsf{str}(X_k(\sigma_1,\ldots,\sigma_{k-1}))}\cdots\forall_{\overline{sab\restriction_{!\odot(X_n)}}\subseteq\sigma_n\in\mathsf{str}(X_n(\sigma_1,\ldots,\sigma_{n-1}))}$
$sa \in P_{X_k(\sigma_1,\ldots,\sigma_{k-1})\Rightarrow\cdots\Rightarrow X_{n+1}(\sigma_1,\ldots,\sigma_n)} \Rightarrow sab\restriction_{!\odot(X_k)} \in P_{X_k(\sigma_1,\ldots,\sigma_{k-1})} \wedge \cdots \wedge$
$sab\restriction_{!\odot(X_n)} \in P_{X_n(\sigma_1,\ldots,\sigma_{n-1})} \wedge sab\restriction_{\odot(X_{n+1})} \in P_{X_{n+1}(\sigma_1,\ldots,\sigma_n)}\},$

which clearly carves out the same plays in $P_{\odot(X_1)\Rightarrow\cdots\Rightarrow\odot(X_n)\Rightarrow\odot(X_{n+1})}$ as our proposed description. □

Remark 4.2.4 (Logical Predicates/Realizability?)**.** *Note that these games of dependent functions lead to quite a non-trivial notion of dependently typed strategy. Indeed, we can send a game B with dependency on A to a function $\mathsf{str}(\odot(A)) \longrightarrow \mathcal{P}\mathsf{str}(\odot(B))$ which assigns a set of consistent strategies $\sigma \mapsto \mathsf{cstr}(B) := \mathsf{str}(B(\sigma)) \subseteq \mathsf{str}(\odot(B))$. One might wonder if this description is enough to recover our model from and if the model can be recast into a realizability style model [99]. In particular, this would mean that we send a pair $(A(\bot), \odot(A))$ to the pair $(\odot(A), \mathsf{cstr}(A) \subseteq \mathsf{str}(\odot(A)))$. In*

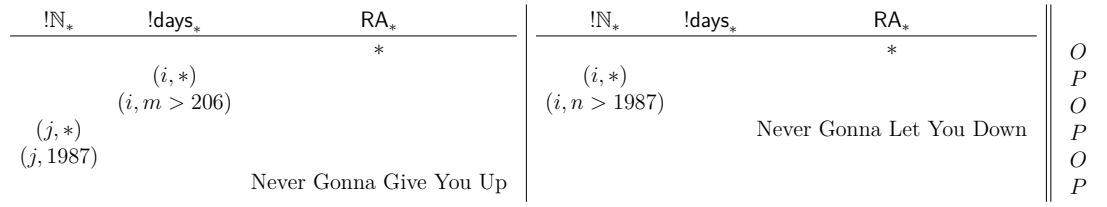


Figure 4.2: Two examples of (partial) strategies on the game $\mathsf{O\text{-}sat}(\Pi_{\mathbb{N}_*}\Pi_{\mathsf{days}_*}\mathsf{RA}_*)$, defining dependent functions of two arguments. Note that these lyrics come from a song released on day 207 of the year 1987, so Player does not constrain the fibre anywhere.

a realizability model, one would expect this class $\mathsf{cstr}(A)$ *of consistent strategies to behave as a logical predicate. In particular, given* $\mathsf{just}\{\mathsf{tt}\} = (\{\mathsf{tt}\}_*, \mathbb{B}_*)$*, if* cstr *were a logical predicate, we would have that* $\mathsf{cstr}((\mathsf{just}\{\mathsf{tt}\} \Rightarrow \mathbb{B}_*) \Rightarrow \mathbb{B}_*) = \mathsf{cstr}((\mathbb{B}_* \Rightarrow \mathbb{B}_*) \Rightarrow \mathbb{B}_*) = \mathsf{str}((\mathbb{B}_* \Rightarrow \mathbb{B}_*) \Rightarrow \mathbb{B}_*)$ *as* $\mathsf{cstr}(\mathbb{B}_*) = \mathsf{str}(\mathbb{B}_*)$*. However, we have that* $\mathsf{cstr}((\mathsf{just}\{\mathsf{tt}\} \Rightarrow \mathbb{B}_*) \Rightarrow \mathbb{B}_*) := \mathsf{str}(\mathsf{O\text{-}sat}((\mathsf{just}_*(\mathsf{tt}) \Rightarrow \mathbb{B}_*) \Rightarrow \mathbb{B}_*)) \subsetneq \mathsf{str}((\mathbb{B}_* \Rightarrow \mathbb{B}_*) \Rightarrow \mathbb{B}_*)$*. Indeed, a consistent strategy on* $(\mathsf{just}\{\mathsf{tt}\} \Rightarrow \mathbb{B}_*) \Rightarrow \mathbb{B}_*$ *cannot play* ff *in* $\mathsf{just}\{\mathsf{tt}\} \trianglelefteq \mathbb{B}_*$*. We see that our notion of consistent strategy does not behave as a logical predicate. We get a more non-trivial notion of higher-order dependent function. The extra requirement that Player is constrained by type dependency in positive occurring types just as she is in strictly positively occurring ones is important to get an exact match with the syntax of dependent type theory.*

For illustration, define a game RA_* depending on the context game $[\mathbb{N}_*, \mathsf{days}_*]$ by

$$\mathsf{RA}(n, m) := \{\text{Rick Astley lyrics from songs released before day } m \text{ of year } n\}$$

Then, the two strategies of figure 4.2 illustrate that a dependent function may query its arguments in unexpected order or may not query some at all. To illustrate the subtle nature of higher-order dependent functions with an example, define the game $\mathsf{holidays}_*$ depending on the context game $[\mathbb{N}_*, \mathsf{days}_*]$ by

$$\mathsf{holidays}(n, m) := \{\text{holidays that are celebrated on day } m \text{ of year } n\}$$

Figure 4.3 illustrates how Player is in charge of providing certain arguments (the positive ones) of dependent games and can therefore choose the fibre in some cases. (Opponent controls the negative arguments to dependent games.) In the figure

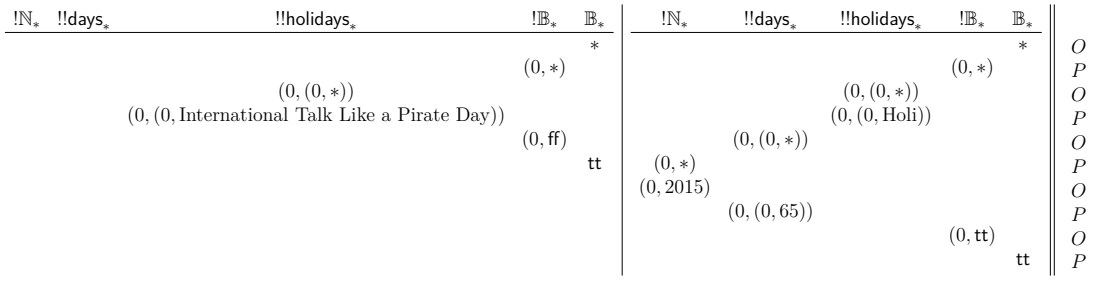


Figure 4.3: Two plays in $\mathsf{O\text{-}sat}(\Pi_{\mathbb{N}_*}\Pi_{\Pi_{\mathsf{days}_*}\Pi_{\mathsf{holidays}_*}\mathbb{B}_*}\mathbb{B}_*)$. For an interpretation, imagine Player is a PhD-student who is trying to decide if he is going on holidays and ends up asking his supervisor (Opponent) if she's okay with him doing so. The first play can be read as the dialogue where the supervisor asks if the student is planning to take any holidays, the student asks if he's allowed to, the supervisor wants to know what the occasion is, the student admits that his best excuse for wanting time off is International Talk Like a Pirate Day, the supervisor tells the student that he can't have time off and, finally, the student tells his supervisor that he's taking time off anyway for this important occasion. Here, Player can choose the holiday 'International Talk Like a Pirate Day' as it is celebrated each year, meaning that Player does not restrict the year we may be talking about (which, as a negative argument, belongs to Opponent). Note that by choosing this particular holiday, Player automatically fixes the day the holiday falls on, which is fine as the subgame days_* occurs positively in the total game we are playing in, meaning that Player is in charge of determining the corresponding argument. The second play corresponds to a dialogue with a more sensible student who uses the more respectable excuse of celebrating Holi to get time off from work. Here, Player has to let Opponent determine the year first, before she can answer with a date for Holi, as the date of Holi on the Gregorian calendar varies (while it is celebrated every year).

below, Player controls the arguments of type days_* and $\mathsf{holidays}_*$, while Opponent is in charge of the type of years $\mathbb{N}_*$. We stress again that although Player has to play in accordance with any choice of year that Opponent could make, the converse is not true: Opponent can do what she likes and does not have to respect Player's choices of day and holiday.

We define a category $\mathsf{Ctxt}(\mathsf{DGame}_!)$ with objects context games $[A_i]_{1\leq i\leq n}$ and morphisms which are defined inductively as (dependent) lists $[\sigma_i]_{1\leq i\leq n}$ of strategies on appropriate games of dependent functions. To keep notation light, we sometimes abuse notation and write σ for the context morphism $[\sigma]$ of length 1.

We show that $\mathsf{Ctxt}(\mathsf{DGame}_!)$ has the structure of a category with families (CwF) (see definition 2.1.6), a canonical notion of model of dependently typed equational logic. This gives a more concise presentation of the resulting strict indexed category with comprehension, where we also add formal Σ-types in the fibres.

Theorem 4.2.5. *We have a CwF* $(\mathsf{Ctxt}(\mathsf{DGame}_!), \mathsf{Ty}, \mathsf{Tm}, \mathbf{p}_,, \mathbf{v}_,, -.-, \langle -, - \rangle)$.

Proof. We define the required structures. All equations follow straightforwardly from the definitions and the two claims stated.

$\boxed{\mathsf{ob}(\mathcal{C}),\ \mathsf{Ty},\ -.-,\ \cdot}$

We define a category $\mathcal{C} := \mathsf{Ctxt}(\mathsf{DGame}_!)$ with context games as objects. We define $\mathsf{Ty}([X_i]_i)$ as the set of **context games with dependency on** $[X_i]_i$: $[Y_j]_j \in \mathsf{Ty}([X_i]_i)$ iff $[X_i]_i.[Y_j]_j := [X_1, \ldots, X_n, Y_1, \ldots, Y_m]$ is a context game, while $\cdot := []$ is the terminal object.

$\boxed{\mathsf{mor}(\mathcal{C}),\ -\{-\}_{\mathsf{Ty}}}$

Next, let $[X_i]_{i\leq n}, [Z_k]_{k\leq n'} \in \mathsf{ob}(\mathcal{C})$, $[Y_j]_{j\leq m} \in \mathsf{Ty}([X_i]_{i\leq n}$ and let

$$☺(Z_1)\& \cdots \&☺(Z_{n'}) \xrightarrow{f} ☺(X_1)\& \cdots \&☺(X_n)$$

be a morphism in $\mathsf{Game}_!$. Then, we define $[Y_j]_{j\leq m}\{f\} \in \mathsf{Ty}([Z_k]_{k\leq n'})$ by

$$☺(Y_j\{f\}) := ☺(Y_j)$$
$$Y_j\{f\}(\sigma_1, \ldots, \sigma_{n'}, \tau_1, \ldots, \tau_{j-1}) := Y_j(\langle \sigma_1, \ldots, \sigma_{n'} \rangle^\dagger; f, \tau_1, \ldots, \tau_{j-1}).$$

This, in turn, lets us define $\mathsf{mor}(\mathcal{C})$:

$$\mathsf{Ctxt}(\mathsf{DGame}_!)([X_i]_{i\leq n}, [Y_j]_{j\leq m}) := \{\ [f_j]_{j\leq m} \mid f_j \in \mathsf{str}(\mathsf{O\text{-}sat}(\Pi_{X_1} \ldots \Pi_{X_n} Y_j\{\langle f_1, \ldots, f_{j-1}\rangle\}))\ \},$$

noting that f_j can, in particular, be interpreted as a morphism

$$☺(X_1)\& \cdots \&☺(X_n) \xrightarrow{f_j} ☺(Y_j) \in \mathsf{Game}_!.$$

$\boxed{\mathsf{id},\ \mathbf{p}_,,\ \mathsf{Tm},\ \mathbf{v}_,,\ \langle \cdot, \cdot \rangle,\ \text{(Cons-Id)}}$

The identities are defined as lists of derelicted copycats. Let us define a strategy $\mathsf{der}_{[X_j]_j, X_i}$ which plays the derelicted copycat on all of $☺(X_i)$: $\mathsf{der}_{[X_j]_j, X_i} := \{s \in P_{\mathsf{O\text{-}sat}(\Pi_{X_1} \ldots \Pi_{X_n} X_i)} \mid \forall_{s' \in P^{\mathsf{even}}_{\mathsf{O\text{-}sat}(\Pi_{X_1} \ldots \Pi_{X_n} X_i)}} s' \leq s \Rightarrow \exists_k s \upharpoonright_{!☺(X_i)} \upharpoonright_k \approx_{☺(X_i)} s \upharpoonright_{☺(X_i)}\}$. We then define $\mathsf{id}_{[X_i]_i} := [\mathsf{der}_{[X_j]_j, X_i}]_i$ and $\mathbf{p}_{[X_i]_i, [Y_j]_j} := [\mathsf{der}_{[X_i]_i.[Y_j]_j, X_k}]_k$. Let us define

$$\mathsf{Tm}([X_i]_{i\leq n},[Y_j]_{j\leq m}) := \left\{\ [f_j]_{j\leq m} \;\middle|\; [\mathsf{der}_{[X_i]_i,X_1},\ldots,\mathsf{der}_{[X_i]_i,X_n},f_1,\ldots,f_m] \in \mathsf{Ctxt}(\mathsf{DGame}_!)([X_i]_i,[X_i]_i.[Y_j]_j)\ \right\}.$$

Then, we can define $\mathbf{v}_{[X_i]_i,[Y_j]_j} := [\mathsf{der}_{[X_i]_i.[Y_j]_j,Y_k}]_k$. Note that these are well-defined because of the following claim.

Claim. $\mathsf{der}_{[X_j]_j,X_i} \in \mathsf{str}(\mathsf{O\text{-}sat}(\Pi_{X_1}\cdots\Pi_{X_n}X_i\{[\mathsf{der}_{[X_j]_j,X_k}]_{k\leq i-1}\}))$.

Proof. Note that Opponent makes every move first in X_i, so Player can copy it freely without restricting the fibre of X_i further. □

We define $\langle [f_j]_{j\leq m},[g_k]_{k\leq l}\rangle := [f_1,\ldots,f_m,g_1,\ldots,g_l]$, after which (Cons-Id) follows trivially.

Composition, $-\{-\}_{\mathsf{Tm}}$, (Cons-Nat)

We define the composition of $[X_i]_{i\leq n} \xrightarrow{[f_j]_j} [Y_j]_{j\leq m} \xrightarrow{[g_k]_k} [Z_k]_k$ in $\mathsf{Ctxt}(\mathsf{DGame}_!)$ by

$$[f_j]_j;[g_k]_k := [\langle f_1,\ldots,f_m\rangle^\dagger; g_k]_k,$$

using the usual (co-Kleisli) composition of strategies on $☺(X_1)\Rightarrow\cdots\Rightarrow ☺(X_n)\Rightarrow (☺(Y_1)\&\cdots\&☺(Y_m))$ and $☺(Y_1)\Rightarrow\cdots\Rightarrow ☺(Y_m)\Rightarrow ☺(Z_k)$. We note that we can assign to this composition a more precise dependent function type.

Claim. *The composition $[f_j]_j;[g_k]_k$ above does, in fact, define a morphism in $\mathsf{Ctxt}(\mathsf{DGame}_!)([X_i]_{i\leq n},[Z_k]_k)$.*

Proof. We need to verify that $\langle f_1,\ldots,f_m\rangle^\dagger; g_k$ defines a winning strategy on $\mathsf{O\text{-}sat}(\Pi_{X_1}\cdots\Pi_{X_n}W\{[f_j]_j\})$, where we write $W := Z_k\{[g_{k'}]_{k'<k}\}$. The winning part of the claim follows trivially from the usual fact that winning strategies compose. What is to be verified is the claim that $\langle f_1,\ldots,f_m\rangle^\dagger; g_k$ is a strategy on $\mathsf{O\text{-}sat}(\Pi_{X_1}\cdots\Pi_{X_n}W\{[f_j]_j\})$.

Recall that, by assumption, g_k is a strategy on $\mathsf{O\text{-}sat}(\Pi_{Y_1}\cdots\Pi_{Y_m}W)$. Suppose $\langle f_1,\ldots,f_m\rangle^\dagger; g_k$ wants to respond with a move b in some X_i or W after a play sa.

Recall that by theorem 4.2.3, we need to verify that for all $\overline{sab \restriction_{!\textcircled{\scriptsize ☺}(X_1)}} \subseteq \sigma'_1 \in \mathsf{str}(X_1()), \ldots, \overline{sab \restriction_{!\textcircled{\scriptsize ☺}(X_n)}} \subseteq \sigma'_n \in \mathsf{str}(X_n(\sigma'_1, \ldots, \sigma'_{n-1}))$, we have that

$$sab \restriction_{!\textcircled{\scriptsize ☺}(X_1)} \in P_{!X_1()} \wedge \cdots \wedge sab \restriction_{!\textcircled{\scriptsize ☺}(X_n)} \in P_{!X_n(\sigma'_1,\ldots,\sigma'_{n-1})} \wedge sab \restriction_{\textcircled{\scriptsize ☺}(W)} \in P_{W\{[f_j]_j\}(\sigma'_1,\ldots,\sigma'_n)},$$

provided that already

$$sa \restriction_{!\textcircled{\scriptsize ☺}(X_1)} \in P_{!X_1()} \wedge \cdots \wedge sa \restriction_{!\textcircled{\scriptsize ☺}(X_n)} \in P_{!X_n(\sigma'_1,\ldots,\sigma'_{n-1})} \wedge sa \restriction_{\textcircled{\scriptsize ☺}(W)} \in P_{W\{[f_j]_j\}(\sigma'_1,\ldots,\sigma'_n)}.$$

Here, all but the last conjunct follow from the fact that the f_k are strategies on $\mathsf{O\text{-}sat}(\Pi_{X_1} \cdots \Pi_{X_n} Y_k\{[f_{k'}]_{k'<k}\})$. (Seeing that g_k will not break the dependency in $[Y_k]_k$ as a strategy on $\mathsf{O\text{-}sat}(\Pi_{Y_1} \cdots \Pi_{Y_m} Z_k\{[g_{k'}]_{k'<k}\})$.)

What remains to be checked, therefore, is that

$$sab \restriction_{\textcircled{\scriptsize ☺}(W)} \in P_{W\{[f_j]_j\}(\sigma'_1,\ldots,\sigma'_n)},$$

or equivalently,

$$sab \restriction_{\textcircled{\scriptsize ☺}(W)} \in P_{W(\langle\sigma'_1,\ldots,\sigma'_n\rangle^\dagger;f_1,\ldots,\langle\sigma'_1,\ldots,\sigma'_n\rangle^\dagger;f_m)}.$$

This follows immediately from the fact that g_k is a strategy on $\mathsf{O\text{-}sat}(\Pi_{Y_1} \cdots \Pi_{Y_m} W)$ if we can show that $[\sigma'_i]_{i\leq n}; [f_j]_{j\leq m} \in \mathsf{Ctxt}(\mathsf{DGame}_!)([], [Y_j]_{j\leq m})$, which is a special case of our claim when $[X_i]_i = []$.

That is, we need to demonstrate that $\langle\sigma'_1, \ldots, \sigma'_n\rangle^\dagger; f_j$ defines a strategy on $\mathsf{O\text{-}sat}(Y_j\{[\sigma'_i]_{i\leq n}; [f_j]_j\})$. (Again, it follows trivially that it will be a winning strategy.) We verify that for any play sab in $\langle\sigma'_1, \ldots, \sigma'_n\rangle^\dagger || f_j$, where b is a Player move in $\textcircled{\scriptsize ☺}(Y_j)$, we have in fact that it is a move in $\mathsf{O\text{-}sat}(Y_j\{[\sigma'_i]_{i\leq n}; [f_j]_j\})$. This follows from the fact that f_j is a strategy on $\mathsf{O\text{-}sat}(\Pi_{X_1} \cdots \Pi_{X_n} Y_j\{[f_{j'}]_{j'<j}\})$, which according to theorem 4.2.3 means, in particular, that for all $\overline{sab \restriction_{!\textcircled{\scriptsize ☺}(X_1)}} \subseteq \sigma'_1 \in \mathsf{str}(X_1()), \ldots \overline{sab \restriction_{!\textcircled{\scriptsize ☺}(X_n)}} \subseteq \sigma'_n \in \mathsf{str}(X_n(\sigma'_1, \ldots, \sigma'_{n-1}))$, we have that

$$sab \restriction_{!\textcircled{\scriptsize ☺}(X_1)} \in P_{!X_1()} \wedge \cdots \wedge sab \restriction_{!\textcircled{\scriptsize ☺}(X_n)} \in P_{!X_n(\sigma'_1,\ldots,\sigma'_{n-1})} \wedge sab \restriction_{\textcircled{\scriptsize ☺}(Y_j)} \in P_{Y_j\{[f_{j'}]_{j'<j}\}(\sigma'_1,\ldots,\sigma'_n)},$$

provided that already

$$sa \restriction_{!\textcircled{\scriptsize ☺}(X_1)} \in P_{!X_1()} \wedge \cdots \wedge sa \restriction_{!\textcircled{\scriptsize ☺}(X_n)} \in P_{!X_n(\sigma'_1,\ldots,\sigma'_{n-1})} \wedge sa \restriction_{\textcircled{\scriptsize ☺}(Y_j)} \in P_{Y_j\{[f_{j'}]_{j'<j}\}(\sigma'_1,\ldots,\sigma'_n)}.$$

The last conjunct is what we are looking for, or rather its reformulation $sab \restriction_{\odot(Y_j)} \in P_{Y_j\{[\sigma'_i]_i;[f_j]_j\}}$.

□

Note that for $[X_i]_i \xrightarrow{[f_j]_j} [Y_j]_j$ and $[Y_j]_j \xrightarrow{\langle [g_k]_k,[h_l]_l\rangle} [Z_k]_k.[W_l]_l$, (Cons-Nat) holds in the sense that

$$[f_j]_j; \langle [[g_k]_k, [h_l]_l\rangle = \langle [f_j]_j; [g_k]_k, [h_l]_l\{[f_j]_j\}\rangle,$$

if we define $-\{-\}_{\mathsf{Tm}}$ by

$$[h_l]_l\{[f_j]_j\} := [\langle f_1, \ldots, f_m\rangle^\dagger; h_l]_l,$$

which then automatically type checks because of (Cons-Nat).

Id. Law, Assoc., (Ty-Id), (Tm-Id), (Ty-Comp), (Tm-Comp), (Cons-L), (Cons-R)

All these identities are direct consequences of the identity and associativity laws of the usual composition of strategies in $\mathsf{Game}_!$. □

Remark 4.2.6. *Note that, in* $\mathsf{Ctxt}(\mathsf{DGame}_!)$, $[A, B] \cong [A\&B]$ *if* A *and* B *are games (without mutual dependency) and* $[] \cong [I]$.

4.3 Semantic Type Formers 1, Σ, Π and Id

We show that our CwF supports 1-, Σ-, Π-, and Id-types. We leave the verification of all term equations (which are inherited from their simply typed equivalents) to section 4.5. As for type equations, we can note that all type formers are preserved by substitution. We characterise some of the properties of the Id-types, marking their place in the intensionality spectrum.

1-types 1-types are interpreted by the context game of length 0. $\langle\rangle$ is interpreted by the list of strategies of length 0.

Σ-Types Σ-types are (formally) defined by concatenation of lists. For $[Z_k]_{k\leq l} \in \mathsf{Ty}([X_i]_{i\leq n}.[Y_j]_{j\leq m})$, we define a Σ-type

$$\Sigma_{[Y_j]_j}[Z_k]_k := [Y_j]_j.[Z_k]_k \in \mathsf{Ty}([X_i]_{i\leq n}).$$

We can interpret $\langle -,-\rangle$ by concatenation $[\sigma_1,\dots,\sigma_m,\tau_1,\dots,\tau_l]$ of lists of strategies $[\sigma_j]_j$ and $[\tau_k]_k$, while we interpret fst as $[\mathsf{der}_{[X_i]_i.[Y_j]_j.[Z_k]_k,Y_{j'}}]_{j'}$ and snd as $[\mathsf{der}_{[X_i]_i.[Y_j]_j.[Z_k]_k,Z_{k'}}]_{k'}$.

Π-Types We have already seen Π-types

$$\Pi_{[Y_j]_{j\leq m}}[Z] := [\Pi_{Y_1}\cdots\Pi_{Y_m}Z] \in \mathsf{Ty}([X_i]_{i\leq n})$$

of dependent games $[Z] \in \mathsf{Ty}([X_i]_{i\leq n}.[Y_j]_{j\leq m})$. They let us define λ-abstraction and evaluation as on the usual simply typed function game $☺(Y_1) \Rightarrow \cdots \Rightarrow ☺(Y_m) \Rightarrow ☺(Z)$. What remains to be defined are Π-types $\Pi_{[Y_j]_j}[Z_k]_k$ of general dependent context games $[Z_k]_k \in \mathsf{Ty}([X_i]_{i\leq n}.[Y_j]_{j\leq m})$. These can be reduced to the former, as $\Sigma_{f:\Pi_{x:A}B}\Pi_{x:A}C[f(x)/y]$ satisfies the rules for $\Pi_{x:A}\Sigma_{y:B}C$. Conclusion: our CwF supports Π-types.

Corollary 4.3.1. *This means that* $\mathsf{Ctxt}(\mathsf{DGame}_!)$ *is in particular a ccc.*

Id-Types We turn to identity types next, which are essentially defined as one would expect from their definition as an inductive family. Interestingly, due to the intensional nature of function types in game semantics, these identity types acquire a very intensional character as well, refuting FunExt.

For $[Y_j]_j \in \mathsf{Ty}([X_i]_i)$, define $\mathsf{Id}_{[Y_j]_j} \in \mathsf{Ty}([X_i]_i.[Y_j]_j.[Y_{j'}]_{j'})$:

$$\mathsf{Id}_{[Y_j]_j}([\sigma_i]_i,[\tau_j]_j,[\tau'_j]_j) := [\ \begin{matrix}\{\mathtt{refl}\}_* & \mathsf{if}\ [\tau_j]_j = [\tau'_j]_j \\ \emptyset_* & \mathsf{else}\end{matrix}\ ].$$

Here, $☺(\mathsf{Id}_{[Y_j]_j}) := \{\mathtt{refl}\}_*$. Note that, by definition, the plays of $\mathsf{Id}_{[Y_j]_j}([\sigma_i]_i,[\tau_j]_j,[\tau'_j]_j)$ are closed under all Opponent moves in $☺(\mathsf{Id}_{[Y_j]_j})$. Note that this means that $\mathsf{O\text{-}sat}(\mathsf{Id}_{[Y_j]_j}([\sigma_i]_i,[\tau_j]_j,[\tau'_j]_j)) = \mathsf{Id}_{[Y_j]_j}([\sigma_i]_i,[\tau_j]_j,[\tau'_j]_j)$.

Id-I is interpreted by the non-strict strategy

$$\begin{aligned}\mathsf{refl}([f_j]_j) := [\{\epsilon, *, *_{\mathtt{refl}}\}] \in \mathsf{Tm}([X_i]_i, \mathsf{Id}_{[Y_j]_j}\{\langle [\mathsf{der}_{X_i}]_i, [f_j]_j, [f_j]_j\rangle\}) \\ = \{[\sigma] \mid \sigma \in \mathsf{str}\{\mathtt{refl}\}_*\}.\end{aligned}$$

For the (strong) Id-E rule, suppose we are given

- $[Z_k]_k \in \mathsf{Ty}([X_i]_i.[Y_j]_j.[Y_j]_j.\mathsf{Id}_{[Y_j]_j})$;
- $[f_k]_k \in \mathsf{Tm}([X_i]_i.[Y_j]_j, [Z_k]_k\{\langle \mathsf{der}_{[X_i]_i}, \mathsf{der}_{[Y_j]_j}, \mathsf{der}_{[Y_j]_j}, \mathsf{refl}(\mathsf{der}_{[Y_j]_j})\rangle\})$.

Then, we produce

$$[f'_k]_k \in \mathsf{Tm}([X_i]_i.[Y_j^{(1)}]_j.[Y_j^{(2)}]_j.\mathsf{Id}_{[Y_j]_j}, [Z_k]_k).$$

Here, f'_k is the strategy which responds to the initial move in $☺(Z_k)$ by opening $☺(\mathsf{Id}_{[Y_j]_j})$, encoding the initial move in the index, and if Opponent responds $\mathtt{refl}$ continues playing f_k using the left hand side copy of Y_j. (Hence, $[f'_k]_k$ does not ever visit $[Y_j^{(2)}]_j$.) Note that such f'_k are well-defined strategies, as

- all fibres of Id-types contain the intial O-move, allowing f'_k to always play it;
- the moment that Opponent plays $\mathtt{refl}$, she excludes the fibres for which the two arguments of type $[Y_j]_j$ are not equal as we have a bijection

  $$\mathsf{Ctxt}(\mathsf{DGame}_!)([], [X_i]_i.[Y_j]_j) \;\cong\; \mathsf{Ctxt}(\mathsf{DGame}_!)([], [X_i]_i.[Y_j]_j.[Y_j]_j.\mathsf{Id}_{[Y_j]_j})$$

  $$\langle [\sigma_i]_i, [\tau_j]_j\rangle \longmapsto \langle [\sigma_i]_i, [\tau_j]_j, [\tau_j]_j, [\mathtt{refl}]\rangle.$$

  Hence we can continue playing f_k from that point.

Noting that Player always has a response in the initial protocol and next follows f_k, it follows that f'_k are winning iff f_k are.

Remark 4.3.2. *There is an alternative, more extensional, definition of* Id*-types which is tempting and which gives* Id*-types satisfying the principle of function extensionality. One reason we have chosen to work with these* Id*-types instead is that the other definition does not generalise to a situation where we are working with non-winning or non-deterministic strategies.*

The idea is to restrict the model to dependent games which send applicatively equivalent strategies[3] *to equal subgames. In that case, we can define the identity type*

$$\mathsf{Id}_{[Y_j]_j}([\sigma_i]_i, [\tau_j]_j, [\tau'_j]_j) := [\mathsf{Id}_{Y_j}]_j([\sigma_i]_i, [\tau_j]_j, [\tau'_j]_j) := [\tau_j^{app} \cap \tau_j'^{app}]_j.$$

Here, $\copyright(\mathsf{Id}_{Y_j}) := \copyright(Y_j)$ *and we write* ϕ^{app} *for the closure of a set of plays* ϕ *under applicative equivalence. Then,* Id*-I is interpreted by* $\mathsf{refl}([f_j]_j) := [f_j]_j \in \mathsf{Tm}([X_i]_i, \mathsf{Id}_{[Y_j]_j}\{\langle[\mathsf{der}_{X_i}]_i, [f_j]_j, [f_j]_j\rangle\}) = \{[g_j]_j \mid g_j \in \mathsf{str}(\mathsf{O\text{-}sat}(\Pi_{[X_i]_i} f_j^{app}\{[g_k]_{k<j}\}))\}$*, where we interpret the non-deterministic strategy* $f_j^{app}\{[g_k]_{k<j}\}$ *as a game (which contains all Opponent moves) depending on* $[X_i]_i$*, noting that for any* $[\sigma_i]_i \in \mathsf{str}(\copyright(X_1)) \times \cdots \times \mathsf{str}(\copyright(X_n))$ *we have that* $f_j^{app}\{[g_k]_{k<j}\}\{[\sigma_i]_i\}$ *defines a non-deterministic strategy on* $\copyright(Y_j)$ *and hence a subgame of* $\copyright(Y_j)$*. To interpret the* $\mathsf{Id} - E$*-rule, we define* f'_k *as the strategy* f_k *where we identify* Y_j *with* Id_{Y_j}*. (Hence,* $[f'_k]_k$ *does not ever visit* $[Y_j^{(1)}]_j$ *or* $[Y_j^{(2)}]_j$*.) Note that such* f'_k *are well-defined strategies, as long as we are only imposing the type dependency condition for a class of maximal strategies (like winning deterministic strategies).*

Remark 4.3.3 (Interpretation of subst)**.** *Note that* $\Gamma, x : A, x' : A, p : \mathsf{Id}_A(x, x') \vdash \mathsf{subst}(p, -) : B \Rightarrow B[x'/x]$ *gets interpreted as an initial protocol querying the identity type, followed by a simple copycat between the two copies of* $[\![B]\!]$ *after Opponent plays* `refl`*.*

In addition to being non-extensional (i.e. refuting the principle of equality reflection), the intensionality of these identity types can be characterised as follows.

Theorem 4.3.4. *Streicher's Criteria of Intensionality are satisfied, i.e.*

(I1) there exist $\vdash A \quad \mathsf{type}$ *such that* $x, y : A, z : \mathsf{Id}_A(x, y) \nvdash x \equiv y : A$*;*

(I2) there exist $\vdash A \quad \mathsf{type}$ *and* $x : A \vdash B \quad \mathsf{type}$ *such that* $x, y : A, z : \mathsf{Id}_A(x, y) \nvdash B \equiv B[y/x] \quad \mathsf{type}$*;*

(I3) for all $\vdash A \quad \mathsf{type}$*,* $\vdash p : \mathsf{Id}_A(t, s)$ *implies* $\vdash t \equiv s : A$*.*

[3] That is, strategies which cannot be distinguished through their interaction with applicative contexts of ground type.

Proof. (I1) Let us write $\mathbf{p}_{[\mathbb{B}_*{}^{(i)}]}$ for $\mathbf{p}_{[\mathbb{B}_*{}^{(1)},\mathbb{B}_*{}^{(2)},\mathsf{Id}_{\mathbb{B}_*}],[\mathbb{B}_*{}^{(i)}]}$ and $[\![-]\!]$ for the interpretation functor from the syntax of $\mathsf{DTT}_{\mathsf{CBN}}-$ into $\mathsf{Ctxt}(\mathsf{DGame}_!)$. (I1) relies on the interpretation of terms carrying intensionality. Take $[\![A]\!] := [\mathbb{B}_*]$. Then, we have to show that $\mathbf{p}_{[\mathbb{B}_*{}^{(1)}]} \neq \mathbf{p}_{[\mathbb{B}_*{}^{(2)}]} \in \mathsf{Tm}([\mathbb{B}_*].[\mathbb{B}_*].\mathsf{Id}_{[\mathbb{B}_*]}, [\mathbb{B}_*])$. We note that $\mathbf{p}_{[\mathbb{B}_*{}^{(1)}]}\{\langle[\bot],[\mathsf{tt}],[\bot]\rangle\} = [\bot]$ while $\mathbf{p}_{[\mathbb{B}_*{}^{(2)}]}\{\langle[\bot],[\mathsf{tt}],[\bot]\rangle\} = [\mathsf{tt}]$, which shows that $(I1)$ holds.

(I2) This property replies on semantic types having intensional features. In this case, our source of intensionality is that dependent games contain redundant information on their value for inconsistent tuples of strategies. For instance, take $[\![A]\!] := [\mathbb{B}_*]$ and $[\![B]\!] := (\mathsf{ff} \mapsto [I], \mathsf{tt} \mapsto [\mathbb{B}_*])$. Then, we have to show that $[\![B]\!]\{\mathbf{p}_{[\mathbb{B}^{(1)}]}\} \neq [\![B]\!]\{\mathbf{p}_{[\mathbb{B}_*{}^{(2)}]}\} \in \mathsf{Ty}([\mathbb{B}_*].[\mathbb{B}].\mathsf{Id}_{[\mathbb{B}]})$. Now, $[\![B]\!]\{\mathbf{p}_{[\mathbb{B}^{(1)}]}\}(\mathsf{ff}, \mathsf{tt}, \mathtt{refl}) = [\![B]\!](\mathsf{ff}) = [I]$ while $[\![B]\!]\{\mathbf{p}_{[\mathbb{B}_*{}^{(2)}]}\}(\mathsf{ff}, \mathsf{tt}, \mathtt{refl}) = [\![B]\!](\mathsf{tt}) = [\mathbb{B}_*]$, so we conclude that $(I2)$ holds.

