The Scientific Proposal

a State of the art and objectives

We propose research in logical methods for computer science. Specifically, in formal languages and mathematical models for computer programming and mathematical proof.

The research is driven by the thesis that

> Languages for computer programming and languages for mathematical proof are of the same character.

This view started developing in the 1960s and is now central to research in programming languages and constructive mathematics.

An aspect of the thesis that is fundamental to us here is usually referred to as the Proofs-as-Programs or Propositions-as-Types correspondence [25, 51]. It can be impressionistically presented as follows

$$\text{proof} : \text{Proposition} \quad \approx \quad \text{program} : \text{Type}$$

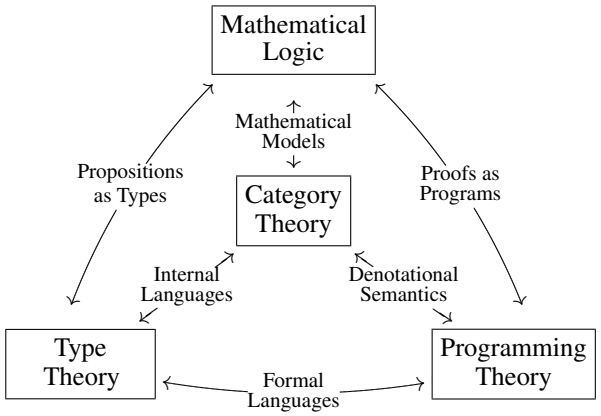
and one intuitively argues for it along these lines: to give a constructive proof of a proposition is to construct a program that certifies the statement, while to type a program is to establish a property of it.

Research stemming from the Propositions-as-Types correspondence has been very fruitful. Many of its mathematical theories have been implemented as computer systems, and used academically and industrially. Cases in point are: programming languages for high-assurance code and proof assistants for computer-aided verification; examples of which are Haskell, an industrial-strength functional programming language, and Coq, a proof assistant in which bodies of mathematics are being verified.

The developments of transfer from mathematical theories to computer systems have required technical/theoretical and technological/applied progress. This has necessarily demanded increasingly specialised, and to some extent fragmentary, research. Importantly, the field has reached a first plateau; encompassing a substantial body of work and expertise. How will it continue to advance? By considering new directions and challenges, of course. But, it is our firm view that, as in the early days of the subject [27, 62, 88, 67], it will be essential to do so generating questions and tackling problems from the perspectives of a variety of research areas. It is both of these aspects that we put forward here.

Two disciplines are clearly involved in investigating the Proofs-as-Programs correspondence: Mathematical Logic and Programming Theory. This is however only half of the picture. The full strength of the correspondence involves also the disciplines of Type Theory and Category Theory. Figure 1 gives a schematic view of the interactions between these four research areas in this context. Each of them is complementary to the others. Together, as a unified whole, they have shaped fields of computer science and mathematics. It is within this framework that we propose cross-cutting research in Category Theory, Mathematical Logic, Programming Theory, and Type Theory. We contend that an approach neglecting any one of them is to the detriment of the others; missing the depth and richness of the subject and, crucially, missing opportunities for research and development.

Figure 1 Research areas and interactions.



The motivations for our research proposal are specifically as follows.

- ★ To search for the next-generation framework of Type Theory relevant to computer science and mathematics.
- ★ To exercise and test Category Theory both as a unifying and as a foundational mathematical language.
- ★ To broaden the current role of Mathematical Logic in computation.
- ★ To design, implement, and experiment within Programming Theory, looking into the languages of the future.

We particularly aim to contribute with: a framework to analyse and synthesise type theories; mathematical theories and models for computational phenomena; formal calculi for proof and/or computation; and experimental high-level programming languages.

