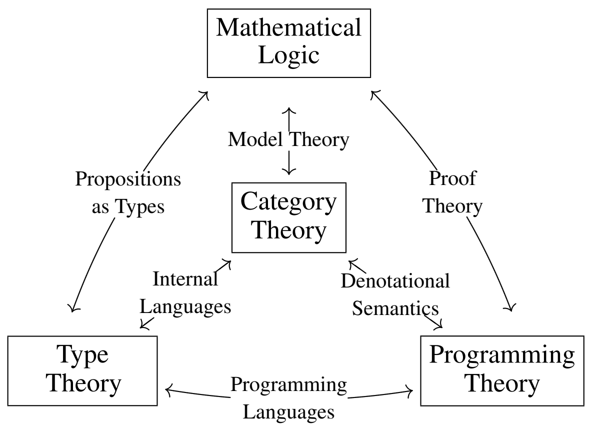
The Scientific Proposal

a State of the art and objectives

We present the general scientific background. This develops in the context of both computer science and mathematics, encompassing four main areas of research: category theory, mathematical logic, programming theory, and type theory. Each of these is regarded as a complimentary part of the others, forming a unified whole through which the subject is to be investigated. Figure 1 gives a schematic view of the areas together with their interactions. We contend that an approach neglecting any one of them is to the detriment of the others, missing out the depth and richness of the subject and, with this, opportunities for research and development.

Figure 1 Research areas and interactions.



In Section (a-1) we outline the origins and influences that lead to this conception. Section (a-2) follows with the current state of the field and raising questions for research. Section (a-3) gives the research objectives that are to be pursued for advancing current knowledge. It also presents the MaStrPLan team that will achieve this. Both these points are respectively expounded upon in Sections (b & c).

a-1 Origins and influences

a-1.i Logic and computation. Computer science was born as a branch of mathematics, specifically mathematical logic, before electronic computers were built. Its inception was the 1928 Entscheidungsproblem (decision problem) of Hilbert asking whether there is an algorithmic procedure for deciding mathematical statements. Negative answers were provided independently by Church [23] and Turing [112] around 1936–1937, giving birth to the mathematical theory of computation. Their completely different approaches gave rise to different branches of theoretical computer science that still permeate the field. On the one hand, the line of development starting with Church's λ-calculus is concerned with prototypical computational languages that are instrumental in studying high-level programming languages. On the other hand, Turing's machines are the most widely used model for analysing computational complexity.

Church's view is central to our concerns here. The λ-calculus is a deceptively simple formal system. Its syntax consists of three types of phrases: variables, applications, and abstraction. Church's seminal innovation was in introducing the latter one, which in modern terminology is referred to as a binding operator, a notion that goes beyond the operators of universal algebra [18]. Binding operators are an integral part of all high-level programming languages. Their mathematical theory is subtle because they define syntax up to the consistent renaming of bound variables (technically referred to as α-equivalence). Consider, for instance, the anecdotal fact that the first definitions of substitution were formally flawed.

Incidentally, the notion of variable binding was initially introduced much earlier, again in the context of mathematical logic. This was done by Frege [49] for the purpose of defining a formal symbolic system axiomatising not only the propositional connectives of Boolean logic [20] but, for the first time, also the quantifiers of predicate logic.

In modern notation, the λ-calculus syntax is given by

$$\begin{array}{rcll} s,t & ::= & & (\lambda\text{-terms}) \\ & | & x & (\text{variables}) \\ & | & (t)s & (\text{application}) \\ & | & \lambda x.\,t & (\text{abstraction}) \end{array}$$

The system has only one rule of computation:

$$(\beta) \quad (\lambda x.\,t)s \longrightarrow t[s/x] \tag{1}$$

by which the result of computing a so-called redex $(\lambda x.\,t)s$ is the λ-term $t[s/x]$ denoting the result of substituting the free occurrences of x in t by s. Non-terminating behaviour arises from self application.

a-1.ii Type theory and logic. The concept of 'type' was also born as a result of solving a foundational problem. In [50], building on [49], Frege put forward a logical system as a foundation for mathematics including arithmetic. By his now famous paradox, Russell [101] observed that one of Frege's axioms led to inconsistency. He was led to this by related contradictions found while developing his account of set theory. Russell's introduction of the 'doctrine of types' [102, Appendix B] came as a direct result to overcome such foundational problems.

Type Theory as we now know it arose however from the integration by Church of types into his λ-calculus [24]. This was a natural step, having in mind the naive interpretation of λ-abstraction as defining functions. The type theory known as the Simply-Typed Lambda Calculus has set of types consisting of basic ones closed under a function-type constructor:

$$\begin{array}{llll} \sigma, \tau & ::= & & \text{(simple types)} \\ & | & \theta & \text{(basic types)} \\ & | & \sigma \to \tau & \text{(function types)} \end{array}$$

In it, λ-terms t are classified according to types τ in contexts Γ assigning types to variables, for which the notation

$$\Gamma \vdash t : \tau$$

is commonly used. This is done according to syntax-directed rules that in the mathematical vernacular are presented as follows

$$\frac{1 \leq i \leq n}{x_1 : \tau_1, \ldots, x_n : \tau_n \vdash x_i : \tau_i}$$

$$\frac{\Gamma \vdash t : \sigma \to \tau \quad \Gamma \vdash s : \sigma}{\Gamma \vdash (t)s : \tau}$$

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x.\, t : \sigma \to \tau} \tag{2}$$

A fundamental discovery in two slightly different, though related, logical contexts (technically, Hilbert-style deduction [57] and Gentzen's natural deduction in sequent form [51]), was made by Curry [30] and Howard [60]. This will be referred to here as the Propositions-as-Types correspondence. Roughly, it establishes a correspondence between terms of types and proofs of propositions. For the Simply-Typed Lambda Calculus, the correspondence can be illustrated by noting that the erasure of term information in its typing rules yields the deduction rules

$$\frac{1 \leq i \leq n}{\tau_1, \ldots, \tau_n \vdash \tau_i}$$

$$\frac{\Gamma \vdash \sigma \to \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash \tau} \qquad \frac{\Gamma, \sigma \vdash \tau}{\Gamma \vdash \sigma \to \tau}$$

of Intuitionistic Logic.

In another very important direction, Church introduced a Simple Theory of Types [24] as an axiomatization of Higher-Order Logic formalised within a Simply-Typed Lambda Calculus with basic types for individuals and propositions enriched with constants for the logical connectives. In doing so, he adopted a radically new perspective, shifting the status of the Simply-Typed Lambda Calculus from that of being a 'language' to becoming a 'metalanguage'; *ie.*, a language for languages, in this particular case for simple type theories. When regarded as a metalanguage, the Simply-Typed Lambda Calculus is considered as an equational theory, with the β-rule (1) stated as an equation together with the extensionality equation

$$(\eta) \quad \lambda x.\,(t)x = t \quad \text{, where } x \text{ is not free in } t$$

Such abstraction processes from languages to metalanguages have become a common activity in computer science and play an important role in our proposed investigations.

a-1.iii Category theory, logic, and type theory. The theory of categories was introduced by Eilenberg and Mac Lane [37]. A category is a mathematical structure consisting of objects and morphisms; where the latter are classified according to pairs of the former: as their domain and codomain. In a category, the notation $f : A \to B$ stipulates that A and B are objects, and that f is a morphism with domain A and codomain B. Categories come equipped with an associative law that composes pairs of morphisms as on the left below

$$\frac{f : A \to B \quad g : B \to C}{g\,f : A \to C} \qquad 1_A : A \to A \tag{3}$$

together with, for every object, an identity morphism as on the right above that is a neutral element for composition. Categories were defined to introduce functors, a notion of morphism between categories, which in turn was defined to introduce natural transformations, a notion of morphism between functors.

It is helpful to appreciate that there are two main uses for categories: in the large and in the small. In the large, categories are seen as mathematical universes of discourse (within which mathematical constructions take place, typically by means of universal properties); like the categories of: sets and functions, algebraic structures and homomorphisms, spaces and continuous functions. In the small, categories are seen as mathematical objects themselves; like a set, a monoid, a preorder, and a graph, all of which can be suitably regarded as a category.

The connection between category theory, logic, and type theory was initiated by Lawvere [74] and Lambek [71], both of whom recognised logical systems and type theories as categories with structure.

Lawvere's insight was to understand logical connectives and type constructors as categorical structures arising from universal properties in the form of adjoint functors, a notion introduced by Kan [65] motivated by homology theory. For instance, the categories with structure corresponding to the Simply-Typed Lambda Calculus with products are Cartesian Closed Categories. These have categorical products and exponentials, respectively denoted × and ⇒, defined by adjoint situations establishing natural bijective correspondences between morphisms as follows:

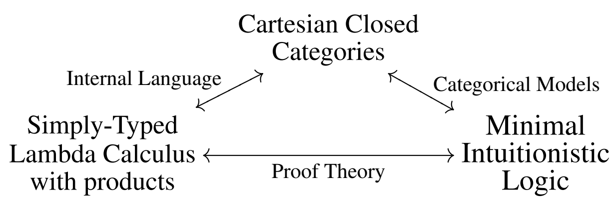
$$\frac{C \to A \;\;,\;\; C \to B}{\overline{\overline{C \to A \times B}}} \qquad \frac{C \times A \to B}{\overline{\overline{C \to A \Rightarrow B}}}$$

The required naturality condition amounts to the β and η laws.