(I3) Given $[\sigma_i]_i, [\tau_i]_i \in \mathsf{Tm}([], [X_i]_i)$ and

$$
\begin{aligned}
[p_i]_i \in \mathsf{Tm}([], \mathsf{Id}_{[X_i]_i}([\sigma_i]_i, [\tau_i]_i)) &:= \{[q] \mid q \in \mathsf{str}\left(\mathsf{O\text{-}sat}\left(\begin{array}{ll} \{\mathtt{refl}\}_* & \text{if } [\sigma_i]_i = [\tau_i]_i \\ \emptyset_* & \text{else} \end{array}\right)\right)\} \\
&\cong \begin{cases} \mathsf{str}(\{\mathtt{refl}\}_*) & \text{if } [\sigma_i]_i = [\tau_i]_i \\ \mathsf{str}(\emptyset_*) & \text{else} \end{cases} \\
&\cong \begin{cases} \{\mathtt{refl}\} & \text{if } [\sigma_i]_i = [\tau_i]_i \\ \emptyset & \text{else} \end{cases},
\end{aligned}
$$

it clearly follows that $[\sigma_i]_i = [\tau_i]_i$.

□

Similar proofs also suffice to establish (I1) and (I2) for the domain model of $\mathsf{DTT}_{\mathsf{CBN}}$. (I3) relies on a crucial difference between the domain and games models: our identity types compare strategies in intension rather than in extension. For similar reasons, FunExt is seen to fail in the games model.

The principle FunExt of function extensionality intuitively states that, from the point of view of the Id-types, functions are extensional objects: black boxes

which merely send inputs to outputs without any internal temporal structure. It is refuted in our model.

Theorem 4.3.5. FunExt *is refuted: for* $\vdash f, g : \Pi_{x:A} B$*, we do not generally have* $z : \Pi_{x:A} \mathsf{Id}_B(f(x), g(x)) \vdash \mathsf{FunExt}_{f,g} : \mathsf{Id}_{\Pi_{x:A} B}(f, g)$.

Proof. For our counter example, we let $[\![A]\!] = [\![B]\!] = [\mathbb{B}_*]$.

Let f be the usual strict strategy that outputs tt (and examines its argument once) and let g by the strategy which always outputs tt but first examines its input twice. Noting that always $[\![f]\!]\{x\} = [\![g]\!]\{x\}$ for all strategies x on $\mathbb{B}_*$, we have an inhabitant $\mathtt{refl} \in \mathsf{str}(\mathsf{O\text{-}sat}(\Pi_{\mathbb{B}_*}\{\mathtt{refl}\}_*)) = \mathsf{Tm}([\mathbb{B}_*], \mathsf{Id}_{[\mathbb{B}_*]}([\![f]\!], [\![g]\!]))$. However, as not $[\![f]\!] = [\![g]\!]$, we do not have an inhabitant of

$$\mathsf{Tm}([], [\mathsf{Id}_{\Pi_{[\mathbb{B}_*]}[\mathbb{B}_*]}]([\![f]\!], [\![g]\!])) = \mathsf{Tm}([], [\emptyset_*]) = \mathsf{str}(\emptyset_*) = \emptyset.$$

□

On the other hand, it turns out that we do have the principle of uniqueness of identity proofs UIP, as the strict strategy which first examines the first copy of $[\![\mathsf{Id}_A]\!]$ and then the second, before replying $\mathtt{refl}$ in $[\![\mathsf{Id}_{\mathsf{Id}_A}]\!]$. We choose this more complicated witness rather than a non-strict one as this will generalise to settings where we consider a broader class of strategies. This principle intuitively says that types have trivial (discrete) spatial structure, from the point of view of the Id-types.

Theorem 4.3.6. *We have* $x, y : A, p, q : \mathsf{Id}_A(x, y) \vdash \mathsf{UIP}_A : \mathsf{Id}_{\mathsf{Id}_A(x,y)}(p, q)$.

Proof. Player can open a copy of $[\![\mathsf{Id}_A]\!]$ as the initial move $*$ is in each fibre. After Player opens a copy of $[\![\mathsf{Id}_A]\!]$, Opponent can only reply $\mathtt{refl}$. Player can then play $\mathtt{refl}$ in $[\![\mathsf{Id}_{\mathsf{Id}_A}]\!]$ as it is in all possible fibres. □

4.4 Ground Types: Finite Dependent Games

In this section, we show how we can additionally give finite inductive type families an interpretation if we restrict to a full subcategory $\mathsf{Ctxt}(\mathsf{DGame}_!)^{\mathsf{fin1\Sigma\Pi Id}}$ of $\mathsf{Ctxt}(\mathsf{DGame}_!)$. Indeed, we use the full subcategory $\mathsf{Ctxt}(\mathsf{DGame}_!)^{\mathsf{fin1\Sigma\Pi Id}}$ on the

hierarchy of context games generated by the semantic constructions interpreting 1-, Σ-, Π- and Id-types and substitution, starting from finite dependent games (as defined below). These finite dependent games will play the rôle of semantic ground types to build a type hierarchy for which we prove completeness results in the next section.

We consider the interpretation of $\mathsf{DTT}_{\mathsf{CBN}}-$ in $\mathsf{Ctxt}(\mathsf{DGame}_!)^{\mathsf{fin1\Sigma\Pi Id}}$ and will denote the interpretation functor by $[\![-]\!]$.

Theorem 4.4.1 (Finite Dependent Game)**.** *A finite inductive type family* $B := (a_i \mapsto_i \{b_{i,j} \mid j\})(x)$ *in context* $x : A$*, where* $B[a_i/x]$ *is generated by* $\{b_{ij} \mid 1 \leq j \leq m_i\}$*, has an interpretation in* $\mathsf{Ctxt}(\mathsf{DGame}_!)^{\mathsf{fin1\Sigma\Pi Id}}$ *as a* ***finite dependent game****:*

$$[\![B]\!] : [\![a_i]\!] \longmapsto \{b_{i,j} \mid j\}_* \qquad \mathsf{else} \longmapsto \emptyset_*$$

and

$$☺([\![B]\!]) = \{b_{i,j} | i, j\}_*.$$

Proof. To be explicit, we interpret the $\mathsf{case}^{p,q}$-constructs rather than the (equivalent) case-constructs, as we shall be using the former later.

The interpretation of the I-rules is clear: $[\![b_{i,j}]\!]$ is the unique strategy on $[\![B[a_i/x]]\!]$ that replies to $*$ with the move $b_{i,j}$.

We inductively construct $[\![\mathsf{case}^{p,q}_{B[a/x],C}(b, \{c_{i,j}\}_{i,j})]\!] : [\![\cdot]\!] \longrightarrow [\![\Pi_{A'} C[b/y]]\!]$, with structural induction on C (apart from the case of $C = 1$, which is trivial). We consider the (more general) base case of arbitrary $[\![C]\!]$ that assign to each $\sigma \in \mathsf{str}(☺([\![A']\!])\&☺([\![B]\!]))$ a finite inductive game with initial move $*$. After that, the case constructs for more general C are obtained from the commutative conversions for Σ- and Π-types induced from those of figure 2.3. Note that substitutions and Id-types are already dealt with because we have been considering the more general base case where some of the constructors of $[\![C]\!]$ can coincide, while substitution commutes with Π and Σ.

Let us consider our base case. We define $[\![\mathsf{case}^{p,q}_{B[a/x],C}(b, \{c_{i,j}\}_{i,j})]\!]$ by noting that

$$[\![\mathsf{case}_{B^T,C^T}]\!]([\![b^T]\!], \{[\![c^T_{i,j}]\!](\mathsf{der}_{A'}, \mathtt{refl}, \mathtt{refl})\}_{i,j})$$

in fact defines a (winning) strategy on $[\![\Pi_{A'} C[b/y]]\!]$, where $(-)^T$ is the syntactic translation from section 2.1.1.

We verify that this yields a strategy $[\![\mathsf{case}^{p,q}_{B[a/x],C}(b, \{c_{i,j}\}_{i,j})]\!]$ on $[\![\Pi_{A'} C[b/y]]\!]$ (which clearly automatically is winning, as usual, as we never restrict O-moves in games of dependent functions). Let us write $[\![A']\!] = [X_i]_i$, so $[\![\mathsf{case}^{p,q}_{B[a/x],C}(b, \{c_{i,j}\}_{i,j})]\!]$ will be a strategy on $\mathsf{O\text{-}sat}(\Pi_{X_1} \cdots \Pi_{X_n} [\![C[b/y]]\!])$. Let $sab \in [\![\mathsf{case}^{p,q}_{B[a/x],C}(b, \{c_{i,j}\}_{i,j})]\!]$. Then, we verify that for all $\overline{sab \restriction_{!\☺(X_1)}} \subseteq \sigma'_1 \in \mathsf{str}(X_1()), \ldots \overline{sab \restriction_{!\☺(X_n)}} \subseteq \sigma'_n \in \mathsf{str}(X_n(\sigma'_1, \ldots, \sigma'_{n-1}))$, we have that $(*)$

$$sab \restriction_{!\☺(X_1)} \in P_{!X_1()} \wedge \cdots \wedge sab \restriction_{!\☺(X_n)} \in P_{!X_n(\sigma'_1,\ldots,\sigma'_{n-1})} \wedge sab \restriction_{\☺([\![C]\!])} \in P_{[\![C[b/y]]\!](\sigma'_1,\ldots,\sigma'_n)},$$

provided that already

$$sa \restriction_{!\☺(X_1)} \in P_{!X_1()} \wedge \cdots \wedge sa \restriction_{!\☺(X_n)} \in P_{!X_n(\sigma'_1,\ldots,\sigma'_{n-1})} \wedge sa \restriction_{\☺([\![C]\!])} \in P_{[\![C[b/y]]\!](\sigma'_1,\ldots,\sigma'_n)}.$$

Because of the type $[\![\Pi_{A'} B[a/x]]\!]$ of $[\![b]\!]$, all Player moves of $[\![\mathsf{case}^{p,q}_{B[a/x],C}(b, \{c_{i,j}\}_{i,j})]\!]$ respect the type $[\![\Pi_{A'} C[b/y]]\!]$ at least until some $[\![c_{i,j}]\!]$ is called. Now, the crux is that $[\![c_{i,j}]\!]$ is only ever called after $[\![b]\!]$ has already replied with the move $b_{i,j}$. This means that for any $[\sigma'_i]_i$ we are considering, we have that $[\sigma'_i]_i; [\![b]\!] = [\![b_{i,j}]\!]$. Moreover, because of the type of b, we have that $[\![b_{i,j}]\!] = [\sigma'_i]_i; [\![b]\!] = [\![b]\!]\{[\sigma'_i]_i\}$ is a winning strategy on $[\![B]\!]\{[\![a]\!]\{[\sigma'_i]_i\}\}$, while $[\![a]\!]\{[\sigma'_i]_i\}$ is a winning strategy on $[\![A]\!]$. Therefore, because of the definition of $[\![B]\!]$, we conclude that $[\sigma'_i]_i; [\![a]\!] = [\![a_i]\!]$, as the fibres of B are disjoint. The upshot is that the semantic type $[\![\Pi_{x':A'} \Pi_{p_{i,j}:\mathsf{Id}_A(a_i,a)} \Pi_{q_{i,j}:\mathsf{Id}_{B[a/x]}(\mathsf{subst}(p_{i,j},b_{i,j}),b)} C[b/y]]\!]$ of $[\![c_{i,j}]\!]$ now gives us that the continuation of the play along $[\![c_{i,j}]\!](\mathsf{der}_{A'}, \mathtt{refl}, \mathtt{refl})$ still respects our condition $(*)$. □

We have obtained the following.

Corollary 4.4.2. $\mathsf{DTT}_{\mathsf{CBN}}-$ *has a sound interpretation in* $\mathsf{Ctxt}(\mathsf{DGame}_!)^{\mathsf{fin1}\Sigma\Pi\mathsf{Id}}$.

We turn to the issue of soundness of the interpretation of $\mathsf{DTT}_{\mathsf{CBN}}$ in the next section.

4.5 Soundness, Faithfulness and Completeness

In this section, we show that the interpretation of $\mathsf{DTT}_{\mathsf{CBN}}$ in $\mathsf{Ctxt}(\mathsf{DGame}_!)^{\mathsf{fin1}\Sigma\Pi\mathsf{Id}}$ is sound and faithful and, if we limit Id-types to only occur strictly positively and at most once, that it is, additionally, fully complete. The proof of soundness and faithfulness follows from the fact that our game semantics for $\mathsf{DTT}_{\mathsf{CBN}}$ factors faithfully over the usual game semantics for simple type theory. The proof of definability proceeds in five steps:

1. interpreting a dependently typed strategy f on a larger (simply typed) game;

2. for a strict f, performing the decomposition of [22] in the simply typed world, as usual, to obtain simply typed strategies g^j and h_y that are called in the execution of f;

3. noting that these g^j and h_y can actually be assigned a more precise dependent type, the trick being that we accumulate appropriate negatively occurring Id-types as the decomposition proceeds inductively;

4. observing that the iterated decomposition of strict strategies strictly decreases a positive integer norm and therefore eventually terminates after finitely many steps, producing only non-strict strategies;

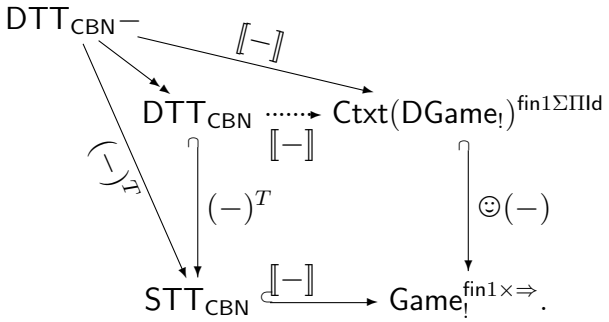
5. for a non-strict f, noting that f is directly definable using the constructors $b_{i,j}$ for finite type families and $\mathsf{Ty\text{-}Ext}$.

4.5.1 Soundness and Faithfulness

We first prove faithfulness of the interpretation of $\mathsf{DTT}_{\mathsf{CBN}}$ in our model.

Theorem 4.5.1 (Soundness and Faithfulness)**.** *The interpretation* $[\![-]\!]$ *of* $\mathsf{DTT}_{\mathsf{CBN}}$ *in* $\mathsf{Ctxt}(\mathsf{DGame}_!)^{\mathsf{fin1}\Sigma\Pi\mathsf{Id}}$ *is sound and faithful. The rule* $\mathsf{Ty\text{-}Ext}$ *is sound for all type families* $x : A \vdash B$ *over a type* A *for which definability holds.*

Proof. We note that we have the following commutative diagram of (non-dashed) functors, where, in the light of corollary 4.4.2, soundness amounts to arguing that our interpretation of $\mathsf{DTT}_{\mathsf{CBN}}-$ factors over $\mathsf{DTT}_{\mathsf{CBN}}$ (denoting the factorisation with the dashed functor)



Here, the top and bottom sides of the outer quadrangle, respectively are the interpretation functor of $\mathsf{DTT}_{\mathsf{CBN}}-$ in our model, which exists according to corollary 4.4.2, and the usual interpretation of simple type theory with finite ground types (or, a total finitary PCF, if you will) in the cartesian category of games and (winning) strategies of [22]. Recall that the latter is (full and) faithful according to theorem 2.4.12. The left side of the inner rectangle is the faithful (non-full) functor defined in section 2.1.1.4. Note that faithfulness of the interpretation of $\mathsf{DTT}_{\mathsf{CBN}}$ automatically follows from the faithfulness of these two functors, if we can prove soundness. Finally, the right side of either quadrangle is the semantic equivalent of this syntactic translation, which we define next.

We have an inductively defined translation $\mathsf{Ctxt}(\mathsf{DGame}_!) \xrightarrow{☺(-)} \mathsf{Game}_!$:

$$☺([A_i]_{1\leq i\leq m}) := \bigamp_{1\leq i\leq m} ☺(A_i)$$
$$☺([]) := I.$$

Note that this also satisfies

$$☺(\Pi_{A_1}\cdots\Pi_{A_n}B) = ☺(A_1) \Rightarrow \cdots \Rightarrow ☺(A_n) \Rightarrow ☺(B)$$
$$☺(\mathsf{Id}_C) = \{\mathtt{refl}\}_*.$$

$\mathbb{\odot}$ automatically extends to a faithful (non-full) functor by interpreting the winning dependently typed strategies on A as simply typed strategies on $\mathbb{\odot}(A)$, which are obviously also winning as we never restrict Opponent moves in our games of dependent functions. Faithfulness of this functor together with commutativity of the outer quadrangle gives us that the dashed arrow is a (unique) well-defined functor, i.e. we have a sound interpretation of $\mathsf{DTT}_{\mathsf{CBN}}$ in $\mathsf{Ctxt}(\mathsf{DGame}_!)^{\mathsf{fin1\Sigma\Pi Id}}$.

We also note that Ty-Ext has a sound interpretation in our model for types B depending on a type $A = [A_i]_i$ for which all morphisms are definable. Indeed, it follows that $\Pi_A B_1 = \Pi_A B_2$ if $\mathbb{\odot}(B_1) = \mathbb{\odot}(B_2)$ and for all $[] \xrightarrow{t} [A_i]_{1 \leq i \leq n}$ we have that $B_1(t) = B_2(t)$. The reason is that the definition of Π-games only relies on $\mathbb{\odot}(B)$ and the evaluation of B_i on these consistent tuples t. If all such t are definable, Ty-Ext follows. □

4.5.2 Full Completeness

Next, we first prove two technical lemmas, which encompass steps 2. and 3. and, respectively, step 4. in the definability proof. We use the notation $[\mathsf{Id}]_{[A_i]_i}([a_i]_i, [a'_i]_i) := [\mathsf{Id}_{A_i}(a_i, a'_i)]_i$.

Lemma 4.5.2 (Decomposition)**.** *Let us suppose we have a context game $[A_i]_{i \leq n}$ in $\mathsf{Ctxt}(\mathsf{DGame}_!)^{\mathsf{fin1\Sigma\Pi Id}}$ with $A_i = \Pi_{B^{i,1}} \dots \Pi_{B^{i,q_i}} Y^i_* = \Pi_{[B^{i,j}]_j} Y^i_*\{c^i\}$ where Y^i_* is a finite inductive dependent game depending on the context game $[C^i_l]_l$ and $[A_k]_{k<i}.[B^{i,j}]_j \xrightarrow{c^i} [C^i_l]_l$. Let us say Y^i_* has constructors y in fibre $Y^i_*\{[c^i_y]_i\}$.*

Then, it follows that, when given a strategy f that does not visit $[\mathsf{Id}]_{[D_k]_k}$,

$$f \in \mathsf{str}(\mathsf{O\text{-}sat}(\Pi_{[A_i]_i} \Pi_{[\mathsf{Id}]_{[D_k]_k}([d^0_k]_k, [d_k]_k)} X_*)),$$

with $\mathsf{str}(\mathbb{\odot}(A_1) \& \dots \& \mathbb{\odot}(A_n)) \xrightarrow{X} \mathcal{P}(\mathbb{\odot}(X))$ a function, where $\mathbb{\odot}(X)$ is some finite set, and context morphisms $[d_k]_k, [d^0_k]_k : [A_i]_i \longrightarrow [D_k]_k$, we can decompose it (uniquely) as follows:

- *if f is non-strict, then $f = [A_i]_i \longrightarrow [] \xrightarrow{x} [\mathbb{\odot}(X_*)]$ for some $x \in \bigcup \mathsf{im}(X)$ such that $x \in X_*([\tau_i]_i)$ for all $[] \xrightarrow{[\tau_i]_i} [A_i]_i$ such that $[d_k]_k\{[\tau_i]_i\} = [d^0_k]_k\{[\tau_i]_i\}$;*

- *if f is strict, then $f = \mathbf{C}_i(g^1, \ldots, g^{q_i}, (h_y \mid y \in \bigcup \mathsf{im}(Y^i)))$ where $\mathbf{C}_i$ embodies a* case*-construct that we shall define in the proof,*

where

$$g^j \in \mathsf{str}(\mathsf{O\text{-}sat}(\Pi_{[A_i]_i}\Pi_{[\mathsf{Id}]_{[D^k]_k}([d_k^0]_k,[d_k]_k)}B^{i,j}\{\langle[\mathsf{der}_{A_l}]_{l<i}, [g^{j'}]_{j'<j}\rangle\}))$$

and

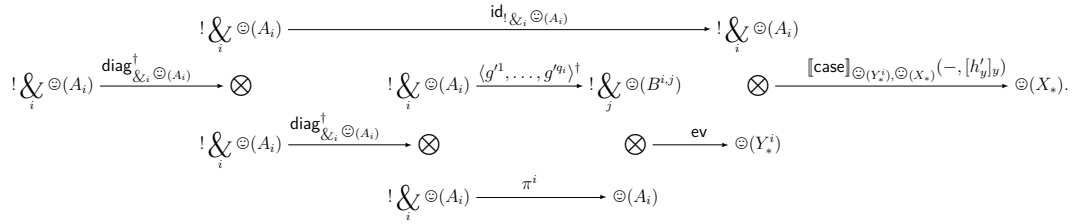
$$h_y \in \mathsf{str}(\mathsf{O\text{-}sat}(\Pi_{[A_i]_i}\Pi_{[\mathsf{Id}]_{[D^k]_k\cdot[C_l^i]_l\cdot[Y_*^i]}(\langle[d_k^0]_k,[c_y^i],[y]\rangle,\langle[d_k]_k,[\widetilde{c^i}],[\phi]\rangle)}X_*)),$$

where $\widetilde{c^i} := \langle[\mathsf{der}_{A_{i'}}]_{i'<i}, [g_j]_j\rangle; c^i$ and $\phi := \lambda_{[\tau_k]_k}\tau_i\{[g^j]_j\{[\tau_k]_k\}\}$ (and we write $\mathsf{im}(Y^i)$ for the image of Y^i and $\lambda_{[\tau_k]_k}$ for the obvious semantic λ-abstraction). Here, neither g^j nor h_y visits the Id*-type.*

Proof. Note that we can consider $☺(f)$ as a strategy on $☺(A_1) \Rightarrow \cdots \Rightarrow ☺(A_n) \Rightarrow ☺(X_*)$ as f does not visit the Id-type. The decomposition lemma [22, 56] for the game semantics of (finitary) PCF now gives us three cases:

- $☺(f) = \bot$
- $☺(f) = \&_i ☺(A_i) \longrightarrow I \xrightarrow{x} ☺(X_*)$ for some $x \in \bigcup \mathsf{im}(X)$;
- $☺(f) = \mathbf{C}'_i(g'^1, \ldots, g'^{q_i}, (h'_y \mid y \in \bigcup \mathsf{im}(Y^i)))$, for a (unique) $1 \leq i \leq n$ and (unique) $g'^j \in \mathsf{str}(☺(A_1) \Rightarrow \cdots \Rightarrow ☺(A_n) \Rightarrow ☺(B^{i,j}))$ and $h'_y \in \mathsf{str}(☺(A_1) \Rightarrow \cdots \Rightarrow ☺(A_n) \Rightarrow ☺(X_*))$, where (writing π^i for the derelicted projection to the i-th component, ev for the obvious evaluation morphism, and denoting the semantic case construct with $[\![\mathsf{case}]\!]$)

$$\mathbf{C}'_i(g'^1, \ldots, g'^{q_i}, (h'_y \mid y \in \bigcup \mathsf{im}(Y^i))) :=$$



Note that the first case cannot occur as f is winning.

For the second case, due to the restriction on P-moves in Π-games and the interpretation of Id-types, a non-strict f needs to respond to $*$ with a move in $\bigcup \mathsf{im}(X)$ such that $x \in X_*([\tau_i]_i)$ for all $[] \xrightarrow{[\tau_i]_i} [A_i]_i$ such that $[d_k]_k\{[\tau_i]_i\} = [d^0_k]_k\{[\tau_i]_i\}$.

For the third case, note the following.

- $g'^j = \mathrel{\textcircled{\scriptsize ☺}}(g^j)$ for (unique)

$$g^j \in \mathsf{str}(\mathsf{O\text{-}sat}(\Pi_{[A_i]_i}\Pi_{[\mathsf{Id}]_{[D^k]_k}([d^0_k]_k,[d_k]_k)}B^{i,j}\{\langle[\mathsf{der}_{A_l}]_{l<i}, [g^{j'}]_{j'<j}\rangle\})).$$

  This will follow once we show that

$$((g^j)^\dagger)^\dagger \in \mathsf{str}(\mathsf{O\text{-}sat}(\Pi_{[A_i]_i}\Pi_{[\mathsf{Id}]_{[D^k]_k}([d^0_k]_k,[d_k]_k)}!!B^{i,j}\{\langle[\mathsf{der}_{A_l}]_{l<i}, [g^{j'}]_{j'<j}\rangle\})),.$$

  The argument will proceed by complete induction on j. Assume the claim holds for g^k with $k < j$. We show it also holds for g^j.

  We need to show that for $s^j = s'ab \in ((g^j)^\dagger)^\dagger$, for any $\overline{s'ab\restriction_{!\mathrel{\textcircled{\scriptsize ☺}}(A_1)}} \subseteq \tau_1 \in \mathsf{str}(A_1()), \ldots, \overline{s'ab\restriction_{!\mathrel{\textcircled{\scriptsize ☺}}(A_n)}} \subseteq \tau_n \in \mathsf{str}(A_n(\tau_1, \ldots, \tau_{n-1}))$ s.t. $[\tau_i]_i; [d^0_k]_k = [\tau_i]_i; [d_k]$, $s'a \in P_{A_1()\Rightarrow\cdots\Rightarrow A_n(\tau_1,\ldots,\tau_{n-1})\Rightarrow !!B^{i,j}(\tau_1,\ldots,\tau_{i-1},\langle\tau_1,\ldots,\tau_n\rangle;g^1,\ldots,\langle\tau_1,\ldots,\tau_n\rangle;g^{j-1})}$ implies that $s'ab \in P_{A_1()\Rightarrow\cdots\Rightarrow A_n(\tau_1,\ldots,\tau_{n-1})\Rightarrow !!B^{i,j}(\tau_1,\ldots,\tau_{i-1},\langle\tau_1,\ldots,\tau_n\rangle;g^1,\ldots,\langle\tau_1,\ldots,\tau_n\rangle;g^{j-1})}$.

  Let us assume that the hypothesis of this implication is true. Now, note that $s^j \in ((g^j)^\dagger)^\dagger$ extends to $tab = *_{X_*}(0,*)_{!Y^i_*}s^1\cdots s^{j-1}s^j \in f$ for any $s^k \in ((g^k)^\dagger)^\dagger$, for $1 \leq k \leq j-1$. We can choose $s^k \in \langle\tau_1, \ldots, \tau_n\rangle||((g^k)^\dagger)^\dagger$ such that $\bigcup\overline{\overline{s^k\restriction_{!!\mathrel{\textcircled{\scriptsize ☺}}(B^{i,k})}}} = \langle\tau_1, \ldots, \tau_n\rangle; g^k$. (We write $\overline{\overline{s}}$ to indicate we apply $\overline{(-)}$ first to s and then again to each member of the resulting set of plays.) Note that we can do this as $\langle\tau_1, \ldots, \tau_n\rangle; g^k$ is finite as a partial function on moves. In fact,

$$s^k \in P_{A_1()\Rightarrow\cdots\Rightarrow A_n(\tau_1,\ldots,\tau_{n-1})\Rightarrow !!B^{i,k}(\tau_1,\ldots,\tau_{i-1},\langle\tau_1,\ldots,\tau_n\rangle;g^1,\ldots,\langle\tau_1,\ldots,\tau_n\rangle;g^{k-1})},$$

  as a consequence of our induction hypothesis.

Then, as $tab \in f$ is a play in

$$\mathsf{O\text{-}sat}(\Pi_{A_1} \cdots \Pi_{A_{i-1}} \Pi_{(\Pi_{B^{i,1}} \cdots \Pi_{B^{i,q_i}} Y^i_*)} \Pi_{A_{i+1}} \cdots \Pi_{A_n} \Pi_{[\mathsf{Id}]_{[D^k]_k}([d^0_k]_k,[d_k]_k)} X_*),$$

we have that for all

$$\overline{tab \restriction_{!\textcircled{\scriptsize\frown}(A_1)}} \subseteq \tau'_1 \in \mathsf{str}(A_1()), \ldots, \overline{tab \restriction_{!\textcircled{\scriptsize\frown}(A_n)}} \subseteq \tau'_n \in \mathsf{str}(A_n(\tau'_1, \ldots, \tau'_{n-1}))$$

s.t. $[\tau'_i]_i; [d^0_k]_k = [\tau'_i]_i; [d_k]$, $ta \in P_{A_1() \Rightarrow \cdots \Rightarrow A_n(\tau'_1,\ldots,\tau'_{n-1}) \Rightarrow X_*(\tau'_1,\ldots,\tau'_n)}$ implies that also $tab \in P_{A_1() \Rightarrow \cdots \Rightarrow A_n(\tau'_1,\ldots,\tau'_{n-1}) \Rightarrow X_*(\tau'_1,\ldots,\tau'_n)}$. Note that by construction of tab, $[\tau_i]_i$ is one such $[\tau'_i]_i$ and is in fact the only one we are interested in, so we simply write $[\tau_i]_i$ for both. Note that the hypothesis of the implication under consideration actually holds by our assumptions about $s'a$ and s^k. Therefore, its conclusion $tab \in P_{A_1() \Rightarrow \cdots \Rightarrow A_n(\tau_1,\ldots,\tau_{n-1}) \Rightarrow X_*(\tau_1,\ldots,\tau_n)}$ follows.

Now, it follows immediately from the restriction on plays tab in

$$\mathsf{O\text{-}sat}(\Pi_{A_1} \cdots \Pi_{A_{i-1}} \Pi_{(\Pi_{B^{i,1}} \cdots \Pi_{B^{i,q_i}} Y^i_*)} \Pi_{A_{i+1}} \cdots \Pi_{A_n} \Pi_{[\mathsf{Id}]_{[D^k]_k}([d^0_k]_k,[d_k]_k)} X_*)$$

that if $((g^j)^\dagger)^\dagger$ makes the move b in $!\textcircled{\scriptsize\frown}(A_l)$, then it also satisfies the rules of $\mathsf{O\text{-}sat}(\Pi_{[A_i]_i} \Pi_{\mathsf{Id}_{[D^k]_k}([d^0_k]_k,[d_k]_k)} !!B^{i,j}\{\langle [\mathsf{der}_{A_l}]_{l<i}, [g^{j'}]_{j'<j} \rangle\})$. The interesting case is when b is a move in $!!B^{i,j}$. Let us presume that Opponent has not been naughty. (Otherwise, anything goes.) To deal with this case, we note that the restriction on plays tab in

$$\mathsf{O\text{-}sat}(\Pi_{A_1} \cdots \Pi_{A_{i-1}} \Pi_{(\Pi_{B^{i,1}} \cdots \Pi_{B^{i,q_i}} Y^i_*)} \Pi_{A_{i+1}} \cdots \Pi_{A_n} \Pi_{[\mathsf{Id}]_{[D^k]_k}([d^0_k]_k,[d_k]_k)} X_*)$$

combined with the definition of $(\Pi_{B^{i,1}} \cdots \Pi_{B^{i,q_i}} Y^j{}_*)(\tau_1, \ldots, \tau_{i-1})$ gives us that there exist $\bigcup \overline{\overline{tab \restriction_{!!\textcircled{\scriptsize\frown}(B^{i,1})}}} \subseteq \sigma^1 \in \mathsf{str}(B^{i,1}(\tau_1, \ldots, \tau_{i-1})), \ldots, \bigcup \overline{\overline{tab^{\restriction}_{!!\textcircled{\scriptsize\frown}(B^{i,q_i})}}} \subseteq \sigma^{q_i} \in \mathsf{str}(B^{i,q_i}(\tau_1, \ldots, \tau_{i-1}, \sigma^1, \ldots, \sigma^{q_i-1}))$, such that

$$tab \restriction_{!\textcircled{\scriptsize\frown}(A_i)} \in P_{!(B^{i,1}(\tau_1,\ldots,\tau_{i-1}) \Rightarrow \cdots \Rightarrow B^{i,q_i}(\tau_1,\ldots,\tau_{i-1},\sigma^1,\ldots,\sigma^{q_i-1}) \Rightarrow Y^i{}_*(\tau_1,\ldots,\tau_{i-1},\sigma^1,\ldots,\sigma^{q_i}))}$$

so, in particular, $tab \restriction_{!\textcircled{\scriptsize\frown}(A_i)} \restriction_{!!\textcircled{\scriptsize\frown}(B^{i,j})} \in P_{!!B^{i,j}(\tau_1,\ldots,\tau_{i-1},\sigma^1,\ldots,\sigma^{j-1})}$.

To complete the argument, we note that by construction of t, we have that $\bigcup \overline{\overline{tab \restriction_{!!\textcircled{\scriptsize\frown}(B^{i,k})}}} = \langle \tau_1, \ldots, \tau_n \rangle; g^k$, for $1 \leq k \leq j-1$. We conclude that $s'ab \restriction_{!!B^{i,j}} \in P_{!!B^{i,j}(\tau_1,\ldots,\tau_{i-1},\langle\tau_1,\ldots,\tau_n\rangle;g^1,\ldots,\langle\tau_1,\ldots,\tau_n\rangle;g^{j-1})}$.

- $h'_y = \copyright(h_y)$ for (unique)

$$h_y \in \mathsf{str}(\mathsf{O\text{-}sat}(\Pi_{A_1} \cdots \Pi_{A_n} \Pi_{[\mathsf{Id}]_{[D^k]_k \cdot [C^i_l]_l \cdot [Y^i_*]}(\langle [d^0_k]_k, [c^i_y], [y] \rangle, \langle [d_k]_k, [\widetilde{c^i}], [\phi] \rangle)} X_*)).$$

Indeed, note that $*s \in h_y$ iff $*(0,*)t(0,y)s \in f$ for some $*ty \in \phi \in \mathsf{str}(\mathsf{O\text{-}sat}(\Pi_{A_1} \cdots \Pi_{A_n} \Pi_{[\mathsf{Id}]_{[D^k]_k}([d^0_k],[d_k]_k)} Y^i_*\{\langle [\mathsf{der}_{A_l}]_{l<i}, [g^j]_j \rangle\}))$. It then follows that $*s \in \mathsf{O\text{-}sat}(\Pi_{A_1} \cdots \Pi_{A_n} \Pi_{[\mathsf{Id}]_{[D^k]_k \cdot [C^i_l]_l \cdot [Y^i_*]}(\langle [d^0_k]_k, [c^i_y], [y] \rangle, \langle [d_k]_k, [\widetilde{c^i}], [\phi] \rangle)} X_*)$ by the following observation. Observing that $\phi\{[\overline{t \restriction_{!A_i}}]_i\} = y$, note that, for winning $[\tau_i]_i \geq [\overline{*s \restriction_{!A_i}}]_i$, we also have that $[\tau_i]_i \geq [\overline{*(0,*)t(0,y)s \restriction_{!A_i}}]_i$ for some $*ty \in \phi$ iff $\phi\{[\tau_i]_i\} \geq y$ i.e. $\phi\{[\tau_i]_i\} = y$ as y is a maximal strategy on $\copyright(Y_*)$. (Indeed, we can take $*ty \in \langle \tau_1, \ldots, \tau_n \rangle || \phi$.) It automatically then follows that also $\widetilde{c^i}\{[\tau_i]_i\} = c^i_y$, by the type of $\langle [\widetilde{c^i}], [\phi] \rangle$ and the disjointness of fibres.