Principal Investigator Marcelo Fiore has conceived the research programme of this proposal, and assembled the team, to precisely target these goals.

a-1 Origins and influences

This section outlines the origins and influences that led to the holistic conception of Figure 1. It is important to appreciate and understand our scientific framework.

a-1.i Logic and computation. Computer science was born as a branch of mathematics, specifically mathematical logic, even before the first electronic computers were built. Its inception was Hilbert's 1928 Entscheidungsproblem (decision problem), asking whether there is an algorithmic procedure for deciding mathematical statements. Negative answers were provided independently by Church [18] in 1936

and Turing [94] in 1937, giving birth to the mathematical theory of computation. Their completely different approaches gave rise to different branches of theoretical computer science that still persist today. On one hand, the line of development starting with Church's λ-calculus concerns itself with prototypical computational languages that are used to study 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 this proposal. 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 [14]. 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).

The λ-calculus syntax is given by

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

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 conceived to solve a foundational problem. In [44], building on [43], Frege proposed a logical system as a foundation for mathematics including arithmetic. In his now famous paradox, Russell [84] observed that one of Frege's axioms led to inconsistency. He was led to this contradiction by related contradictions he found while developing his account of set theory. To overcome such foundational problems, Russell introduced the 'doctrine of types' [85, Appendix B] — typed systems can avoid these inconsistencies.

Type Theory, as we now know it, arose from the integration by Church of types into his λ-calculus [19]. This was a natural step, particularly if one held the naive interpretation of λ-abstraction as defining functions. The type theory known as the Simply-Typed Lambda Calculus has a set of types consisting of basic ones closed under a function-type constructor:

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

In the Simply-Typed Lambda Calculus, λ-terms t are classified by 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

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

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

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

A fundamental discovery relating Mathematical Logic and Type Theory was made by Curry [25] and Howard [51] in two slightly different, though related, contexts (technically, Hilbert-style deduction and Gentzen's natural deduction in sequent form [45]). We will refer to this discovery as the Propositions-as-Types correspondence. Roughly, it establishes a correspondence between terms having types and proofs proving propositions. For the Simply-Typed Lambda Calculus, the correspondence can be illustrated by noting that the erasure of term information in the typing rules (given above) yields the deduction rules

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

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

of Intuitionistic Logic.

In another very important direction Church introduced a Simple Theory of Types [19] as an axiomatization of Higher-Order Logic. This axiomatization was formalised within a Simply-Typed Lambda Calculus with basic types for both individuals and propositions, further 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 a 'language' to a 'metalanguage'; *ie.*, a language in which it is possible to represent and work with other 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 computational β-rule (1) stated as an equation together with the extensionality equation

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

The processes of abstracting from languages to metalanguages has become a common activity in computer science, and plays 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 [32]. A category is a mathematical structure consisting of objects and morphisms. Morphisms are classified by pairs of objects. 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 together with, for every object, an identity morphism that is a neutral element for composition.

In understanding categories, it is helpful to have in mind that they have two kinds of uses: 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.

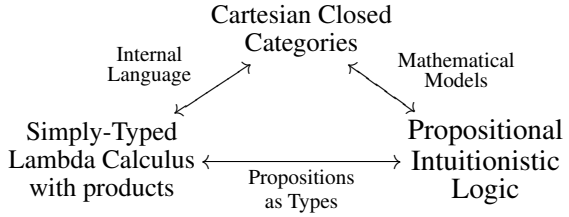
Category Theory analyses mathematical structure by isolating the principles for which mathematical theories work. Because of this attention to essentials, it has had considerable success in unifying ideas from many areas of mathematics. The connection between Category Theory, Logic, and Type Theory was initiated by Lawvere [62] and Lambek [59], 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 [56] 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 $\times$ and $\Rightarrow$, defined by adjoint situations establishing natural bijective correspondences between morphisms as follows:

$$\frac{C \to A \ , \ C \to B}{C \to (A \times B)} \qquad \frac{(C \times A) \to B}{C \to (A \Rightarrow B)}$$

The required naturality condition amounts to the computational β laws and the extensionality 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 the trinity below



(3)

and is the standard with respect to which analogous developments in the area are measured.