The similarity between the bijective correspondence on the right above and the typing rule for λ-abstraction (2) is not casual, and it is now well-understood that the Simply-Typed Lambda Calculus with products provides an internal language for Cartesian Closed Categories; namely, it is the calculus of all such models. This view further enriches the

Propositions-as-Types correspondence as in Figure 2. It is by now the standard with respect to which analogous developments in the area are measured.

Figure 2 Propositions-As-Types correspondence



a-1.iv Type theory and programming. The Simply-Typed Lambda Calculus, regarded as a language rather than as a metalanguage, is a prototypical functional programming language. The change of perspective from Type Theory to Programming Theory is not straightforward and comes with considerations that enrich both subjects.

In the context of programming languages, Milner understood early on that while a programming language should come with a type discipline to classify programs according to their type invariants, programmers would be better served if type annotations were inferred automatically.

The problem of type inference (by which given a program one wishes to compute the best possible type for it) became of central practical importance. For Combinatory Logic[1] this problem was solved by Hindley [58]. Independently, however, Milner [89] solved it in a context more relevant to programming; namely, for the Simply-Typed Lambda Calculus with parametric polymorphism. The algorithm is now known as the Hindley-Milner type inference method. The approach is very robust, having proved to extend broadly.

Polymorphism in programming refers to languages that support abstraction mechanisms by which a program (function or procedure) can be used with a variety of types. The concept was introduced by Strachey [109], who further classify it as ad-hoc polymorphism and parametric polymorphism. The former is also referred to as overloading and accounts for uses of a program with different types (like integers and reals) by means of different algorithms. The latter indicates the use of a uniform program for all types (like a duplicator program making copies of its input).

Reynolds [99] formalised the programming intuition by introducing the Polymorphic Typed Lambda Calculus; an extension of the Simply-Typed Lambda Calculus with polymorphic types. Roughly, these are abstract parametric types whose programs can be used for all instances of the parameter. Strikingly, this system had already been proposed several years earlier by Girard [53] under the name of System F, as the type-theoretic counterpart of Second-Order Propositional Logic in the context of the Propositions-as-Types correspondence. Further, Girard had also introduced System F_ω; the type-theoretic counterpart of Higher-Order Propositional Logic. These logical systems widely extend the expressiveness of the Symply-Typed Lambda Calculus, notably by the possibility of encoding inductive data types [19]. System F_ω lies at the core of the Haskell programming language.

The type systems mentioned above aim at providing logical foundations and, as such, when viewed as programming languages can only introduce terminating computations. There are various ways in which one can extend them to Turing complete computational languages. In this direction, and motivated by model-theoretic studies of the λ-calculus, Scott [104] introduced an extension of Typed Combinatory Logic with a fixpoint combinator for general recursion. Plotkin [95] studied it as a programming language, shifting from Typed Combinatory Logic to Simply-Typed Lambda Calculus. In doing so, he opened up further possible distinctions in the study of type theories for computation; namely, the consideration of equational theories for different function call mechanisms: by value or by name [94], as in ML or Haskell.

The influence of model theories on type theories, logical systems, and programming languages plays a central role throughout the proposal.

a-2 State of the art: Questions and pathways for research

Having introduced the scientific framework, we turn attention to topics of active research in the area. The intent is not to be comprehensive. Rather we identify main questions and pathways for research that are to be the stepping stones for the research plan detailed in Section (b).

a-2.i Foundations. We have already mentioned several type theories, and we will mention a few more in the sequel. However, the following fundamental open question remains open:

Research question see (b-1.i)
- What is a type theory?

Section (b-1.i) aims at a comprehensive mathematical answer, that will also serve as a framework for our other type-theoretic developments.

a-2.ii Logical wiring. Lambek [71] recognised the similarity between the basic structure of a category, given by identities and composition (see (3)), and the wiring of logical deduction systems, specifically Gentzen's sequent calculi [51], given by the axiom and cut rules. Intuitionistic sequents led Lambek [72] to axiomatize their algebra under the concept of multicategory. This correspondence is roughly as outlined

[1] Combinatory Logic is an important symbolic formalism introduced by Schönfinkel [105] closely related to the λ-calculus, but based on algebraic combinators, that enjoys the Propositions-as-Types correspondence with respect to the Hilbert-style deduction system for Intuitionistic Logic.

below (where the vector notation $\vec{\cdot}$ stands for finite sequences of objects):

Intuitionistic Sequent Calculus	Multicategories
(Axiom) $\dfrac{}{P \vdash P}$	(Identity) $(A) \xrightarrow{1_A} A$
(Cut) $\dfrac{\Gamma \vdash P \quad \Gamma_1, P, \Gamma_2 \vdash Q}{\Gamma_1, \Gamma, \Gamma_2 \vdash Q}$	(Multicomposition) $\dfrac{\vec{Y} \xrightarrow{f} A \quad \vec{X}, A, \vec{Z} \xrightarrow{g} B}{\vec{X}, \vec{Y}, \vec{Z} \xrightarrow{g \circ_{\vec{X}, \vec{Z}} f} B}$

The analogous for classical sequents was done by Szabo [110] with the introduction of polycategories.

The structure of multicategory appeared independently in a very different mathematical context, the work of May [83] on homotopical algebra, under the name of Operad. Operads are now central to studies in higher-dimensional algebra (see *eg.* [76, 80]).

> Research pathway — see (b-1.ii)
> - Investigate interactions between logical sequent calculi and algebraic operad theory.

a-2.iii Dependent types. Dependent type theory is a formalism introduced by de Bruijn [32] that extends that of simple type theories by allowing types to be indexed or parameterised by other types. Such objects abound in computer science and mathematics. For instance, in combinatorics one is interested in the type of permutations $\mathfrak{S}(n)$ on n elements, as the index (or parameter) n ranges over the natural numbers $\mathbb{N}$. In modern notation, this is presented by a judgement of the form

$$n : \mathbb{N} \vdash \mathfrak{S}(n) \ \mathsf{type}$$

There are two fundamental constructions on such dependent judgements as follows

$$\frac{i : I \vdash T(i) \ \mathsf{type}}{\vdash \Sigma i : I.\, T(i) \ \mathsf{type}} \qquad \frac{i : I \vdash T(i) \ \mathsf{type}}{\vdash \Pi i : I.\, T(i) \ \mathsf{type}}$$

respectively known as dependent sums and dependent product types (see *eg.* [**?**]). These generalise the product and function types of Simply-Typed Lambda Calculus and, under the Propositions-as-Types correspondence, amount to existential and universal quantification. We omit discussing the syntax of terms for these types. As for their equational theory, we only mention that dependent sums may be weak or strong and that dependent products may be intensional (under β equality) or extensional (under $\beta\eta$ equality).

A basic question in this area, for which there is as yet no satisfactory answer is worth noting:

REVISE

> Research question — see (b-2.i)
> - What is a model of intensional type theory?

This is part of the proposed investigations in Section (b-2.i).

The passage from sum and product types to their dependent versions required new type theories. Categorical models suggest further generalisations. These are the subject of Section (b-3) under the following.

> Research pathway — see (b-3)
> - Develop type theories from mathematical models.

a-2.iv Mathematical universes. The ability to construct new mathematical universes of discourse from old ones is a fundamental part of the technical toolkit of researchers in semantics, be it either in logic or computation.

One technique to do so is to enrich the semantic universe with a mode of variation. In its basic form, starting from the universe of sets and functions $\mathcal{S}$ one considers the universe $\mathcal{S}^{\mathbb{C}}$ of so-called presheaves consisting of the $\mathbb{C}$-variable sets for a small category $\mathbb{C}$. The importance of this passage is that the kind of variation embodied in the parameter small category translates to new, often surprising, internal structure in the universe of presheaves.

The presheaf construction was introduced by Grothendieck together with a more sophisticated and important refinement of it known as the sheaf construction [7] (in the context of the Weil conjectures in cohomology theory). Sheaves are a central object of study in the area of mathematics known as Topos Theory [62]; a topos being a universe of discourse for Higher-Order Intuitionistic Logic [73].

In view of the many possible applications, it is natural to ask:

> Research questions — see (b-2.ii)
> - Which mechanisms are there for changing from a type theory to another one as universes of discourse?
>
>   Can this be done while maintaining the relevant computational properties and then incorporated into mechanical proof assistants?

This question should not only be considered from the topos-theoretic viewpoint mentioned above; but also from other approaches, notably that of the related forcing technique of Cohen [25] (introduced by him to prove the independence of both the axiom of choice and the continuum hypothesis from Zermelo-Fraenkel Set Theory) and its elaboration by Scott and Solovay as Boolean-valued models [103].

a-2.v Indexed programming. The use of presheaf categories in computer science applications has been prominent; *eg.* in programming language theory, lambda calculus, domain theory, concurrency theory, and type theory. In particular, in their most elementary discrete form, presheaves can be found in programming languages as indexed datatypes; programming with which will be generically referred to as indexed programming.

Indexed programming developed from two main influences (none to do with presheaves though): the practical needs of supporting data structures able to maintain strong data invariants, like nested and generalised datatypes in functional programming [108, 117]; and the experimentation with dependently-typed programming languages [9, 84] as a by-product of dependent type theory. These two views somehow pull in opposite directions and, as such, lead to conceptually different languages. One is lead to investigate the following.