- We can now note that $f = \mathbf{C}_i(g^1, \ldots, g^{q_i}, (h_y \mid y \in \bigcup \mathsf{im}(Y^i)))$, where $\mathbf{C}_i$ is defined exactly as $\mathbf{C}'_i$ but using instead the dependently typed substitution and the dependently typed construct $[\![\mathsf{case}^{p,q}]\!]_{Y^i_*\{\langle [\mathsf{der}_{A_l}]_{l<i}, [g^j]_{j \leq q_i} \rangle\}, X_*}$. (That is, $\mathbf{C}_i$ and $\mathbf{C'}_i$ are the same, except for typing.) Note that $\mathsf{ev}(\pi^i, \langle g^1, \ldots, g^{q_i} \rangle)$ and h_y feed into this case construct.

- Finally, to see that g^j and h_y define winning strategies, we note that their infinite plays are Player-wins as they arise as labelled subtrees of f which is winning. We need to verify that they are total. This also follows immediately from the totality of f together with the fact that Opponent moves are, by definition, not restricted in (O-saturated) games of dependent functions. Indeed, if $s \in g^j$ and sa is a valid extension of the play, then $*(0,*)sa$ is a valid extension of $*(0,*)s \in f$ to which f hence g^j has a response b. Similarly, if $*s \in h_y$ and $*sa$ is a valid extension of the play, then there exists a suitable t such that $*(0,*)t(0,y)sa$ is a valid extension of $*(0,*)t(0,y)s \in f$ to which f has a response b, being a total strategy. Therefore, $*sab \in h_y$.

□

Lemma 4.5.3 (Norm for dependent strategies)**.** *Let* $[A_i]_i \xrightarrow{[d_k]} [D_k]_k$ *and* X_* *as in the previous lemma. Let us write* $E := \Pi_{[A_i]_i}\Pi_{[\mathsf{Id}]_{[D^k]_k}([d_k^0],[d_k]_k)}X_*$*. Then, we have a norm* $||-||_E : \mathsf{str}(\mathsf{O\text{-}sat}(E)) \longrightarrow \mathbb{N}$ *(we sometimes leave out the subscript* E*) for any such* E *such that* $f = \mathbf{C}_i(g^1, \ldots, g^{q_i}, (h_y \mid y \in \bigcup \mathsf{im}(Y^i)))$ *implies that*

$$||g^i||, ||h_y|| < ||f||.$$

Proof. We define a norm $||-||_{☺(E)} : \mathsf{str}(☺(E)) \longrightarrow \mathbb{N}$ for games $☺(E)$ of the $I\& \Rightarrow$-hierarchy over finite flat games and extend this to a norm on $\mathsf{str}(\mathsf{O\text{-}sat}(E))$ by precomposition with the injection $\mathsf{str}(\mathsf{O\text{-}sat}(E)) \xrightarrow{☺(-)} \mathsf{str}(☺(E))$. The idea behind this norm is that winning strategies on games of the $I\& \Rightarrow$-hierarchy over finite flat games are finite objects in the sense that they only contain finitely many finite plays if we do not allow Opponent to open multiple threads of the same game – remember that infinite plays in winning strategies are always due to Opponent opening an infinite number of threads of the same game.

Inductively, if T is a type of $\mathsf{STT}_{\mathsf{CBN}}$ (i.e. formed from finite ground types G by the grammar $T ::= G \mid \top \mid \& \mid \Rightarrow$), we define a type LT of intuitionistic linear logic over finite types (i.e. formed from finite ground types G by the grammar $LT ::= G \mid !LT \mid LT \multimap LT \mid LT \otimes LT \mid LT\&LT \mid I \mid \top$, where we note that in our interpretation $[\![\top]\!] = [\![I]\!]$ and where we identify the cartesian type $A \Rightarrow B$ with the linear type $!A \multimap B$) by removing each positive occurrence of ! in T or, equivalently, replacing each even-depth occurrence of $\Rightarrow$ with $\multimap$. Essentially, $[\![LT]\!]$ is obtained from the game $[\![T]\!]$ by not allowing Opponent to open more than one thread of any game. Note that we have a canonical winning strategy representing a generalised dereliction $[\![T]\!] \xrightarrow{\mathsf{gder}_{[\![LT]\!]}} [\![LT]\!]$ which is defined in the obvious way from dereliction maps on subtypes using the functoriality of $\top$, $\&$, $\Rightarrow$ and $\multimap$.

Now, if we can show that $W_{[\![LT]\!]} = \emptyset$, it follows that the norm $||\sigma||_{[\![T]\!]} := \Sigma_{s \in \sigma; \mathsf{gder}_{[\![LT]\!]}/\approx_{[\![LT]\!]}} \mathsf{length}(s)$ is well-defined for $\sigma \in \mathsf{str}([\![T]\!])$. (Here, we mean some skeleton for $\sigma; \mathsf{gder}_{[\![LT]\!]}$ when we write $\sigma; \mathsf{gder}_{[\![LT]\!]} / \approx_{[\![LT]\!]}$,) Indeed, there are only finitely many Opponents for $[\![LT]\!]$ as Opponent can only make a choice between finitely many alternatives for each connective in formula LT, of which there are

finitely many. Moreover, interactions with Player never become unboundedly long because $W_{[\![LT]\!]} = \emptyset$.

We show that $W_{[\![LT]\!]} = \emptyset$. Define classes of formulas $\mathtt{AllWin}, \mathtt{NoWin}$ by mutual induction as follows. In this definition, we use G to stand for any game all of whose maximal positions are of length 2, F^A (respectively, F^N) and their subscripted versions to range over $\mathtt{AllWin}$ (respectively, $\mathtt{NoWin}$) games.

$$\begin{aligned} \mathtt{AllWin} :: \quad & G \mid F_1^A \otimes F_2^A \mid \ F_1^A \& F_2^A \mid !F^A \mid F^N \multimap F^A \\ \mathtt{NoWin} :: \quad & G \mid F_1^N \otimes F_2^N \mid \ F_1^N \& F_2^N \mid F^A \multimap F^N. \end{aligned}$$

It follows from a simple inductive argument that

- for all $\mathtt{AllWin}$ games F^A, $W_{F^A} = P^{\infty}_{F^A}$;
- for all $\mathtt{NoWin}$ formulas F^N, $W_{F^N} = \emptyset$.

Now, to conclude that $W_{[\![LT]\!]} = \emptyset$, we observe that $LT \in \mathtt{NoWin}$, as all occurrences of ! are negative.

Finally, if $f = \mathbf{C}_i(g^1, \ldots, g^{q_i}, (h_y \mid y \in \bigcup \mathsf{im}(Y^i)))$, then, plays of g^j and h_y properly extend to plays of f as discussed in the previous proof. Therefore, it follows that $||g^i||, ||h_y|| < ||f||$. □

Now, we combine steps 1.-4. to reduce the definability of strict strategies to that of non-strict ones.

Lemma 4.5.4 (Defining Strict Strategies from Non-Strict Ones)**.** *All morphisms in* $\mathsf{Ctxt}(\mathsf{DGame}_!)^{\mathsf{fin1\Sigma\Pi}}$ *are definable in* $\mathsf{DTT}_{\mathsf{CBN}}$ *if we assume that the non-strict ones are, where we write* $\mathsf{Ctxt}(\mathsf{DGame}_!)^{\mathsf{fin1\Sigma\Pi}}$ *for the full subcategory of* $\mathsf{Ctxt}(\mathsf{DGame}_!)$ *on the objects formed by the interpretation of types of* $\mathsf{DTT}_{\mathsf{CBN}}$ *formed without* Id*-constructors.*

Proof. Let T be a type of $\mathsf{DTT}_{\mathsf{CBN}}$ with Π, Σ, 1 and finite inductive type families and let $f \in \mathsf{Ctxt}(\mathsf{DGame}_!)([], [\![T]\!])$. If $T = \Sigma_{x_1:T^1} \ldots \Sigma_{x_{n-1}:T^{n-1}} T^n$ (including the case of $T = 1$ if $n = 0$), then, we know that both in the syntax and semantics f

decomposes as $\langle f_1, \ldots, f_n \rangle$. The interesting remaining case to deal with therefore is definability for $T = \Pi_{x:T'} S[q/x']$ where $x' : Q \vdash S \quad \mathsf{type}$ and $x : T' \vdash q : Q$, i.e. for $T = \Pi_{x:T'} S[q/x']$ where S is a finite inductive type family. (In that case $[\![S[q/x']]\!] = X_*$ has finite inductive games as fibres.)

From here, the argument to show that $f \in \mathsf{Ctxt}(\mathsf{DGame}_!)([], [\![T]\!]) = \mathsf{str}(\mathsf{O\text{-}sat}([\![T]\!]))$ is definable in $\mathsf{DTT}_{\mathsf{CBN}}$ will proceed by complete induction on $||f||$, which terminates according to lemma 4.5.3. For the sake of our inductive argument, let us consider the more general case of $f \in \mathsf{str}(\mathsf{O\text{-}sat}([\![\Pi_{x:T'} \Pi_{[\mathsf{Id}]_D(d,d^0)} S[q/x']]\!]))$ which does not visit the Id-type. Note that we may assume WLOG that $T' = \Sigma_{T^1} \ldots \Sigma_{T^{n-1}} T^n$ with $T^i = \Pi_{T'^1} \cdots \Pi_{T'^{q_i}} U[v/x'']$, where $x'' : V \vdash U \quad \mathsf{type}$ and $x_1 : T^1, \ldots, x_{i-1} : T^{i-1}, x'_1 : T^{1\prime}, \ldots, x'_{q_i} : T^{q_i\prime} \vdash v : V$ and where $[\![U[v/x'']]\!] = Y^i_*$. This is where we invoke lemma 4.5.2.

If f is strict, then f can be expressed as

$$\mathbf{C}_i(g^1, \ldots, g^{q_i}, (h_y \mid y \in \bigcup \mathsf{im}(Y^i))) =$$
$$[\![\lambda_{x:T'} \mathsf{case}^{p,q}_{U[v/x''][\mathsf{fst}(x)/x_1, \ldots, (i-1)-\mathsf{th}(x)/x_{i-1}, G^1 x/x'_1, \ldots, G^{q_i} x/x'_{q_i}], S[q/x']}(x_i(G^1 x) \cdots (G^{q_i} x), \{H_y x\}_y)]\!],$$

where by the induction hypothesis $g^i = [\![G^i]\!]$ and $h_y = [\![H_y]\!]$.

If f is non-strict, it is definable by assumption.

We conclude that f is definable in $\mathsf{DTT}_{\mathsf{CBN}}$. □

Next, we complete the definability proof by showing how to define non-strict strategies from the syntax of $\mathsf{DTT}_{\mathsf{CBN}}$ using the extensionality of types.

Theorem 4.5.5 (Full Completeness at Id-free type hierarchy)**.** *All morphisms in* $\mathsf{Ctxt}(\mathsf{DGame}_!)^{\mathsf{fin1\Sigma\Pi}}$ *are definable in* $\mathsf{DTT}_{\mathsf{CBN}}$*, where we write* $\mathsf{Ctxt}(\mathsf{DGame}_!)^{\mathsf{fin1\Sigma\Pi}}$ *for the full subcategory of* $\mathsf{Ctxt}(\mathsf{DGame}_!)$ *on the objects formed by the interpretation of types of* $\mathsf{DTT}_{\mathsf{CBN}}$ *formed without* Id*-constructors.*

Proof. To show definability, by lemma 4.5.4, all that remains to be done is demonstrate definability for non-strict f.

If f is non-strict, we know from lemma 4.5.2 that f answers with some move a s.t. for all $\vdash t : T'$ and $\vdash \overrightarrow{k} : \overrightarrow{\mathsf{Id}}_{\overrightarrow{D}[t/x']}(\overrightarrow{d^0}[t/x'], \overrightarrow{d}[t/x']), [] \xrightarrow{a} [\![S[q[t/x]/x']]\!]$. (Where,

to keep notation light, we write $\overrightarrow{D}$ for the list of types $D^1, \ldots, D^m$ and similarly for terms.) Now, as S is a finite inductive type family, we know that $a = [\![s_0]\!]$ for some $\vdash s_0 : S[q_0/x']$ where we write $q_0 := q[t/x]$ (noting that $q[t/x]$ is independent of t as the fibres of S are disjoint and the interpretations of constructors of finite inductive types is faithful).

Now, in particular, $S[q[t/x]/x'] = S[q_0/x']$. Moreover, clearly, $S[q[t/x]/x']^T = S^T = S[q_0/x']^T$. Therefore, by $\mathsf{Ty\text{-}Ext}$ (which, by theorem 4.5.1, we know to hold for $S[q/x']$ and $S[q_0/x']$, as, by induction[4], we may assume that we have already established definability for $\Sigma_{x':T'} \overrightarrow{\mathsf{Id}}_{\overrightarrow{D}}(\overrightarrow{d^0}, \overrightarrow{d})$), it follows that $\vdash \Pi_{x':T'} \Pi_{\overrightarrow{p}:\overrightarrow{\mathsf{Id}}_{\overrightarrow{D}}(\overrightarrow{d^0}, \overrightarrow{d})} S[q/x'] = \Pi_{x':T'} \Pi_{\overrightarrow{p}:\mathsf{Id}_{\overrightarrow{D}}(\overrightarrow{d^0}, \overrightarrow{d})} S[q_0/x']$. Therefore, by $\mathsf{Ty\text{-}Conv}$, we have $x' : T', \overrightarrow{p} : \mathsf{Id}_{\overrightarrow{D}}(\overrightarrow{d^0}, \overrightarrow{d}) \vdash s_0 : S[q/x']$ which is interpreted as f. □

We have obtained the following, combining theorems 4.5.1 and 4.5.5.

Corollary 4.5.6 (Full and Faithful Completeness at Id-free type hierarchy)**.** *All morphisms in* $\mathsf{Ctxt}(\mathsf{DGame}_!)^{\mathsf{fin1}\Sigma\Pi}$ *are faithfully definable in* $\mathsf{DTT}_{\mathsf{CBN}}$.

Next, we show that full (and faithful) completeness still holds if we allow one strictly positive occurrence[5] of an Id-type. This shows, in particular, that the notion of propositional identity coincides in syntax and semantics for open terms of the Id-free type hierarchy.

Theorem 4.5.7 (Full and Faithful Completeness for strictly positive Id-types)**.** *All morphisms in* $\mathsf{Ctxt}(\mathsf{DGame}_!)([], [\![\Pi_{x:A} \mathsf{Id}_B(f, g)]\!])$ *for* $x : A \vdash f, g : B$ *are faithfully definable in* $\mathsf{DTT}_{\mathsf{CBN}}$*, if* $\vdash A \quad \mathsf{type}$ *and* $x : A \vdash B \quad \mathsf{type}$ *are types built without* Id*-constructors.*

[4]Indeed, definability is only non-trivial for function types constructors. By induction, we have already established definability for T'. It then follows trivially for $\Sigma_{x':T'} \overrightarrow{\mathsf{Id}}_{\overrightarrow{D}}(\overrightarrow{d^0}, \overrightarrow{d})$, as all closed witnesses of Σ- and Id-types are canonical, both in syntax and semantics.

[5]Recall that we say that a subformula B occurs strictly positively in a type A if it does not appear as the antecedent of any function types. In particular, in the case of $\mathsf{DTT}_{\mathsf{CBN}}$, we say that B occurs strictly positively in A if it does not occur as the left hand side argument of a Π-type constructor.

Proof. Faithfulness has already been argued in theorem 4.5.1.

Given an inhabitant p of $\mathsf{Ctxt}(\mathsf{DGame}_!)([\![1]\!], [\![\Pi_{x:A}\mathsf{Id}_B(f,g)]\!])$, for any $\cdot \vdash a : A$, evaluating p at $[\![a]\!]$ gives $p\{[\![a]\!]\} \in \mathsf{Ctxt}(\mathsf{DGame}_!)([\![1]\!], [\![\mathsf{Id}_B(f[a/x], g[a/x])]\!])$, which by (I3) of theorem 4.3.4 implies that $[\![f[a/x]]\!] = [\![f]\!]\{[\![a]\!]\} = [\![g]\!]\{[\![a]\!]\} = [\![g[a/x]]\!]$. Seeing that our model is faithful at the $1\Sigma\Pi$-hierarchy over finite inductive type families, we conclude that $\cdot \vdash f[a/x] \equiv g[a/x] : B$ for all $\cdot \vdash a : A$.

Noting that $\vdash \mathsf{Id}_B(f,g)^T = \{\mathtt{refl}\} = \mathsf{Id}_B(f,f)^T$, we conclude, by Ty-Ext, that $\vdash \Pi_{x:A}\mathsf{Id}_B(f,g) = \Pi_{x:A}\mathsf{Id}_B(f,f)$. Ty-Conv now reduces full completeness for the former type to full completeness for the latter. We further note that we have an isomorphism of types $\vdash \Pi_{x:A}\mathsf{Id}(f,f) \cong \Pi_{x:A}\{\mathtt{refl}\}$. For this last type, full completeness has already been established in theorem 4.5.5. Our claim therefore follows. □

Remark 4.5.8. *We would like to point out to the reader the phenomenon that (full) completeness at types involving positively occurring* Id*-type constructors crucially relies on faithfulness of the model. This is illustrated here for the case of one strictly positively occurring* Id*-type. The question rises what the status is of full completeness results for general types of* $\mathsf{DTT}_{\mathsf{CBN}}$*, in which* Id*-types are also allowed to occur negatively and (non-strictly) positively. It seems very believable that our completeness proof could be adapted to this setting as well as the more general setting of arbitrary inductive families [23, 100] of which* Id*-types are a special case. Indeed, the idea will just be to perform the decomposition in the simply typed translation, accumulating* Id*-types as we progress in the decomposition, after which definability of non-strict strategies should follow by using constructors for inductive families together with Ty-Ext. We note, in particular, that* let p be $\mathsf{refl}(x)$ in d *is entirely analogous to a simple* case *construct and* $\mathsf{refl}(x)$ *simply behaves as a constructor* `refl` *for an inductive type.*

4.6 Dependent Games for Effects

The whole previous development was carefully set up to be robust under failure of any combination of the conditions on strategies of being winning, history-free,

well-bracketed or deterministic. All stated definitions would stay the same, where one should just interpret the word strategy and the set $\mathsf{str}(A)$ differently. All results would remain true with the exception of completeness results. In particular, we get sound faithful interpretations of $\mathsf{DTT}_{\mathsf{CBN}}$. As we shall see, the completeness properties will be more subtle and require further study.

Remark 4.6.1 (Types as Homomorphisms?)**.** *The reader may wonder if we should not require that types are continuous or at the very least monotone. Indeed, the intuition may be that types should arise from strategies into some universe. However, we believe this point of view is not the most productive. Indeed, as we shall see in chapter 5, types are best thought of as functions on values (like thunks of computations) rather than some sort of homomorphism into which we can effectfully substitute computations. It should be noted that even if we work with a type universe, it is not clear that the codes (which strategies on the universe represent) correspond to types in a monotone way. However, universes* $\mathcal{U}$ *should be thought of as value types (as we have a family* El *depending on them), hence cannot be expected to exist in the same way in a type theory with unrestricted effects. For instance, we should not expect* $\mathsf{diverge}_{\mathcal{U}}$ *to code for a type.*

As a concrete example, we note that, in Martin-Löf's partial type theory [83], essentially Martin-Löf type theory with fixpoint combinators for all type families, types are not monotone. Indeed, $\mathsf{Id}_{\mathbb{B}\Rightarrow\mathbb{B}}(\lambda_{x:\mathbb{B}}\mathsf{case}(x,\mathsf{tt},\mathsf{diverge}),\lambda_{x:\mathbb{B}}\mathsf{case}(x,\mathsf{tt},\mathsf{ff}))$ *has* $\mathsf{diverge}$ *as only inhabitant while* $\mathsf{Id}_{\mathbb{B}\Rightarrow\mathbb{B}}(\lambda_{x:\mathbb{B}}\mathsf{case}(x,\mathsf{tt},\mathsf{diverge}),\lambda_{x:\mathbb{B}}\mathsf{case}(x,\mathsf{tt},\mathsf{diverge}))$ *has two distinct inhabitants* $\mathsf{diverge}$ *and* $\mathsf{refl}(\lambda_{x:\mathbb{B}}\mathsf{case}(x,\mathsf{tt},\mathsf{diverge}))$*. At the same time,* $\lambda_{x:\mathbb{B}}\mathsf{case}(x,\mathsf{tt},\mathsf{diverge}) \leq \lambda_{x:\mathbb{B}}\mathsf{case}(x,\mathsf{tt},\mathsf{ff})$*. Type monotonicity is seen to be restored, for instance, if we impose the axiom of function extensionality on the* Id*-types, but it is not a feature of bare intensional type theory with recursion.*

In the simply typed world, the idea is precisely (in the sense that we get full abstraction results) that dropping winning conditions allows us to interpret fixpoint combinators [22], dropping history-freeness allows us to interpret local references of ground type [64], dropping determinism allows us to interpret erratic

non-deterministic choice primitives [60] and switching from well-bracketed to weakly well-bracketed strategies allows us to interpret the universal control operator $\mathsf{call/cc}$ [61]. We make some observations on the extent to which our context games with morphisms composed of the suitable strategies give a sound interpretation of various primitives for effects and we briefly discuss the status of definability results.

4.6.1 Recursion

We note that we can interpret fixpoint combinators in the world of partial (i.e. non-winning) strategies for all context games of length 1 exactly as in [22]. The only difficulty is posed by Σ-types. We can obtain an interpretation of fixpoint combinators from a general construction (see [22]) if we can show that $\mathsf{Ctxt}(\mathsf{Game}_!)$ is a rational category. This is not at all guaranteed if we impose no conditions on dependent games and define them simply as unconstrained functions on strategies. Indeed, while our homsets are always pointed partial orders, we cannot always take the colimit of chains $f^{(k)}$ of repeated function applications (starting from $\bot$).

To achieve this, it would be sufficient to demand that dependent games are continuous functions. However, this is easily seen to be inconsistent with our interpretation of Id-types. A weaker and still sufficient condition is to demand that for a dependent game B, for each infinite ascending chain $[\sigma_i^0]_i, [\sigma_i^1]_i, \ldots$ in $\mathsf{str}(☺(A_1)\& \cdots \&☺(A_n))$ and for every $s \in P_{☺(B)}$, if there exists some integer N[6] such that $s \in P_{B(\sigma_1^k,\ldots,\sigma_n^k)}$ for all $k \geq N$, then $s \in P_{B(\bigcup_k \sigma_1^k,\ldots,\bigcup_k \sigma_n^k)}$. This condition is easily seen to be preserved by all our type formers. We obtain a model of $\mathsf{DTT}_{\mathsf{CBN}}$ extended with fixpoint combinators at all types.

Further, we suspect[7] that for compact elements (noting that these have a finite norm [22]), our definability proof of theorem 4.5.5, can be adapted to this setting by making h_y explore the newly accumulated Id-type. The decomposition lemma then leaves us to define a strategy which visits all Id-types from right to

[6] Indeed, given $f = \langle f_1, \ldots, f_n\rangle : \Sigma(A_1, \ldots, A_n) \longrightarrow \Sigma(A_1, \ldots, A_n)$, define the increasing ω-chain $\sigma^N := f^N(\bot)$. We want that $\bigcup_{N\in\mathbb{N}} f_k(\sigma^N) \in \mathsf{str}(A_k\{[f_{k'}]_{k'<k}\}(\bigcup_{N\in\mathbb{N}} \sigma^N))$. We have that $f_k(\sigma^N) \in \mathsf{str}(A_k\{[f_{k'}]_{k'<k}\}(\sigma^N))$, so we precisely need the extra condition that B does not shirk in the limit $N \to \infty$.

[7] The details remain to be verified.

left and then gives a non-strict reply. This can then be defined using constructors, subst-operators (to visit all Id-types) and Ty-Ext.

4.6.2 Local Ground References

Local references of ground type (for instance, of integer type) can be added to $\mathsf{DTT}_{\mathsf{CBN}}$ in exactly the same way as they are to a simply typed language. Indeed, the new terms for handling state have types only involving (closed) inductive types X and reference types $\mathsf{Ref}(X)$. This is described very clearly in [64] where a slightly different definition of ! and $\multimap$ are used (threads are not distinguished by labelling moves with a thread number), meaning that justifiers are not uniquely determined by the type structure and have to be specified as part of the play. All results of [64] transfer to our setting, as long as we take care to include the obvious thread labelling in our interpretation of the terms for manipulating state. In particular, we get a sound interpretation of $\mathsf{DTT}_{\mathsf{CBN}}$ extended with local ground type references.

Definability[8] (and with that full abstraction) in the simply typed world, depends on the result that there is a universal history-sensitive (well-bracketed deterministic) strategy cell_X on $!\mathsf{Ref}(X)$, for any ground type X, such that any history-sensitive strategy σ on a game A in the simply typed hierarchy over ground types factors as cell_X for a suitably large X (this needs to be countably infinite, at least, to encode the whole history of the play in A) followed by a history-free strategy τ on $\mathsf{Ref}(X) \Rightarrow A$ (which keeps updating $\mathsf{Ref}(X)$ to hold the current history of the play in A and then responds as σ would). We note that we should impose a visibility condition on strategies (a condition to exclude the use of higher-order references, which follows automatically from the restriction on plays for our type hierarchy) if we want the same factorisation result for more general games A. The factorisation theorem (and with that the definability result) of [64] does not have an obvious generalisation to all types of $\mathsf{Ctxt}(\mathsf{Game}_!)$. Indeed, context games of length greater than 1 (Σ-types) can pose problems. If we apply the simply typed factorisation construction to a consistent tuple $[\sigma_i]_i$ of strategies, there is no

[8] The usual definability proof relies crucially on having partial strategies. It is not clear if it can be made to work in the world of winning strategies.

guarantee that the resulting tuple $[\tau_i]_i$ is still consistent. Indeed, we only know that its interaction with cell_X would be consistent.

4.6.3 Finite Non-Determinism

Lifting the determinacy condition for strategies gives us a model of dependent type theory with non-deterministic features. In fact, following [60], we can clearly interpret the non-deterministic choice primitive $\vdash \mathsf{or}_A : A \Rightarrow A \Rightarrow A$ for any type A which gets interpreted as a game (rather than proper context game), by using the (well-bracketed history-free winning) strategy which plays a copycat between both $A^{(3)}$ and, non-deterministically, both $A^{(1)}$ and $A^{(2)}$. It is not clear that we can interpret this primitive for context games (or types containing Σ-constructors), however. Indeed, to get the correct simply typed translation, we should define $\mathsf{or}_{[A_j]}$ by $[f_j]_j \ \mathsf{or} \ [g_j]_j := [f_j \ \mathsf{or} \ g_j]_j$. However, we can easily see that this is well-defined for all context morphisms if and only if the dependent games A_j are monotone functions of their inputs. (Indeed, $f_j \ \mathsf{or} \ g_j$ represents the union of f_j and g_j.) This is a condition that is incompatible with the current interpretation of Id-types. Hence, we are faced with a choice: interpreting Id-types or or-primitives at Σ-types.

Definability (and, with that, full abstraction with respect to "may-observational-equivalence") in the simply typed world depends on the result that there is a universal non-deterministic strategy oracle on $\mathbb{N}_* \Rightarrow \mathbb{N}_*$ (which is strict and responds to a number n, non-deterministically, with some number between 0 and n) such that any finitely non-deterministic strategy σ on A factors as oracle, followed by some deterministic strategy $\mathsf{det}(\sigma)$ (it responds with the number of different moves that σ could make, in the left most copy of $\mathbb{N}_*$, and makes the n-th move of σ in A in response to n; it is winning, well-bracketed and history-free if σ was such) on $(\mathbb{N}_* \Rightarrow \mathbb{N}_*) \Rightarrow A$. However, it is easily seen[9] that the simply typed factorisation, when performed on arbitrary context morphisms $[] \xrightarrow{[\sigma_i]_i} [A_i]_i$ (rather than just

[9] Indeed, take $\mathsf{contradict}$ be a game depending on $\mathbb{B}_*$ such that $\mathsf{contradict} : \mathbb{B}_* \mapsto \mathbb{B}_*$ and $\mathsf{contradict} : \mathsf{else} \mapsto \emptyset_*$. We have a strategy $\sigma = \langle \mathbb{B}_*, \mathsf{tt} \rangle$ on $\Sigma(\mathbb{B}_*, \mathsf{contradict})$. Then $(\lambda_x 1); \mathsf{det}(\sigma) = \langle \mathsf{tt}, \mathsf{tt} \rangle$, which is not a strategy on $\Sigma(\mathbb{B}_*, \mathsf{contradict})$.

individual strategies, or context morphisms of length 1), can result in inconsistent lists of strategies $[\tau_i]_i$, not defining a context morphism $[\mathbb{N}_* \Rightarrow \mathbb{N}_*] \longrightarrow [A_i]_i$.

4.6.4 Control Operators

The interpretation of control operators in game semantics seems to be more fragile than that of the other effects we have considered (partiality, local ground references, non-determinism). That is, [61] shows that $\mathsf{Game}_!$, for weakly well-bracketed strategies, interprets (with a history-free winning deterministic strategy violating the well-bracketing condition) the universal control operator[10] $\mathsf{call/cc}_{X_*,Y_*} : ((X_* \Rightarrow Y_*) \Rightarrow X_*) \Rightarrow X_*$ (which plays a copycat between $X_*^{(1)}$ and $X_*^{(3)}$ and between $X_*^{(2)}$ and $X_*^{(3)}$ and which opens $X_*^{(1)}$ in response to the initial question in Y_*) for flat games X_* and Y_*. Next, it shows that any strategy σ on a game A **in the simply typed hierarchy over ground types** (rather than a general game) can be factored as $\mathsf{call/cc}_{X_*,X_*}; \tau$ for (appropriate X_* and) a well-bracketed strategy τ on $(((X_* \Rightarrow Y_*) \Rightarrow X_*) \Rightarrow X_*) \Rightarrow A$.

We note that we can define

$$\begin{aligned}
\mathsf{call/cc}_{I,C}() &:= \langle\rangle \\
\mathsf{call/cc}_{A\&B,C}(f,g) &:= \langle \mathsf{call/cc}_{A,C}(\lambda_\phi f(\lambda_x \phi(\mathsf{fst}\ (x))), \mathsf{call/cc}_{B,C}(\lambda_\phi f(\lambda_x \phi(\mathsf{fst}\ (x)))) \rangle \\
\mathsf{call/cc}_{A\Rightarrow B,C}(f) &:= \mathsf{call/cc}_{B,C}(\lambda_\phi t(\lambda_y \phi(y(x))(x))).
\end{aligned}$$

That is, adding the control operator $\mathsf{call/cc}$ at ground types to an intuitionistic type theory with $1, \times, \Rightarrow$-types gives us the control operator at all types, making it into a constructive classical type theory. In particular, we can interpret these control operators in our game semantics.

Now, it is well-known that constructive classical dependent type theory is degenerate in the sense that it identifies all terms (propositionally) [101]. The situation in our model is that the obvious candidate for $\mathsf{call/cc}$ on Σ-types (based on the simply typed translation) does not define a context morphism. Indeed,

[10] This is also known as Peirce's law in logic, which is easily seen to be equivalent to the principle of double negation elimination (take Y_* to be a false formula). This law (with its computation rules) is the defining feature of constructive classical logic.

in particular, for the type $A = \Sigma_{x:\mathbb{B}}\mathsf{Id}_{\mathbb{B}}(\mathsf{tt},x)$ used by Herbelin to derive his degeneracy result, our candidate for $\mathsf{call/cc}$ (for $C = \mathsf{Id}_{\mathbb{B}}(\mathsf{tt},\mathsf{ff})$ an inconsistent proposition) does not type check. Indeed, such an appropriate term $\mathsf{call/cc}_A$ of type $((A \Rightarrow \mathsf{Id}_{\mathbb{B}}(\mathsf{tt},\mathsf{ff})) \Rightarrow A) \Rightarrow A$ would decompose as $\langle \mathsf{call/cc}^1_A, \mathsf{call/cc}^2_A\rangle$, where $\vdash \mathsf{call/cc}^1_A : ((\Sigma_{x:\mathbb{B}}\mathsf{Id}_{\mathbb{B}}(\mathsf{tt},x) \Rightarrow \mathsf{Id}_{\mathbb{B}}(\mathsf{tt},\mathsf{ff})) \Rightarrow \Sigma_{x:\mathbb{B}}\mathsf{Id}_{\mathbb{B}}(\mathsf{tt},x)) \Rightarrow \mathbb{B}$ and $\vdash \mathsf{call/cc}^2_A : \Pi_{t:(\Sigma_{x:\mathbb{B}}\mathsf{Id}_{\mathbb{B}}(\mathsf{tt},x)\Rightarrow\mathsf{Id}_{\mathbb{B}}(\mathsf{tt},\mathsf{ff}))\Rightarrow\Sigma_{x:\mathbb{B}}\mathsf{Id}_{\mathbb{B}}(\mathsf{tt},x)}\mathsf{Id}_{\mathbb{B}}(\mathsf{tt},\mathsf{call/cc}^1_A(t))$. The equivalent of the usual interpretation of $\mathsf{call/cc}$ of [61], which plays a copycat back and forth between $A^{(3)}$ and $A^{(2)}$, which plays the initial move in $A^{(1)}$ if $\mathsf{Id}_{\mathbb{B}}(\mathsf{tt},\mathsf{ff})$ is opened by Opponent and which copies back Opponent's response in $A^{(1)}$ to $A^{(3)}$, is seen not to yield a sound interpretation of $\mathsf{call/cc}_A$ in our model. Indeed, $[\![\mathsf{call/cc}^2_A]\!]$ violates the rules of the game

$$\mathsf{O\text{-}sat}([\![\Pi_{t:(\Sigma_{x:\mathbb{B}}\mathsf{Id}_{\mathbb{B}}(\mathsf{tt},x)\Rightarrow\mathsf{Id}_{\mathbb{B}}(\mathsf{tt},\mathsf{ff}))\Rightarrow\Sigma_{x:\mathbb{B}}\mathsf{Id}_{\mathbb{B}}(\mathsf{tt},x)}\mathsf{Id}_{\mathbb{B}}(\mathsf{tt},\mathsf{call/cc}^1_A(t))]\!]),$$

in particular, its play $*(0,*)(0,(0,*))(0,(0,(0,*)))(0,(0,(0,\mathtt{refl})))\mathtt{refl}$ does so with its last move. Indeed, this Player move excludes the value

$$\tau = [\![\lambda_{k:A\Rightarrow\mathsf{Id}_{\mathbb{B}}(\mathsf{tt},\mathsf{ff})}\langle \mathsf{ff}, k(\langle \mathsf{tt}, \mathsf{refl}(\mathsf{tt})\rangle)\rangle]\!]$$

for $[\![t]\!]$, while only Opponent is allowed to restrict the fibre.

Moreover, the factorisation construction of [61] does not give valid well-bracketed context morphisms when applied naively in $\mathsf{Ctxt}(\mathsf{Game}_!)$, so it is not clear how a definability result could be established for this model.

4.6.5 Lessons for Combining Dependent Types and Effects

The models of dependent type theory presented in this section can be seen as assigning dependent types to effectful programs under a CBN equational theory. We mean that in the following way. The strategies we have considered are known to correspond very closely to programs with recursion, local ground references, finite non-determinism and control operators with CBN evaluation [22, 60, 62, 64]. Rather than the usual simple types, modelled in usual game semantics, we have considered more precise types for these programs here, modelled by dependent

games. While what we arrive at clearly represents game theoretic model of a CBN dependent type theory with some effects, it should be further examined how freely these effects are allowed to occur.

One thing that the semantics suggests is the interpretation of fixpoint combinators and non-deterministic choice may be difficult at (projection) Σ-types, unless types behave as suitable homomorphisms. This, however, seems to cause a tension with the interpretation of Id-types. There is a real question, therefore, whether types should act as homomorphisms. Even stronger is the result that we saw that the interpretation of universal control operators at Σ-types can fail in this model and, generally, is known to cause degeneracy. In many cases, we see that the interpretation of effects at (projection) Σ-types can be problematic. That does not mean, however, that we should not include primitives for effects (like fixpoint combinators) at other types.

A more general, but related question that this semantics raises is whether one should aim for dependent type theory with unrestricted effects in the first place as this seems to lead to many technical challenges. We address this question further in the next chapter. In hindsight, based on lessons learnt both in this chapter and in the other chapters of this thesis, we now believe the answer should be no. In most cases, it appears both safer and more useful to us to restrict the use of effects with the type system. A prime aim for future work should, therefore, in our opinion, be the development of a game semantics for dependently typed CBPV.

The impossible often has a kind of integrity which the merely improbable lacks.

— Douglas Adams

5

Dependently Typed Call-by-Push-Value (dCBPV)

Dependent types [20] are slowly being taken up by the functional programming community and are in the transition from a quirky academic hobby into a practical approach to building certified software. Purely functional dependently typed languages like Coq [102] and Agda [103] have existed for a long time. If the technology is to become more widely used in practice, however, it is crucial that dependent types can be smoothly combined with the wide range of effects that programmers make use of in their day to day work, like non-termination and recursion, mutable state, input and output, non-determinism, probability and non-local control.

Although some languages exist which combine dependent types and effects, like Cayenne [104], ΠΣ [105], Zombie [106], Idris [107], Dependent ML [108] and F$\star$ [109], there have always been some strict limitations. For instance, the first four only combine dependent types with unrestricted recursion (although Idris has good support for emulating other effects), Dependent ML constrains types to depend only on static natural numbers and F$\star$ does not allow types to depend on effectful terms at all (including non-termination). Somewhat different is Hoare Type Theory (HTT) [110], which defines a programming language for writing effectful programs as well as a separation logic encoded in a system of dependent types

for reasoning about these programs. We note that the programming fragment is not merely an extension of the logical one, which would be the elegant solution suggested by the Curry-Howard correspondence.

The sentiment of most papers discussing the marriage of these ideas seems to be that dependent types and effects form a difficult though not impossible combination. However, as far as we are aware, treatment has so far been on a case-by-case basis and no general theoretical analysis has been given which discusses, on a conceptual level, the possibilities, difficulties and impossibilities of combining general computational effects and dependent types.

In a somewhat different vein, there has long been an interest in combining linearity and dependent types. This combination was first studied from the point of view of syntax in Cervesato and Pfenning's LLF [69]. To this, the author added a semantic perspective, as described in chapter 3, which has proved important e.g. in the development of the game semantics for dependent types of chapter 4. One aspect that this abstract semantics as well as the study of particular models highlight is – more so than in the simply typed case – the added insight and flexibility obtained by decomposing the !-comonad into an adjunction[1]. This corresponds to working with dependently typed version of Benton's LNL calculus [33] rather than Barber and Plotkin's DILL [34], as was done in [78].

Similarly, it has proved problematic to give a dependently typed version of Moggi's monadic metalanguage [32]. We hope that this chapter illustrates that also in this case a decomposed adjunction perspective, like that of CBPV [30], is more flexible than a monadic perspective. (Recall that if we decompose both linear logic and the monadic metalanguage into an adjunction, we can see the former to be a restricted case of the latter which only describes (certain) commutative effects.) In particular, it turns out that the distinction that CBPV makes between **dynamic computations** and **static values** (including thunks of computations) is crucial.

[1]Indeed, connectives seem to be most naturally formulated on either the linear or cartesian side: Σ- and Id-constructors operate on cartesian types while Π-constructors operate on linear types.

In this chapter, we show that the analysis of dDILL of chapter 3 generalises straightforwardly to general (non-commutative) effects to give a dependently typed CBPV calculus that we call dCBPV-, which allows types to depend on values (including thunks of computations) but which lacks a Kleisli extension (or sequencing) principle for dependent functions. This calculus is closely related to Harper and Licata's dependently typed polarized intuitionistic logic [111]. Its categorical semantics is obtained from that (see section 3.3) for the dependent LNL calculus, by relaxing a condition on the adjunction which would normally imply, among other things, the commutativity of the effects described (and by dropping the symmetric monoidal closed structure on $\mathcal{D}$).

It straightforwardly generalises Levy's adjunction models for CBPV [37] (from locally indexed categories to more general comprehension categories [25]) and, in a way, simplifies Moggi's strong monad models for the monadic metalanguage [32], as was already anticipated by Plotkin in the late 80s: in a dependently typed setting the monad strength follows straightforwardly from the natural demand that its adjunction is compatible with substitution and, similarly, the distributivity of coproducts follows from their compatibility with substitution. In fact, we believe the categorical semantics of simply typed CBPV is most naturally understood as a special case of a that of dCBPV-. Particular examples of models are given by models of the dependent LNL calculus and by Eilenberg-Moore adjunctions for strict indexed monads on models of pure DTT. The small-step operational semantics for CBPV of [30] transfers to dCBPV- without any difficulties with the expected subject reduction and (depending on the effects considered) strong normalization and determinacy results.

When formulating candidate CBV and CBN translations of DTT into dCBPV-, it becomes apparent that the latter is only well-defined if we work with the weak (non-dependent) elimination rules for positive connectives, while the former is ill-defined altogether. To obtain a CBV translation and the CBN translation in all its glory, we have to add a principle of Kleisli extensions (or sequencing) for dependent functions to dCBPV-. Such a principle also seems appealing from the

point of view of compositionality of the system. We call the resulting calculus dCBPV+, to which we can easily extend our categorical and operational semantics. Normalization and determinacy results for the operational semantics of the pure calculus remain the same. However, depending on the effects we consider, subject reduction may fail. We analyse on a case-by-case basis the principle of dependent Kleisli extensions in dCBPV- models of a range of effects. We see that it is not always valid, depending on the effects under consideration. These technical challenges make it questionable if the extra expressive power of dCBPV+ is worth the extra complications. Therefore, as an alternative, we discuss the possibility of extending dCBPV- with some extra connectives to a dependently typed eriched effect calculus (EEC) [112]. This increases its expressive power in a slightly different way, but we argue that, similarly to dependent Kleisli extensions, it also restores compositionality, in a sense that we make precise.

On the one hand, we hope this analysis gives a helpful theoretical framework in which we can study various combinations of dependent types and effects from an algebraic, denotational and operational point of view. It gives a robust motivation for the equations we should expect to hold in both CBV and CBN versions of effectful DTT, through their translations into dCBPV, and it guides us in modelling dependent types in effectful settings like game semantics.

On the other, noting that not all effects correspond to sound logical principles, an expressive system like CBPV or a monadic language, with fine control over where effects occur, is an excellent combination with dependent types as it allows us to use the language both for writing effectful programs and pure logical proofs about these programs. Similar to HTT in aim, but different in implementation, we hope that dCBPV can be expanded in future to an elegant language, serving both for writing effectful programs and for reasoning about them.

In section 5.1, we explain why the combination of effects and dependent types is not straightforward. Next, in sections 5.2 and 5.3, we study the syntax, categorical semantics, a range of concrete models and operational semantics for, respectively, dCBPV- and dCBPV+. After that, we discuss dependent projection products

(additive Σ-types) in section 5.4 and show that we encounter technical challenges, similar to those caused by dependent Kleisli extensions. In section 5.5, we discuss the pros and cons of dependent Kleisli extensions, to introduce the dependently typed enriched effect calculus in section 5.6 as a better behaved extension of dCBPV-. We end on a brief comparison with HTT in section 5.7.

Remark 5.0.1 (Related Publications)**.** *This chapter is largely based on [16, 17]. In these preprints we incorrectly conjectured that subject reduction of dCBPV+ could be restored through appropriate subtyping conditions. While necessary, we have since realised that such subtyping conditions are likely not to be sufficient. Independently from our work [16] on dependently typed CBPV, [113] arrived at a syntax very similar to dCBPV- and an equivalent categorical semantics, presented in terms of fibrations rather than indexed categories. Where we additionally give a study of operational semantics, dCBPV+ and CBV and CBN translations, concrete models coming from indexed monads and dLNL and more connectives, [113] gives a detailed exposition of algebraic effects in dCBPV-.*

5.1 Dependent Types and Effects?

We believe that it is clear that the combination of dependent types and effects is important, being both of a fundamental theoretical interest and of a very practical interest in verifying real world code. Why is the combination not straightforward, however?

A first obstacle that we discussed in chapter 1 is that dependent types are largely useful for verification purposes. Therefore, it is important to be able to guarantee the logical consistency of a dependently typed language. At the same time, effects tend to introduce inconsistency. This is easily addressed by encapsulating effects in (strong) monads, to be thought of as logical modalities on the type system. This suggests we should pursue a dependently typed version of Moggi's monadic metalanguage rather than a dependent type theory with unrestricted effects.

A second conundrum that we face in formulating such a (monadic) dependently typed language is related to sequencing of computations. A **pure dependent type theory** involves the crucial dependent composition operation

$$f : A \Rightarrow B, g : \Pi_{y:B} C \vdash f; g : \Pi_{x:A} C[f(x)/y],$$

which can be interpreted as saying that if we can prove that a predicate is universally true and we provide a witness, we can derive that the predicate holds for the witness. Similarly, a **monadic effectful simple type theory** gets much of its power from the sequencing operation

$$f : A \Rightarrow TB, g : B \Rightarrow TC \vdash f; g^* : A \Rightarrow TC,$$

which lets us first perform one effectful computation f and then take its result as an input when we perform g next. One would expect an effectful dependent type theory to combine both substitution operations. In particular, we would perhaps expect to be able to substitute effectful computations into dependent functions like

$$f : A \Rightarrow TB, g : \Pi_{y:B} TC \vdash f; g^* : \Pi_{x:A} TC[f(x)/y].$$

What could this mean? TC is a type depending on $y : B$, while we are trying to substitute a value $f(x) : TB$ in it. Semantically, this principle would correspond to having a Kleisli extension principle for dependent functions, which has so far – as far as we are aware – only been considered in [89] for a limited class of modalities T and where TC is a predicate on TB which can be restricted to B along $B \xrightarrow{\eta_B} TB$.

A third, closely related challenge is how we should interpret a type $B[M/x]$ into which we have substituted an effectful computation M. That is, what does type checking $N : B[M/x]$ constitute? We seem to have at least two choices. Do we first evaluate M (as a dynamic computation) and substitute the result V into the type and type check $N : B[V/x]$? Or do we consider B as expressing a property of the effectful computation M and not just its outcome, meaning that we normalise the thunk of M (as a static value) and type check $N : B[(\mathsf{thunk}\ M)_{\mathsf{nf}}/x]$? We see that the dual rôles of M as a dynamic computation and $\mathsf{thunk}\ M$ as a static value

are crucial. This suggests that a CBPV perspective which distinguishes between values and computations is more suitable for understanding effectful dependent type theory than a monadic language without such an explicit distinction.

In formulating a dependently typed version of CBPV, we need to decide whether types can depend on identifiers of computation types or on those of value types (or on both). This precisely corresponds to the two interpretations of type checking $N : B[M/x]$ described above: type checking $N : B[M/\mathsf{nil}\,]$ or $N : B[\mathsf{thunk}\ M/x]$. We believe that type dependency on computations is problematic in a similar way that type dependency on identifiers of linear type is. (See chapter 3.) Indeed, the dynamic nature of computations means that their evaluation might be non-deterministic or non-terminating or might even produce other side effects like write to disk. This means that type dependency on computations would mean throwing overboard the idea of types as providing static guarantees; it would erase the significance of the phase distinction in typed programming. One could argue that what we are left with could also be achieved by writing an untyped program with some suitably placed `Assert` statements. Non-determinism (and reading state) would mean that type checking no longer provides guarantees. Indeed, our program may have passed the type check by chance as it happened to have chosen a safe trace. Non-termination would clearly make type checking undecidable. Other side effects like writing to disk could change type checking from a harmless procedure to something to think about carefully. We hope to have convinced the reader that we should focus on types depending solely on values.

Having addressed the first and third concerns though, the second concern still remains and we shall answer it in the course of this chapter. Indeed, part of the original motivation for CBPV was that both CBV and CBN versions of effectful type theories can be encoded in it. For that purpose, it is crucial to have sequencing operations for computations. Accordingly, we shall see that we need a dependent Kleisli extension like sequencing principle for dependently typed computations to obtain CBV and CBN translations from dependent type theory with unrestricted

effects. We study both a calculus, dCBPV-, without and one, dCBPV+, with such a dependent effectful sequencing principle and discuss their virtues and vices.

5.2 dCBPV without Dependent Kleisli Extensions (dCBPV-)

In this section, we show how the results of section 2.2 have an elegant dependently typed generalization, by allowing types to depend on values. We first consider a system in which we only allow sequencing M to x in N of a dependent function N if the result type of N does not depend on the identifier x that the result of M is bound to.

5.2.1 Syntax

The syntax of CBPV generalises straightforwardly to dependent types. As anticipated already by Levy [30], we only need to take care in the rule for M to x in N. He suggested that the return type $\underline{B}$ of N should not depend on x in this rule. We shall apply this restriction as well for the moment. We call the resulting system **dependently typed call-by-push-value without dependent Kleisli extensions**, or dCBPV-. We shall later revisit this assumption and study a system dCBPV+ in which we do allow such Kleisli extensions for dependent functions.

We distinguish between the following objects: contexts $\Gamma; \Delta$, where Γ is a (cartesian) region consisting of identifier declarations of value types and Δ is a (linear) region for identifier declarations of computation type and where we write Γ as a shorthand for $\Gamma; \cdot$, value types A, computation types $\underline{B}$, values V, computations M and stacks K. The type theory talks about these objects using the judgements of figure 5.1.

To derive these judgements, we have, to start with, rules, which we shall not list, which state that all judgemental equalities are equivalence relations and that all term, type and context constructors as well as substitutions respect judgemental equality. In similar vein, we have conversion rules which state that we may swap contexts and types for judgementally equal ones in all judgements. Additionally,

Judgement	Intended meaning
$\vdash \Gamma; \Delta$ ctxt	$\Gamma; \Delta$ is a valid context
$\Gamma \vdash A$ vtype	A is a value type in context Γ
$\Gamma \vdash \underline{B}$ ctype	$\underline{B}$ is a computation type in context Γ
$\Gamma \vdash V : A$	V is a value of type A in context Γ
$\Gamma; \Delta \vdash K : \underline{B}$	K is a computation/stack of type $\underline{B}$ in context $\Gamma; \Delta$
$\vdash \Gamma; \Delta = \Gamma'; \Delta'$	$\Gamma; \Delta$ and $\Gamma'; \Delta'$ are judgementally equal contexts
$\Gamma \vdash A = A'$	A and A' are judgementally equal value types in context Γ
$\Gamma \vdash \underline{B} = \underline{B}'$	$\underline{B}$ and $\underline{B}'$ are judgementally equal computation types in context Γ
$\Gamma \vdash V = V' : A$	V and V' are judgementally equal values of type A in context Γ
$\Gamma; \Delta \vdash K = K' : \underline{B}$	K and K' are judgementally equal computations/stacks of type $\underline{B}$ in context $\Gamma; \Delta$

Figure 5.1: Judgements of dependently typed CBPV.

$$\frac{}{\cdot; \cdot \text{ ctxt}}$$

$$\frac{\vdash \Gamma; \Delta \text{ ctxt} \qquad \Gamma \vdash A \text{ vtype}}{\vdash \Gamma, x : A; \Delta \text{ ctxt}} \qquad \frac{\vdash \Gamma; \cdot \text{ ctxt} \qquad \Gamma \vdash \underline{B} \text{ ctype}}{\vdash \Gamma; \underline{B} \text{ ctxt}}$$

Figure 5.2: Rules for forming contexts, where x and nil are assumed to be fresh identifiers.

$$\frac{\Gamma, x : A, \Gamma' \vdash A' \text{ vtype} \qquad \Gamma \vdash V : A}{\Gamma, \Gamma'[V/x] \vdash A'[V/x] \text{ vtype}} \qquad \frac{\Gamma, x : A, \Gamma' \vdash \underline{B} \text{ ctype} \qquad \Gamma \vdash V : A}{\Gamma, \Gamma'[V/x] \vdash \underline{B}[V/x] \text{ ctype}}$$

$$\frac{\Gamma \vdash \underline{B} \text{ ctype}}{\Gamma \vdash U\underline{B} \text{ vtype}} \qquad \frac{\Gamma \vdash A \text{ vtype}}{\Gamma \vdash FA \text{ ctype}}$$

$$\frac{\{\Gamma \vdash A_i \text{ vtype}\}_{1 \le i \le n}}{\Gamma \vdash \Sigma_{1 \le i \le n} A_i \text{ vtype}} \qquad \frac{\{\Gamma \vdash \underline{B_i} \text{ ctype}\}_{1 \le i \le n}}{\Gamma \vdash \Pi_{1 \le i \le n} \underline{B_i} \text{ ctype}}$$

$$\frac{\Gamma, x : A \vdash A' \text{ vtype}}{\Gamma \vdash \Sigma_{x:A} A' \text{ vtype}} \qquad \frac{\Gamma, x : A \vdash \underline{B} \text{ ctype}}{\Gamma \vdash \Pi^{\multimap}_{F(x:A)} \underline{B} \text{ ctype}}$$

$$\frac{\vdash \Gamma \text{ ctxt}}{\Gamma \vdash 1 \text{ vtype}}$$

$$\frac{\Gamma \vdash V : A \qquad \Gamma \vdash V' : A}{\Gamma \vdash \mathsf{Id}_A(V, V') \text{ vtype}}$$

Figure 5.3: Rules for type formation.

we demand the obvious (admissible) substitution rules for both kinds of identifiers in all judgements as well as weakening rules for identifiers of value type (but not computation type). To form contexts, we have the rules of figure 5.2.

To form types, we have the rules of figure 5.3. For these types, we consider the values, computations and stacks formed using the rules of figure 5.4.

We generate judgemental equalities for values and computations through the rules of figure 2.9 and 5.5. Note that we are using extensional Id-types, in the sense of Id-types with an η-rule. This is only done for the aesthetics of the categorical

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A} \qquad \frac{\Gamma \vdash V : A \qquad \Gamma, x : A, \Gamma' \vdash W : A'}{\Gamma, \Gamma'[V/x] \vdash \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ W : A'[V/x]}$$

$$\frac{\Gamma \vdash V : A \qquad \Gamma, x : A, \Gamma'; \Delta \vdash K : \underline{B}}{\Gamma, \Gamma'[V/x]; \Delta[V/x] \vdash \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ K : \underline{B}[V/x]}$$

$$\frac{}{\Gamma; \mathsf{nil}\ : \underline{B} \vdash \mathsf{nil}\ : \underline{B}} \qquad \frac{\Gamma; \Delta \vdash K : \underline{B} \qquad \Gamma; \mathsf{nil}\ : \underline{B} \vdash L : \underline{C}}{\Gamma; \Delta \vdash \mathsf{let\ nil}\ \mathsf{be}\ K\ \mathsf{in}\ L : \underline{B}}$$

$$\frac{\Gamma \vdash V : A}{\Gamma; \cdot \vdash \mathsf{return}\ V : FA} \qquad \frac{\Gamma; \Delta \vdash K : FA \qquad \Gamma, x : A, \Gamma'; \cdot \vdash N : \underline{B} \qquad \vdash \Gamma, \Gamma'; \mathsf{nil}\ : \underline{B}\ \mathsf{ctxt}}{\Gamma, \Gamma'; \Delta \vdash K\ \mathsf{to}\ x\ \mathsf{in}\ N : \underline{B}}$$

$$\frac{\Gamma; \cdot \vdash M : \underline{B}}{\Gamma \vdash \mathsf{thunk}\ M : U\underline{B}} \qquad \frac{\Gamma \vdash V : U\underline{B}}{\Gamma; \cdot \vdash \mathsf{force}\ V : \underline{B}}$$

$$\frac{\Gamma \vdash V_i : A_i}{\Gamma \vdash \langle i, V_i \rangle : \Sigma_{1 \le i \le n} A_i} \qquad \frac{\Gamma \vdash V : \Sigma_{1 \le i \le n} A_i \qquad \{\Gamma, x : A_i \vdash W_i : A'[\langle i, x \rangle / z]\}_{1 \le i \le n}}{\Gamma \vdash \mathsf{pm}\ V\ \mathsf{as}\ \langle i, x \rangle\ \mathsf{in}\ W_i : A'[V/z]}$$

$$\frac{\Gamma \vdash V : \Sigma_{1 \le i \le n} A_i \qquad \{\Gamma, x : A_i; \Delta[\langle i, x \rangle / z] \vdash K_i : \underline{B}[\langle i, x \rangle / z]\}_{1 \le i \le n}}{\Gamma; \Delta[V/z] \vdash \mathsf{pm}\ V\ \mathsf{as}\ \langle i, x \rangle\ \mathsf{in}\ K_i : \underline{B}[V/z]}$$

$$\frac{}{\Gamma \vdash \langle\rangle : 1} \qquad \frac{\Gamma \vdash V : 1 \qquad \Gamma \vdash W : A'[\langle\rangle/z]}{\Gamma \vdash \mathsf{pm}\ V\ \mathsf{as}\ \langle\rangle\ \mathsf{in}\ W : A'[V/z]}$$

$$\frac{\Gamma \vdash V : 1 \qquad \Gamma; \Delta[\langle\rangle/z] \vdash K : \underline{B}[\langle\rangle/z]}{\Gamma; \Delta[V/z] \vdash \mathsf{pm}\ V\ \mathsf{as}\ \langle\rangle\ \mathsf{in}\ K : \underline{B}[V/z]}$$

$$\frac{\Gamma \vdash V_1 : A_1 \qquad \Gamma \vdash V_2 : A_2[V_1/x]}{\Gamma \vdash \langle V_1, V_2 \rangle : \Sigma_{x:A_1} A_2} \qquad \frac{\Gamma \vdash V : \Sigma_{x:A_1} A_2 \qquad \Gamma, x : A_1, y : A_2 \vdash W : A'[\langle x, y \rangle / z]}{\Gamma \vdash \mathsf{pm}\ V\ \mathsf{as}\ \langle x, y \rangle\ \mathsf{in}\ W : A'[V/z]}$$

$$\frac{\Gamma \vdash V : \Sigma_{x:A_1} A_2 \qquad \Gamma, x : A_1, y : A_2; \Delta[\langle x, y \rangle / z] \vdash K : \underline{B}[\langle x, y \rangle / z]}{\Gamma; \Delta[V/z] \vdash \mathsf{pm}\ V\ \mathsf{as}\ \langle x, y \rangle\ \mathsf{in}\ K : \underline{B}[V/z]}$$

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \mathsf{refl}(V) : \mathsf{Id}_A(V, V)} \qquad \frac{\Gamma \vdash V : \mathsf{Id}_A(V_1, V_2) \qquad \Gamma, x : A \vdash W : A'[x/x', \mathsf{refl}(x)/p]}{\Gamma \vdash \mathsf{pm}\ V\ \mathsf{as}\ (\mathsf{refl}(x))\ \mathsf{in}\ W : A'[V_1/x, V_2/x', V/p]}$$

$$\frac{\Gamma \vdash V : \mathsf{Id}_A(V_1, V_2) \qquad \Gamma, x : A; \Delta[x, x/x', \mathsf{refl}(x)/p] \vdash K : \underline{B}[x/x', \mathsf{refl}(x)/p]}{\Gamma; \Delta[V_1/x, V_2/x', V/p] \vdash \mathsf{pm}\ V\ \mathsf{as}\ (\mathsf{refl}(x))\ \mathsf{in}\ K : \underline{B}[V_1/x, V_2/x', V/p]}$$

$$\frac{\{\Gamma; \Delta \vdash K_i : \underline{B_i}\}_{1 \le i \le n}}{\Gamma; \Delta \vdash \lambda_i K_i : \Pi_{1 \le i \le n} \underline{B_i}} \qquad \frac{\Gamma; \Delta \vdash K : \Pi_{1 \le i \le n} \underline{B_i}}{\Gamma; \Delta \vdash i\text{`}K : \underline{B_i}}$$

$$\frac{\Gamma, x : A; \Delta \vdash K : \underline{B}}{\Gamma; \Delta \vdash \lambda_x K : \Pi^{\multimap}_{F(x:A)} \underline{B}} \qquad \frac{\Gamma \vdash V : A \qquad \Gamma; \Delta \vdash K : \Pi^{\multimap}_{F(x:A)} \underline{B}}{\Gamma; \Delta \vdash V\text{`}K : \underline{B}[V/x]}$$

Figure 5.4: Values and computations of dCBPV-. To aid legibility, we have left implicit one of the obvious assumptions $\Gamma, z : A'' \vdash A'\ \ \mathsf{vtype}$, $\Gamma, z : A'' \vdash \underline{B}\ \ \mathsf{ctype}$, $\Gamma, x, x' : A, p : \mathsf{Id}_A(x, x') \vdash A'\ \ \mathsf{vtype}$ and $\Gamma, x, x' : A, p : \mathsf{Id}_A(x, x') \vdash \underline{B}\ \ \mathsf{ctype}$, in each of the rules for forming pattern matching eliminators $\mathsf{pm}\ V\ \mathsf{as}\ R\ \mathsf{in}\ S$ for values V of type A''.

semantics. They may not be suitable for an implementation, however, as they can (in the presence of Π-types, c.f. section 5.6) make type checking undecidable for the usual reasons [20]. The syntax and semantics can just as easily be adapted to

$$\mathsf{pm}\ \mathsf{refl}(V)\ \mathsf{as}\ (\mathsf{refl}(x))\ \mathsf{in}\ R = R[V/x] \qquad R[V_1/x, V_2/y, V/z] \overset{\#w}{=} \mathsf{pm}\ V\ \mathsf{as}\ (\mathsf{refl}(w))\ \mathsf{in}\ R[w/x, w/y, \mathsf{refl}(w)/z]$$

Figure 5.5: Equations for terms involving reflexivity witnesses. Again, these rules should be read as equations of typed terms in context: they are assumed to hold if we can derive that both sides of the equation are terms of the same type in the same context.

CBV type	**CBPV type**	**CBV term**	**CBPV term**
$\Gamma \vdash A[M/x]$ type	$\boldsymbol{UF}\Gamma^v \vdash A^v[(\mathsf{thunk}\ M^v)^*/x]$ vtype	$x_1 : A_1, \ldots, x_m : A_m \vdash M : A$	$x_1 : A_1^v, \ldots, x_m : A_m^v[\ldots \mathsf{tr}\ x_i/z_i \ldots] ; \cdot \vdash M^v : F(A^v[\mathsf{tr}\ x_1/z_1, \ldots, \mathsf{tr}\ x_n/z_n])$
		x	$\mathsf{return}\ x$
		$\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N$	$M^v\ \mathsf{to}\ x\ \mathsf{in}\ N^v$
$\Sigma_{1\le i\le n} A_i$	$\Sigma_{1\le i\le n} A_i^v$	$\langle i, M\rangle$	$M^v\ \mathsf{to}\ x\ \mathsf{in}\ \mathsf{return}\ \langle i, x\rangle$
		$\mathsf{pm}\ M\ \mathsf{as}\ \langle i, x\rangle\ \mathsf{in}\ N_i$	$M^v\ \mathsf{to}\ z\ \mathsf{in}\ (\mathsf{pm}\ z\ \mathsf{as}\ \langle i, x\rangle\ \mathsf{in}\ N_i^v)$
$\Pi_{1\le i\le n} A_i$	$U\Pi_{1\le i\le n} F A_i^v$	$\lambda_i M_i$	$\mathsf{return\ thunk}\ (\lambda_i M_i^v)$
		$i\text{`}N$	$N^v\ \mathsf{to}\ z\ \mathsf{in}\ (i\text{`force}\ z)$
$\Pi_{x:A} A'$	$U(\Pi^{\circ}_{F(x:A^v)} F A'^v[\mathsf{tr}\ x/z])$	$\lambda_x M$	$\mathsf{return\ thunk}\ \lambda_x M^v$
		$M\text{`}N$	$M^v\ \mathsf{to}\ x\ \mathsf{in}\ (N^v\ \mathsf{to}\ z\ \mathsf{in}\ (x\text{`force}\ z))$
1	1	$\langle\rangle$	$\mathsf{return}\ \langle\rangle$
		$\mathsf{pm}\ M\ \mathsf{as}\ \langle\rangle\ \mathsf{in}\ N$	$M^v\ \mathsf{to}\ z\ \mathsf{in}\ (\mathsf{pm}\ z\ \mathsf{as}\ \langle\rangle\ \mathsf{in}\ N^v)$
$\Sigma_{x:A} A'$	$\Sigma_{x:A^v} A'^v[\mathsf{tr}\ x/z]$	$\langle M, N\rangle$	$M^v\ \mathsf{to}\ x\ \mathsf{in}\ (N^v\ \mathsf{to}\ y\ \mathsf{in}\ \mathsf{return}\ \langle x, y\rangle)$
		$\mathsf{pm}\ M\ \mathsf{as}\ \langle x, y\rangle\ \mathsf{in}\ N$	$M^v\ \mathsf{to}\ z\ \mathsf{in}\ (\mathsf{pm}\ z\ \mathsf{as}\ \langle x, y\rangle\ \mathsf{in}\ N^v)$
$\mathsf{Id}_A(M, N)$	$\mathsf{Id}_{UFA^v}(\mathsf{thunk}\ M^v, \mathsf{thunk}\ N^v)$	$\mathsf{refl}(M)$	$M^v\ \mathsf{to}\ z\ \mathsf{in\ return\ refl(tr}\ z)$
		$\mathsf{pm}\ M\ \mathsf{as}\ (\mathsf{refl}(x))\ \mathsf{in}\ N$	$M^v\ \mathsf{to}\ z\ \mathsf{in}\ (\mathsf{pm}\ z\ \mathsf{as}\ (\mathsf{refl}(y))\ \mathsf{in}\ (\mathsf{force}\ y\ \mathsf{to}\ x\ \mathsf{in}\ N^v))$

Figure 5.6: A translation of dependently typed CBV into dCBPV. We write tr as an abbreviation for $\mathsf{thunk\ return}$, $\boldsymbol{UF}\Gamma := z_1 : UFA_1, \ldots, z_n : UFA_n$ for a context $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ and V^* for $\mathsf{thunk}\ (\mathsf{force}\ z_1\ \mathsf{to}\ x_1\ \mathsf{in}\ \ldots \mathsf{force}\ z_n\ \mathsf{to}\ x_n\ \mathsf{in\ force}\ V)$.

CBN type	**CBPV type**	**CBN term**	**CBPV term**
$\Gamma \vdash \underline{B}[M/x]$ type	$\boldsymbol{U}\Gamma^n \vdash \underline{B}^n[\mathsf{thunk}\ M/x]$ ctype	$x_1 : \underline{B}_1, \ldots, x_m : \underline{B}_m \vdash M : \underline{B}$	$x_1 : U\underline{B}_1^n, \ldots, x_m : U\underline{B}_m^n ; \cdot \vdash M^n : \underline{B}^n$
		x	$\mathsf{force}\ x$
		$\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N$	$\mathsf{let}\ x\ \mathsf{be}\ (\mathsf{thunk}\ M^n)\ \mathsf{in}\ N^n$
$\Sigma_{1\le i\le n} \underline{B}_i$	$F\Sigma_{1\le i\le n} U\underline{B}_i^n$	$\langle i, M\rangle$	$\mathsf{return}\ \langle i, \mathsf{thunk}\ M^n\rangle$
		$\mathsf{pm}\ M\ \mathsf{as}\ \langle i, x\rangle\ \mathsf{in}\ N_i$	$M^n\ \mathsf{to}\ z\ \mathsf{in}\ (\mathsf{pm}\ z\ \mathsf{as}\ \langle i, x\rangle\ \mathsf{in}\ N_i^n)$
$\Pi_{1\le i\le n} \underline{B}_i$	$\Pi_{1\le i\le n} \underline{B}_i^n$	$\lambda_i M_i$	$\lambda_i M_i^n$
		$i\text{`}M$	$i\text{`}M^n$
$\Pi_{x:\underline{B}} \underline{B'}$	$\Pi^{\circ}_{F(x:U\underline{B}^n)} \underline{B'^n}$	$\lambda_x M$	$\lambda_x M^n$
		$N\text{`}M$	$(\mathsf{thunk}\ N^n)\text{`}M^n$
1	$F1$	$\langle\rangle$	$\mathsf{return}\ \langle\rangle$
		$\mathsf{pm}\ M\ \mathsf{as}\ \langle\rangle\ \mathsf{in}\ N$	$M^n\ \mathsf{to}\ z\ \mathsf{in}\ (\mathsf{pm}\ z\ \mathsf{as}\ \langle\rangle\ \mathsf{in}\ N^n)$
$\Sigma_{x:\underline{B}} \underline{B'}$	$F(\Sigma_{x:U\underline{B}^n} U\underline{B'^n})$	$\langle M, N\rangle$	$\mathsf{return}\ \langle \mathsf{thunk}\ M^n, \mathsf{thunk}\ N^n\rangle$
		$\mathsf{pm}\ M\ \mathsf{as}\ \langle x, y\rangle\ \mathsf{in}\ N$	$M^n\ \mathsf{to}\ z\ \mathsf{in}\ (\mathsf{pm}\ z\ \mathsf{as}\ \langle x, y\rangle\ \mathsf{in}\ N^n)$
$\mathsf{Id}_{\underline{B}}(M, M')$	$F(\mathsf{Id}_{U\underline{B}}(\mathsf{thunk}\ M^n, \mathsf{thunk}\ M'^n))$	$\mathsf{refl}(M)$	$\mathsf{return\ refl(thunk}\ M^n)$
		$\mathsf{pm}\ M\ \mathsf{as}\ (\mathsf{refl}(x))\ \mathsf{in}\ N$	$M^n\ \mathsf{to}\ z\ \mathsf{in}\ (\mathsf{pm}\ z\ \mathsf{as}\ (\mathsf{refl}(x))\ \mathsf{in}\ N^n)$

Figure 5.7: A translation of dependently typed CBN into dCBPV. We write $\boldsymbol{U}\Gamma := x_1 : UA_1, \ldots, x_n : UA_n$ for a context $\Gamma = x_1 : A_1, \ldots, x_n : A_n$.

intensional Id-types, which are the obvious choice for an implementation.

Figures 5.6 and 5.7 indicate the natural candidate CBV and CBN translations of DTT into dCBPV, where we interpret Σ-types as having a pattern matching eliminator, as opposed to projection eliminators[2].

[2]To give the translations of projection Σ-types, we would need dependent connectives generalising $\Pi_{1\le i\le n}$ on computation types. These are the equivalents of additive Σ-types and

However, it turns out that without dependent Kleisli extensions, the CBV translation is not well-defined as it results in untypable terms. The CBN translation is, but only if we restrict to the weak (non-dependent) elimination rules for $\Sigma_{1\leq i\leq n}$-, 1-, Σ- and Id-types, meaning that the type we are eliminating into does not depend on the type being eliminated from. For an alternative to the CBV translation, we would expect the CBV translation to factorise as a translation into a dependently typed equivalent of Moggi's' monadic metalanguage, followed by a translation from this monadic language into dCBPV. It is, in fact, the former that is ill-defined if we do not have a principle of Kleisli extensions in our monadic language (or, correspondingly, in dCBPV). What we can define is a translation from a dependently typed monadic language (without dependent Kleisli extensions) into dCBPV-. In this case, we can use the strong (dependent) elimination rules for all positive connectives.

By analogy with the simply typed scenario, it seems very likely that one would be able to state soundness and completeness results for these translations, if one used the canonical equational theories for CBV and CBN dependent type theory. As we are not aware of any such equational theories being described in literature, one could imagine **defining** the CBV and CBN equational theory on dependent type theories through their translations into CBPV.

5.2.2 Categorical Semantics

We have now reached the point in the story that was our initial motivation to study dependently typed CBPV: its very natural categorical semantics. Note that we have the following elegant generalization of our reformulated notion of categorical model for simple CBPV of section 2.2.2.

Definition 5.2.1 (dCBPV- Model)**.** *By a categorical model of dCBPV-, we shall mean the following data.*

- *an indexed category* $\mathcal{B}^{op} \xrightarrow{\mathcal{C}} \mathsf{Cat}$ *of* ***values*** *with full and faithful democratic comprehension (including an indexed terminal object* 1*);*

they are similarly problematic. We discuss them in section 5.4.