Research pathway see (b-4.i)

- Develop foundational type theories for indexed datatypes.

  Design and implement indexed programming languages from these and pragmatics.

a-2.vi Resources, effects, modalities. In the late 1980s, two important analyses of computation, respectively for resources and effects, were proposed by Girard [54] and by Moggi [90]. The former was in the contexts of logic and proof theory; the latter in that of denotational semantics and category theory. Both, however, have had tremendous impact in programming language theory. The resource analysis in the form of Linear Logic calculi; the effect analysis in the form of Computational λ-calculi.

From the viewpoint of categorical models, the resource and effect management structures are respectively seen as comonadic and monadic structures. Comonads and monads being dual categorical concepts that arose in the context of cohomology theory in the 1960s [12].

A classical basic result of category theory establishes a strong correspondence between (co)monads and adjoint functors (see *eg.* [81]). One aspect of this is that every adjunction

$$\begin{tikzcd} \mathcal{D} \arrow[r, bend left, "\mathsf{G}"] \arrow[r, phantom, "\perp"] & \mathcal{C} \arrow[l, bend left, "\mathsf{F}"] \end{tikzcd} \tag{4}$$

with F and G respectively left and right adjoint to each other, gives rise (by composition) to a comonad on $\mathcal{D}$ and a monad on $\mathcal{C}$. This viewpoint gave impetus to further analyses based on the more primitive notion of adjunction.

Models of Linear Logic are founded on the theory of monoidal categories [81, Chapter VII.1]. For them, one requests that the adjunction be monoidal with respect to linear structure on $\mathcal{D}$ and cartesian (or multiplicative) structure on $\mathcal{C}$ (see *eg.* [85]). On the other hand, models of Computational λ-calculi rely on enriched category theory [66]. Here, the structure is roughly given by an enriched adjunction with $\mathcal{C}$ cartesian and $\mathcal{D}$ with powers (or exponentials) relative to $\mathcal{C}$.

In the context of linear logic, examples are the mixed linear/cartesian models and calculi of Benton and Wadler [17] and of Barber and Plotkin [10]. In the context of effect calculi, examples are the Call-By-Push-Value of Levy [77] and the Enriched Effect Calculus of Egger, Møgelberg, and Simpson [36]. Metaphorically speaking, both these lines of work regard resources and effects as being opposite sides of the same coin. However, this is not so in all models; and the following question remains unanswered.

Research question see (b-2.iii)

- How can resources and effects be reconciled and unified?

To answer it, a broader view of the subject involving the orthogonal notion of polarisation seems to be needed. From the programming-language viewpoint, this further enriches the computational picture with the ability of distinguishing between eager *vs.* lazy modes of computation and data structure. We incorporate this into the following.

Research pathway see (b-2.iii & b-4.ii)

- Study and develop the theory of resource management, computational effects, and polarisation. Percolate this down into the design of programming languages.

From the logical point of view, resource comonads and effect monads are so-called modal operators, and some of the literature has indeed considered them as such (see *eg.* [68]). The field of modal logics is broad, with many subfields of specialised logics motivated by computation, linguistics, and philosophy. While a class of modal logics known as temporal logics have played a prominent role, and been very successful, in the area of computer aided verification; the impact of modal logics on programming languages has been peripheral. It is thus natural and important to reconsider them in this context.

Research pathway see (b-2.iv & b-4.iii)

- Investigate the Propositions-As-Types correspondence for modal logics, and apply it to programming languages.

Let us stress that we are specially interested in modalities for computation with reflection.

a-2.vii Sequent calculi. A theme that runs orthogonal to the logics under consideration is whether they are specified in natural deduction or sequent calculus style (see *eg.* [114]).

Most of the work on the Propositions-As-Types correspondence has been done for natural deduction systems; especially in relation to programming language theory, where the logical syntax matches that of functional languages.

On the other hand, there is as yet no established syntax for sequent-style calculi. A question at the core of this situation is:

Research question see (b-4.ii)

- What is the categorical algebra of classical sequent calculi?

Nevertheless, a syntactic system that is proving robust in applications seems to be emerging. This is the System L of Curien and Herbelin [28], applied further in *eg.* [116, 91, 29].

The main novel features of System L, and its philosophy, are an intrinsic symmetry (reflecting the computation roles of program and environment) and a close connection with abstract machines (which are internalised into the calculus). From this perspective we ask:

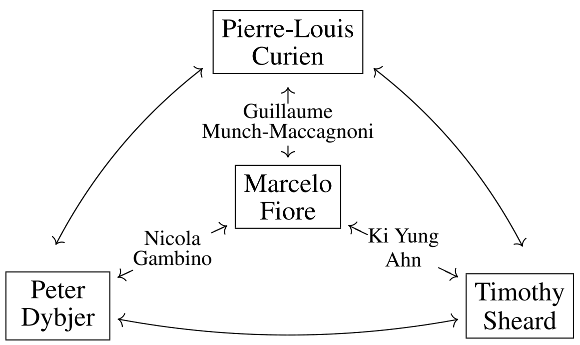
Research question see (b-4.ii)
- What can the proof theory of sequent calculi do for programming?

a-3 Research objectives

The general aim of MaStrPLan is to build a group dedicated to combined research in category theory, mathematical logic, programming theory, and type theory to advance current knowledge in their mathematical foundations and practical applications. To this end, MaStrPLan assembles an international team of world-leading experts in each of the project research areas.

The team consists of *Principal Investigator* Marcelo Fiore; *Senior Researchers* Pierre-Louis Curien, Peter Dybjer, and Timothy Sheard; and *Junior Researchers* Ki Yung Ahn, Nicola Gambino, and Guillaume Munch-Maccagnoni. Figure 3 gives a schematic presentation of the team that matches the research areas as rendered in Figure 1. Further details are deferred to Section (c).

Figure 3 MaStrPLan team



The overall research objectives that we are to undertake are classified under four headings as follows.

1 Foundations: A comprehensive research programme on the metamathematics of type theories.

We will provide an algebraic answer to what type theories are, establishing connections with areas of abstract algebra where related structures play a central role.

2 Models: Study of mathematical models for type theories and logical systems.

We will investigate model theories targeting semantic problems at the forefront of current understanding, specifically for intensional type theory and polarised logic.

3 Calculi: Development of formalisms of deduction as internal languages of mathematical theories.

We will research type theories that go beyond current logical frameworks, especially motivated by (higher-dimensional) category theory.

4 Programming: Design and implementation of, and experimentation with, novel computational languages.

We will target languages with first-class indexed data structures, programming paradigms stemming from sequent calculi embodying computational effects, and explore extensions for metaprogramming.

b Methodology

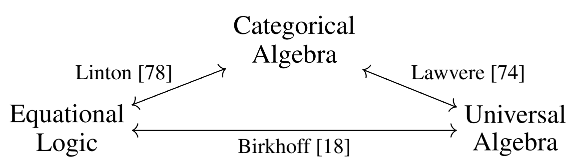
This section expands our research objectives providing a plan together with the methodology for its completion.

b-1 Foundations

b-1.i Algebraic Type Theory. To understand our approach it is best to start by stripping type theories down to their essential bare structure. To do so, let us eliminate type constructors and binding operators from them. What one is left with is a notion of type theory that reduces to that of many-sorted algebraic theory [18], a thoroughly studied area of algebra.

The modern understanding of many-sorted algebraic equational theories is through three interrelated perspectives as in the Algebraic Trinity of Figure 4. Equational Logic is the metalanguage of algebraic theories, while Universal Algebra is its model theory. They were both introduced by Birkhoff [18], with the former as a by-product of the latter related by a soundness and completeness theorem. The categorical viewpoint of algebraic theories came later, with the work of Lawvere [74] and Linton [78]. Two crucial categorical structures that play a pivotal role here are: Lawvere theories (closely related to Hall's abstract clones) and finitary monads (a notion arising in algebra from free constructions).

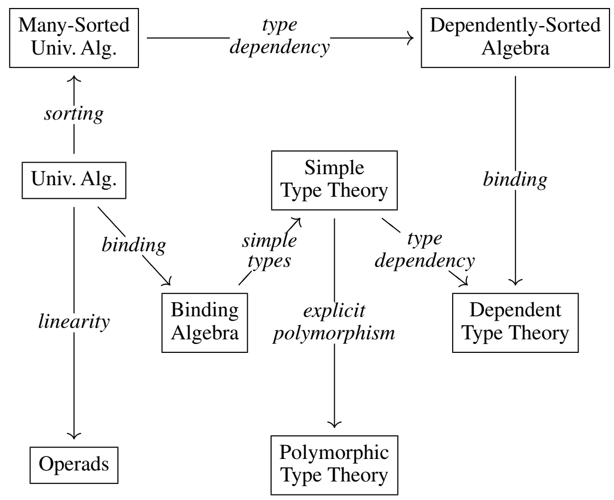
Figure 4 Algebraic Trinity



Each of the approaches to algebraic theories in the Algebraic Trinity gives a different viewpoint of the subject. Thus, it is important to consider them all. For instance, Equational Logic provides the deductive system for formal reasoning about equations from axioms; Universal Algebra provides a general notion of model from which one derives an abstract notion of syntactic structure by means of freely generated mod-

REVISE

Figure 5 Algebraic Type Theory research space



els; Categorical Algebra provides an invariant notion of theory together with a notion of translation between them that lifts to adjoint functors between categories of models.

Having deconstructed type theories down to many-sorted algebraic theories: Can one reconstruct them back, and in the process enrich them, as a unified mathematical theory encompassing all aspects of the Algebraic Trinity? This is the main general goal of this research track.

To make substantial progress in the area, our proposal is to systematically explore a broad spectrum of key features present in type theories. The scale at which we will be attempting this is unprecedented, and is graphically represented by the research space of investigation in Figure 5, where the *linearity* dimension could be transported along or mixed with all the other ones.

Initial investigations in this direction have been carried out by Fiore and PhD students [46, 47]. These extend the Algebraic Trinity to the realm of Binding Algebra, *viz.* algebraic languages with binding operators. The extension to simple type theories should follow along similar lines further taking into account the algebraic structure of types. As for the other points of the research space in Figure 5, they are yet to be investigated. For these we have the following specific goals.

[1] To develop a mathematical algebraic framework for the semantics of language phrases, whereby free models universally characterise the abstract syntax of the language.

[2] To synthesise metalanguages for type theories in the form of formal systems for equational deduction, that are sound and complete for the model theory.

[3] To extract syntactic notions of translations between type theories as suggested by the mathematical models, and to use these to provide constructions for the modular combination of type theories.

[4] To test and apply the above mathematical theories by formalising them in Coq or Agda, while at the same time exercising these systems to the limit to identify shortcomings triggering new research in the context of proof assistants.

The outcome of this work will be mathematical foundations for the aforementioned aspects of type theories that are currently treated in an ad hoc fashion.

b-1.ii Wiring structure. A basic concern in the investigation of Section (b-1.i) is the understanding of the mathematical structure of sequents and judgements. This subject has already received attention, but new perspectives are emerging that call for its reconsideration.

Remitting ourselves back to Section (a-2.ii), the first thing to notice is that there are two possible axiomatizations for multicategory composition: the one exhibited there, known as partial composition, and a total (or simultaneous) one of the form

$$\frac{g : A_1, \ldots, A_n \to B \qquad f_i : \vec{X_i} \to A_i \;\; (1 \le i \le n)}{g \circ (f_1, \ldots, f_n) : \vec{X_1}, \ldots, \vec{X_n} \to B} \tag{5}$$

In the presence of identities, both notions become equivalent. An interesting observation from the theory of operads is that this is not so otherwise [82].

Algebraically, the total composition structure (5) is well understood as given by semigroup structure (technically in the category of Joyal Linear Species [63] with respect to the substitution tensor product [67, 64]). But, what about the concept of partial composition? That, after all, is the predominant in logic and type theory. For it, Fiore has recently uncovered an unexpected connection with structure stemming from Lie-algebra theory; specifically, the pre-Lie algebra structure of Gerstenhaber [52]. Furthermore, and analogous, though more interesting situation occurs with polycategories [110] and the Lie-admissible algebraic structure of Albert [4].

We regard these observations as the starting point for interactions between operad theory and logic and type theory for pursuing research as follows.

From operad theory to logic and type theory:

[1] Kontsevitch introduced the notion of a meager PROP (renamed as $\frac{1}{2}$PROP by Markl [82]) sitting in between that of operad and polycategory. Is there any use for these in logic and type theory?

[2] What about other generalisations, like the preshuffle algebras of Ronco [100] and the permutads of Ronco and Loday [79]? In particular, the former embodies the algebra needed in computational contexts with effects, where the notion of total composition does not make sense, and the notion of partial composition is subject to weaker axioms to those of multicategory. This is relevant to our proposed investigations in Sections (b-2.iii & b-4.ii).

[3] There is a rich variety of notions of operads, and notions of composition between them, many of which are motivated by combinatorics. What can they offer to logic and type theory? This question is not as vacuous as it may seem at first sight. For instance, the shuffle and quasi-shuffle composition (see *eg.* [2]) seem to be related to a type theory for non-commutative linear logic of Polakow and Pfenning [98].

From logic and type theory to operad theory:

[4] The new Lie-algebraic view of multicategory composition mentioned above gives a rational reconstruction of an explicit characterisation of the pre-Lie operad by Chapoton and Livernet [22]. The case of polycategory composition leads to a conjectured characterisation for the Lie-admissible operad. A possible approach to proving it leads to the introduction of new interesting algebraic structures.

[5] A combination of the *linearity* dimension of Figure 5 with the other dimensions (that take place in the traditional *cartesian* setting, where all structural rules on contexts are allowed) leads to a notion of mixed operad [42], whose associated composition operation is new to the mathematical theory.

Mixed operads are the wiring of linear/cartesian languages, which since the work of Bellantoni and Cook [15] play a role in implicit complexity theory. It would be interesting to develop this connection. A main aim would be to introduce new mathematical structure in implicit complexity theory that would help with its problems. This is relevant to our proposed investigations in Section (b-3.iii).

[6] The algebraic structure on operads considered in mathematics is in the tradition of linear algebra (*ie.* first-order). However, the binding operators of logic and type theory transport to operads as a new kind of algebraic structure which thereby enriches the theory and is worth investigating.

b-2 Models

b-2.i Intensional dependent type theory. We have referred to intensional and extensional type theory in Section (a-2.iii). Recall that the essential difference between them is that the latter extends the computational (or β) rules of the former with extensionality (or η) rules. Since type checking for intensional type theory is decidable, while it is not so for extensional type theory, the former has become the core of foundational systems for proof assistants as metalanguages for constructive mathematics; as Automath, Coq, and Agda. All these systems are based on very rich type theories. For instance, the two current mainstream systems Coq and Agda are respectively based on the Calculus of Constructions [27] and Martin-Löf Type Theory [92]. The proposed investigations below are in this context. Some aspects of these arose and are under discussion with Warren (IAS School of Mathematics, Princeton).

The overall goal here is to:

[1] Develop concrete and abstract models of intensional dependent type theories; and use them both to establish relationships between type theories and between type theory and homotopical mathematical theories. Research on the latter point has only recently gathered momentum in the community.

Conceptually, one may regard a type theory as a basic deduction system extended with type constructors. For instance, Martin-Löf Type Theory embodies dependent sums and dependent products, as well as Identity Types, W Types, and Universes [92]. As such, a type theory is to be thought not just as a single universe of discourse, but rather as a variety of universes of discourse corresponding to the various possible restrictions. From this perspective, it is important to understand the interaction of modularly extending type theories, central to which is the notion of conservative extension. Informally, one system is a conservative extension of another if it faithfully embeds it.

Conservativity has been studied for some simple type theories, but it has been overlooked for dependent type theories. We thus propose here to:

[2] Develop a model-theoretic framework for establishing conservative extension results for Martin-Löf Type Theory and Pure Type Systems.

The programme will lead to sophisticated model constructions that will deepen our understanding of the subject. In particular because this was already the case in proving the conservative extension of the Simply-Typed Lambda Calculus extended with (non-dependent) extensional strong sums [44].

To pursue the program for intensional type theory, some fundamental problems will need to be addressed. For a start:

[3] What is a model of intensional dependent type theory?

Despite the fundamental character of the question there is as yet no established answer. This is in opposite contrast with the extensional case, where categorical models are well understood [**?**]. There are various reasons for this situation.

Firstly, current categorical models are not flexible enough to dispense with extensionality. In this respect, the work on Algebraic Type Theory of Section (b-1.i) will provide new directions. Moreover, concrete models of intensional type theory that researchers work with (like groupoids, strict ω-groupoids, and simplicial sets) interpret dependent sums and dependent products as adjoints and are hence extensional. A question thus arises: REVISE — [34]

[4] Are there natural non-extensional models of dependent type theory?

Secondly, the study of intensional models with

Identity Types [59] is very much under development. In fact, Identity Types are currently receiving renewed special attention, steaming from connections with homotopy theory (and higher-dimensional category theory) that are feeding back into mathematical foundations, specifically Voevodsky's Univalent Foundations of Mathematics program. A main open question [**?**] to be considered is:

[5] What is the relationship between the syntactic model of Identity Types and the mathematical Homotopy Types?

Steps into this problem have been made, see *eg.* [8]. But there are fundamental matters to be understood; for instance, when seen from the viewpoint of Algebraic Type Theory:

[6] What is the relationship between the algebraic theory of Identity Types and the algebraic theory of Weak Higher-Dimensional Categories?

This is to be investigated specially noting that the former crucially relies on binding operators while the latter does not.

The Univalent Foundations of Mathematics program mentioned above is based on Voevodsky's Univalence Axiom. This axiom has been shown consistent with Martin-Löf Type Theory from the outset, but:

[7] Is the extension of Martin-Löf Type Theory with the Univalence Axiom conservative?

Also, it is an intriguing fact that the Univalence Axiom somehow endows Identity Types with logical character, very roughly in that the Identity Type constructor logically commutes with all the other type constructors up to equivalence. It would be interesting to investigate the converse:

[8] Do Logical Identity Types imply the Univalence Axiom?

Finally, it is a main open problem as to how to give a constructive interpretation of the Univalence Axiom. One could instead aim to:

[9] Design a constructive type theory with built-in Logical Identity Types.

Here, we will:

[10] Investigate possible connections to Strachey's notion of parametric polymorphism [109] in computer science as formalised by Reynolds [99] using logical relations.