- *an indexed category* $\mathcal{B}^{op} \xrightarrow{\mathcal{D}} \mathsf{Cat}$ *of* ***computations and stacks****;*
- *strong* $0, +$*-types in* $\mathcal{C}$ *such that, additionally, the following induced maps are bijections:*

$$\mathcal{D}(C.\Sigma_{1\leq i\leq n}C_i)(\underline{D},\underline{D'}) \longrightarrow \Pi_{1\leq i\leq n}\mathcal{D}(C.C_i)(\underline{D}\{\mathbf{p}_{C,\langle i,\mathsf{id}_{C_i}\rangle}\},\underline{D'}\{\mathbf{p}_{C,\langle i,\mathsf{id}_{C_i}\rangle}\});$$

- *an indexed adjunction* $\mathcal{D} \underset{U}{\overset{F}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathcal{C}$;
- $\Pi^{\multimap}_{F(-)}$*-types in* $\mathcal{D}$ *in the sense of having right adjoint functors* $-\mathcal{D}(\mathbf{p}_{A,B}) \dashv \Pi^{\multimap}_{F(B)}$ *satisfying the right Beck-Chevalley condition for* $\mathbf{p}$*-squares;*
- *finite indexed products* $(\top, \&)$ *in* $\mathcal{D}$*;*
- *strong* Σ*-types in* $\mathcal{C}$*;*
- *strong extensional* Id*-types in* $\mathcal{C}$[3]*.*

Again, this semantics is sound and complete.

Theorem 5.2.2 (dCBPV- Semantics)**.** *We have a sound interpretation of dCBPV- in a dCBPV- model:*

$$\begin{array}{ll}
[\![\cdot]\!] = \cdot & [\![\Gamma;\cdot]\!] = F1 \\
[\![\Gamma, x:A]\!] = [\![\Gamma]\!].[\![A]\!] & [\![\Gamma;\mathsf{nil} : \underline{B}]\!] = [\![\underline{B}]\!] \\
[\![\Gamma \vdash A]\!] = \mathcal{C}([\![\Gamma]\!])(1,[\![A]\!]) & [\![\Gamma;\Delta \vdash \underline{C}]\!] = \mathcal{D}([\![\Gamma]\!])([\![\Delta]\!],[\![\underline{C}]\!]) \\
[\![A[V/x]]\!] = [\![A]\!]\{\mathbf{q}_{\langle \mathsf{id}_{[\![\Gamma]\!]},[\![V]\!]\rangle,[\![\Gamma']\!]}\} & [\![\underline{B}[V/x]]\!] = [\![\underline{B}]\!]\{\mathbf{q}_{\langle \mathsf{id}_{[\![\Gamma]\!]},[\![V]\!]\rangle,[\![\Gamma']\!]}\} \\
[\![U\underline{B}]\!] = U[\![\underline{B}]\!] & [\![FA]\!] = F[\![A]\!] \\
[\![\Sigma_{1\leq i\leq n}A_i]\!] = (\cdot([\![A_1]\!] + [\![A_2]\!]) + \cdots) + [\![A_n]\!]) & [\![\Pi_{1\leq i\leq n}\underline{B}_i]\!] = (\cdot([\![\underline{B}_1]\!]\&[\![\underline{B}_2]\!])\&\cdots)\&[\![\underline{B}_n]\!]) \\
[\![\Sigma_{x:A}A']\!] = \Sigma_{[\![A]\!]}[\![A']\!] & [\![\Pi^{\multimap}_{F(x:A)}\underline{B}]\!] = \Pi^{\multimap}_{F([\![A]\!])}[\![\underline{B}]\!] \\
[\![1]\!] = 1 & \\
[\![\mathsf{Id}_A(V,V')]\!] = \mathsf{Id}_{[\![A]\!]}\{\langle \mathsf{id}_{[\![\Gamma]\!]},[\![V]\!]\rangle,[\![V']\!]\rangle\}, &
\end{array}$$

together with the obvious interpretation of terms. The interpretation in such categories is complete in the sense that an equality of terms of types holds in all interpretations iff it is provable in the syntax of dCBPV-. In fact, the interpretation

[3]In case we work with intensional Id-types, we should add the additional condition, which corresponds to pattern matching for stacks, that says that the canonical map $\mathcal{D}(A.A'.A'.\mathsf{Id}_{A'})(\underline{B},\underline{B'}) \longrightarrow \mathcal{D}(A.A')(\underline{B}\{\langle \mathsf{diag}_{A,A'}, \mathsf{refl}(A')\rangle\},\underline{B'}\{\langle \mathsf{diag}_{A,A'}, \mathsf{refl}(A')\rangle\})$ is a retraction. This map is automatically an isomorphism in our case of extensional Id-types.

defines a 1-1 correspondence between categorical models and syntactic theories in dCBPV- which satisfy mutual soundness and completeness results.

Proof (sketch). The proof goes almost entirely along the lines of the soundness and completeness proofs for linear dependent type theory in chapter 3. Nothing surprising happens in the soundness proof. For the completeness result, we build a syntactic category. □

Performing the CBN translation in the semantics, this leads to an induced notion of model for CBN dependent type theory.

Theorem 5.2.3 (Dependent CBN Semantics 1)**.** *The (semantic equivalent of the) CBN translation of DTT with* $\Sigma_{1\leq i\leq n}$*-,* 1*-,* Σ*-,* Id*-,* $\Pi_{1\leq i\leq n}$*-,* Π*-types, where we use the weak (non-dependent) elimination rules for all positive connectives, into dCBPV-, lets us construct a categorical model of CBN dependent type theory with the connectives above out of any model of dCBPV- by taking the co-Kleisli (indexed) category for* $! := FU$*. The interpretation of CBN dependent type theory is sound and complete for the equational theory induced from dCBPV-:*

$$[\![\underline{B}_1, \cdots, \underline{B}_n \vdash \underline{B}]\!] = \mathcal{D}(U[\![\underline{B}_1]\!].\cdots.U[\![\underline{B}_n]\!])(F1, [\![\underline{B}]\!]) \cong \mathcal{D}_!(U[\![\underline{B}_1]\!].\cdots.U[\![\underline{B}_n]\!])(\top, [\![\underline{B}]\!]).$$

We note that this co-Kleisli category, our notion of a model of CBN dependent type theory, is very close to the usual notion of a model of pure DTT. (We have seen this in chapter 4, in the context of CBN game semantics!) We note that even if we start with extensional Id-types in dCBPV-, we may obtain intensional Id-types in dependent CBN.

Theorem 5.2.4 (Dependent CBN Categories)**.** *The co-Kleisli category* $\mathcal{D}_!$ *is an indexed category with full and faithful (possibly undemocratic) comprehension with fibred finite products* $\Pi_{1\leq i\leq n}$ *as well as* $\Pi^{\circ}_{F(-)}$*-types. It supports weak* $\Sigma_{1\leq i\leq n}$*-,* Σ*- and* Id*-types (non-dependent elimination rules, failure of the general* η*-rules).*

Proof. ! being an indexed comonad, it follows that $\mathcal{D}_!$ is an indexed category. $\mathcal{D}_!$ satisfies the comprehension axiom in the sense that we have homset isomorphism

$$\begin{aligned}\mathcal{D}_!(\Gamma')(\top, B\{f\}) &= \mathcal{D}(\Gamma')(FU\top, B\{f\}) \\ &\cong \mathcal{D}(\Gamma')(F1, B\{f\}) \\ &\cong \mathcal{C}(\Gamma')(1, U(B\{f\})) \\ &= \mathcal{C}(\Gamma')(1, U(B)\{f\}) \\ &\cong \mathcal{B}/\Gamma(f, \mathbf{p}_{\Gamma,UB}).\end{aligned}$$

As the comprehension functor $\mathcal{D}_!(\Gamma)(B, B') \cong \mathcal{C}(\Gamma)(UB, UB') \longrightarrow \mathcal{B}/\Gamma(\mathbf{p}_{\Gamma,UB}, \mathbf{p}_{\Gamma,UB'})$ is a special case of the comprehension functor for $\mathcal{C}$, we know it to be full and faithful. Note that the comprehension may be undemocratic as $\mathcal{D}_!(\cdot)$ is equivalent to the full subcategory of $\mathcal{C}$ on the objects in the image of U, which may give a proper subcategory of $\mathcal{C}(\cdot) \cong \mathcal{B}$.

We know from the simply typed case that fibre-wise products in $\mathcal{D}$ give rise to products in $\mathcal{D}_!$. These are stable under change of base, by assumption.

Note that Π-types directly follow as a special case of $\Pi^{-\circ}_{F(-)}$-types in $\mathcal{D}$:

$$\begin{aligned}\mathcal{D}_!(\Gamma.UA)(B\{\mathbf{p}_{\Gamma,UA}\}, C) &= \mathcal{D}(\Gamma.UA)(FU(B\{\mathbf{p}_{\Gamma,UA}\}), C) \\ &= \mathcal{D}(\Gamma.UA)((FUB)\{\mathbf{p}_{\Gamma,UA}\}, C) \\ &\cong \mathcal{D}(\Gamma)(FUB, \Pi^{-\circ}_{F(UA)}C) \\ &= \mathcal{D}_!(\Gamma)(B, \Pi^{-\circ}_{F(UA)}C).\end{aligned}$$

For Σ-types, we note that we have maps back and forth, given by the unit and counit of the adjunction between F and U which satisfy a β-law given by one of

the triangle identities for the adjunction:

$$\begin{aligned}
\mathcal{D}_!(\Gamma.UA)(B, C\{\mathbf{p}_{\Gamma,UA}\}) &= \mathcal{D}(\Gamma.UA)(FU(B), C\{\mathbf{p}_{\Gamma,UA}\}) \\
&\cong \mathcal{C}(\Gamma.UA)(UB, U(C\{\mathbf{p}_{\Gamma,UA}\})) \\
&= \mathcal{C}(\Gamma.UA)(UB, (UC)\{\mathbf{p}_{\Gamma,UA}\}) \\
&\cong \mathcal{C}(\Gamma)(\Sigma_{UA}UB, UC) \\
&\cong \mathcal{D}(\Gamma)(F\Sigma_{UA}UB, C) \\
&\leftrightarrows \mathcal{D}(\Gamma)(FUF\Sigma_{UA}UB, C) \\
&= \mathcal{D}_!(\Gamma)(F\Sigma_{UA}UB, C).
\end{aligned}$$

The same argument gives us the corresponding statement for $\Sigma_{1\leq i\leq n}$- and Id-types, using their definition as left adjoint functors. □

We postpone the categorical discussion of models for dependently typed CBV until we add dependent Kleisli extensions to dCBPV- in section 5.3. For now, we would just like to point out that $\mathcal{C}$ equipped with the indexed monad $T := UF$ defines what should be regarded as a model of a dependently typed equivalent of Moggi's monadic metalanguage, without dependent Kleisli extensions.

Theorem 5.2.5 (Dependent monadic metalanguage models)**.** *Given a model $\mathcal{C} \leftrightarrows \mathcal{D}$ of dCBPV-, $T := UF$ defines an indexed monad on $\mathcal{C}$, which has a generalized notion of strength $\Sigma_A TB \xrightarrow{s_{A,B}} T\Sigma_A B$.*

Proof. As $F \dashv U$ is an indexed adjunction, T is an indexed monad. We note that, starting from $\mathsf{id}_{\Sigma_A B}$, we can obtain a generalised notion of strength for T:

$$\begin{aligned}
\mathcal{C}(\Gamma)(\Sigma_A B, \Sigma_A B) &\cong \mathcal{C}(\Gamma.A)(B, (\Sigma_A B)\{\mathbf{p}_{\Gamma,A}\}) \\
&\xrightarrow{T} \mathcal{C}(\Gamma.A)(TB, T(\Sigma_A B)\{\mathbf{p}_{\Gamma,A}\}) \\
&= \mathcal{C}(\Gamma.A)(TB, (T\Sigma_A B)\{\mathbf{p}_{\Gamma,A}\}) \\
&\cong \mathcal{C}(\Gamma)(\Sigma_A TB, T\Sigma_A B).
\end{aligned}$$

In particular (for the case where $\Gamma = \cdot$, using full and faithful comprehension), we get $\Gamma.TA \longrightarrow T(\Gamma.A) \in \mathcal{B}$. □

Remark 5.2.6. *Note that we cannot, in general, define a costrength* $\Sigma_{TA}B \longrightarrow T\Sigma_A B\{\mathbf{p}_{\Gamma,\eta_A}\}$ *or, therefore, a pairing* $\Sigma_{TA}TB \longrightarrow T\Sigma_A B\{\mathbf{p}_{\Gamma,\eta_A}\}$. *This asymmetry does not occur in the simply typed setting. It can be mended by the addition of Kleisli extensions for dependent functions.*

In the simply typed setting, one can factor the CBV translation from the λ-calculus into CBPV through the monadic metalanguage. While the translation from the dependently typed monadic metalanguage with dependent Kleisli extensions in dCBPV- works fine, we cannot define the obvious CBV translation from dependent type theory into the dependently typed monadic metalanguage, unless we have dependent Kleisli extensions.

5.2.3 Some Basic Models

We can first note that any model of pure dependent type theory is, by using the identity adjunction, in particular, a model of dependently typed CBPV, which shows consistency of the calculus.

Theorem 5.2.7. *dCBPV- is consistent both in the sense that not all terms are identified and in the sense that not all types are inhabited.*

More interestingly, any model of the dependent LNL calculus supporting the appropriate connectives (see chapter 3) gives rise to a model of dependently typed CBPV without dependent Kleisli extensions, modelling commutative effects.

Theorem 5.2.8. *The notion of model given by section 3.3 for the dLNL calculus of [78] with the additional connective of finite disjunctions for cartesian types (indexed finite distributive coproducts in* $\mathcal{C}$*) is precisely a dCBPV- model such that we have symmetric monoidal closed structures* $(I, \otimes, \multimap)$ *on the fibres of* $\mathcal{D}$*, stable under change of base, (*$\mathcal{D}$ *is an indexed symmetric monoidal closed category) s.t.* F *consists of strong symmetric monoidal functors (sending nullary and binary products in* $\mathcal{C}$ *to* I *and* $\otimes$ *in* $\mathcal{D}$*) and which supports* $\Sigma^{\otimes}_{F(-)}$*-types (see section 5.6).*

As in the simply typed setting, models of pure DTT on which we have an indexed monad are again a source of examples of dCBPV- models. This shows that dCBPV- is compatible with a wide range of effects.

Theorem 5.2.9. *Let $\mathcal{B}^{op} \xrightarrow{\mathcal{C}} \mathsf{Cat}$ be a model of pure DTT (with all type formers discussed) on which we have an indexed monad T. Then, the indexed Eilenberg-Moore adjunction $\mathcal{C} \leftrightarrows \mathcal{C}^T$ gives a model of dCBPV-.*

Proof. A product of algebras is just the product of their carriers equipped with the obvious algebra structure. Indeed, it is a basic result in category theory that the forgetful functor from the Eilenberg-Moore category creates limits. Given an object $TB \xrightarrow{k} B$ of $\mathcal{C}^T(\Gamma.A)$, we note that we also obtain a canonical T-algebra structure on Π-types of carriers (starting from the identity on $\Pi_A B$):

$$\begin{aligned}
\mathcal{C}(\Gamma)(\Pi_A B, \Pi_A B) &\cong \mathcal{C}(\Gamma.A)((\Pi_A B)\{\mathbf{p}_{\Gamma,A}\}, B) \\
&\xrightarrow{T} \mathcal{C}(\Gamma.A)(T((\Pi_A B)\{\mathbf{p}_{\Gamma,A}\}), TB) \\
&\cong \mathcal{C}(\Gamma.A)((T\Pi_A B)\{\mathbf{p}_{\Gamma,A}\}, TB) \\
&\xrightarrow{-;k} \mathcal{C}(\Gamma.A)((T\Pi_A B)\{\mathbf{p}_{\Gamma,A}\}, B) \\
&\cong \mathcal{C}(\Gamma)(T\Pi_A B, \Pi_A B).
\end{aligned}$$

We leave the verification of the T-algebra axioms to the reader. We define the result to be $\Pi^{\multimap}_{F(A)} k$. Note that it is precisely defined so that, for an algebra $TC \xrightarrow{l} C$, the isomorphism $\mathcal{C}(\Gamma.A)(C\{\mathbf{p}_{\Gamma,A}\}, B) \cong \mathcal{C}(\Gamma)(C, \Pi_A B)$ restricts to $\mathcal{C}^T(\Gamma.A)(l\{\mathbf{p}_{\Gamma,A}\}, k) \cong \mathcal{C}^T(\Gamma)(l, \Pi^{\multimap}_{F(A)} k)$. □

A concrete example to which we can apply the previous theorem is obtained for any monad T on Set. Indeed, we can lift T (point-wise) to an indexed monad on the usual families of sets model $\mathsf{Fam}(\mathsf{Set})$ of pure DTT[4]. In a different vein, given a model $\mathcal{C}$ of pure DTT, the usual exception $((-) + E)$, global state $(S \Rightarrow (- \times S))$, reader $(S \Rightarrow (-))$, writer $((-) \times M)$ and continuation monads $(((-) \Rightarrow R) \Rightarrow R)$, which we form using objects of $\mathcal{C}(\cdot)$, and, if we are dealing with a higher-order

[4]Recall that $\mathsf{Fam}(\mathsf{Set})$ is defined as the restriction to $\mathsf{Set} \subseteq \mathsf{Cat}$ of the (Cat-enriched) hom-functor into Set: $\mathsf{Set}^{op} \subseteq \mathsf{Cat}^{op} \xrightarrow{\mathsf{Cat}(-,\mathsf{Set})} \mathsf{Cat}$.

Transitions

$\mathsf{pm}\ \mathsf{refl}(V_{!\mathsf{nf}})\ \mathsf{as}\ (\mathsf{refl}(x))\ \mathsf{in}\ M \quad , \quad K \quad \rightsquigarrow \quad \mathsf{pm}\ \mathsf{refl}(V_{\mathsf{nf}})\ \mathsf{as}\ (\mathsf{refl}(x))\ \mathsf{in}\ M \quad , \quad K$

$\mathsf{pm}\ \mathsf{refl}(V_{\mathsf{nf}})\ \mathsf{as}\ (\mathsf{refl}(x))\ \mathsf{in}\ M \quad , \quad K \quad \rightsquigarrow \quad M[V_{\mathsf{nf}}/x] \quad , \quad K$

Terminal Configuration

$\mathsf{pm}\ V^{x'}_{\mathsf{nf}}\ \mathsf{as}\ (\mathsf{refl}(x))\ \mathsf{in}\ M \quad , \quad K$

Figure 5.8: The additional transition and terminal configuration that specify the operational behaviour of identity witnesses.

logic, power set monad $\mathcal{P}(-)$ give rise to indexed monads, hence we obtain models of dCBPV-. More exotic examples are the many indexed monads that arise from homotopy type theory, like truncation modalities or cohesion (shape and sharp) modalities [89, 114, 115]. A caveat there is that the identity types in the model are intensional and that many equations are often only assumed up to propositional rather than judgemental equality.

5.2.4 Operational Semantics and Effects

We define an operational semantics for dCBPV-. It is a basic result in dependent type theory that the (parallel nested) β-reductions for values are strongly normalizing [68] (according to a variation on Tait's logical relations argument). Let us write again, V_{nf} for the normal form of a value V and $V_{!\mathsf{nf}}$ to make explicit that V is not in normal form. We again define a configuration to be a pair M, K of a dCBPV-computation $\Gamma; \cdot \vdash M : \underline{B}$ and a stack $\Gamma; \mathsf{nil} : \underline{B} \vdash K : \underline{C}$. The CK-machine that evaluates our computations is again just that of figure 2.12 where we add the extra transitions and terminal configuration of figure 5.8. As before, we can add the effects of figure 2.13 together with their operational semantics of figures 2.14 and 2.15 and equations of figure 2.17. We get the same determinism, strong normalization and subject reduction results as in the simply typed case.

Theorem 5.2.10 (Determinism, Strong Normalization and Subject Reduction)**.** *Every transition respects the type of the configuration. No transition occurs precisely if we are in a terminal configuration. In absence of erratic choice, at most one transition applies to each configuration. In absence of divergence and recursion, every configuration reduces to a terminal configuration in a finite number of steps.*

Proof. The important observation will be that types only depend on values. Therefore, the only real difference in this proof from the simply typed case are the rules involving the reduction of values.

We recall from [68] that value types are closed under the untyped β-reductions for values. (This result applies as values form a conventional cartesian dependent type theory.) This implies that $\Gamma \vdash V_{\mathsf{!nf}} = V_{\mathsf{nf}} : A$ for any $\Gamma \vdash V_{\mathsf{!nf}} : A$. Therefore, it follows that $\Gamma \vdash \underline{B}[V_{\mathsf{!nf}}/x] = \underline{B}[V_{\mathsf{nf}}/x]$ for any $\Gamma, x : A \vdash \underline{B}$ ctype. It follows that the rules involving value normalization also satisfy subject reduction.

As all transitions are defined on untyped terms, determinism and strong normalization results are no different from the simply typed case. □

Remark 5.2.11 (Type Checking)**.** *While the operational semantics discussed here is very relevant as it describes the execution of a program of dCBPV-, one could argue that a type checker is as important an operational aspect to the implementation of a dependent type theory. We leave the description of a type checking algorithm to future work. We note that the core step in the implementation of a type checker is a normalization algorithm for directed versions (from left to right) of the equations for values of figures 2.9 and 5.5 (with congruence laws) and perhaps some equations for values induced from computation equations of figure 2.17 and from the specific equational theories for the effects under consideration, as this would give us a normalization procedure for types. One might be able to construct such an algorithm using normalization by evaluation by combining the techniques of [116] and [117]. Our hope is that this will lead to a proof of decidable type checking of the system at least in absence of the η-law for* Id*-types. We note that the complexity of a type checking algorithm can vary widely depending on which equations we include for specific effects. The idea is that one only includes a basic set of program equations as judgemental equalities to be able to decide type checking and one postulates other equations as propositional equalities, which can be used for manual or tactic-assisted reasoning about effectful programs.*

$$\frac{\Gamma, z : UFA, \Gamma' \vdash \underline{B}\ \mathsf{ctype} \qquad \Gamma; \cdot \vdash M : FA \qquad \Gamma, x : A, \Gamma'[\mathsf{tr}\ x/z]; \cdot \vdash N : \underline{B}[\mathsf{tr}\ x/z]}{\Gamma, \Gamma'[\mathsf{thunk}\ M/z]; \cdot \vdash M\ \mathsf{to}\ x\ \mathsf{in}\ N : \underline{B}[\mathsf{thunk}\ M/z].}$$

Figure 5.9: The rule for dependent Kleisli extensions in dCBPV. As before, we write tr as an abbreviation for $\mathsf{thunk}\ \mathsf{return}$.

5.3 dCBPV with Dependent Kleisli Extensions (dCBPV+)

While the system dCBPV- is very clean in its syntax, operational semantics, categorical semantics and admits plenty of concrete models, it may be a bit of a disappointment to the reader who was expecting to see a proper combination of effects and dependent types, rather than a system that keeps both features side by side without them interacting meaningfully[5]. In particular, one might find it unsatisfactory that the CBV translation from dependent type theory into dCBPV- fails and that the CBN translation only goes through to a limited extent.

To address these issues, we introduce a more expressive system in this section which we call dCBPV+ and which extends dCBPV- with Kleisli extensions for dependent functions. We shall later discuss, in section 5.5, whether such dependent Kleisli extensions are desirable.

5.3.1 Syntax

We have seen the need to add dependent Kleisli extensions in the form of the rule shown in figure 5.9 if we want to obtain a dependently typed equivalent of the CBV translation into CBPV or if we want to model dependent elimination rules for the positive connectives in the CBN translation. We use the name dCBPV+ to explicitly refer to the resulting system of the rules of dCBPV- (figures 5.2, 5.3, 5.4, 2.9 and 5.5) and dependent Kleisli extensions (figure 5.9).

We note that, in the presence of this extra rule, the translations of figures 5.6 and 5.7 are finally well-defined. We would like to highlight the fact that a type $x_1 : A_1, \ldots, x_n : A_n \vdash A\ \mathsf{type}$ gets translated into a type $z_1 : UFA_1, \ldots, z_n :$

[5] As we shall discuss later, this is not entirely fair on dCBPV-, as it does allow us to form types (predicates) depending on thunks of effectful computations.

$UFA_n \vdash A^v$ vtype by the CBV translation. Briefly, this is necessitated by the CBV translation of substitution of terms in types. For example, to substitute a term $x : B \vdash M : A$ into $x : A \vdash C$ type in the CBV translation, we have to be able to substitute $(x : B)^v; \cdot \vdash M^v : FA$ (or equivalently $(x : B)^v \vdash \mathsf{thunk}\ M^v : UFA$) into $(x : A)^v \vdash C^v$ vtype. This forces us to define the CBV translation $(x : A)^v$ of an identifier declaration in the context of a type well-formedness judgement as $z : UFA$ if we are to use the usual type substitution of CBPV (after taking the Kleisli extension of $\mathsf{thunk}\ M^v$).

We would like to say that the CBV and CBN translations are sound and complete. However, as no notion of a CBV or CBN equational theory has been formulated for dependent type theory, as far as we are aware, we take the equational theories induced by these translations as their definitions. Unsurprisingly, Σ-types behave equationally exactly like $\times$-types and $\Pi^{\circ}_{F(-)}$-types do as ${}_F{-\!\circ}$-types. The interesting connective to study is the Id-type.

Theorem 5.3.1. *Figures 5.6 and 5.7 define CBV and CBN translations of dependent type theory with $\Sigma_{1\leq i\leq n}$-, 1-, Σ-, Id-, $\Pi_{1\leq i\leq n}$- and Π-types (with dependent elimination rules for all positive connectives) into dCBPV+. In fact, they allow us to transfer an arbitrary theory in CBV or CBN dependent type theory to one on dCBPV+ such that we again get well-defined CBV and CBN translations. As expected, CBN Id-types (even extensional ones) satisfy the β-law but may not satisfy the η-law. More surprising, perhaps, is that the same is true for CBV Id-types.*

Proof. It is easily seen that dependent Kleisli extensions make the translations of figures 5.6 and 5.7 well-defined.

In the previous section, we have already seen the statement about CBN Id-types in case we use a non-dependent elimination rule. The case with a dependent elimination rules works similarly.

The interesting case here are the Id-types in the CBV translation. For the β-rule, note that

$(\mathsf{pm}\ \mathsf{refl}(M)\ \mathsf{as}\ (\mathsf{refl}(x))\ \mathsf{in}\ N)^v =$

$((M^v)\ \mathsf{to}\ z\ \mathsf{in}\ \mathsf{return}\ \mathsf{refl}(\mathsf{tr}\ z))\ \mathsf{to}\ \zeta\ \mathsf{in}\ \mathsf{pm}\ \zeta\ \mathsf{as}\ (\mathsf{refl}(y))\ \mathsf{in}\ (\mathsf{force}\ y)\ \mathsf{to}\ x\ \mathsf{in}\ N^v =$

$M^v\ \mathsf{to}\ z\ \mathsf{in}\ \mathsf{pm}\ \mathsf{refl}(\mathsf{tr}\ z)\ \mathsf{as}\ (\mathsf{refl}(y))\ \mathsf{in}\ (\mathsf{force}\ y)\ \mathsf{to}\ x\ \mathsf{in}\ N^v =$

$M^v\ \mathsf{to}\ z\ \mathsf{in}\ (\mathsf{force}\ \mathsf{thunk}\ \mathsf{return}\ z)\ \mathsf{to}\ x\ \mathsf{in}\ N^v =$

$M^v\ \mathsf{to}\ z\ \mathsf{in}\ (\mathsf{return}\ z)\ \mathsf{to}\ x\ \mathsf{in}\ N^v =$

$M^v\ \mathsf{to}\ x\ \mathsf{in}\ N^v =$

$(\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N)^v.$

To see that the η-rule may fail, consider dCBPV+ with divergence. We note that in case the η-law held for Id-types in CBV type theory, it would imply the following principle of reflection [28]:

$$\frac{x_1 : A_1, \ldots, x_n : A_n \vdash P : \mathsf{Id}_A(M, N)}{x_1 : A_1, \ldots, x_n : A_n \vdash M = N : A,}$$

which, in dCBPV+ translates to the rule that

$$\frac{x_1 : A_1^v, \ldots, x_n : A_n^v[\ldots \mathsf{tr}\ x_i/z_i \ldots]; \cdot \vdash P : F\mathsf{Id}_{A^v}(\mathsf{thunk}\ M^{v*}, \mathsf{thunk}\ N^{v*})}{x_1 : A_1^v, \ldots, x_n : A_n^v[\ldots \mathsf{tr}\ x_i/z_i \ldots]; \cdot \vdash M^v = N^v : FA^v,}$$

In particular, presence of divergence would make CBV type theory identify all terms in that case. In particular, this would mean that the terms $x : A^v, y : A^v; \cdot \vdash \mathsf{return}\ x : FA^v$ and $x : A^v, y : A^v; \cdot \vdash \mathsf{return}\ y : FA^v$ are judgementally equal in dCBPV+ with divergence, which they clearly are not. For a formal proof that they are not, we note that we can see this in the families of domains model given in section 5.3.3. For a more syntactic intuition, we note that Id-η is less harmful in CBPV with effects than it is in CBV or CBN with effects due to the strict distinction between values and computations, as the obvious reflection rule it implies is the following which does not identify all terms in the presence of divergence, as it does not trivially let us satisfy the hypothesis of the rule, in the way it did in CBV type theory, given that divergence is a computation and not a value.

$$\frac{\Gamma \vdash V : \mathsf{Id}_A(V_1, V_2)}{\Gamma \vdash V_1 = V_2 : A.}$$

□

5.3.2 Categorical Semantics

To formulate a categorical semantics of dCBPV+, we need a dependently typed generalization of the notion of Kleisli triple. A similar notion of dependently typed Kleisli extension has been proposed before in [89] (section 7.7), be it for a more limited class of modalities. In practice, we shall see that, for a given indexed adjunction, dependent Kleisli extensions may not exist.

Definition 5.3.2 (dCBPV+ Model)**.** *By a dCBPV+ model, we shall mean a dCBPV- model $F \dashv U : \mathcal{C} \leftrightarrows \mathcal{D}$ equipped with **dependent Kleisli extensions**. That is, maps*

$$\mathcal{C}(\Gamma.A.\Gamma'\{\mathbf{p}_{\Gamma,\eta_A}\})(1, UB\{\mathbf{q}_{\mathbf{p}_{\Gamma,\eta_A},\Gamma'}\}) \xrightarrow{(-)^*} \mathcal{C}(\Gamma.UFA.\Gamma')(1, UB),$$

where η is the unit of the adjunction $F \dashv U$, such that the following laws hold for members of the same homset:

- *unitality: $b^*\{\mathbf{q}_{\mathbf{p}_{\Gamma,\eta_A},\Gamma'}\} = b$;*
- *composition: $b^*\{\mathbf{q}_{\langle \mathsf{id}_\Gamma, a^*\rangle,\Gamma'}\} = (b^*\{\mathbf{q}_{\langle \mathsf{id}_\Gamma, a\rangle,\Gamma'}\})^*$;*
- *agreement with the usual non-dependent Kleisli extension $(-)^\star$ for the adjunction $F \dashv U$:*

$$\begin{array}{ccc} \mathcal{C}(\Gamma)(A, UB) & \xrightarrow{\cong} & \mathcal{C}(\Gamma.A)(1, UB\{\mathbf{p}_{\Gamma,A}\}) = \mathcal{C}(\Gamma.A)(1, UB\{\mathbf{p}_{\Gamma,UFA}\}\{\mathbf{p}_{\Gamma,\eta_A}\}) \\ (-)^\star \downarrow & & \downarrow (-)^* \\ \mathcal{C}(\Gamma)(UFA, UB) & \xrightarrow{\cong} & \mathcal{C}(\Gamma.UFA)(1, UB\{\mathbf{p}_{\Gamma,UFA}\}). \end{array}$$

Remark 5.3.3. *Note that it is enough to just specify the dependent Kleisli extensions of the form*

$$\mathcal{C}(\Gamma.A.\Gamma'\{\mathbf{p}_{\Gamma,\eta_A}\})(1, UFA'\{\mathbf{q}_{\mathbf{p}_{\Gamma,\eta_A},\Gamma'}\}) \xrightarrow{(-)^*} \mathcal{C}(\Gamma.UFA.\Gamma')(1, UFA').$$

Then, we can define, more generally, $f^* := \lambda_{x:\Gamma.UFA.\Gamma'}(f; \eta_{UB})^*\{x\}; U\epsilon_{B(x)}$, where η is the counit of the adjunction $F \dashv U$.