Were these connections to materialise, it will open up a new flow of ideas between hitherto disconnected fields.

b-2.ii Mathematical universes. We speculate here on directions in relation to Section (a-2.iv), where we are interested in building new models from old ones. A main source of inspiration and guidance for the development comes from Topos Theory [62].

Our proposal is to:

[1] Investigate presheaf, orthogonality, sheaf, glueing, and forcing constructions on type theories.

with the general goal to:

[2] Build a mathematical theory of constructions on type theories that produce new type theories from old ones, together with an interpretation (or compilation) of the latter into the former.

[3] Implement the mathematics in proof assistants, and exercise it in applications.

Initial work in this line has been done by Jaber, Tabareau, and Sozeau [61], who formalised the presheaf construction on a partial order in the Calculus of Constructions, and then re-interpreted back the Calculus of Constructions in the internal presheaf model. However, not only a wide spectrum of other possible constructions (as detailed in item **b-2.ii [1]** above) remains to be explored; but, even in this basic setting, work remains to be done: notably to incorporate Inductive Types.

What we envisaged is also more general, in that we take the view that the new type theory need not be of the same character as the old one. Thus there are two aspects to our proposed investigations: the mathematical one of studying model constructions and the type theoretic one of designing internal languages. To fix ideas, consider as an example that the presheaf construction on a monoidal category endows the new universe of discourse with monoidal closed structure (formally via Day's convolution monoidal structure [31]), and thereby introduces new linear structure. This is important in many applications, for which the reader may consult [41], and complementary to our proposed investigations in Section (b-3.iii).

The study of Cohen Forcing for type theory was stated as an open problem by Beeson [14], who wrote: "Forcing has yet to be worked out directly for Martin-Löf's system.". Work in this direction has only recently begun. Specifically, by Coquand and Jaber [26] who presented an interesting example. Much remains to be explored both with respect to the vast literature on set-theoretic forcing and, more relevantly, with respect to Krivine's Classical Realizability [69], an extension of forcing.

b-2.iii Polarised logic. We propose here a model-theoretic study of a rich variety of logical systems encompassing aspects of resource management and computational effects (recall Section (a-2.vi)). This research constitutes work under discussion with Curien and Munch-Maccagnoni (Laboratoire PPS, Université Paris Diderot - Paris 7). A novelty of our approach is that this will be pursued as informed by the logical notion of polarisation.

The notion of explicit polarisation in logical systems was introduced by Andreoli [5] in his study of proof search in Linear Logic, in particular by focalisation. According to it, logical connectives are classified as either being positive or negative; with these two worlds being dual to each other (categorically by adjunction). Following Andreoli's work, Girard understood the relevance of focalisation via polarisation as a way to tame down the inherent non-determinism in computation (by cut elimination) in classical logic;

and, in this direction, introduced the first explicitly polarised logic: LC [55]. A crucial aspect of this system, is to make eager (*viz.* call-by-value) and lazy (*viz.*, call-by-name) modes of computation explicit, allowing for their combination in a framework with eager and lazy data structure.

Polarisation, though not recognised as such, is also present in the Computational λ-calculi based on adjunctions as mentioned in Section (a-2.vi). This key observation leads to a refinement of the model (4) as follows

$$\begin{tikzcd} \overset{\text{(negative)}}{\mathcal{N}} \arrow[dr, bend left] \arrow[dr, phantom, "\dashv"] & & \overset{\text{(multiplicative)}}{\mathcal{M}} \arrow[dl, bend left] \arrow[dl, phantom, "\dashv"] \\ & \underset{\text{(positive/linear)}}{\mathcal{P}} \arrow[ul, bend left] \arrow[ur, bend left] & \end{tikzcd} \tag{6}$$

with comonadic resource structure given by the adjunction on the right and monadic effect structure given by the adjunction on the left.

When the resource structure is trivial, the above restricts to models of Linear Logic [85]; while, when the linear structure collapses to cartesian structure, one recovers the models of Call-By-Push-Value [77]. One is thus led to the following programme.

[1] Devise a sound and complete logical system for (6), and relate it to existing polarised systems [91, 29].

[2] A first model-theoretic result shows that the effect structure on $\mathcal{P}$ lifts to one on $\mathcal{M}$. Show that this corresponds to an encoding of Call-By-Push-Value in the devised logical system.

[3] Develop a generic syntactic theory able to incorporate concrete computational effects into the devised calculus. Two research possibilities are: (i) considering algebraic theories of effects following the work of Plotkin and Power [96], and (ii) revisiting and refining Filinski's result on representing monads [38] that reduces general monadic effects to the storage and escape effects. Either of these approaches will necessarily have to overcome serious shortcomings. For instance, on the one hand, there is as yet no general operational theory of algebraic effects; and, on the other, a logical system for storage is not yet in place. These problems will be investigated.

[4] The previous development will provide foundational metalanguages. The next step in the programme is to promote them to programming languages, where the effects are implicit, by putting them in so-called direct-style. The kind of model theory involved in this development will be discussed below.

The model theory of (6) above can be further specialised to models with enough internal structure so as to extend the picture as follows

$$\begin{tikzcd} & \mathcal{N} \arrow[dl, bend left] \arrow[dl, phantom, "\dashv"] \arrow[dr, bend left] \arrow[dr, phantom, "\dashv"] & & \mathcal{M} \arrow[dl, bend left] \arrow[dl, phantom, "\dashv"] \\ \mathcal{P}^{\circ} \arrow[ur, bend left] & & \mathcal{P} \arrow[ul, bend left] \arrow[ur, bend left] & \end{tikzcd} \tag{7}$$

where the new adjunction between the negatives and the opposite dual of the positives enriches the models with control structure of the linear and/or delimited continuation kind, though this is to be investigated.

The model theory (7) refines that of two recent developments: the aforementioned Enriched Effect Calculus, which amounts to the case in which the linear structure collapses to cartesian one; and the Tensor Logic of Melliès and Tabareau [86], where the effect structure is collapsed. Our programme for (6) will be then also pursued for (7) drawing connections with these works.

Let us now return to the distinction between metalanguages and programming languages briefly mentioned in item **b-2.iii [4]** above. Model theoretical, this can be roughly seen as follows: whereas the metalanguage is an internal language for co/monadic structure; the programming language is the internal language for the derived structure of co/free algebras for the co/monad—typically defined by the categorical co/Kleisli construction [81]. In particular, the Kleisli category of a computational monad provides a model of call-by-value [90]; while the coKleisli category of a linear exponential comonad provides one of call-by-name (*ie.*, a cartesian closed category) [106].

The programming languages corresponding to the metalanguages of the polarised models (6) and (7) should be the internal languages for an analogous, but more intricate, construction relative to an adjunction rather than a co/monad. In this context, the following will be investigated.

[5] What is the mathematical structure of such construction? The question is subtle. For instance, we envisaged models of LC to arise in this manner and, because of the interaction with call-by-value and call-by-name modes of computation, the structure should be more general that that of a category.

[6] What would then be the mathematical theory of universal constructions for these structures giving rise to logical connectives? Notice again from considering LC that also this will be subtle, requiring constructions of objects with mixed polarities.

[7] What can the resulting direct-style calculi contribute to programming? For discussion on this see Section (b-4.ii).

b-2.iv Modal logics. As mentioned in Section (a-2.vi), resource comonads and effect monads are modalities. Polarisation in the context of modal logics has not been considered yet. We will thus complement the previous section with the

[1] Investigation of Polarised Modal Logics.

Our overall goal here is to

[2] Develop the Propositions-As-Types correspondence for modal logics.

specially in establishing the categorical model theory vertex of it (see Figure 2), that has been scarcely investigated.

Looking ahead in connection to the proposed research in Section (b-4.iii), we will be particularly interested in studying: Borghuis' Modal Pure Type Systems [21]; Artemov's Logic of Proofs [6] and related systems; Mendler's Multimodal CK [87]; and Park and Im's Calculus S_Δ [93].

b-3 Calculi

Categorical Type Theory. The evolution of term and type structure in type theories from its origin to the current state can be succinctly represented as follows

	Term Structure	Type Structure
Equational Logic	Algebraic	—
Simple Type Theory	Binding	Algebraic
Polymorphic Type Theory	Binding	Binding
Dependent Type Theory	Binding	

and it has not changed since de Bruijn's Automath in the late 1960s. We believe that further evolution will come from mathematical input, and in this direction propose two lines of research for type theories as formal languages of categorical structures. These are presented in the two sections below.

b-3.i Generalised Type Theory. Section (b-2.ii) considers presheaf categories in the large as mathematical universes of discourse (or gross toposes). There is however an alternative view of them in the small as spaces (or petit toposes) with morphisms between them given by linear (or cocontinuous) functors, resulting in the bicategory of profunctors (or distributors) [16]. Lawvere [75] considered these structures in the general context of enriched category theory and synthesised a Generalised Logical Calculus out of them. The enrichments give rise to different interpretations, *eg.* in preorders, categories, and metric spaces.

The general goal of our research here is to:

[1] Develop a type theory providing a Propositions-As-Types interpretation of the Generalised Logical Calculus.

This is to be investigated in two stages, respectively corresponding to enrichment over cartesian closed categories and over symmetric monoidal closed categories. In the first case, one roughly has the following translation table between the Generalised Logical Calculus and Predicate Logic.

sum	disjunction
coend	existential quantification
product	conjunction
end	universal quantification
exponential	implication
hom	equality
presheaf application	predicate membership

The second stage introduces linearity.

Success in this endevour will yield a system where the mathematical calculations done with profunctors (*eg.* in the context of the coherence of profunctor composition [16], the mathematical theory of substitution [41], generalised species of structures [45], and generalised polynomial functors [43]) can be established formally.

The coend and end constructions mentioned in the table above are intuitively quotients of dependent sums under a compatibility condition and restrictions of dependent products under a parametricity condition. Taking this view seriously, we will aim to:

[2] Extend item **b-3.i [1]** above to a fully-fledged, possibly higher-dimensional, dependent type theory.

Finally, in an intriguing complementary direction, Fiore [41] has suggested a graphical reading of the Generalised Logical Calculus that is surprisingly close to formalisms for categorical graphical languages [107], and proof [54]/interaction [70] nets. This aspect deserves detailed investigation.

b-3.ii Directed Type Theory. Homotopy Type Theory (under the acronym HoTT, and the slogan 'types are spaces') is the name being used for the body of work at the boundary between Homotopy Theory and Dependent Type Theory through Identity Types.

The question arises:

[1] Is there a notion of Directed Type connecting Higher-Dimensional Category Theory and Dependent Type Theory on which to establish a body of work on Directed Type Theory (DiTT)?

In this context, while an Identity Type

$$\mathrm{Id}_{\mathsf{T}}(x, y)$$

is intended to establish the intensional equality of elements of a type T so that $\mathrm{Id}_{\mathsf{T}}(x, y)$ and $\mathrm{Id}_{\mathsf{T}}(y, x)$ are equivalent; the intuitive idea behind a Directed Type

$$\mathrm{Di}_{\mathsf{T}}(x, y)$$

would be to classify the possible ways in which an element of a type T may evolve to another one in a not possibly reversible manner; as it happens, for instance, in computation. Thereby leading to not necessarily equivalent types $\mathrm{Di}_{\mathsf{T}}(x, y)$ and $\mathrm{Di}_{\mathsf{T}}(y, x)$, and to the DiTT slogan 'types are directed spaces'.

An analysis of the elimination rule for Identity Types reveals that there are at least two sources leading to their inherent reversible character. We mention them below as possible directions of research for investigating Directed Types.

[2] The use of contexts allowing the commutativity of two consecutive variables of the same type, which suggests moving on to a non-commutative setting (as *eg.* in non-commutative linear logic).

[3] The lack of a type duality, a property that seems to be inherent to directionality, whereby every type T has an associated dual type T° for which $\mathrm{Di}_{\mathsf{T}}(x, y)$ and $\mathrm{Di}_{\mathsf{T}^\circ}(y, x)$ are equivalent.

The previous item is of course related to the considerations of the previous section. Pursuing this further, it would be natural to generalise from presheaves, now regarded as discrete fibrations, to the general notion of fibration and aim to:

[4] Develop a type theory modelled on Grothendieck fibrations [56].

As a final point, we note that yet another approach would be to:

[5] Investigate a directed-cylinder construction, intuitively adding a new base point to a type that canonically evolves to every element.

This would generalise the lifting construction of Domain Theory [39] and possibly establish connections with the Complete Cuboidal Sets model of Axiomatic Domain Theory [48] opening new directions.

b-3.iii Substructural Type Theory. The work of Bellantoni and Cook [15] on implicit computational complexity gave a characterisation of the polynomial-time computable functions on finite strings by means of a safe-recursion scheme that restricted primitive recursion. A crucial ingredient of their approach is to divide function arguments into normal and safe ones. Recursion is allowed on normal arguments but not on safe ones. In the terminology that we have been using, normal arguments are linear while safe ones are cartesian. In fact, the notion of safe composition coincides with that for mixed linear/cartesian operads (see item **b-1.ii [5]**). Since then, the role of resource management in taming complexity has been prominently recognised, and a big body of work on implicit computational complexity (*eg.* on soft, bounded, light, and linear logics and types) has followed.

Recently, Beckmann, Buss, and Friedman [13] have extended the safe-recursion scheme to arbitrary set functions. Changing viewpoint form the mathematical universe of sets to that of types, one is lead to:

[1] Extend dependent type theory to the mixed linear/cartesian setting.

In particular, in what concerns to taming computational complexity:

[2] Investigate a safe version of the Induction-Recursion scheme of Dybjer [35].

This development should be also accompanied by a model theory, possibly based on

[3] Realisability models relative to the Linear Combinatory Algebras of Abramsky, Haghverdi, and Scott [1].

b-4 Programming

This track of the proposal is concerned with the design and implementation of programming languages from first principles and pragmatics, followed up by their subsequent test and use.

Three directions for this research are presented. These will be considered as research units in their own right, and also in relation to each other.

b-4.i Indexed programming. This section outlines work under discussion with Ahn and Sheard (Department of Computer Science, Portland State University) on indexed programming (recall Section (a-2.v)).

The main problem to be addressed here is to:

[1] Design a language for programming indexed data structures that will be archetypal for the next generation of such languages.

The task can be approached from two different viewpoints. In a top-down fashion, one may turn dependent type theories into programming languages; conversely, in a bottom-up fashion, one may extend functional programming languages with indexing structure. These two approaches pull the programming language design into two opposite directions. This is pictorially presented in Figure 6, where the main features inherent to each approach are also listed.

Figure 6 Indexed-programming design space

Functional Programming Language  Constructive Proof Assistant

ML, Haskell	Coq, Agda
System F_ω	MLTT, ICC
Polymorphism	Type dependency
Impredicativity	Universes
Type inference	Type checking
Recursive types	Inductive types
Equality types	Identity types

Most of the work in this area has concentrated on exploring the top-down approach (see *eg.* [9] and [84]). Here we will rather pursue research on the bottom-up approach, which has not been explored systematically. There is a strong pragmatic reason for this: our outmost interest is in a language targeted to programmers. However, this should not be taken to mean that we are not interested in the activity of proving as in constructive proof assistants; but indeed our main goal, under the Propositions-As-Types paradigm, is to:

[2] Prove as a by-product of programming; rather than extract programs from proving.

For this to be meaningful, both in theory and practice, the language will need to guarantee logical consistency in a setting that naturally allows programming strong invariants for rich indexing type structure. For instance, so that the code of a compiler guarantees its correctness.

In the above direction, we are investigating:

[3] System F_i: an extension of System F_ω with first-class static type-indexing structure.

which is to serve as the mathematical foundation for designing, and giving operational semantics to, our programming language. We stress here that indices are static, *ie.* determined at compile time. This is a main feature pulling us away from traditional dependently-typed formalisms.

The logical consistency of System F_i amounts to establishing its strong normalisation; while as an extension of System F_ω it will embody rich type structure. In this setting, following Ahn and Sheard [3], we will:

[4] Study programming primitives corresponding to a variety of induction proof principles.

These are to be incorporated in the programming language design. Here, it is interesting to note that the scheme provided by Mendler's iterator [88], that crucially takes advantage of parametric polymorphism, allows for terminating iteration over recursively defined datatypes of possibly mixed variance. Thus, going beyond the typical inductive types of type theory stemming from Dybjer's Inductive Families [33]. It would be interesting to establish a formal connection between these approaches so that they can inform each other; in particular in the context of the termination checkers.

Dually, we will also:

[5] Develop programming primitives corresponding to coinduction proof principles.

which have received less attention.

The above design considerations are to be shaped by the following maxim:

[6] Design language constructs with minimal type annotation supporting maximal type inference.

This is a main open problem in dependently-typed programming language theory.

Finally, let us mention an intriguing possibility suggested by the model-theoretic considerations of Section (b-2.ii). The indexing structure in dependent type theory is discrete. In many applications, however, one is interested in indexed types that furthermore relate structure on indices to structure on the associated indexed family of types. The question arises as to how to:

[7] Build idioms or language abstractions for defining and then programming with indexed types equipped with internal varying structure.

A specific motivation for this comes from the possibility of directly programming, as opposed to having to code, presheaf structure; *eg.* as it is needed in Normalisation by Evaluation [40].

b-4.ii Effects. Our main aim here is to consider the model-theoretic investigations of Section (b-2.iii) from a proof-theoretic viewpoint and percolate this down to programming language theory.

A first main novelty in our approach is that the proof theory to be considered is based on sequent calculus, rather than the traditional line followed so far in programming language foundations based on sequent-style natural deduction.

The proof-theoretic formalism that we will be adhering to is System L, as recalled in Section (a-2.vii). A first main question that arises is:

[1] What is the programming paradigm stemming from sequent calculi in general, and System L in particular?

For instance,

[2] Does the inherent symmetry of System L lead to a new programming style?

[3] How can the close connection between System L and abstract machines be exploited in programming?

These investigations will also require mathematical principles to be developed. Specifically, we will

[4] Establish the Propositions-As-Types axis corresponding to internal languages (see Figure 2) relating System L to adjoints.

As for polarisation, we believe that programmers will be able to intuitively assimilate the eager *vs.* lazy modes of computation and data structure underlying it; but, pragmatically,

[5] How will a programmer be able to easily code polarisation in a programming language?