Remark 5.3.4 (Dependent Costrength)**.** *Note that dependent Kleisli extensions allow us, in particular, to define the* ***dependent costrength*** $s'_{A,B}$ *for the monad* $T := UF$ *that we were missing (starting from the identity on* $F\Sigma_A(B\{\mathbf{p}_{\Gamma,\eta_A}\})$*):*

$$\begin{aligned}
&\mathcal{D}(\Gamma)(F\Sigma_A(B\{\mathbf{p}_{\Gamma,\eta_A}\}), F\Sigma_A(B\{\mathbf{p}_{\Gamma,\eta_A}\}))\\
\cong\ &\mathcal{C}(\Gamma)(\Sigma_A B\{\mathbf{p}_{\Gamma,\eta_A}\}, T\Sigma_A(B\{\mathbf{p}_{\Gamma,\eta_A}\}))\\
\cong\ &\mathcal{C}(\Gamma.A.B\{\mathbf{p}_{\Gamma,\eta_A}\})(1, T\Sigma_A(B\{\mathbf{p}_{\Gamma,\eta_A}\})\{\mathbf{p}_{\Gamma,A}\}\{\mathbf{p}_{\Gamma.A,B\{\mathbf{p}_{\Gamma,\eta_A}\}}\})\\
\cong\ &\mathcal{C}(\Gamma.A.B\{\mathbf{p}_{\Gamma,\eta_A}\})(1, T\Sigma_A(B\{\mathbf{p}_{\Gamma,\eta_A}\})\{\mathbf{p}_{\Gamma,TA}\}\{\mathbf{p}_{\Gamma.TA,B}\}\{\mathbf{q}_{\mathbf{p}_{\Gamma,\eta_A},B}\})\\
\xrightarrow{(-)^*}\ &\mathcal{C}(\Gamma.TA.B)(1, T\Sigma_A(B\{\mathbf{p}_{\Gamma,\eta_A}\})\{\mathbf{p}_{\Gamma,TA}\}\{\mathbf{p}_{\Gamma.TA,B}\})\\
\cong\ &\mathcal{C}(\Gamma)(\Sigma_{TA}B, T\Sigma_A(B\{\mathbf{p}_{\Gamma,\eta_A}\})).
\end{aligned}$$

As a consequence, we are able to define both a left and a right pairing (which will in general not coincide for non-commutative effects):

$$\Sigma_{TA}TB \xrightarrow{s'} T\Sigma_A TB\{\mathbf{p}_{\Gamma,\eta_A}\} \xrightarrow{Ts} T^2\Sigma_A B\{\mathbf{p}_{\Gamma,\eta_A}\} \xrightarrow{\mu} T\Sigma_A B\{\mathbf{p}_{\Gamma,\eta_A}\}$$

$$\Sigma_{TA}TB \xrightarrow{s} T\Sigma_{TA}B \xrightarrow{Ts'} T^2\Sigma_A B\{\mathbf{p}_{\Gamma,\eta_A}\} \xrightarrow{\mu} T\Sigma_A B\{\mathbf{p}_{\Gamma,\eta_A}\}.$$

Theorem 5.3.5 (dCBPV+ Semantics)**.** *We have a sound interpretation of dCBPV+ in a dCBPV+ model. The interpretation in such categories is complete in the sense that an equality of values or computations holds in all interpretations iff it is provable in the syntax of dCBPV+. In fact, the interpretation defines a 1-1 correspondence between categorical models and syntactic theories in dCBPV+ which satisfy mutual soundness and completeness results.*

Proof. This follows from theorem 5.2.2 together with the observation that we can interpret the rule of figure 5.9 by dependent Kleisli extensions combined with composition. Conversely, we can apply the rule of figure 5.9 with $\Gamma, x : UFA; \cdot \vdash \mathsf{force}\ x : FA$ for M to derive the rule for dependent Kleisli extensions. □

Theorem 5.3.6 (Dependent CBN Semantics 2)**.** *The (semantic equivalent of the) CBN translation of DTT with* $\Sigma_{1\leq i\leq n}$*-,* 1*-,* Σ*-,* Id*-,* $\Pi_{1\leq i\leq n}$*-,* Π*-types, where we use the strong (dependent) elimination rules for all positive connectives, into dCBPV+, lets us construct a categorical model of CBN dependent type theory with*

the connectives above out of any model of dCBPV+ by taking the co-Kleisli category for $! = FU$*. The interpretation of CBN dependent type theory is sound and complete for the equational theory induced from dCBPV+:*

$$[\![\underline{B_1}, \cdots, \underline{B_n} \vdash \underline{B}]\!] = \mathcal{D}(U[\![\underline{B_1}]\!].\cdots.U[\![\underline{B_n}]\!])(F1, [\![\underline{B}]\!]) \cong \mathcal{D}_!(U[\![\underline{B_1}]\!].\cdots.U[\![\underline{B_n}]\!])(\top, [\![\underline{B}]\!]).$$

Theorem 5.3.7 (Dependent CBV Semantics)**.** *The (semantic equivalent of the) CBV translation of DTT with* $\Sigma_{1\leq i\leq n}$*-,* 1*-,* Σ*-,* Id*-,* $\Pi_{1\leq i\leq n}$*-,* Π*-types, where we use the strong (dependent) elimination rules for all positive connectives, into dCBPV+, lets us construct a categorical model of CBV dependent type theory with the connectives above out of any model of dCBPV+ by taking the Kleisli category for* $T = UF$*. The interpretation of CBN dependent type theory is sound and complete for the equational theory induced from dCBPV+:*

$$\begin{aligned}[\![A_1, \cdots, A_n \vdash A]\!] &= \mathcal{D}([\![A_1]\!].\cdots.[\![A_n]\!]\{\eta_{[\![A_1]\!],\ldots,[\![A_n]\!]}\})(F1, F[\![A]\!]\{\eta_{[\![A_1]\!],\ldots,[\![A_n]\!]}\}) \\ &\cong \mathcal{C}_T([\![A_1]\!].\cdots.[\![A_n]\!]\{\eta_{[\![A_1]\!],\ldots,[\![A_n]\!]}\})(1, [\![A]\!]\{\eta_{[\![A_1]\!],\ldots,[\![A_n]\!]}\}).\end{aligned}$$

Here, $\eta_{[\![A_1]\!],\ldots,[\![A_n]\!]}$ *is inductively defined by*

$$\eta_{[\![A_1]\!],\ldots,[\![A_k]\!]} := \mathbf{q}_{\eta_{[\![A_1]\!],\ldots,[\![A_{k-1}]\!]},[\![A_k]\!]}; \mathbf{p}_{[\![UFA_1]\!].\cdots.[\![UFA_{k-1}]\!],\eta_{[\![A_k]\!]}}.$$

Remark 5.3.8. *We have finally arrived at a notion of a model for CBV dependent type theory. It seems much less straightforward than the corresponding notion of a model for CBN dependent type theory as a particular kind of model of pure dependent type theory in which the* η*-laws for positive connectives may fail. Then again, a similar phenomenon is already seen in the simply typed case.*

5.3.3 Some Basic Models and Non-Models

As for dCBPV-, we can note that the identity adjunction on any model of pure DTT (in particular, the families of sets model) gives a model of dCBPV+, which demonstrates consistency.

Theorem 5.3.9 (Consistency)**.** *dCBPV+ is consistent both in the sense that not all terms are identified and in the sense that not all types are inhabited.*

Indeed, the identity monad on any model of DTT trivially admits dependent Kleisli extensions.

However, as we shall see, it is not the case that any model of dCBPV- extends to a model of dCBPV+. In particular, not every indexed monad on a model of pure DTT admits dependent Kleisli extensions. As it turns out, the existence of dependent Kleisli extensions needs to be assessed on a case-by-case basis. As we shall see, in the case of various set-theoretic models, dependent Kleisli extensions naturally lead to certain subtyping conditions as a necessary requirement which can't always be satisfied. Therefore, we treat some dCBPV- models for common effects and discuss the (im)possibility of dependent Kleisli extensions.

5.3.3.1 A Non-Model and A Model: Writing

We let $\mathcal{B}$ be Set and $\mathcal{C}$ be $\mathsf{Fam}(\mathsf{Set})$. Let M be a non-trivial monoid, for instance a monoid of strings of ASCII characters. Then, we let $\mathcal{D}$ be the Eilenberg-Moore category for the indexed monad $- \times M$. Now, we note that dependent Kleisli extensions do not have a sound interpretation in this model of dCBPV-. Indeed, it would amount to giving appropriate maps

$$\mathsf{Fam}(\mathsf{Set})(\Gamma.A)(1, B\{\langle \mathsf{id}_\Gamma, \mathsf{id}_A, 1_M\rangle\} \times M) \xrightarrow{(-)^*} \mathsf{Fam}(\mathsf{Set})(\Gamma.(A \times M))(1, B \times M)$$

$$\Pi_{\langle c,a\rangle \in \Gamma.A} B(c, a, 1_M) \times M \xrightarrow{\qquad (-)^* \qquad} \Pi_{\langle c,a,m\rangle \in \Gamma.(A\times M)} B(c, a, m) \times M$$

$$f = \langle f_B, f_M\rangle \longmapsto \lambda_{c,a,m}\langle ?, f_M(c, a) * m\rangle.$$

We see that this is not always possible. For instance, let $\Gamma = 1 = A$ and let B be a predicate that expresses that $m = 1_M$ (a predicate which says that no printing happens). In that case, any f^* cannot be a total function as it cannot send, for instance, $(*, *, \texttt{hello world})$ anywhere.

We would like to stress that this does not show that dependent Kleisli extensions are incompatible with printing. Indeed, it only shows that this particular model of printing does not admit dependent Kleisli extensions. One could conceive of, for example, a model of printing where types depending on TA are not allowed to refer to what is being printed, in which case we could define $f^*(c, a, m) :=$

$\langle f_B(c,a), f_M(c,a) * m\rangle$. More generally, such a definition could work if, for all $m \in M,\ B(c,a,1_M) \subseteq B(c,a,m)$.

A concrete instantiation of this idea can be given by considering the setoid model of dependent type theory instead [27], which has as objects sets with an equivalence relation, as morphisms functions which send equivalent elements to equivalent elements and as dependent types equivalence respecting families. Note that any monoid M in Set can be equipped with the codiscrete equivalence relation which identifies all elements to give a monoid internal to the category of setoids. This, in turn, defines an indexed monad $- \times M$ on the setoid model of type theory, which lets us model printing. Note that in this case, predicates cannot distinguish between functions with the same input output behaviour but different printing behaviour. The result is a model of dCBPV+ which models printing (which happens to have intensional Id-types). However, note that from the point of view of the identity types, M only appears to have one element (although the judgemental equality can distinguish between the elements of M).

5.3.3.2 A Non-Model: Reading

We let $\mathcal{B}$ be Set and $\mathcal{C}$ be $\mathsf{Fam}(\mathsf{Set})$. Let S be some non-trivial set (that is, not 0 or 1), which we think of as a set of states for a storage cell. Then, we let $\mathcal{D}$ be the Eilenberg-Moore category for the indexed monad $(-)^S$. Now, we note that dependent Kleisli extensions do not have a sound interpretation in this model of dCBPV-. Indeed, it would amount to giving appropriate maps

$$\mathsf{Fam}(\mathsf{Set})(\Gamma.A)(1, B\{\lambda_s\langle \mathsf{id}_\Gamma, \mathsf{id}_A\rangle\}^S) \xrightarrow{(-)^*} \mathsf{Fam}(\mathsf{Set})(\Gamma.(A^S))(1, B^S)$$

$$\Pi_{\langle c,a\rangle\in\Gamma.A} B(s \mapsto \langle c,a\rangle)^S \xrightarrow{(-)^*} \Pi_{(s\mapsto\langle c,a_s\rangle)\in\Gamma.(A^S)} B(s \mapsto \langle c,a_s\rangle)^S$$

$$f \longmapsto \lambda_{s\mapsto\langle c,a_s\rangle}\lambda_{s'}?.$$

We see that this is not always possible. For instance, let $\Gamma = 1$ and $A = 2$ and let B be a predicate that expresses that $s \mapsto \langle *, a_s\rangle$ is constant. In that case, any f^* cannot be a total function as it cannot send a non-constant $s \mapsto \langle *, a_s\rangle$ anywhere.

If we want to define, as usual, $f^*(s \mapsto \langle c, a_s \rangle)(s') := f(c, a_{s'})(s')$, we require that for all fixed $s' \in S$, $B(s \mapsto \langle c, a_{s'} \rangle) \subseteq B(s \mapsto \langle c, a_s \rangle)$, which is easily seen to be equivalent to B being constant on A^S.

5.3.3.3 A Non-Model: Global State

Similarly, for global state, we let $\mathcal{B}$ be Set and $\mathcal{C}$ be $\mathsf{Fam}(\mathsf{Set})$ and we take $T := (- \times S)^S$, where S is a non-trivial set, and let $\mathcal{D}$ be the Eilenberg-Moore category for T. Then, dependent Kleisli extensions would amount to appropriate maps

$$\mathsf{Fam}(\mathsf{Set})(\Gamma.A)(1, (B\{\lambda_s\langle \mathsf{id}_\Gamma, \mathsf{id}_A\rangle\} \times S)^S) \xrightarrow{(-)^*} \mathsf{Fam}(\mathsf{Set})(\Gamma.((A \times S)^S))(1, (B \times S)^S)$$

$$\Pi_{\langle c,a\rangle \in \Gamma.A}(B(s \mapsto \langle c, a, s\rangle) \times S)^S \xrightarrow{(-)^*} \Pi_{(s \mapsto \langle c,a_s,t_s\rangle) \in \Gamma.((A\times S)^S)}(B(s \mapsto \langle c, a_s, t_s\rangle) \times S)^S$$

$$f \longmapsto \lambda_{s \mapsto \langle c,a_s,t_s\rangle}\lambda_{s'}?.$$

Now, B could express the property that $a_s = a$ (a_s is independent of s) and $t_s = s$. In that case, no such dependent Kleisli extension exists.

One could imagine a different model of global state, however, in which, for every fixed $s' \in S$, $B(s \mapsto \langle c, a_{s'}, s\rangle) \subseteq B(s \mapsto \langle c, a_s, t_s\rangle)$. In that case, one could define as one normally (for non-dependent Kleisli extensions) would $f^*(s \mapsto \langle c, a_s, t_s\rangle)(s') := f(c, a_{s'})(s')$. At present, it is not clear to the author if non-trivial models along these lines exist.

5.3.3.4 A Model: Exceptions or Divergence

We consider a model for exceptions or divergence, where we use the monad $T = E + (-)$ on $\mathsf{Fam}(\mathsf{Set})$, for some fixed set E whose elements we think of as either exceptions or, perhaps, in the case of $E = 1$, divergence. We let $\mathcal{B}$ be Set and $\mathcal{C}$ be $\mathsf{Fam}(\mathsf{Set})$ and we take for $\mathcal{D}$ the Eilenberg-Moore category for T. In this

case, we in fact have maps

$$\mathsf{Fam}(\mathsf{Set})(\Gamma.A)(1, E + B\{\langle \mathsf{id}_\Gamma, \mathsf{inr}\rangle\}) \xrightarrow{(-)^*} \mathsf{Fam}(\mathsf{Set})(\Gamma.(E+A))(1, E+B)$$

$$\Pi_{\langle c,a\rangle \in \Gamma.A} E + B(c, \mathsf{inr}\ a) \xrightarrow{\quad (-)^* \quad} \Pi_{\langle c,t\rangle \in \Gamma.(E+A)} E + B(c,t)$$

$$f \longmapsto \lambda_c[\mathsf{inl}, f(c,-)].$$

These are easily seen to give a sound interpretation of dependent Kleisli extensions. They indeed model the propagation of exceptions one would expect.

5.3.3.5 A Dubious Model: Erratic Choice

We consider a model for erratic choice, where we use the powerset monad $T = \mathcal{P}$ on $\mathsf{Fam}(\mathsf{Set})$. We let $\mathcal{B}$ be Set and $\mathcal{C}$ be $\mathsf{Fam}(\mathsf{Set})$ and we take for $\mathcal{D}$ the Eilenberg-Moore category for T. Dependent Kleisli extensions would amount to appropriate maps

$$\mathsf{Fam}(\mathsf{Set})(\Gamma.A)(1, \mathcal{P}B\{\langle \mathsf{id}_\Gamma, x \mapsto \{x\}\rangle\}) \xrightarrow{(-)^*} \mathsf{Fam}(\mathsf{Set})(\Gamma.(\mathcal{P}A))(1, \mathcal{P}B)$$

$$\Pi_{\langle c,a\rangle \in \Gamma.A} \mathcal{P}B(c, \{a\}) \xrightarrow{\quad (-)^* \quad} \Pi_{\langle c,t\rangle \in \Gamma.(\mathcal{P}A)} \mathcal{P}B(c,t)$$

$$f \longmapsto \lambda_{c,t}?.$$

We can, in principle, define $f^*(c,t) := (\bigcup_{a\in t} f(c,a)) \cap B(c,t)$ to obtain a dependent Kleisli extension. However, this model might not correspond to the expected operational semantics. It would be preferable to consider, instead, a model of type theory $\mathcal{C}$ in which it is always the case that $\bigcup_{a\in t} B(c,\{a\}) \subseteq B(c,t)$, in which case we can just define $f^*(c,t) := \bigcup_{a\in t} f(c,a)$ (cf. reader monad). At present, it is not clear to the author how a model with such properties can be constructed.

5.3.3.6 A Puzzle: Control Operators

We consider a dCBPV- model for control operators, where we use a continuation monad $T = R^{(R^-)}$ on $\mathsf{Fam}(\mathsf{Set})$, for some non-trivial set R. We let $\mathcal{B}$ be Set and $\mathcal{C}$ be $\mathsf{Fam}(\mathsf{Set})$ and we take for $\mathcal{D}$ the Eilenberg-Moore category for T. Dependent

Kleisli extensions would amount to appropriate maps

$$\mathsf{Fam}(\mathsf{Set})(\Gamma.A)(1,(R^{(R^{B\{\langle \mathsf{id}_\Gamma, x\mapsto \mathsf{ev}_x\rangle\}})}) \xrightarrow{(-)^*} \mathsf{Fam}(\mathsf{Set})(\Gamma.(R^{(R^A)}))(1,R^{(R^B)})$$

$$\Pi_{\langle c,a\rangle\in\Gamma.A}(R^{(R^{B(c,\mathsf{ev}_a)})}) \xrightarrow{(-)^*} \Pi_{\langle c,t\rangle\in\Gamma.(R^{(R^A)})}(R^{(R^{B(c,t)})})$$

$$f \longmapsto \lambda_{c,t}?.$$

In order to match the expected operational semantics, it is tempting to try to define, just as in the simply typed case, $f^*(c,t)(k) := t(\lambda_a f(c,a)(k))$. However, this is only well-defined if we have $\forall_{a\in A(c)} R^{B(c,t)} \subseteq R^{B(c,\mathsf{ev}_a)}$. In particular, we would have that $B(c,\mathsf{ev}_a) = B(c,\mathsf{ev}_{a'})$ for all $a,a' \in A(c)$. This suggests a kind of incompatibility between control operators and dependent Kleisli extensions. We would like to further investigate the combination of dCBPV with control operators in future work, especially given the correspondence with classical logic. In the light of [101], we already know that the combination of dependent types and control operators can easily lead to degeneracy of the system (in the sense that all programs get equated propositionally).

5.3.3.7 A Model: Recursion

Note that the model of the dependent LNL calculus of section 3.5.4 in particular gives us a model of dCBPV-. The model clearly supports recursion, as we can define our usual fixpoint combinators. This model is easily seen further to support dependent Kleisli extensions: similar to our previous model of divergence, for a dependent function f, we define the Kleisli extension f^* as sending the new bottom element to bottom and otherwise acting as f.

5.3.4 Operational Semantics and Effects

Using the CK-machine, we can again define an operational semantics for dCBPV+.

The definition of the operational semantics does not change in the presence of dependent Kleisli extensions and is exactly as that described in section 5.2.4. In particular, figures 2.12 and 5.8 define a CK-machine on which we evaluate the computations of pure dCBPV+. As before, we can add the effects of figure 2.13

together with their operational semantics of figures 2.14 and 2.15 and equations of figure 2.17. We still have the same determinacy and strong normalization results as before, as the essentially untyped proofs remain valid.

Theorem 5.3.10 (Determinacy, Strong Normalization)**.** *No transition occurs precisely if we are in a terminal configuration. In absence of erratic choice, at most one transition applies to each configuration. In absence of divergence and recursion, every configuration reduces to a terminal configuration in a finite number of steps.*

However, the results of section 5.3.3 are reflected at the level of the operational semantics. While for some effects like divergence, exceptions and recursion, subject reduction can be established, certain subtyping conditions are necessary to obtain subject reduction in the presence of printing, global state and erratic choice. It is at present not clear if a these conditions are compatible with, for instance, $\Pi^{\multimap}_{F(-)}$-types.

Theorem 5.3.11 (Limited Subject Reduction)**.** *In absence of printing, global state and erratic choice, if the sequence of reductions of a well-typed computation M passes through a well-typed configuration M, K and later another configuration M', K, then the latter configuration is also well-typed and has the same type as the former.*

Proof. It is easy to see that all transitions preserve the type of a configuration as for dCBPV-, with the exception of the transitions for M to x in N and return V_{nf}. Both involve a stack $[\cdot]$ to x in $N :: K$ which is untypable when x is free in the type of N. The crux, however, is that these transitions always occur in pairs and, in this case, two wrongs make a right. Say that we are evaluating a well-typed computation and that the former transition occurs from M to x in N, K to $M, [\cdot]$ to x in $N :: K$ where $\Gamma; \cdot \vdash M$ to x in $N : \underline{B}[\mathsf{thunk}\ M/z]$. After that, several other transitions may occur, but if we return to another configuration M', K, we know that the last transition that occurred was that from $\mathsf{return}\ V_{\mathsf{nf}}, [\cdot]$ to x in $N :: K \rightsquigarrow N[V_{\mathsf{nf}}/x], K$ (and so, $M' = N[V_{\mathsf{nf}}/x]$).

Our claim is that also $\Gamma; \cdot \vdash N[V_{\mathsf{nf}}/x] : \underline{B}[\mathsf{thunk}\ M/z]$ (if so, then the theorem follows). The important thing to notice is that, from inversion on $\Gamma; \cdot \vdash M$ to x in N :

$$\frac{\Gamma;\Delta \vdash K : \underline{B}[\mathsf{thunk}\ M/z]}{\Gamma;\Delta \vdash K : \underline{B}[\mathsf{thunk}\ (\mathsf{print}\ m\ .\ M)/z]} \qquad \frac{\Gamma;\Delta \vdash K : \underline{B}[\mathsf{thunk}\ M_{i'}/z]}{\Gamma;\Delta \vdash K : \underline{B}[\mathsf{thunk}\ (\mathsf{choose}_i(M_i))/z]}$$

$$\frac{\Gamma;\Delta \vdash K : \underline{B}[\mathsf{thunk}\ M)/z]}{\Gamma;\Delta \vdash K : \underline{B}[\mathsf{thunk}\ (\mathsf{write}\ s\ .\ M)/z]} \qquad \frac{\Gamma;\Delta \vdash K : \underline{B}[\mathsf{thunk}\ M_{s'}/z]}{\Gamma;\Delta \vdash K : \underline{B}[\mathsf{thunk}\ (\mathsf{readto}_s(M_s))/z]}$$

Figure 5.10: Extra rules that are necessary in dCBPV+ to establish subject reduction in the presence of printing, global state and erratic choice.

$\underline{B}[\mathsf{thunk}\ M/z]$, we have that $\Gamma, x : A; \cdot \vdash N : \underline{B}[\mathsf{thunk}\ \mathsf{return}\ x/z]$. Therefore, it follows that $\Gamma; \cdot \vdash N[V/x] : \underline{B}[\mathsf{thunk}\ \mathsf{return}\ V/z]$. Our claim follows by noting that $\mathsf{return}\ V = M$ as all reductions that could have been applied to M are also equalities (seeing that the only effects we allow are recursion, divergence and errors, all of whose transitions are equations, as are β-reductions). □

The proof above shows why subject reduction may fail in the presence of printing, global state and erratic choice: their transitions $M, K \rightsquigarrow M', K$ of figures 2.14 and 2.15 on computations are not contained in the judgemental equalities we consider (see figure 2.17) in the sense that not $M = M'$. They represent real dynamics. In this sense, they differ from the other transitions we have considered. In fact, it is not reasonable to demand such an equality. In particular, in the case of reading global state and erratic choice, that would lead to all computations of the same type being equated.

Remark 5.3.12. *On closer inspection, however, it seems that what we really needed to establish subject reduction was an inclusion of computation types, whenever* $M, K, m, s \rightsquigarrow M', K, m', s'$,

$$\frac{\Gamma;\Delta \vdash K : \underline{B}[\mathsf{thunk}\ M'/z]}{\Gamma;\Delta \vdash K : \underline{B}[\mathsf{thunk}\ M/z].}$$

The idea is that the type of a computation becomes more determined in the computation progresses. We list concrete instantiations of this rule in figure 5.10. It is clear that admissability of these rules is a necessary condition to establish subject reduction property for dCBPV+ (compare this to the results in section 5.3.3!). What

$$\frac{\vdash \Gamma, z_1 : U\underline{B_1}, \ldots, z_n : U\underline{B_n} \ \mathsf{ctxt}}{\Gamma \vdash \Pi^{z_1,\ldots,z_n}_{1\leq i\leq n} \underline{B_i} \ \mathsf{ctype}} \qquad \frac{\{\Gamma; \cdot \vdash M_i : \underline{B_i}[\mathsf{thunk}\ M_1/z_1, \ldots, \mathsf{thunk}\ M_{i-1}/z_{i-1}]\}_{1\leq i\leq n}}{\Gamma; \cdot \vdash \lambda_i M_i : \Pi^{z_1,\ldots,z_n}_{1\leq i\leq n} \underline{B_i}}$$

$$\frac{\Gamma; \cdot \vdash M : \Pi^{z_1,\ldots,z_n}_{1\leq i\leq n} \underline{B_i}}{\Gamma; \cdot \vdash i`M : \underline{B_i}[\mathsf{thunk}\ 1`M/z_1, \ldots, \mathsf{thunk}\ (i-1)`M/z_{i-1}]}$$

Figure 5.11: Rules for dependent projection products. We also demand the obvious β- and η-laws.

is less clear, is if as adding them to the type system is sufficient, as this complicates the usual subject reduction proof, which relies on inversion on the typing rules.

5.4 Dependent Projection Products?

It was somewhat surprising, perhaps, that while dependent pattern matching products arise so naturally in CBPV, dependent projection products seem less natural. The reader should compare this to the status of additive Σ-types, their cousins in linear logic, which often fail to be supported in natural models. In principle, we could include the system of rules of figure 5.11 in dCBPV to replace $\Pi_{1\leq i\leq n}$-types. This allows us to define the appropriate CBV and CBN translations for dependent projection products in dCBPV, exactly as one defines the translation for simple projection products. This translation re-enforces the idea that the CBV translation of a type $x_1 : A_1, \ldots, x_n : A_n \vdash A \ \mathsf{type}$ should be $z_1 : UFA^v_1, \ldots, z_n : UFA^v_n \vdash A^v \ \mathsf{vtype}$. We note that we have CBV and CBN translations of dependent projection products (which have a dependent/strong elimination principle) even in dCBPV-. Moreover, we can use the usual operational semantics of computations of type $\Pi_{1\leq i\leq n}\underline{B_i}$ for these types.

Although we can formulate a sound and complete categorical semantics for dependent projection products (we demand strong n-ary Σ-types in $\mathcal{D}$ in the sense of objects $\Pi^{dep}_{1\leq i\leq n}\underline{B_i}$ such that $\mathbf{p}_{\Gamma, U\Pi^{dep}_{1\leq i\leq n}\underline{B_i}} = \mathbf{p}_{\Gamma.U\underline{B_1}.\cdots.U\underline{B_{n-1}}, U\underline{B_n}}; \ldots; \mathbf{p}_{\Gamma, U\underline{B_1}}$), many models fail to support these connectives in practice. In particular, they are hard to obtain in models of linear logic, where they would give additive Σ-types in the sense of objects $\Sigma^{\&}_A B$ such that $!\Sigma^{\&}_A B \cong \Sigma^{\otimes}_{!A} !B$, and are difficult to give a satisfactory

interpretation in models of the monadic metalanguage, where they would correspond to the construction of a T-algebra structure on $\Sigma_{Uk}Ul$, given $l \in \mathcal{C}^T(\Gamma.Uk)$.

A related phenomenon is that subject reduction for dependent projection products can be problematic to establish (for the obvious operational semantics on untyped terms which is identical to that for $\Pi_{1\leq i\leq n}$-types). We encourage the reader to think of dependent projection products in a similar way to dependent Kleisli extensions, as their problems with subject reduction have a similar origin. That is, they lead to types depending on (thunks of) computations which might not be static objects during reduction (in the sense that some reductions might not be equalities in the presence of some effects). In that case, we are faced with a choice, either subject reduction fails or we have to make types into dynamic objects as well, meaning that they no longer provide the static guarantees which are their primary raison d'être.

Theorem 5.4.1 (Limited Subject Reduction)**.** *Let us consider dCBPV- with dependent projection products. In absence of printing, global state and erratic choice, if the sequence of reductions of a well-typed computation M passes through a well-typed configuration M, K and later another configuration M', K, then the latter configuration is also well-typed and has the same type as the former.*

Proof. The proof is very similar to that of theorem 5.3.11. Indeed, starting from a well-typed configuration $\Gamma; \cdot \vdash j\text{`}M : \underline{B}_j[\mathsf{thunk}\ 1\text{`}M/z_1, \ldots, \mathsf{thunk}\ (j-1)\text{`}M/z_{j-1}]$, $\Gamma; \mathsf{nil} : \underline{B}_j[\mathsf{thunk}\ 1\text{`}M/z_1, \ldots, \mathsf{thunk}\ (j-1)\text{`}M/z_{j-1}] \vdash K : \underline{C}$, we transition into $M, j :: K$, where $j :: K := \mathsf{let}\ \mathsf{nil}_1\ \mathsf{be}\ j\text{`}\mathsf{nil}_2\ \mathsf{in}\ K$ is an untypable stack. Eventually, if have transitioned into a configuration $\lambda_i M_i, j :: K$, we next transition into M_j, K, where we can derive by inversion on $\lambda_i M_i$ that $\Gamma; \cdot \vdash M_i : \underline{B}_i[\mathsf{thunk}\ M_1/z_1, \ldots, \mathsf{thunk}\ M_{i-1}/z_{i-1}]$. We have now arrived at a well-typed configuration again if we can show that $\underline{B}_j[\mathsf{thunk}\ M_1/z_1, \ldots, \mathsf{thunk}\ M_{j-1}/z_{j-1}] = \underline{B}_j[\mathsf{thunk}\ 1\text{`}M/z_1, \ldots, \mathsf{thunk}\ (j-1)\text{`}M/z_{j-1}]$. This follows if we can show that $M_i = i\text{`}M$ for all $1 \leq i \leq j-1$, which we know to be generally true in absence of of printing, global state and erratic choice and false otherwise. □

One could add similar negative versions of the other positive connectives like identity types (which we have called additive identity types in the context of linear logic). Their categorical semantics would correspond to having computation type formers $R(\underline{B_1}, \ldots, \underline{B_n})$ that U maps to $R'(U\underline{B_1}, \ldots, U\underline{B_n})$ where R' is the corresponding positive type former. In the obvious operational semantics, destructors push to the stack and constructors pop the stack and substitute.

Let us briefly consider some specific models. We have already seen in section 3.5.4 that the domain model of dCBPV+ supports additive Σ-types as well. Similarly, the error monad admits a satisfactory definition of dependent projection products. We define the algebra structure $\Sigma_k l$, as expected, by $(\Sigma_k l)(e) := \langle k(e), l(k(e))(e)\rangle$.

Note that for the writer monad, we cannot use the expected generalisation of the product algebra structure on $\Sigma_{Uk} Ul$. Instead, we can use the trivial algebra structure. (Note that dependent projection products are only a generalisation of the product in a weak sense. In particular, they are far from unique.) That may not be what we are hoping for though, as the individual algebra structures k and l on Uk and Ul are ignored in the construction. (The product action may not respect the fibres of the Σ-type if l is not invariant under k.) Similarly, for the reader and global state monads, we can equip $\Sigma_{Uk} Ul$ with an algebra structure by evaluating at an arbitrary state. Again, similarly, note that an algebra for the powerset monad is a join-semi-lattice. Therefore, assuming the axiom of choice (or rather, its equivalent, the well-ordering principle), we can define dependent projection products by equipping $\Sigma_{Uk} Ul$ with some well-order. However, this is not the algebra structure we would be expecting (the product order), as this may fail to be a join-semi-lattice (take, for instance, $k = \{0 \leq 1\} = l(0)$ and $l(1) = \{0\}$; then $\langle 0, 1\rangle$ and $\langle 1, 0\rangle$ do not have a join).

We have already seen in section 3.5.5 that additive Σ-types are not always supported in models of linear dependent type theory.

As projection products are more natural than pattern matching products in CBN (or, at least, more customary), we see that the CBN-translation into dCBPV runs into similar problems as the CBV. Where the latter requires us to extend

dCBPV- with dependent Kleisli extensions, the former at least strongly suggests adding dependent projection products to dCBPV-[6]. Both these extensions lead to similar challenges with constructing concrete models and with establishing subject reduction.

5.5 Dependent Kleisli Extensions: a Bug or a Feature?

5.5.1 Unrestricted Effects and Dependent Types?

In sections 5.2 and 5.3, we introduced two systems which combine dependent types with computational effects: dCBPV- and dCBPV+. Recall that the latter extends the former with a rule for Kleisli extensions of dependent functions.

One motivation for studying dCBPV+ is the possible criticism that can be made that in dCBPV- dependent types and effects sit side-by-side and do not interact meaningfully. (More about that later.) Another is the observation that we need dependent Kleisli extensions to obtain a well-defined CBV or CBN translation of dependent type theory with unrestricted effects (which are not encapsulated by the type system) into dCBPV. This may be interesting as real world languages like Agda and Idris include unrestricted recursion and only exclude non-terminating terms at a later stage through a termination check which is separate from type checking [103, 118]. Moreover, models of dependent type theory in categories of domains and games naturally model unrestricted effects. Therefore, we believed it to be important, from a theoretical point of view at least, to study the system dCBPV+.

It should be clear to the reader, however, that the expressive power of dCBPV+ comes at a cost of simplicity and, in particular, in many cases, subject reduction. The question is if dCBPV- or dCBPV+ is more suitable for practical implementations. We would like to argue that dCBPV+ does not add much practical value over dCBPV- as a programming language.

[6] For instance, we require these if we want the co-Kleisli category for $! = FU$ to give a model of DTT indexed over $\mathcal{D}(\cdot)_!$ (recall that this category is equivalent to full category of $\mathcal{B}$ on the objects in image of U), rather than merely over $\mathcal{B}$.

Indeed, let us return to the primary practical motivation for wanting to combine dependent types and effects: having a single elegant language in which we can both write practical software and perform its verification. For these purposes, as argued in section 1.1.5, it is crucial that we constrain where effects are allowed to occur using the type system, for instance using modalities, as effects usually do not correspond to sound logical principles so should be excluded from proofs. For this reason, dependent type theory with unrestricted effects (and with it the corresponding CBV and CBN translations) is not what we are most interested in. Rather, a modal and ideally adjunction language like dCBPV is closer to what we are looking for.

5.5.2 Fundamentalist vs Pragmatic Dependent Types

Observe that, in practice, dependent types tend to be used in two closely related but slightly different styles[7]. On the one hand we have a style of programming where we build up the program immediately from dependently typed building blocks $c : C$, where C may be formed using inductive families and other dependently typed constructs, by writing the code and proving its properties simultaneously, fundamentalist dependently typed programming, if you will. Some examples for C include a type of lists of a fixed length, sorted lists, heaps or binary search trees, red-black trees, suitably balanced trees and a type of λ-terms up to $\beta\eta$-equality. On the other hand, we can write simply typed programs $a : A$ first, where A is a datatype formed from simple inductive types and simple connectives, like mere lists or binary trees, and only later prove the required properties $a' : A'[a/x]$ (where A' is a proof-relevant predicate, like the BST property, formed using inductive families and other dependently typed constructions). This is a more pragmatic stance where dependent types are simply seen as a tool for expressing appropriate program properties that we want to verify.

The latter style seems to be more popular in practice and more suitable for the creation of large modular code bases. The reason for this is that we often only

[7]These can be seen to be closely related to the two traditional schools of thought about types: typing à la Church and à la Curry [119]. Also closely related to the fundamentalist school is the correctness-by-construction programming methodology advocated by Dijkstra and others [120].

decide on the properties we need to verify after we have already written code and, in fact, we often need to verify different properties of the same code in different contexts. It should be noted that both points of view are equivalent, that the distinction is mostly a matter of style and that both styles can be combined well.

To illustrate the distinction, let us consider lists of length n. We can either view them directly as single inductive family $x : \mathbb{N} \vdash \mathsf{ListOfLen}(x)\ \ \mathsf{vtype}$ (corresponding to the former style) or as a predicate $x : \mathbb{N}, y : \mathsf{List} \vdash \mathsf{has}-\mathsf{length}(y, x)\ \ \mathsf{vtype}$ from which we form $x : \mathbb{N} \vdash \Sigma_{y:\mathsf{List}}\mathsf{has}-\mathsf{length}(y, x)\ \ \mathsf{vtype}$ (corresponding to the latter). If we want to write a program that for every $x : \mathbb{N}$ returns a list of length x, we write, in the fundamentalist view, directly, proof-carrying code $\vdash f : \Pi_{x:\mathbb{N}}\mathsf{ListOfLen}(x)$, which can be though of as both an algorithm producing a list and a proof that that list has length x in one. Meanwhile, in the more pragmatic view of post hoc verification using dependent types, we first write the algorithm $\vdash g : \mathbb{N} \Rightarrow \mathsf{List}$ and then a separate proof $\vdash p : \Pi_{x:\mathbb{N}}\mathsf{has}-\mathsf{length}(g(x), x)$ about g.

The latter point of view generalises without problems to dCBPV-. Indeed, by keeping the (simply typed and effectful) algorithm separate from the (dependently typed and pure) proof, all we need is a sequencing operation from simply typed effectful computations and a regular composition operation for pure dependent functions, both of which are available in dCBPV-. To be precise, we write an effectful simply typed algorithm $\vdash g : \mathbb{N} \ {}_F\!\multimap F\mathsf{List}$ and a separate pure dependently typed proof $\vdash p : \mathsf{my}-\mathsf{favourite}-\mathsf{property}(\mathsf{thunk}\ g)$ about g. Here, we would like to point out that $z : U(\mathbb{N} \ {}_F\!\multimap F\mathsf{List}) \vdash \mathsf{my}-\mathsf{favourite}-\mathsf{property}(z)\ \ \mathsf{vtype}$. Therefore, to make dCBPV- into a practical system for verification of effectful programs, it is crucial that we extend it with mechanisms for defining interesting (value) types depending on types of thunks of effectful computations. In the case that we are working with printing with values in some monoid M internal to the type theory, a simple example of a property to express using a type family could be that the program does not print anything or that its return value has a specific property. In particular, any type depending on $A \times M$ should give rise to a type depending on

UFA. The design of good mechanisms for defining types depending on types of thunks of effectful computations is planned to be a central theme in our future work.