Once the above is sorted, one can consider experimenting with programming languages based on the various systems of Section (b-2.iii) further enriched with computational effects. To this end, there is a large body of theoretical work on effects that needs to be examined, evaluated, and reconsidered. An interesting stepping stone here is the recent work of Bauer and Pretnar [11] on the experimental programming language Eff.

Eff is based on the algebraic theory of effects and handlers of Plotkin *et al.* [96, 97]; but, for the pragmatics of supporting seemingly non-algebraic control operators, it goes beyond algebraic models. This is indeed so with respect to (first-order) algebraic theories. However, as it has transpired in conversation between Fiore and Staton, the consideration of the richer Second-Order Algebraic Theories of Fiore *et al.* [46, 47] seems to enrich the class of specifiable effects to incorporate aspects of control. Thus, our research into this topic will also

[6] Investigate Second-Order (and, when they are developed, Polymorphic) Algebraic Theories as a mathematical basis for an algebraic theory of effects encompassing control.

Targeted goals here will be program logics for effects and control, and formal correctness proofs of (continuation and/or abstract machine) implementations of effects.

b-4.iii Metaprogramming.

⇝ generic programming

⇝ reflection

⇝ TDPE with sums: reflection with delimited control operators

⇝ ***System F^*_ω: Tillmann Rendel, Klaus Ostermann, Christian Hofer (2009),Typed Self-Representation***

c Resources

Team. During the 2011-12 academic year, I was on sabbatical leave. Two important events for this grant proposal happen then as follows.

On the one hand, I was awarded a *Research in Paris* grant from the *programme d'accueil des chercheurs étrangers de la Ville de Paris* to visit Laboratoire PPS at Université Paris Diderot - Paris 7 for three months. There, I mainly interacted with Pierre-Louis Curien and his PhD student Guillaume Munch-Maccagnoni, establishing the Mathematical Logic axis of Figure 3.

On the other hand, Timothy Sheard (Department of Computer Science, Portland State University), who was also on sabbatical leave during the 2011-12 academic year, visited Microsoft Research Cambridge during October–December 2011 together with his PhD student Ki Yung Ahn who instead visited the Computer Laboratory (University of Cambridge). The Programming Language axis of Figure 3 was established then.

To complete the team, for the Type Theory axis of Figure 3 it was most natural to recruit Peter Dybjer (Department of Computer Science and Engineering, Chalmers University of Technology) and Nicola Gambino (Dipartimento di Matematica e Applicazioni, Università degli Studi di Palermo).

⇝ ***Makoto — Marco — Michael***

Principal investigator. As Principal Investigator, ***Marcelo Fiore*** would be concerned with all areas of the project and head the MaStrPLan team, comprising both Senior and Junior Researchers.

- For the Principal Investigator, salary is requested for eight months per year for the five years of the duration of the project.

The reduction acknowledges that the Principal Investigator will concentrate on the project while continuing with some teaching to attract new PhD students and still have some departmental administration duties.

Senior researchers. The Senior Researchers Pierre-Louis Curien, Peter Dybjer, and Timothy Sheard are renown researchers in their respective field of expertise.

- For the Senior Researchers, funding is requested for annual one-month research visits to the Computer Laboratory (University of Cambridge) for the five years of the duration of the project.

Our collaboration will however go beyond each visit; continuing by email, conference calls, and/or visits to their respective sites. Their academic credentials together with the specific areas of the project to which they will be engaged follow.

Pierre-Louis Curien. ...

Section	Keyword
(b-1.ii)	operads
(b-2.i)	dependent types, parametricity
(b-2.iii)	polarisation
(b-2.iv)	Linear Logic
(b-4.ii)	System L

Peter Dybjer. ...

Section	Keyword
(b-1.i)	Internal Type Theory
(b-2.i)	Dependent Type Theory
(b-2.ii)	NbE, glueing
(b-3.iii)	induction-recursion
(b-4.i)	Agda

Timothy Sheard. ...

Section	Keyword
(b-2.iv)	MetaML
(b-3.iii)	iterators
(b-4.i)	Nax, Omega
(b-4.ii)	PL design & implementation
(b-4.iii)	MetaML

Junior researchers. The Junior Researchers Ki Yung Ahn, Nicola Gambino, and Guillaume Munch-Maccagnoni will be based at the Computer Laboratory (University of Cambridge), working in the project full time. Ahn and Munch-Maccagnoni are currently finishing their PhD dissertations. Nicola Gambino is already an established researcher. All would be ready to join the project as it starts.

- Salary is requested to employ each Junior Researcher full-time for the five years of duration of the project.

Ki Yung Ahn. ...

Section	Keyword
(b-3.iii)	iterators
(b-4.i)	Nax
(b-4.ii)	Haskell

Nicola Gambino. ...

Section	Keyword
(b-1.ii)	generalised species
(b-2.i)	HoTT
(b-2.ii)	sheaves, Algebraic Set Theory
(b-3.i)	generalised species
(b-3.ii)	HoTT

Guillaume Munch-Maccagnoni. ...

Section	Keyword
(b-2.iii)	polarisation
(b-2.iv)	Linear Logic
(b-4.ii)	System L, delimited control

Additional costs.

- Travel funding is requested for attending workshops and conferences, and for visiting and/or inviting researchers.
- Further funds are requested for computer support in the form of laptops for the Junior Researchers and the cost of broadband for the Principal Investigator.