The former point of view, however, is more difficult to generalise without dependent Kleisli extensions, it would seem at first sight. Indeed, if our basic building blocks are dependent effectful functions $\vdash f : \Pi^{-\circ}_{F(x:\mathbb{N})} F\mathsf{ListOfLen}(x)$, we want to be able to compose them with each other, at the very least. In particular, we want to be able to precompose f with some effectful function $\vdash h : \mathbb{N} \;{}_F\!\multimap F\mathbb{N}$. To do this, however, we precisely need a sequencing principle for dependent effectful functions, or a principle of dependent Kleisli extensions. As it turns out, compositionality in this paradigm can be restored to a satisfying extent by considering dCBPV- with $\Sigma^{\otimes}_{F(-)}$-types, a much less intrusive and more well-behaved extension than dependent Kleisli extensions. We discuss this in section 5.6.

On the whole, we are inclined to view dependent Kleisli extensions as technical devices that were important to study for theoretical reasons, but which may not be very suitable for practical implementations of dCBPV. The extra complexity they introduce into the implementation of a type checker for may not be justified. Therefore, in the rest of this chapter, we focus on dCBPV- and add extra type formers to it to increase its expressive power.

5.6 Dependent Enriched Effect Calculus and More Connectives

In this section, we show how to increase the power of dCBPV- by extending it with Π-, $\Sigma^{\otimes}_{F(-)}$- and $\overset{U}{\multimap}$-types. First we motivate why we might want to include them in our calculus. Next, we show that they are unproblematic from the points of view of categorical semantics, concrete models and operational semantics.

Levy did not include function type formers for value types in his CBPV as he was mainly interested in (the CBV and CBN translations for) effectful programs, for which they are unnecessary. We, however, are also interested in pure proofs of universally quantified formulas. For those purposes, value function types are of crucial importance. This leads us to consider Π-types.

As discussed in section 5.5.2, it is not as important as one might think to be able to substitute effectful computations into dependent functions. However, it might sometimes still be practically convenient. We would like to suggest that $\Sigma^{\otimes}_{F(-)}$-types give an alternative, more lightweight method of achieving this compared to dependent Kleisli extensions.

Recall that, given a dependent function $\Gamma, x : A \vdash M : B$ in pure type theory, we can transform it into a simple function $\Gamma, x : A \vdash \langle x, M\rangle : \Sigma_{x:A} B$ by viewing it as a section of $\Gamma, z : \Sigma_{x:A} B \vdash \mathsf{fst}\ (z) : A$. Precomposition with $\Gamma, y : C \vdash N : A$ then gives $\Gamma, y : C \vdash \langle N, M[N/x]\rangle : \Sigma_{x:A} B$. We can employ a similar trick to get around the effectful composition of certain dependent functions in bare dCBPV- already. Indeed, we can represent any effectful dependent function $\Gamma, x : A; \cdot \vdash M : FA'$ as an effectful simple function $\Gamma, x : A; \cdot \vdash M\ \mathsf{to}\ z\ \mathsf{in\ return}\ \langle x, z\rangle : F\Sigma_{x:A} A'$. In this representation, we can use usual simple sequencing of effectful computations to achieve effectful precomposition: given $\Gamma, y : C; \cdot \vdash N : FA$, we have the effectful composition $\Gamma, y : C; \cdot \vdash N\ \mathsf{to}\ z\ \mathsf{in}\ M\ \mathsf{to}\ z\ \mathsf{in\ return}\ \langle x, z\rangle : F\Sigma_{x:A} A'$ without using dependent Kleisli extensions.

We are in trouble, however, if M is of the more general form $\Gamma, x : A; \cdot \vdash M : \underline{B}$. In order to repeat the trick above, we introduce the type $\Sigma^{\otimes}_{F(x:A)} \underline{B}$ to generalise $F\Sigma_{x:A} A' \cong \Sigma^{\otimes}_{F(x:A)} FA'$. This lets us define a simply typed effectful function $\Gamma, x : A; \cdot \vdash \mathsf{return}\ x \otimes M : \Sigma^{\otimes}_{F(x:A)} \underline{B}$ out of M and therefore a precomposition $\Gamma, y : C; \cdot \vdash N\ \mathsf{to}\ x\ \mathsf{in\ return}\ x \otimes M : \Sigma^{\otimes}_{F(x:A)} \underline{B}$. The problem with sequencing an effectful computation N into a dependent function M was, essentially, that we do not know what fibre of the return type $\underline{B}$ the result would land in. Indeed, N may, for instance, exhibit non-determinism or use state. $\Sigma^{\otimes}_{F(-)}$ solves this problem by bundling all fibres together and saying that we are not interested in the particular fibre it lands is, as long as there is one.

Finally, to reason about effectful programs and their evaluation, it can be very useful to include a type not just of arbitrary functions, but also a type of homomorphisms or stacks. While it is well-known that the sets of homomorphisms for a commutative monad T on a cartesian closed category admit a natural T-algebra

$$\frac{\Gamma, x : A \vdash A' \ \mathsf{vtype}}{\Gamma \vdash \Pi_{x:A} A' \ \mathsf{vtype}} \qquad \frac{\Gamma, x : A \vdash V : A'}{\Gamma \vdash \lambda_x V : \Pi_{x:A} A'} \qquad \frac{\Gamma \vdash V : A \qquad \Gamma; \Delta \vdash W : \Pi_{x:A} A'}{\Gamma; \Delta \vdash V \text{`} W : A'[V/x]}$$

$$\frac{\Gamma, x : A \vdash \underline{B} \ \mathsf{ctype}}{\Gamma \vdash \Sigma^{\otimes}_{F(x:A)} \underline{B} \ \mathsf{ctype}}$$

$$\frac{\Gamma \vdash V : A \qquad \Gamma; \Delta \vdash K : \underline{B}[V/x]}{\Gamma; \Delta \vdash \mathsf{return}\ V \otimes K : \Sigma^{\otimes}_{F(x:A)} \underline{B}} \qquad \frac{\Gamma, x : A; \mathsf{nil} \ : \underline{B} \vdash K : \underline{C} \qquad \Gamma \vdash \underline{C} \ \mathsf{ctype} \qquad \Gamma; \Delta \vdash L : \Sigma^{\otimes}_{F(x:A)} \underline{B}}{\Gamma; \Delta \vdash L \ \mathsf{to}\ \mathsf{return}\ x \otimes \mathsf{nil} \ \mathsf{in}\ K : \underline{C}}$$

$$\frac{\Gamma \vdash \underline{B} \ \mathsf{ctype} \qquad \Gamma \vdash \underline{C} \ \mathsf{ctype}}{\Gamma \vdash \underline{B} \overset{U}{\multimap} \underline{C} \ \mathsf{vtype}}$$

$$\frac{\Gamma; \mathsf{nil} \ : \underline{B} \vdash K : \underline{C}}{\Gamma \vdash \lambda_{\mathsf{nil}} K : \underline{B} \overset{U}{\multimap} \underline{C}} \qquad \frac{\Gamma \vdash V : \underline{B} \overset{U}{\multimap} \underline{C} \qquad \Gamma; \Delta \vdash K : \underline{B}}{\Gamma; \Delta \vdash K \text{`} V : \underline{C}}$$

Figure 5.12: Rules for forming Π-, $\Sigma^{\otimes}_{F(-)}$- and $\overset{U}{\multimap}$-types and their terms.

$$V \text{`} \lambda_x W = W[V/x] \qquad V \overset{\#x}{=} \lambda_x x \text{`} V$$

$$\mathsf{return}\ V \otimes K \ \mathsf{to}\ \mathsf{return}\ x \otimes \mathsf{nil} \ \mathsf{in}\ L = L[V/x, K/\mathsf{nil}\,] \qquad K[L/\mathsf{nil}_{\,1}] \overset{\#x,\mathsf{nil}_{\,2}}{=} K\ \mathsf{to}\ \mathsf{return}\ x \otimes \mathsf{nil}_{\,2}\ \mathsf{in}\ L[\mathsf{return}\ x \otimes \mathsf{nil}_{\,2}/\mathsf{nil}_{\,1}]$$

$$K \text{`} \lambda_{\mathsf{nil}} L = L[K/\mathsf{nil}\,] \qquad K \overset{\#\mathsf{nil}}{=} \lambda_{\mathsf{nil}}\, \mathsf{nil} \text{`} K$$

Figure 5.13: Equations we impose for the terms of Π-, $\Sigma^{\otimes}_{F(-)}$- and $\overset{U}{\multimap}$-types.

structure themselves [48] (leading us to models of linear logic), it should be familiar from the theory of monoids that such an algebra structure might not be available for non-commutative monads [52]. This shows that, in general, for non-commutative effects, we cannot expect the type of homomorphisms/stacks from $\underline{B}$ to $\underline{C}$ to be a computation type itself. Luckily, we can often interpret it as a value type $\underline{B} \overset{U}{\multimap} \underline{C}$.

We include type and term forming rules for Π-, $\Sigma^{\otimes}_{F(-)}$- and $\overset{U}{\multimap}$-types in figure 5.12, their equations in figure 5.13 and their operational semantics in figure 5.14. We then see that the results on the categorical semantics, concrete models and operational semantics of dCBPV- smoothly extend to these connectives.

Theorem 5.6.1 (Categorical Semantics)**.** *A dCBPV- model $F \dashv U : \mathcal{C} \leftrightarrows \mathcal{D}$ supports*

- Π*-types iff we have* Π*-types in* $\mathcal{C}$*;*
- $\Sigma^{\otimes}_{F(-)}$*-types iff we have* $\Sigma^{\otimes}_{F(-)}$*-types in* $\mathcal{D}$ *in the sense of having left adjoint functors* $\Sigma^{\otimes}_{F(A')} \dashv \mathcal{D}(\mathbf{p}_{A,A'})$ *satisfying the left Beck-Chevalley condition for*

Transitions

M to return $x \otimes \mathsf{nil}$ in L	,	K	$\rightsquigarrow$	M	,	$[\cdot]$ to return $x \otimes \mathsf{nil}$ in $L :: K$
return $V_{!\mathsf{nf}} \otimes M$	,	K	$\rightsquigarrow$	return $V_{\mathsf{nf}} \otimes M$	,	K
return $V_{\mathsf{nf}} \otimes M$	,	$[\cdot]$ to return $x \otimes \mathsf{nil}$ in $L :: K$	$\rightsquigarrow$	$L[V_{\mathsf{nf}}/x, M/\mathsf{nil}\,]$	,	K
M`$V_{!\mathsf{nf}}$	,	K	$\rightsquigarrow$	M`V_{nf}	,	K
M`$\lambda_{\mathsf{nil}}\, L$	,	K	$\rightsquigarrow$	$L[M/\mathsf{nil}\,]$	,	K

Terminal Configurations

return $V_{\mathsf{nf}} \otimes M$	,	nil
M`return $V_{\mathsf{nf}}^{x'}$	,	K

Figure 5.14: The additional transitions and terminal configurations that specify the operational behaviour of terms of Π-, $\Sigma^{\otimes}_{F(-)}$- and $\overset{U}{\multimap}$-types. Here, we use the abbreviation $[\cdot]$ to return $x \otimes \mathsf{nil}$ in $L :: K$ for let nil $_1$ be nil $_2$ to return $x \otimes \mathsf{nil}$ $_3$ in L in K. Note that the transitions for Π-types simply are contained in the (β) normalization rules of values.

p-*squares;*

- $\overset{U}{\multimap}$*-types iff we have objects* $B \overset{U}{\multimap} C$ *in* $\mathcal{C}$ *that are stable under change of base in the sense that* $(B \overset{U}{\multimap} C)\{f\} \cong B(\{f\} \overset{U}{\multimap} C\{f\}$ *such that we have natural bijections*

$$\mathcal{D}(\Gamma)(B, C) \cong \mathcal{C}(\Gamma)(1, B \overset{U}{\multimap} C).$$

This semantics is both sound and complete in the usual sense of categorical semantics, leading to a 1-1 correspondence between models and theories supporting the appropriate connectives.

Proof.

- This is a standard result in the semantics of dependent type theory [28], seeing that the value judgements form an ordinary (cartesian) dependent type theory.

- This is precisely analogous to the situation in linear dependent type theory of chapter 3. The introduction rule, by definition, corresponds to a natural transformation

$$\Sigma_{V \in \mathcal{C}(\Gamma)(1,A)} \mathcal{D}(\Gamma)(\Delta, B\{\langle \mathsf{id}_\Gamma, V\rangle\}) \longrightarrow \mathcal{D}(\Gamma)(\Delta, \Sigma^{\otimes}_{F(A)} B),$$

which can be equivalently represented, by taking $V = \mathbf{v}_{\Gamma,A}$ for the first argument and id_B for the second, as another natural transformation

$$\mathcal{D}(\Gamma.A)(\Delta, B) \longrightarrow \mathcal{D}(\Gamma.A)(\Delta, \Sigma^{\otimes}_{F(A)} B\{\mathbf{p}_{\Gamma,A}\}).$$

This, by naturality in Δ is precisely determined by the image of id_B which is an element of

$$\mathcal{D}(\Gamma.A)(B, \Sigma^{\otimes}_{F(A)} B\{\mathbf{p}_{\Gamma,A}\}).$$

By a simple variation on the Yoneda lemma, we see that this is the same as specifying a natural transformation

$$\mathcal{D}(\Gamma)(\Sigma^{\otimes}_{F(A)} B, C) \longrightarrow \mathcal{D}(\Gamma.A)(B, C\{\mathbf{p}_{\Gamma,A}\}).$$

(This is one of the two defining natural transformations of the adjunction.)

The elemination rule corresponds by definition to a natural transformation

$$\mathcal{D}(\Gamma.A)(B, C\{\mathbf{p}_{\Gamma,A}\}) \times \mathcal{D}(\Gamma)(\Delta, \Sigma^{\otimes}_{F(A)} B) \longrightarrow \mathcal{D}(\Gamma)(\Delta, C),$$

which by naturality in Δ is equivalent to a natural transformation

$$\mathcal{D}(\Gamma.A)(B, C\{\mathbf{p}_{\Gamma,A}\}) \longrightarrow \mathcal{D}(\Gamma)(\Sigma^{\otimes}_{F(A)} B, C)$$

(where we have specialised to $\Delta = \Sigma^{\otimes}_{F(A)} B$ and have substituted $\mathsf{id}_{\Sigma^{\otimes}_{F(A)} B}$ for the second argument). (This is the other defining natural transformation of the adjunction.) The β- and η-rules precisely state that both defining natural transformations of the adjunction are inverse. As usual, the Beck-Chevalley condition corresponds to the compatibility of $\Pi^{\multimap}_{F(-)}$-types with substitution.

- Note that the introduction rule, by definition, corresponds precisely with the natural transformation from left to right in the categorical semantics. The elimination rule by definition corresponds to a natural transformation

$$\mathcal{C}(\Gamma)(1, B \overset{U}{\multimap} C) \times \mathcal{D}(\Gamma)(\Delta, B) \longrightarrow \mathcal{D}(\Gamma)(\Delta, C),$$

which by naturality in Δ is equivalent to a natural transformation

$$\mathcal{C}(\Gamma)(1, B \overset{U}{\multimap} C) \longrightarrow \mathcal{D}(\Gamma)(B, C)$$

(where we have specialised to $\Delta = B$ and have substituted id_B for the second argument). The β- and η-laws precisely translate to these functions being

inverse. Naturality of the bijections corresponds to compatibility of term formers with substitution. Compatibility of the syntactic type formers with substitution corresponds with stability under change of base in the semantics.

□

Let us provide some context for thinking about $\Sigma^{\otimes}_{F(-)}$- and $\overset{U}{\multimap}$-types. As observed by Benton and Wadler [38], linear logic can be seen as the term calculus of stacks for certain commutative effects. The question remained, if more general, possibly non-commutative effects would give rise to a certain kind of generalized, possibly non-commutative linear logic. In particular, the question was if one could define a monoidal-like structure on stacks in a general model of CBPV which generalizes the tensor of linear logic and similarly for the lollipop. A partial positive answer to this was given by the Enriched Effect Calculus (EEC) [112], telling us that any model of simple CBPV fully and faithfully embeds into a model where we have a binary operation $F(-) \otimes -$ (conventionally, somewhat misleadingly, written $!(-) \otimes -$) which takes a value type and a computation type and produces a computation type and a binary operation $- \overset{U}{\multimap} -$ (conventionally written $- \multimap -$) which takes two computation types to a value type. Our notation is chosen to be suggestive as these operations do not generalize the plain linear logic operations $\otimes$ and $\multimap$ but rather the composite connectives $F(-) \otimes (-)$ and $U(- \multimap -)$ that one can define in the LNL calculus [33].

Independently, linear dependent type theory forces a similar operation on us if we wish to extend $- \otimes -$ to a dependent connective [12]. Because types are only allowed to depend on cartesian assumptions and not linear ones, the best we can do is a multiplicative Σ-type $\Sigma^{\otimes}_{F(-)}-$. Seemingly for two very different reasons, the connective $F(-) \otimes -$ seems to be a preferred over $- \otimes -$, if one wants to generalize. We believe this is not a coincidence as the semantics of simply typed CBPV already forces various notions from dependent type theory on us.

In analogy with linear logic, we have the following isomorphisms of types, motivating some of our use of notation.

Theorem 5.6.2 (Type Isomorphisms)**.** *We have type isomorphisms*

$$U\Pi^{\multimap}_{F(x:A)}\underline{B} \cong \Pi_{x:A}U\underline{B} \qquad FA \overset{U}{\multimap} \underline{B} \cong U(A \ {}_F\!\multimap \underline{B}) \qquad \begin{array}{l} F\Sigma_{x:A}A' \cong \Sigma^{\otimes}_{F(x:A)}FA' \\ \Sigma^{\otimes}_{F(x:1)}\underline{B} \cong \underline{B} \\ \Sigma^{\otimes}_{F(x:A)}F1 \cong FA. \end{array}$$

Proof. These are straightforward consequences of the universal properties in the categorical semantics of the various connectives involved, together with their compatibility with substitution. □

Let us say a few words about the interpretation of these connectives on some concrete classes of models.

Theorem 5.6.3 (Concrete Models)**.** *We have the following results on interpreting these connectives in concrete models.*

- *An indexed Eilenberg-Moore $\mathcal{C}^T$ category for an indexed monad T on a model $\mathcal{B}^{op} \xrightarrow{\mathcal{C}} \mathsf{Cat}$ of pure dependent type theory with* 1-, ×-, 0-, +-, Σ-, Id- *and* Π*-types gives a model of dCBPV- with* Π*-types. If $\mathcal{C}$ has indexed equalisers, we can interpret $\overset{U}{\multimap}$-types and if $\mathcal{C}^T$ has indexed reflexive coequalisers, then we can interpret $\Sigma^{\otimes}_{F(-)}$-types. All these conditions are satisfied for $\mathcal{C} = \mathsf{Fam}(\mathsf{Set})$ and T any finitary indexed monad like one of the usual reader, writer, state or exceptions monads.*

- *A model for the dependently typed LNL calculus with sum types, in the style of section 3.3 gives a model of dCBPV- with* Π*, $\Sigma^{\otimes}_{F(-)}$- and $\overset{U}{\multimap}$-types. The dependent LNL calculus model of continuous families of predomains and domains is a specific example of this (see section 3.5.4).*

Proof.

- The interpretation of Π-types is obvious.

  Let us write $F \dashv U$ for the Eilenberg-Moore adjunction inducing T. As $T = UF$ is an indexed monad, recall that we have a dependent strength $\Sigma_A UFUk \xrightarrow{s_{A,Uk}} UF\Sigma_A Uk$. We note that we have a reflexive fork

$$F(\Sigma_A Uk) \xrightarrow{F\Sigma_A \eta_{Uk}} F(\Sigma_A UFUk) \underset{F(s_{A,Uk});\epsilon_{F\Sigma_A Uk}}{\overset{F\Sigma_A k}{\rightrightarrows}} F(\Sigma_A Uk).$$

Now, we can define $\Sigma^{\otimes}_{F(A)}k$ as the coequaliser of the reflexive pair. Note that a morphism $k \xrightarrow{\phi} l$ gives a natural transformation between the coequaliser diagrams for $\Sigma^{\otimes}_{F(A)}k$ and $\Sigma^{\otimes}_{F(A)}l$, or equivalently, a morphism $\Sigma^{\otimes}_{F(A)}k \xrightarrow{\Sigma^{\otimes}_{F(A)}\phi} \Sigma^{\otimes}_{F(A)}l$. This is easily seen to make $\Sigma^{\otimes}_{F(A)}-$ into a functor. Let us convey to the reader how we arrived at this definition: noting that $F(\Sigma_A A') \cong \Sigma^{\otimes}_{F(A)}FA'$ if we can prove that $\Sigma^{\otimes}_{F(A)}- \dashv -\{\mathbf{p}_{\Gamma,A}\}$, we are defining $\Sigma^{\otimes}_{F(A)}k$ above as the coequaliser of

$$\Sigma^{\otimes}_{F(A)}FUFUk \underset{\Sigma^{\otimes}_{F(A)}\epsilon_{FUk}}{\overset{\Sigma^{\otimes}_{F(A)}Fk}{\rightrightarrows}} \Sigma^{\otimes}_{F(A)}FUk,$$

showing that we are simply computing a $\mathcal{B}$-indexed variation of Linton's construction of Set-indexed coproducts of algebras [49]. We now verify that indeed $\Sigma^{\otimes}_{F(A)}- \dashv \{\mathbf{p}_{\Gamma,A}\}$. We can easily[8] see that we have natural bijections between the following morphisms

$$\begin{array}{lll}
\phi \in \mathcal{C}(\Gamma)^T(\Sigma^{\otimes}_{F(A)}k, l) & & \\
\phi' \in \mathcal{C}(\Gamma)^T(F(\Sigma_A Uk), l) & \text{s.t.} & F(\Sigma_A k); \phi' = F(s_{A,Uk}); \epsilon_{F\Sigma_A Uk}; \phi' \\
\psi \in \mathcal{C}(\Gamma)(\Sigma_A Uk, Ul) & \text{s.t.} & F(\Sigma_A k); F(\psi); \epsilon_l = F(s_{A,Uk}); \epsilon_{F\Sigma_A Uk}; F(\psi); \epsilon_l \\
\psi \in \mathcal{C}(\Gamma)(\Sigma_A Uk, Ul) & \text{s.t.} & F(\Sigma_A k; \psi); \epsilon_l = F(s_{A,Uk}; T\psi; l); \epsilon_l \\
\psi \in \mathcal{C}(\Gamma)(\Sigma_A Uk, Ul) & \text{s.t.} & \Sigma_A k; \psi = s_{A,Uk}; T\psi; l \\
\psi' \in \mathcal{C}(\Gamma.A)(Uk, Ul\{\mathbf{p}_{\Gamma,A}\}) & \text{s.t.} & \Sigma_A k; \Sigma_A \psi'; \mathsf{snd} = s_{A,Uk}; T(\Sigma_A \psi'; \mathsf{snd}); l \\
\psi' \in \mathcal{C}(\Gamma.A)(Uk, Ul\{\mathbf{p}_{\Gamma,A}\}) & \text{s.t.} & \Sigma_A(k; \psi'); \mathsf{snd} = \Sigma_A T\psi'; s_{A,Ul\{\mathbf{p}_{\Gamma,A}\}}; T\mathsf{snd}; l \\
\psi' \in \mathcal{C}(\Gamma.A)(Uk, Ul\{\mathbf{p}_{\Gamma,A}\}) & \text{s.t.} & \Sigma_A(k; \psi'); \mathsf{snd} = \Sigma_A T\psi'; \mathsf{snd}; l \\
\psi' \in \mathcal{C}(\Gamma.A)(Uk, Ul\{\mathbf{p}_{\Gamma,A}\}) & \text{s.t.} & \Sigma_A(k; \psi'); \mathsf{snd} = \Sigma_A(T(\psi'); l\{\mathbf{p}_{\Gamma,A}\}); \mathsf{snd} \\
\psi' \in \mathcal{C}(\Gamma.A)(Uk, Ul\{\mathbf{p}_{\Gamma,A}\}) & \text{s.t.} & k; \psi' = T(\psi'); (l\{\mathbf{p}_{\Gamma,A}\}) \\
\psi'' \in \mathcal{C}(\Gamma.A)^T(k, l\{\mathbf{p}_{\Gamma,A}\}) & &
\end{array}$$

Note that a sufficient condition to have reflexive coequalisers in $\mathcal{C}^T$ is to have them in $\mathcal{C}$ and to have T preserve them. For a cartesian closed category $\mathcal{C}$, a broad class of monads that preserve reflexive coequalisers are those arising from a finitary algebraic theory [51] (section D5.3).

[8] These bijections can, in order, be motivated by the universal property of the coequaliser defining $\Sigma^{\otimes}_{F(A)}k$, the homset bijection of $F \dashv U$, naturality of ϵ and the observation that $\epsilon_l = l$, the homset bijection of $F \dashv U$, the homset bijection of $\Sigma_A \dashv -\{\mathbf{p}_{\Gamma,A}\}$, the naturality of s and functoriality of T, a triangle identity for $\Sigma_A \dashv -\{\mathbf{p}_{\Gamma,A}\}$, the naturality of snd, the homset bijection of $\Sigma_A \dashv -\{\mathbf{p}_{\Gamma,A}\}$, and, finally, the definition of a homorphism from k to l.

Note that $\overset{U}{\multimap}$-types can be constructed exactly as in the proof of theorem 2.3.3. We cannot usually construct an appropriate algebra structure on $k \overset{U}{\multimap} l$ (unless T is a commutative monad).

- We interpret $B \overset{U}{\multimap} C$ as $U(B \multimap C)$. The rest should be obvious.

□

As an example, consider the writing monad $- \times M$ on Set (which, as we have seen, does not admit dependent Kleisli extensions). We can note that its Eilenberg-Moore category is equivalent to the indexed category $\mathsf{Fam}(\mathsf{Set}^M)$ of families of M-modules (also known as M-sets or sets with an M-action). Being a presheaf category (if we consider M as a one-object category), this is a topos and, in particular, we can construct $\Sigma^{\otimes}_{F(-)}$-types. If we calculate the coequaliser above (as colimits are computed pointwise), we find that $\Sigma^{\otimes}_{F(A)}k(s)$ has as carrier the quotient of $(\Sigma_A Uk)(s) \times M$ by the relation $(a, b, m \cdot m') \sim (a, b \cdot m, m')$ and the algebra structure that the quotient induces starting from the free one. Similarly, we have a (non-symmetric) indexed premonoidal structure $\otimes$ on $\mathsf{Fam}(\mathsf{Set}^M)$, where the carrier of $(k \otimes l)(s)$ is obtained by the quotient of $(Uk \times Ul)(s) \times M$ by the transitive closure of $(a, b, m \cdot m' \cdot m'') \sim (a \cdot m, b \cdot m', m'')$ and the algebra structure is obtained from the free one under the quotient. We can easily compute that $k \overset{U}{\multimap} l$ consists of the set of equivariant functions from k to l. Note that $k \overset{U}{\multimap} l$ does not admit an appropriate algebra structure by [52]. In this example, obviously, we have Π-types as usual sets of dependent functions.

Next, we consider the operational behaviour of terms of these new types. It turns out to be entirely well-behaved.

Theorem 5.6.4 (Determinism, Strong Normalization, Subject Reduction)**.** *Π-, $\Sigma^{\otimes}_{F(-)}$- and $\overset{U}{\multimap}$-types do not alter any of the determinism, strong normalization or subject reduction results for dCBPV- of theorem 5.2.10.*

Proof. For Π-types, we note that we can still rely on the subject reduction and strong normalization proofs for β-reductions in pure dependent type theory of [68].

Determinism and strong normalization of reductions for $\Sigma^{\otimes}_{F(-)}$- and $\overset{U}{\multimap}$-types is no different than for the other type formers. We verify subject reduction. In both cases it is clear from the subject reduction results for pure type theory that the transitions involving normalization of values satisfy subject reduction, so we focus on the remaining two transitions.

Let us assume that we start with a well-typed configuration $\Gamma; \cdot \vdash \mathsf{return}\ V_{\mathsf{nf}} \otimes M : \Sigma^{\otimes}_{F(A)} \underline{B}$, $\Gamma; \mathsf{nil} : \Sigma^{\otimes}_{F(A)} \underline{B} \vdash [\cdot]\ \mathsf{to}\ \mathsf{return}\ x \otimes \mathsf{nil}\ \mathsf{in}\ L :: K : \underline{C}$. Then, on the one hand, by inversion on the introduction rule for $\Sigma^{\otimes}_{F(-)}$-types, we have that $\Gamma \vdash V_{\mathsf{nf}} : A$ and $\Gamma; \cdot \vdash M : \underline{B}[V_{\mathsf{nf}}/x]$ (where $\Gamma, x : A \vdash \underline{B}\ \ \mathsf{ctype}$). On the other hand, noting that $[\cdot]\ \mathsf{to}\ \mathsf{return}\ x \otimes \mathsf{nil}\ \mathsf{in}\ L :: K$ is an abbreviation for $\mathsf{let}\ \mathsf{nil}_1\ \mathsf{be}\ \mathsf{nil}_2\ \mathsf{to}\ \mathsf{return}\ x \otimes \mathsf{nil}_3\ \mathsf{in}\ L\ \mathsf{in}\ K$ by inversion on the rules for $\mathsf{let}\ \mathsf{nil}_1\ \mathsf{be}\ \mathsf{in}$ and the elimination rule for $\Sigma^{\otimes}_{F(-)}$-types, we have that $\Gamma, x : A; \mathsf{nil} : \underline{B} \vdash L : \underline{D}$ for some $\Gamma \vdash \underline{D}\ \ \mathsf{ctype}$ and that $\Gamma; \mathsf{nil} : \underline{D} \vdash K : \underline{C}$. Therefore, because of the substitution property, we have that $\Gamma; \cdot \vdash L[V/x, M/\mathsf{nil}\,] : \underline{D}$. Hence, $L[V/x, M/\mathsf{nil}\,], K$ is a well-typed configuration.

Let us assume that we start with a well-typed configuration $\Gamma; \cdot \vdash M \text{`} \lambda_{\mathsf{nil}}\, L : \underline{B}$, $\Gamma; \mathsf{nil} : \underline{B} \vdash K : \underline{C}$. Then, inversion on the elimination and introduction rules for $\overset{U}{\multimap}$-types gives us that $\Gamma; \cdot M : \underline{D}$ and that $\Gamma; \mathsf{nil} : \underline{D} \vdash L : \underline{B}$ for some $\Gamma \vdash \underline{D}\ \ \mathsf{ctype}$. Therefore, the substitution property gives us that $\Gamma; \cdot \vdash L[M/\mathsf{nil}\,] : \underline{B}$, which means that $L[M/\mathsf{nil}\,], K$ is a well-typed configuration. □

Remark 5.6.5 ($\mathsf{Id}^{\otimes}_{F(-)}$-types)**.** *We could have included rules for $\mathsf{Id}^{\otimes}$-types, similarly to chapter 3, connectives such that $F\mathsf{Id}_A \cong \mathsf{Id}^{\otimes}_{FA}$. While such connectives seem interesting from the point of view of linear logic, their use in CBPV is less clear. We are really interested in pure proofs of equality, rather than effectful ones (as, for instance, divergence can trivially inhabit any type $\mathsf{Id}^{\otimes}_{FA}$), so the use of $\mathsf{Id}^{\otimes}_{F(-)}$-types from the point of view of program verification is unclear.*

Remark 5.6.6 (Universes)**.** *We have so far not considered higher-order quantification, which can be expressed in dependent type theory through universes, or types whose terms are (codes for) types. Universes (à la Tarski) arise as a special*

case of induction-recursion, a generalisation of more traditional, weaker induction schemes [121]. As a rule of thumb, inductive-recursive families are more like an inductive than a coinductive construction, hence one would expect them to arise most naturally as value types. In the particular case of universes, we would expect separate universes $\mathcal{U}_v$ and $\mathcal{U}_c$ (both of which are value types) to classify value and computation types respectively. Like in pure type theory, one could include rules like

$$\frac{\vdash \Gamma \ \mathsf{ctxt}}{\Gamma \vdash 1 : \mathcal{U}_v}$$

to build values of the universes which code for types and rules like

$$\frac{}{x : \mathcal{U}_v \vdash \mathsf{El}_v(x) \ \ \mathsf{vtype}}$$

to make types out of codes.

Very recently, [122] has further pursued a system resembling dCBPV- extended with universes.

5.7 Comparison with HTT

We make a few observations on the relationship between dCBPV and an existing successful framework for certified effectful programming which is also based on dependent type theory: Hoare Type Theory (HTT) [110] (implemented in Ynot [123]).

Regarding the motivation behind both systems, HTT seems to have been developed from the start with the practical syntactic goal in mind of a language for verifying effectful programs. By contrast, dCBPV arose almost entirely from semantic considerations. In particular, dCBPV was motivated by the study of models of DTT which naturally model effects, like its domain semantics [83] and game semantics [14], the question if dDILL [13] could be interpreted as a DTT with commutative effects and the existing categorical semantics of CBPV which strongly suggests a dependently typed generalization [37].

Regarding their implementation, HTT expresses a property ϕ of an effectful program V of type A by saying that V inhabits a type $T_\phi A$, where T_ϕ are monads which are indexed by formulae ϕ formed using an (external) separation logic. dCBPV

sticks closer to the Curry-Howard correspondence in its formulation of properties ϕ of an effectful program M of type FA: they are types ϕ depending on thunks of type UFA and into which we can, in particular, substitute $\mathsf{thunk}\ M$ to see if we can construct an inhabitant witnessing the truth of $\phi(\mathsf{thunk}\ M)$.

I may not have gone where I intended to go, but I think I have ended up where I intended to be.

— Douglas Adams

6 Conclusions and Future Work

6.1 Conclusions

In this thesis, we have examined the relationship between programming languages and formal logic, specifically the combination of computational effects with dependent types. We did this by analysing dependent types from three separate but related points of view on effects:

1. linear logic, which, as we argued, represents a type system for computations exhibiting commutative effects;

2. game semantics, a setting to provide, in a unified way, models with strong completeness properties for a wide range of effectful type theories;

3. CBPV, an elegant framework for representing type theories with a wide range of effects with a fine-grained evaluation strategy that encompasses both traditional CBN and CBV.

We believe these three perspectives often complement and sometimes reinforce each other.

Firstly, we constructed a dependently typed version of DILL, and showed it admits an elegant categorical semantics as well as a wide range of concrete

models. Secondly, We constructed a CBN game semantics for dependent type theory, validated it by showing that it exhibits the usual completeness properties one expects of a game semantics and showed that it can be generalised to model effectful dependent type theory by relaxing the conditions on strategies. Thirdly, we studied a dependently typed version of CBPV and showed it has a simple categorical semantics, admits classes of models arising from both linear dependent type theory and indexed monads and that it has a well-behaved operational semantics.

We learned that one principal source of the tension between type dependency and effects is the phenomenon that effectful computations are dynamic (in the sense that their evaluation can break equality) while we use types to provide static guarantees about programs. If types depend on effectful computations, therefore, they are at risk of losing their static nature.

Working in an adjunction language like CBPV (or the LNL calculus, for commutative effects) which distinguishes between (dynamic) computations and their thunks (which are static values), we can use types depending on thunks of effectful programs to express complex properties that we might want to verify for effectful programs. Analogously, in linear logic, while types depending on linear terms are problematic, (linear) types depending on cartesian terms are entirely harmless.

If we impose this restriction, one consequence is that we do not have CBV and CBN translations of type theory with unrestricted effects into CBPV (or dependently typed Girard translations, in the case of linear logic). Our view is that this is not at all a problem. Indeed, dependently typed languages with unrestricted effects are of limited value, anyway, as effects render the language inconsistent as a logic, while the prime purpose of dependent types is to prove properties about programs. Moreover, the substitution of effectful computations in types introduces various technical challenges, as witnessed by our effectful game semantics for dependent type theory. One effect that could possibly be interesting to include in a dependent type theory in unrestricted fashion is that of non-local control operators because of its close relation to the classical principle of double negation elimination. However, it is already known that a constructive classical dependent

type theory (i.e. a dependent type theory with call/cc at each type) is degenerate in the sense that it equates all programs [101] (propositionally). Therefore, any language that combines dependent types and effects in a meaningful way needs to have a mechanism for controlling the occurrence of effects. We hope to have demonstrated that modalities on the type system are an excellent tool for this purpose, in particular half-modalities (or adjunctions).

If one wants to obtain full CBV and CBN translations for dependent type theory with unrestricted effects, we showed that one needs to include Kleisli extensions for dependent functions in dCBPV. We have seen that these are not always supported in concrete models and can lead to problems with subject reduction in the operational semantics[1]. Especially given that a similar effect can be achieved with the entirely unproblematic $\Sigma^{\otimes}_{F(-)}$-types, we believe such dependent Kleisli extensions are not a desirable feature of a dependently typed effectful language. Similar technical challenges arise with dependent projection products (or their linear logic equivalent, additive Σ-types). Indeed, the fact that these connectives were not naturally supported in categories of games and strategies is one of the prime reasons that a game semantics for dependent type theory had so far been absent and, more generally, that models of dependent type theory in computational settings of categories of cofree !-coalgebras had been missing.

Summarising, dependent types and computational effects form a delicate though not impossible combination. We hope to have demonstrated that robust systems can be achieved, as long as one is prepared to restrict type dependency to static values and exclude dependency on dynamic computations.

[1] We would like to point out that dependent types are indeed combined with certain unrestricted effects, like recursion, in practice: Agda and Idris support unrestricted recursion and perform a separate optional termination check. It is no coincidence that recursion is precisely one of the effects for which dependent Kleisli extensions are well-defined. Indeed, its computations are not really dynamic in the sense that their evaluation respects equality. Therefore, they are much easier to combine with type dependency.

6.2 Future Work

We describe some interesting directions for future research, suggested by the work presented in this thesis.

6.2.1 Linear Dependent Functions

McBride [124] presented a type system in which linear types depend on linear assumptions and with a type $\Pi^{\multimap}_{x:A} B$ of dependent functions from A to B that use x exactly once (and in which types are allowed to refer to identifiers arbitrarily often). His solution relies on an unorthodox new view on linear logic in which we do not have separate classes of cartesian and linear types, but only one kind of type, achieving the linearity through annotation of identifier declarations with a count. We would like to analyse, using semantic methods, how this system relates to our work.

6.2.2 Stable Homotopy as Effectful Homotopy?

An indexed category of spectra up to homotopy, indexed over topological spaces, has been studied in e.g. [74, 76], as a setting for stable homotopy theory. We can interpret this as a model of the dLNL calculus. It has been shown to admit I-, $\otimes$-, $\multimap$-, and $\Sigma^{\otimes}_F$-types. The natural candidate for a comprehension adjunction, here, is that between the infinite suspension spectrum and the infinite loop space: $F \dashv U \;=\; \Sigma^\infty \dashv \Omega^\infty$. What is particularly fascinating is that the corresponding monad $T = \Omega^\infty\Sigma^\infty$ seems to be very closely related (if not identical) to an important homotopical construction known as the Goodwillie exponential [125]. This raises the question whether one could phrase stable homotopy theory as an (commutative) effectful version of homotopy type theory and, if so, what the computational interpretation of the Goodwillie calculus in terms of effects should be.

6.2.3 Dependently Typed Quantum Programming?

Another fascinating possibility is that of models related to quantum mechanics. Non-dependent linear type theory has found very interesting interpretations in quantum computation (see e.g. [126]). The question rises if the extension to

dependent linear types has a natural counterpart in physics. In [77], it was recently sketched how linear dependent types can serve as a language to talk about quantum field theory and quantisation. On a related note, one could well imagine using an extension of the dLNL calculus as a type system for a language in which we both have (cartesian) types for classical data and (linear) types for quantum data which may depend on the former. Such a precise type system may be useful for catching bugs in quantum programs.

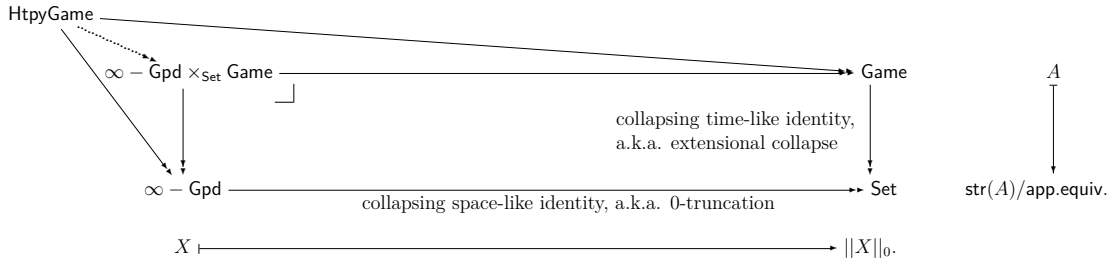
6.2.4 Extending CBN Game Semantics for Dependent Types

We see a few interesting directions for continuing the work we started in chapter 4. One obvious continuation would be to try to extend the (full and faithful) completeness proof to the complete type hierarchy of $\mathsf{DTT}_{\mathsf{CBN}}$. A next step is to study the interpretation of more general inductive families [23, 127] and inductive-recursive definitions (of which type universes are an obviously interesting example) [121]. Such a study of universes should also lead to a more intensional notion of a dependent game as a kind of strategy.

On a different note, it would be desirable to find an alternative, less technical presentation of a suitable category of !-coalgebras extending $\mathsf{Ctxt}(\mathsf{Game}_!)$ which also models dependent type theory (cf. section 3.4).

Finally, note that our games model of dependent types has identity types that are intensional in orthogonal ways compared to the homotopy semantics [88]. In our case, all non-trivial propositional identities concern a kind of homotopy in the time direction (applicative equivalence of functions), rather than in the space direction, in the sense that our ground types are discrete and we accumulate non-trivial propositional identities if we ascend the function hierarchy. By contrast, in the homotopy semantics of dependent types, all non-trivial propositional identities exist on ground types and we do not acquire any non-trivial identifications of functions (beyond their pointwise identity). We propose to pursue a notion of what one might call a category of **homotopy games**, which should factor over the pullback of the

spatial and temporal extensional collapse of the two models,



That is, we are looking for a setting to model DTT which combines the possibility of non-trivial propositional identity on ground types of the (∞-)groupoid model of DTT [88] with the failure of function extensionality of the game semantics. We hope this would not only result in a satisfactory game semantics for quotient types and higher inductive types, but would also give deeper insights into the subtle shades of intensionality that arise in dependent type theory, by cleanly separating out the time-like and space-like aspects of propositional identity.

6.2.5 Game Semantics for dCBPV

The practical challenge of constructing a CBN game semantics for dependent type theory was important for us in developing our understanding of the interaction between effects and type dependency. Moreover, we hope that it can be a useful addition to the large family of game semantics for various logics and programming languages.

However, in hindsight, we believe that dependently typed languages with unrestricted effects such as those modelled by our CBN game semantics are not the most interesting combination of dependent types and effects (though also not uninteresting). What would be really interesting is to construct a game semantics for dCBPV in the style of [30, 37]. We hope it could both be simpler are more practically relevant. It should be emphasized though that the experience of working with the CBN game semantics of dependent types was necessary for us to reach this insight.

6.2.6 Certified Real-World Programming in dCBPV-

Cervesato and Pfenning pioneered the use of systems combining dependent types and linearity to reason about effectful computations in [69]. We hope that our

system dCBPV- can be a step forward for this purpose, through its generalisation to non-commutative effects and the extra expressive power obtained by distinguishing between values and computations, where the value judgement can be seen as a pure logic that be used for reasoning about (thunks of) effectful computations, defined with the computation judgement. It is particularly salient that this distinction allows us to use Id-types, which were painfully absent from Cervesato and Pfenning's system.

In particular, we hope that dCBPV- can serve as an alternative to Hoare Type Theory that sticks closer to the elegance of the Curry-Howard correspondence and that it can be extended to a practical language for both writing and verifying real world effectful code. For that purpose, the next step is to add mechanisms for forming types that express delicate properties of thunks of computations exhibiting specific effects, like properties of the state before and after a computation is run. [128] recently proposed using effect handlers, which have a semantics as monad algebras $TA \xrightarrow{k} A$, as a way of lifting predicates on A to ones on TA. We believe this idea sounds very promising and deserves to be explored further. Another important step would be the implementation of a type checker for the resulting system.

Appendices

Huh?!

— Sylvester Stallone

A Summary for a General Audience

Over the past decades, we – both as individuals and as a society as a whole – have very rapidly become reliant on computer systems, to the point that we trust them with our private data (e.g. mobile phone communications and medical records), our critical resources (e.g. bank transactions), the smooth running of society (e.g. elections, financial markets, and classified government documents) and even our lives (e.g. auto-pilots in air planes, self-driving cars, medical devices and missile detection systems on which decisions whether or not to engage in nuclear war are based). While many people will agree that computers have changed our lives for the better, there have been enough incidents to make us question how much trust we should put into computer systems for critical applications: e.g.[1] false alarms in both US (in 1980) and Soviet (in 1983) missile detection systems which could have easily led to nuclear war, Therac-25 medical radiation therapy devices administering deadly doses of radiation, the destruction of the Ariane 5 rocket (costing $370 million dollar) and regularly uncovered cryptography bugs like Heartbleed which enable hackers to get into critical computer systems (like those of the Democratic National Convention, in the context of the 2016 US election). Moreover, software

[1] Links to news stories on many fascinating software bugs can be found on [129].

bugs are unbelievably expensive, annually costing an estimated $312 billion[2], with software developers spending on average half their time debugging [131].

These bugs are often not introduced into software due to negligence on behalf of the programmer. Rather, we learn from experience that bugs are almost unavoidable when writing software. The human mind is simply rather ill-suited for writing watertight computer software. Many bugs can be found through testing, but, depending on the application, that may not be enough: real world software, particularly concurrent software, can have a space of possible executions that is simply too large to explore through testing. For instance, new critical bugs are found in web-protocols every week despite extensive testing. For the most critical of applications, only a formal machine-checked proof is enough to guarantee the correctness of a piece of code.

Currently, while very suitable programming languages for writing computer software exist as well as good logical frameworks for writing proofs (formal arguments demonstrating the truth of a proposition), there is no single satisfactory system in which we can both write production code and write and check a proof about the correctness of this code. We believe an important reason for the absence of such a system is a lack of fundamental understanding of how real-world programming languages relate to logical frameworks. The aim of this thesis is to improve on the state of the art of such an understanding and, as a consequence, to work towards the dream of having a single elegant language for writing provably correct production code.

It is clear that programs written in very simple programming languages, so-called purely functional languages, are effectively the same thing as mathematical proofs. This idea is called the Curry-Howard correspondence. Similar to how we can organise proofs by the proposition (e.g. A and B implies C or D) whose validity they demonstrate, we can classify computer programs according to their so-called type (e.g. the type of programs that take two inputs, one of type A, one

[2]To give a sense of scale, according to a 2015 U.N. report, it would cost around $267 billion annually to bring the roughly 800 million people world-wide living in extreme poverty up to the World Bank's poverty line immediately [130].

of type B and produce an output which will either be of type C or type D). That is, under the Curry-Howard correspondence, types correspond to propositions in the same way that purely functional programs correspond to proofs. Types can be thought of as expressing properties of programs. Some examples of types are the type of booleans, the type of integers, the type whose elements consist of a pair of a boolean and a string, the type whose elements are either a boolean or an integer, the type of programs that take two integers as input and produce five booleans and the type of programs that take a program from booleans to integers as input and produce an integer as output.

Real world programming languages and useful logical frameworks are not the same thing, however! On the one hand, there are more good computer programs than acceptable proofs. For instance (this is just one of many examples), a computer program can loop indefinitely (like an operating system), but a circular argument is unacceptable as a proof. These extra programs are very useful in practical software development. On the other hand, the most useful logics include more propositions than there are types in real world programming languages. Indeed, while these programming languages include so-called simple types, which correspond to propositions formed by the logical connectives "and", "or" and "implies", dependent types are missing, corresponding to the crucial propositions of the form "$P(x)$ holds for all x" or "$P(x)$ holds for some x". (An example would be the statement "all Buddhists are happy".)

This thesis examines how this gap between programming languages and logical frameworks can be bridged and how a single language can be designed that can serve for writing both real world code as well as formal, machine-checked proofs that this code has the properties that one desires. Before tackling, head on, the question of how effectful programs (programs that do not correspond to proofs) can be given dependent types, we first study the closely related topics of how so-called linear logic can be extended with dependent types and how a game theoretic interpretation can be given to logical frameworks with dependent types.

Linear logic is a logic in which we keep track of how often each assumption is used in a proof: assumptions cannot be copied or discarded freely. Linear logic proofs are closely related to effectful programs. The intuition is that effectful programs could, for instance, make a random choice, meaning that two executing copies of the same program may later cease to be equal.

A formal logic can be equivalently phrased in terms of game theory by interpreting a proposition as a turn-based two-player game (think of it as the game of formal debates about the proposition, analogous to Socratic dialogues) and a proof of that proposition as a certain kind of winning strategy for that game (if we can win any debate about a proposition, it must be true and vice versa). The charm of game semantics is that we can weaken the conditions we put on the strategies we consider to obtain various effectful programs. For instance, partial strategies (in which we do not always have a response to everything our opponent says in a debate, meaning that we do not always win) correspond to programs which may loop for ever and never return an output.

This thesis first presents a dependently typed linear logic and game theoretic interpretation for dependent types. This helps us build an understanding that is useful to, next, present an elegant language which can both serve as an effectful programming language for writing software and as a pure logic to prove properties about the software we write in it. We hope that this work, on the one hand, contributes to a better understanding of the foundations of the disciplines of mathematics and computer science and their relationship and, on the other, ultimately will help us to work towards a world in which one can safely rely on critical computer systems, both as individuals and as a society.

References

[1] Stephen C Kleene. "Origins of recursive function theory". In: **Annals of the History of Computing** 3.1 (1981), pp. 52–67.

[2] Robin Gandy. "The confluence of ideas in 1936". In: **The Universal Turing Machine a Half-Century Survey** (1995), pp. 51–102.

[3] Robert I Soare. "The history and concept of computability". In: **Studies in Logic and the Foundations of Mathematics** 140 (1999), pp. 3–36.

[4] TIOBE Software BV. **TIOBE Index**. September 2016. URL: `http://www.tiobe.com/tiobe-index/`.

[5] Conor McBride. **on termination**. May 2003. URL: `https://mail.haskell.org/pipermail/haskell-cafe/2003-May/004343.html`.

[6] Eugenio Moggi. **Computational lambda-calculus and monads**. 1989, pp. 14–23.

[7] Philip Wadler. "Monads for functional programming". In: **International School on Advanced Functional Programming**. Springer. 1995, pp. 24–52.

[8] P Nick Benton, Gavin M. Bierman, and Valeria CV de Paiva. "Computational types from a logical perspective". In: **Journal of Functional Programming** 8.02 (1998), pp. 177–193.

[9] Timothy G Griffin. "A formulae-as-type notion of control". In: **Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages**. ACM. 1989, pp. 47–58.

[10] Joachim Lambek and Philip J Scott. **Introduction to higher-order categorical logic**. Vol. 7. Cambridge University Press, 1988.

[11] Samson Abramsky et al. "Applying game semantics to compositional software modeling and verification". In: **International Conference on Tools and Algorithms for the Construction and Analysis of Systems**. Springer. 2004, pp. 421–435.

[12] Matthijs Vákár. "Syntax and Semantics of Linear Dependent Types". In: **arXiv preprint arXiv:1405.0033** (2014).

[13] Matthijs Vákár. "A categorical semantics for linear logical frameworks". In: **Foundations of Software Science and Computation Structures**. Springer. 2015, pp. 102–116.

[14] Samson Abramsky, Radha Jagadeesan, and Matthijs Vákár. "Games for Dependent Types". In: **Automata, Languages and Programming**. Springer, 2015, pp. 31–43.

[15] Matthijs Vákár, Radha Jagadeesan, and Samson Abramsky. "Game Semantics for Dependent Types". In: **Information & Computation** (2016). To appear. URL: http://users.ox.ac.uk/~magd3996/research/I%26C%20Submission.pdf.

[16] Matthijs Vákár. "A Framework for Dependent Types and Effects". In: **arXiv preprint arXiv:1512.08009** (2015).

[17] Matthijs Vákár. "An Effectful Treatment of Dependent Types". In: **arXiv preprint arXiv:1603.04298** (2016).

[18] John Cartmell. "Generalised algebraic theories and contextual categories". In: **Annals of Pure and Applied Logic** 32 (1986), pp. 209–243.

[19] Andrew M Pitts. "Categorical logic". In: **Handbook of Logic in Computer Science, Volume 5**. Ed. by S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum. OUP, 2000, pp. 39–128.

[20] Martin Hofmann. **Extensional Constructs in Intensional Type Theory**. Springer, 1997.

[21] Neil Ghani. "$\beta\eta$-Equality for Coproducts". In: **In Typed -calculus and Applications, number 902 in Lecture Notes in Computer Science**. Springer Verlag, 1995, pp. 171–185.

[22] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. "Full abstraction for PCF". In: **Information and Computation** 163.2 (2000), pp. 409–470.

[23] Peter Dybjer. "Inductive families". In: **Formal aspects of computing** 6.4 (1994), pp. 440–465.

[24] F William Lawvere. "Equality in hyperdoctrines and comprehension schema as an adjoint functor". In: **Applications of Categorical Algebra** 17 (1970), pp. 1–14.

[25] Bart Jacobs. "Comprehension categories and the semantics of type dependency". In: **Theoretical Computer Science** 107.2 (1993), pp. 169–207.

[26] Peter Dybjer. "Internal type theory". In: **International Workshop on Types for Proofs and Programs**. Springer. 1995, pp. 120–134.

[27] Thomas Streicher. **Investigations into intensional type theory**. Habilitation thesis. 1993. URL: http://www.mathematik.tu-darmstadt.de/~streicher/HabilStreicher.pdf.

[28] Bart Jacobs. **Categorical logic and type theory**. Elsevier, 1999.

[29] Claudio Hermida and Bart Jacobs. "Structural induction and coinduction in a fibrational setting". In: **Information and computation** 145.2 (1998), pp. 107–152.

[30] Paul Blain Levy. **Call-by-push-value: A Functional/imperative Synthesis**. Vol. 2. Springer Science & Business Media, 2012.

[31] Hagen Huwig and Axel Poigné. "A note on inconsistencies caused by fixpoints in a cartesian closed category". In: **Theoretical Computer Science** 73.1 (1990), pp. 101–112.

[32] Eugenio Moggi. "Notions of computation and monads". In: **Information and computation** 93.1 (1991), pp. 55–92.

[33] P Nick Benton. "A mixed linear and non-linear logic: Proofs, terms and models". In: **Computer Science Logic**. Springer. 1995, pp. 121–135.

[34] Andrew Barber and Gordon Plotkin. **Dual intuitionistic linear logic**. University of Edinburgh, Department of Computer Science, 1996.

[35] Gordon D. Plotkin. "Call-by-name, call-by-value and the λ-calculus". In: **Theoretical computer science** 1.2 (1975), pp. 125–159.

[36] Paul Blain Levy. "Jumbo λ-calculus". In: **Automata, Languages and Programming**. Springer, 2006, pp. 444–455.

[37] Paul Blain Levy. "Adjunction models for call-by-push-value with stacks". In: **Theory and Applications of Categories** 14.5 (2005), pp. 75–110.

[38] Nick Benton and Philip Wadler. "Linear logic, monads and the lambda calculus". In: **Logic in Computer Science, 1996. LICS'96. Proceedings., Eleventh Annual IEEE Symposium on**. IEEE. 1996, pp. 420–431.

[39] Anders Kock. "Strong functors and monoidal monads". In: **Archiv der Mathematik** 23.1 (1972), pp. 113–120.

[40] William W Tait. "Intensional interpretations of functionals of finite type I". In: **The journal of symbolic logic** 32.02 (1967), pp. 198–212.

[41] Paul Blain Levy. "Call-by-push-value: Decomposing call-by-value and call-by-name". In: **Higher-Order and Symbolic Computation** 19.4 (2006), pp. 377–414.

[42] Gordon Plotkin and John Power. "Notions of computation determine monads". In: **Foundations of Software Science and Computation Structures**. Springer. 2002, pp. 342–356.

[43] Jean-Yves Girard. "Linear logic". In: **Theoretical computer science** 50.1 (1987), pp. 1–101.

[44] Samuel Eilenberg and G Max Kelly. "Closed categories". In: **Proceedings of the Conference on Categorical Algebra**. Springer. 1966, pp. 421–562.

[45] Samson Abramsky. "No-cloning in categorical quantum mechanics". In: **Semantic Techniques in Quantum Computation** (2009), pp. 1–28.

[46] Paul-André Melliès. "Categorical semantics of linear logic". In: **Panoramas et Syntheses** 27 (2009), pp. 15–215.

[47] GM Bierman. **On intuitionistic linear logic**. Tech. rep. UCAM-CL-TR-346. University of Cambridge, Computer Laboratory, 1994.

[48] Anders Kock. "Closed categories generated by commutative monads". In: **Journal of the Australian Mathematical Society** 12.04 (1971), pp. 405–424.

[49] Fred EJ Linton. "Coequalizers in categories of algebras". In: **Seminar on triples and categorical homology theory**. Springer. 1969, pp. 75–90.

[50] William F Keigher. "Symmetric monoidal closed categories generated by commutative adjoint monads". In: **Cahiers de Topologie et Géométrie Différentielle Catégoriques** 19.3 (1978), pp. 269–293.

[51] Peter T Johnstone. **Sketches of an elephant: A topos theory compendium**. Vol. 2. Oxford University Press, 2002.

[52] François Foltz, Christian Lair, and GM Kelly. "Algebraic categories with few monoidal biclosed structures or none". In: **Journal of Pure and Applied Algebra** 17.2 (1980), pp. 171–177.

[53] Anders Kock. "Commutative monads as a theory of distributions". In: **Theory and Applications of Categories** 26.4 (2012), pp. 97–131.

[54] John Power and Edmund Robinson. "Premonoidal categories and notions of computation". In: **Mathematical structures in computer science** 7.05 (1997), pp. 453–468.

[55] Brian Day. "On closed categories of functors". In: **Reports of the Midwest Category Seminar IV**. Springer. 1970, pp. 1–38.

[56] Samson Abramsky. "Axioms for Definability and Full Completeness". In: **Proof, Language and Interaction: Essays in Honour of Robin**. MIT Press, 2000, pp. 55–75.

[57] Ana C Calderon and Guy McCusker. "Understanding game semantics through coherence spaces". In: **Electronic Notes in Theoretical Computer Science** 265 (2010), pp. 231–244.

[58] J Martin E Hyland and C-HL Ong. "On full abstraction for PCF: I, II and III". In: **Information and computation** 163.2 (2000), pp. 285–408.

[59] Samson Abramsky and Radha Jagadeesan. "Game semantics for access control". In: **Electronic Notes in Theoretical Computer Science** 249 (2009), pp. 135–156.

[60] Russell Harmer and Guy McCusker. "A fully abstract game semantics for finite nondeterminism". In: **Logic in Computer Science, 1999. Proceedings. 14th Symposium on**. IEEE. 1999, pp. 422–430.

[61] James Laird. "Full abstraction for functional languages with control". In: **Logic in Computer Science, 1997. LICS'97. Proceedings., 12th Annual IEEE Symposium on**. IEEE. 1997, pp. 58–67.

[62] James David Laird. **A semantic analysis of control**. University of Edinburgh. College of Science and Engineering. School of Informatics., 1999.

[63] Samson Abramsky, Kohei Honda, and Guy McCusker. "A fully abstract game semantics for general references". In: **Logic in Computer Science, 1998. Proceedings. Thirteenth Annual IEEE Symposium on**. IEEE. 1998, pp. 334–344.

[64] Samson Abramsky and Guy McCusker. "Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions". In: **Electronic Notes in Theoretical Computer Science** 3 (1996), pp. 2–14.

[65] Samson Abramsky et al. "Game semantics for programming languages". In: **Lecture notes in computer science** 1295 (1997), pp. 3–4.

[66] Guy McCusker. "Games and full abstraction for FPC". In: **Logic in Computer Science, 1996. LICS'96. Proceedings., Eleventh Annual IEEE Symposium on**. IEEE. 1996, pp. 174–183.

[67] G. McCusker. **Games and Full Abstraction for a Functional Metalanguage with Recursive Types**. Distinguished Dissertations. Springer London, 2012. URL: https://books.google.co.uk/books?id=0oXbBwAAQBAJ.

[68] Per Martin-Löf. “An intuitionistic theory of types”. In: **Twenty-five years of constructive type theory**. Vol. 36. Oxford University Press, 1998, pp. 127–172.

[69] Iliano Cervesato and Frank Pfenning. “A linear logical framework”. In: **LICS’96. Proceedings.** IEEE. 1996, pp. 264–275.

[70] Ugo Dal Lago and Marco Gaboardi. “Linear dependent types and relative completeness”. In: **LiCS 2011. Proceedings.** IEEE. 2011, pp. 133–142.

[71] Barbara Petit et al. “Linear dependent types in a call-by-value scenario”. In: **Proceedings of the 14th symposium on Principles and practice of declarative programming**. ACM. 2012, pp. 115–126.

[72] Marco Gaboardi et al. “Linear dependent types for differential privacy”. In: **ACM SIGPLAN Notices**. Vol. 48. 1. ACM. 2013, pp. 357–370.

[73] Kevin Watkins et al. **A concurrent logical framework I: Judgments and properties**. Tech. rep. DTIC Document, 2003.

[74] J Peter May and Johann Sigurdsson. **Parametrized homotopy theory**. 132. American Mathematical Soc., 2006.

[75] Michael Shulman. “Enriched indexed categories”. In: **Theory and Applications of Categories** 28.21 (2013), pp. 616–695.

[76] Kate Ponto and Michael Shulman. “Duality and traces for indexed monoidal categories”. In: **Theory and Applications of Categories** 26.23 (2012), pp. 582–659.

[77] Urs Schreiber. “Quantization via Linear homotopy types”. In: **arXiv preprint arXiv:1402.7041** (2014).

[78] Neelakantan R Krishnaswami, Pierre Pradic, and Nick Benton. “Integrating Linear and Dependent Types”. In: **Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. ACM. 2015, pp. 17–30.

[79] Bart Jacobs. “Semantics of weakening and contraction”. In: **Annals of pure and applied logic** 69.1 (1994), pp. 73–106.

[80] Lewis Carroll, John Tenniel, and Martin Gardner. **The Annotated Alice**. Penguin books, 1965.

[81] Nax Paul Mendler. “Predictive type universes and primitive recursion”. In: **Logic in Computer Science, 1991. LICS’91., Proceedings of Sixth Annual IEEE Symposium on**. IEEE. 1991, pp. 173–184.

[82] Martin Hyland and Andrea Schalk. “Glueing and orthogonality for models of linear logic”. In: **Theoretical computer science** 294.1-2 (2003), pp. 183–231.

[83] Erik Palmgren and Viggo Stoltenberg-Hansen. “Domain interpretations of Martin-Löf’s partial type theory”. In: **Annals of Pure and Applied Logic** 48.2 (1990), pp. 135–196.

[84] Claude Bertrand. “A Natural Semantics of First-order Type Dependency”. In: **Theor. Comput. Sci.** 123.1 (Jan. 1994), pp. 31–53. URL: http://dx.doi.org/10.1016/0304-3975(94)90067-1.

[85] Antonio Bucciarelli et al. “On linear information systems”. In: **arXiv preprint arXiv:1003.5518** (2010).

[86] James McKinna. “Why dependent types matter”. In: **ACM Sigplan Notices**. Vol. 41. 1. ACM. 2006, pp. 1–1.

[87] Conor McBride. “Faking it Simulating dependent types in Haskell”. In: **Journal of functional programming** 12.4-5 (2002), pp. 375–392.

[88] Steve Awodey and Michael A Warren. “Homotopy theoretic models of identity types”. In: **Mathematical Proceedings of the Cambridge Philosophical Society** 146.01 (2009), pp. 45–55.

[89] U HoTTbaki. **Homotopy Type Theory: Univalent Foundations of Mathematics**. Institute for Advanced Study: `http://homotopytypetheory.org/book`, 2013.

[90] Samson Abramsky and Radha Jagadeesan. “Games and full completeness for multiplicative linear logic”. In: **The Journal of Symbolic Logic** 59.02 (1994), pp. 543–574.

[91] Hanno Nickau. “Hereditarily sequential functionals”. In: **International Symposium on Logical Foundations of Computer Science**. Springer. 1994, pp. 253–264.

[92] Samson Abramsky and Radha Jagadeesan. “A game semantics for generic polymorphism”. In: **Annals of Pure and Applied Logic** 133.1 (2005), pp. 3–37.

[93] Andrzej S Murawski and Nikos Tzevelekos. “Game semantics for good general references”. In: **Logic in Computer Science (LICS), 2011 26th Annual IEEE Symposium on**. IEEE. 2011, pp. 75–84.

[94] Vincent Danos and Russell S Harmer. “Probabilistic game semantics”. In: **ACM Transactions on Computational Logic (TOCL)** 3.3 (2002), pp. 359–382.

[95] Samson Abramsky et al. “Nominal games and full abstraction for the nu-calculus”. In: **Logic in Computer Science, 2004. Proceedings of the 19th Annual IEEE Symposium on**. IEEE. 2004, pp. 150–159.

[96] Samson Abramsky and Guy McCusker. “Call-by-value games”. English. In: **Computer Science Logic**. Ed. by Mogens Nielsen and Wolfgang Thomas. Vol. 1414. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998, pp. 1–17. URL: `http://dx.doi.org/10.1007/BFb0028004`.

[97] Marc Bezem, Thierry Coquand, and Simon Huber. “A model of type theory in cubical sets”. In: **19th International Conference on Types for Proofs and Programs (TYPES 2013)**. Vol. 26. 2014, pp. 107–128.

[98] Norihiro Yamada. “Game Semantics for Martin-Löf Type Theory”. In: **arXiv preprint arXiv:1610.01669** (2016).

[99] Catarina Coquand. “A realizability interpretation of Martin-Löf’s type theory”. In: **Twenty-Five Years of Constructive Type Theory** (1998).

[100] Peter Dybjer. “Inductive sets and families in Martin-Löf’s type theory and their set-theoretic semantics”. In: **Logical frameworks**. Ed. by G. Huet and G. Plotkin. Vol. 2. Cambridge Univ Press, 1991, p. 6.

[101] Hugo Herbelin. “On the degeneracy of Σ-types in presence of computational classical logic”. In: **Typed Lambda Calculi and Applications**. Springer, 2005, pp. 209–220.

[102] The Coq development team. **The Coq proof assistant reference manual**. Version 8.0. LogiCal Project. 2004. URL: http://coq.inria.fr.

[103] Ulf Norell. **Towards a practical programming language based on dependent type theory**. Vol. 32. Chalmers University of Technology, 2007.

[104] Lennart Augustsson. "Cayenne — a language with dependent types". In: **ACM SIGPLAN Notices**. Vol. 34. 1. ACM. 1998, pp. 239–250.

[105] Thorsten Altenkirch et al. "ΠΣ: Dependent types without the sugar". In: **Functional and Logic Programming**. Springer Berlin Heidelberg, 2010, pp. 40–55.

[106] Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. "Combining proofs and programs in a dependently typed language". In: **ACM SIGPLAN Notices** 49.1 (2014), pp. 33–45.

[107] Edwin Brady. "Idris, a general-purpose dependently typed programming language: Design and implementation". In: **Journal of Functional Programming** 23.05 (2013), pp. 552–593.

[108] Hongwei Xi and Frank Pfenning. "Dependent types in practical programming". In: **Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages**. ACM. 1999, pp. 214–227.

[109] Nikhil Swamy et al. "Dependent types and multi-monadic effects in F". In: **ACM SIGPLAN Notices**. Vol. 51. 1. ACM. 2016, pp. 256–270.

[110] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. "Polymorphism and separation in hoare type theory". In: **ACM SIGPLAN Notices**. Vol. 41. 9. ACM. 2006, pp. 62–73.

[111] Daniel R Licata and Robert Harper. "Positively dependent types". In: **Proceedings of the 3rd workshop on Programming languages meets program verification**. ACM. 2009, pp. 3–14.

[112] Jeff Egger, Rasmus Ejlers Møgelberg, and Alex Simpson. "Enriching an Effect Calculus with Linear Types." In: **CSL**. Vol. 5771. Springer. 2009, pp. 240–254.

[113] Danel Ahman, Neil Ghani, and Gordon D Plotkin. "Dependent types and fibred computational effects". In: **International Conference on Foundations of Software Science and Computation Structures**. Springer. 2016, pp. 36–54.

[114] Michael Shulman. "Brouwer's fixed-point theorem in real-cohesive homotopy type theory". In: **arXiv preprint arXiv:1509.07584** (2015).

[115] Urs Schreiber and Michael Shulman. "Quantum gauge field theory in cohesive homotopy type theory". In: **arXiv preprint arXiv:1408.0054** (2014).

[116] Andreas Abel, Klaus Aehlig, and Peter Dybjer. "Normalization by evaluation for Martin-Löf type theory with one universe". In: **Electronic Notes in Theoretical Computer Science** 173 (2007), pp. 17–39.

[117] Danel Ahman and Sam Staton. "Normalization by evaluation and algebraic effects". In: **Electronic Notes in Theoretical Computer Science** 298 (2013), pp. 51–69.

[118] Ana Bove and Peter Dybjer. "Dependent types at work". In: **Language engineering and rigorous software development**. Springer, 2009, pp. 57–99.

[119] Henk Barendregt, Wil Dekkers, and Richard Statman. **Lambda calculus with types**. 2013.

[120] Derrick G Kourie and Bruce W Watson. **The Correctness-by-Construction Approach to Programming**. Springer Science & Business Media, 2012.

[121] Peter Dybjer. "A general formulation of simultaneous inductive-recursive definitions in type theory". In: **The Journal of Symbolic Logic** 65.02 (2000), pp. 525–549.

[122] Pierre-Marie Pédrot and Nicolas Tabareau. "An Effectful Way to Eliminate Addiction to Dependence". In: **Logic in Computer Science (LICS), 2017 32nd Annual ACM/IEEE Symposium on**. Reykjavik, Iceland, June 2017, p. 12. URL: https://hal.inria.fr/hal-01441829.

[123] Aleksandar Nanevski et al. "Ynot: dependent types for imperative programs". In: **ACM Sigplan Notices**. Vol. 43. 9. ACM. 2008, pp. 229–240.

[124] Conor McBride. "I got plenty o'nuttin'". In: **A List of Successes That Can Change the World**. Springer, 2016, pp. 207–233.

[125] Gregory Arone and Marja Kankaanrinta. **The Goodwillie tower of the identity is a logarithm**. Citeseer, 1995. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.8306.

[126] Samson Abramsky and Ross Duncan. "A categorical quantum logic". In: **Mathematical Structures in Computer Science** 16.03 (2006), pp. 469–489.

[127] Pierre Clairambault. "Least and greatest fixpoints in game semantics". In: **International Conference on Foundations of Software Science and Computational Structures**. Springer. 2009, pp. 16–31.

[128] Danel Ahman. **Handling fibred algebraic effects**. Preprint. URL: https://danelahman.github.io/drafts/handling_fibred_algebraic_effects.pdf.

[129] Thomas Huckle. **Collection of software bugs**. 2015. URL: https://www5.in.tum.de/~huckle/bugse.html.

[130] Joseph D'Urso. **How much would it cost to end hunger?** 2015. URL: https://www.weforum.org/agenda/2015/07/how-much-would-it-cost-to-end-hunger/.

[131] Tom Britton et al. "Reversible debugging software". In: **University of Cambridge-Judge Business School, Tech. Rep** (2013).