REFERENCES

1 S. Abramsky, E. Haghverdi and P. Scott (2002). Geometry of Interaction and linear combinatory algebras. *MSCS* 12.
2 M. Aguiar and S. Mahajan (2010). *Monoidal functors, species, and Hopf algebras*. CRM Monograph Series, AMS.
3 K.Y. Ahn and T. Sheard (2011). A Hierarchy of Mendler style Recursion Combinators. In *ICFP'11*.
4 A.A. Albert (1948). Power-associative rings. *Trans. Amer. Math. Soc.*, 64(3):552–593.
5 J.-M. Andreoli (1992). Logic programming with focusing proof in linear logic. *J. of Logic and Computation*, 2:297–347.
6 S. Artemov (2001). Explicit provability and constructive semantics. *Bulletin for Symbolic Logic*, 7(1):1–36.
7 M. Artin, A. Grothendieck, and J.-L. Verdier, eds. (1972). *SGA 4*, LNM 269.
8 S. Awodey, P. Hofstra and M. Warren (2012). Martin-Löf complexes. In `http://arxiv.org/abs/0906.4521`.
9 L. Augustsson (1998). Cayenne — a language with dependent types. In *ICFP'98*, pp 239–250.
10 A. Barber (1996). Dual Intuitionistic Linear Logic. PhD thesis, University of Edinburgh.
11 A, Bauer and M. Pretnar (2012). Programming with Algebraic Effects and Handlers.
12 J. Beck (1967). *Triples, algebras and cohomology*. *Reprints in TAC* 2, 2003.
13 A. Beckmann, S. Buss, and S. Friedman (2012). Safe Recursive Set Functions.
14 M. Beeson (1984). Foundations of constructive mathematics. Springer Verlag.
15 S. Bellantoni and S. Cook (1992). A new recursion-theoretic characterisation of the polytime functions. *Computational Complexity*, 2:97–110.
16 J. Bénabou (1967). *Introduction to bicategories*. In LNM 47, pp 1–77.
17 N. Benton and P. Wadler (1996). Linear logic, monads, and the lambda calculus. In *11th LICS*.
18 G. Birkhoff (1935). On the structure of abstract algebras. *P. Camb. Philos. Soc.*, 31:433–454.
19 C. Boehm and A. Berarducci (1985). Automatic Synthesis of Typed Lambda-Programs on Term Algebras. *TCS* 39.
20 G. Boole (1847). *The Mathematical Analysis of Logic*. Macmillan, Barclay, & Macmillan.
21 T Borghuis (1998). Modal Pure Type Systems: Type Theory for Knowledge Representation. *Journal of Logic, Language, and Information*, 7:265–296.
22 F. Chapoton and M. Livernet (2001). Pre-Lie algebras and the rooted trees operad. *Int. Math. Res. Not.*, 8:395–408.
23 A. Church (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363.
24 A. Church (1940). A formulation of the simple theory of types. *J. Symbolic Logic*, 5:56–68.
25 P. Cohen (1966). *Set theory and the continuum hypothesis*. W A Benjamin.
26 T. Coquand and G. Jaber (2012). A note on forcing and type theory. *Fundamenta Informatica* 100:43–52.
27 T. Coquand and G. Huet (1988). The Calculus of Constructions. *Information and Computation*, 76(2–3).
28 P.-L. Curien and H. Herbelin (2000). The duality of computation. In *ICFP'00*, pp 233–243.
29 P.-L. Curien and G. Munch-Maccagnoni (2010). The duality of computation under focus. In *IFIP TCS*.
30 H. Curry (1934). Functionality in Combinatory Logic. In *Proceedings of the National Academy of Sciences*, 20:584–590.
31 B. Day (1970). On closed categories of functors. In LNM 137.
32 N.G. de Bruijn (1968). Automath, a language for mathematics. In *Automation and Reasoning*, pp 159–200, 1983.
33 P. Dybjer (1994). Inductive families. *Formal Aspects of Computing*, 6:440–465.
34 P. Dybjer (1996). Internal Type Theory. In *TYPES*, LNCS 1158, pp 120–134.
35 P. Dybjer (2000). A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory. *JSL* 65.
36 J. Egger, R. Møgelberg, and A. Simpson (2009). Enriching an Effect Calculus with Linear Types. In *CSL'09*.
37 S. Eilenberg and S. Mac Lane (1945). General Theory of Natural Equivalences. *Trans. Amer. Math. Soc.*, 58(2):231–294.
38 A. Filinski (1994). Representing monads. In *POPL'94*.
39 M. Fiore (1995). Lifting as a KZ-doctrine. In *CTCS'96*.
40 M. Fiore (2002). Semantic analysis of Normalisation By Evaluation for typed lambda calculus. In *PPDP'02*.
41 M. Fiore (2005). Mathematical models of computational and combinatorial structures. In *FOSSACS'05*, LNCS 3441.
42 M. Fiore (2006). On the structure of substitution. MFPS XXII invited address.
43 M. Fiore (2012). Discrete Generalised Polynomial Functors. In *ICALP'12*, LNCS 7392, pp 214–226.
44 M. Fiore, R. Di Cosmo and V. Balat (2002). Remarks on isomorphisms in typed lambda calculi with empty and sum types. In *LICS'02*, pp 147-156.
45 M. Fiore, N. Gambino, M. Hyland, and G. Winskel (2008). The cartesian closed bicategory of generalised species of structures. *J. London Math. Soc.*, 77:203-220.
46 M. Fiore and C.-K. Hur (2010). Second-order equational logic. In *CSL'10*, LNCS 6247, pp 320–335.
47 M. Fiore and O. Mahmoud (2010). Second-order algebraic theories. In *MFCS'10*, LNCS 6281, pp 368–380.
48 M. Fiore, G. Plotkin and A.J. Power (1997). Complete cuboidal sets in axiomatic domain theory. In *LICS'97*.
49 G. Frege (1879). *Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought*. In *[113]*, pp 1–82.
50 G. Frege (1893/1903). *Grundgesetze der Arithmetik*. Vols. I/II.
51 G. Gentzen (1935). Untersuchungen über das logische Schließen. Mathematische Zeitschrift 39:176–210.
52 M. Gerstenhaber (1963). The cohomology structure of an associative ring. *Annals of Mathematics*, 78(2):267–288.
53 J.-Y. Girard (1972). *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thèse de doctorat d'état, Université Paris VII.
54 J.-Y. Girard (1987). Linear logic. *TCS*, 50:1–101.
55 J.-Y. Girard (1991). A new constructive logic: Classical logic. *Math. Struct. Comp. Sci.*, 1(3).
56 A. Grothendieck and M. Raynaud (1971). *SGA 1*, LNM 224.
57 D. Hilbert (1927). The foundations of mathematics. In *[113]*, pp 464–479.
58 R. Hindley (1969). The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.*, 146:29–60.
59 M. Hofmann and T. Streicher (1998). The groupoid interpretation of type theory. In *Twenty-five years of constructive type theory*, pp 83–111.
60 W. Howard (1969). The formulae-as-types notion of construction. In *[111]*, pp 479–490.
61 G. Jaber, N. Tabareau and M. Sozeau (2012). Extending Type Theory with Forcing. In *LICS'12*, pp 395–404.
62 P. Johnstone (2002). Sketches of an Elephant: A Topos Theory Compendium. Oxford University Press.
63 A. Joyal (1981). Une théorie combinatoire des séries formelles. *Advances in Mathematics*, 42:1–82.
64 A. Joyal (1986). Foncteurs analytiques et espèces de structures. *Combinatoire Énumérative*, LNM 1234, pp 126–159.
65 D. Kan (1958). Adjoint functors. *Trans. Amer. Math. Soc.* 87.
66 G.M. Kelly (1982). *Basic Concepts of Enriched Category Theory*. *Reprints in TAC* 10, 2005.
67 G.M. Kelly (1972). On the operads of J.P. May. *Reprints in TAC* 13, 2005.
68 S. Kobayashi (1997). Monad as modality. In *TCS*, 175:29–74.
69 J.-L. Krivine (2011). Realizability algebras: a program to well order R. *LMCS*-7(3:2).
70 Y. Lafont (1990). Interaction nets. In *POPL'90*.
71 J. Lambek (1968). Deductive systems and categories I. *J. Math. Systems Theory*, 2:278–318.
72 J. Lambek (1969). Deductive systems and categories II. LNM 86, pp 76–122.
73 J. Lambek and P. Scott (1986). *Introduction to Higher Order Categorical Logic*. Cambridge University Press.
74 F.W. Lawvere (1963). Functorial Semantics of Algebraic Theories. *Reprints in TAC* 5, 2004.
75 F.W. Lawvere (1973). Metric spaces, generalized logic and closed categories. *Reprints in TAC* 1, 2002.
76 T. Leinster (2004). *Higher Operads, Higher Categories*. LMS Lecture Note Series 298, Cambridge University Press.
77 P. Levy (2005). Adjunction models for Call-By-Push-Value with stacks. *TAC*, 14(5):75–110.
78 F. Linton (1966). Some aspects of equational theories. In *Proc. Conf. on Categorical Algebra at La Jolla*, pp 84–95.
79 J.-L. Loday and M. Ronco (2012). Permutads.

80 J.-L. Loday and B. Vallette (2012). *Algebraic Operads*. Grundlehren der mathematischen Wissenschaften.
81 S. Mac Lane (1945). Categories for the Working Mathematician (Second Edition). Springer Verlag, 1998.
82 M. Markl (2006). Operads and PROPs.
83 J.P. May (1972). *The geometry of iterated loop spaces*. LNM 271.
84 C. McBride. Epigram: practical programming with dependent types. In *AFP'04*, pp i130–170.
85 P.-A. Melliès (2009). Categorical Semantics of Linear Logic. In *Panoramas et synthèsis*, 27:1–196.
86 P.-A. Melliès and N. Tabareau (2010). Resource modalities in tensor logic. *APAL*, 161(5):632–653.
87 M. Mendler and S. Scheele (2011). Cut-free Gentzen calculus for multimodal CK. *Information and Computation*, 209(12):1465–1490.
88 N. Mendler (1991). Inductive types and type constraints in the second-order lambda calculus. *APAL*, 51(1–2):159–172.
89 R. Milner (1978). A Theory of Type Polymorphism in Programming. *J. of Computer and System Sciences*, 17:348–375.
90 E. Moggi (1991). Notions of computation and monads. *Information And Computation*, 93(1).
91 G. Munch-Maccagnoni (2009). Focalisation and classical realisability. In *CSL'09*, LNCS 5771, pp 409–423.
92 B. Nordström, K. Petersson, and J. Smith (1990). Programming in Martin-Löf Type Theory: An Introduction. OUP.
93 S. Park and H. Im (2011). A modal logic internalizing normal proofs. *Information and Computation*, 209:1519–1535.
94 G. Plotkin (1975). Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159.
95 G. Plotkin (1977). LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255.
96 G. Plotkin and A.J. Power (2003). Algebraic operations and generic effects. *Applied Categorical Structures* 11.
97 G. Plotkin and M. Pretnar (2009). Handlers of algebraic effects. *Programming Languages and Systems*, pp 80–94.
98 J. Polakow and F. Pfenning (1999). Natural Deduction for Intuitionistic Non-Commutative Linear Logic. In *TLCA'99*.
99 J. Reynolds (1983). Types, abstraction and parametric polymorphism. *Information Processing* 83, pp 513–523.
100 M. Ronco (2008). Shuffle bialgebras.
101 B. Russell (1902). Letter to Frege. In *[113]*, pp 124–125.
102 B. Russell (1903). *The principles of mathematics*. Cambridge University Press.
103 D. Scott (1967). A proof of the independence of the continuum hypothesis *Theory of Computing Systems*, 1(2):89–111.
104 D. Scott (1969). A type-theoretical alternative to ISWIM, CUCH, OWHY In *TCS*, 121(1–2):411–440, 1993.
105 M. Schönfinkel (1924). On the building blocks of mathematical logic. In *[113]*, pp 355–366.
106 R. Seely (1989). Linear logic, *-autonomous categories and cofree coalgebras *Contemporary Mathematics* 92.
107 P. Selinger (2011). A survey of graphical languages for monoidal categories. In LNP 813, pp 289–355.
108 T. Sheard (2004). Languages of the future. In *SIGPLAN Notices*.
109 C. Strachey (1967). Fundamental Concepts in Programming Languages. In *HOSC*, 13:11–49, 2000.
110 M. Szabo (1975). Polycategories. *Communications in Algebra*, 3(8):663–689.
111 J. Seldin and J.R. Hindley (1980). *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.*
112 A. Turing (1937). On computable numbers, with an application to the Entscheidungsproblem. *Proc. LMS*, 42:230–265.
113 J. van Heijenoort, ed. (1967). *From Frege to Gödel: A source book in mathematical logic, 1879–1931*.
114 J. von Plato (2012). Gentzen's proof systems: byproducts in a work of genius. *Bull. Symbolic Logic*, 18(3):313–367.
115 D. Vytiniotis, S. Peyton Jones, and J.P. Magalhaes (2012). Equality proof and deferred type errors. In *ICFP'12*.
116 P, Wadler (2003). Call-by-value is dual to call-by-name. In *ICFP'03*, pp 189–201.
117 The Haskell Programming Language. `http://www.haskell.org/haskellwiki/Haskell`.