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 [50]. Independently, Milner [74] solved it in a context more relevant to programming; namely, for the Simply-Typed Lambda Calculus with parametric polymorphism. The algorithm is known as the Hindley-Milner type inference method. The approach is very robust, extending broadly to many related systems.

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 [92], who further classified the phenomena into 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 [83] 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 [46] 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 Simply-Typed Lambda Calculus, notably by the possibility of encoding inductive data types [15]. System F_ω lies at the core of the Haskell programming language.

The type systems mentioned above aim at providing logical foundations. When viewed as pro-

[1] Combinatory Logic is an important symbolic formalism introduced by Schönfinkel [89] 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.

gramming languages, they 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 [87] introduced an extension of Typed Combinatory Logic with a fixpoint combinator for general recursion. Plotkin [80] 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 [79], as in ML or Haskell.

The influence of mathematical models 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 each subsection below we identify broad questions and/or pathways for research. In Section (b) we return to them, point by point, and construct a detailed research plan.

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 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 Dependent types. Dependent type theory is a formalism introduced by de Bruijn [27] that extends 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.* [54]). 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 or extensional.

A fundamental problem in the area is to:

Research pathway see (b-2.i)
- Investigate equality and identity in dependent type theory.

This lays at the core of our 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.iii 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 [6] (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 [55]; a topos being a universe of discourse for Higher-Order Intuitionistic Logic [60].

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 [21] (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 [86].

a-2.iv 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 algebraic datatypes (GADTs) in functional programming [91, 98]; and the experimentation with dependently-typed programming languages [8, 69] 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.v Resources, effects, modalities. In the late 1980s, two important analyses of computation, respectively for resources and effects, were proposed by Girard [47] and by Moggi [75]. 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 [11].

A classical basic result of category theory establishes a strong correspondence between (co)monads and adjoint functors (see *eg.* [66]). One aspect of this is that every adjunction

$$\mathcal{D} \underset{\mathsf{F}}{\overset{\mathsf{G}}{\rightleftarrows}} \mathcal{C} \qquad (4)$$

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 [66, 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.* [70]). On the other hand, models of Computational λ-calculi rely on enriched category theory [57]. 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 [13] and of Barber and Plotkin [9]. In the context of effect calculi, examples are the Call-By-Push-Value of Levy [64] and the Enriched Effect Calculus of Egger, Møgelberg, and Simpson [31]. 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 structures. 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.* [58]). 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.

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

a-2.vi 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.* [96]).

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 formalism that is proving robust in applications seems to be emerging. This will be referred to as the calculus, formalism, or system L; and underlies the ones developed in *eg.* [23, 97, 76, 24].

The main novel features of the formalism L, and its philosophy, are an intrinsic symmetry (corresponding to the computation roles of program and environment) with syntax that reflects an adjoint situation and is closely connected to abstract machines (which are in fact internal to the calculus). From this perspective we ask:

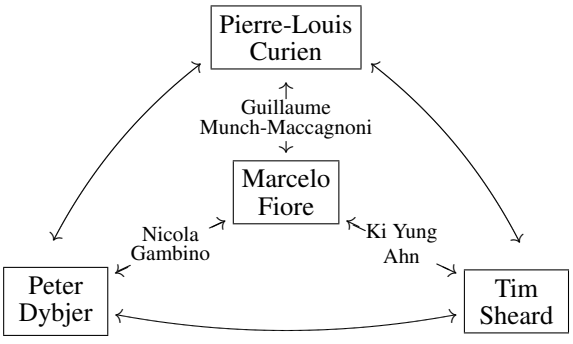
Research question see (b-4.ii)
- What can the proof theory of sequent calculi do for programming?

a-3 Research objectives

The general aim of the project is to build a group dedicated to combined research in Category Theory, Mathematical Logic, Programming Theory, and Type Theory. Specifically, to understand and develop formal languages and mathematical models for computer programming and mathematical proof. 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 Visiting Researchers Pierre-Louis Curien, Peter Dybjer, and Tim Sheard; and Research Associates Ki Yung Ahn, Nicola Gambino, and Guillaume Munch-Maccagnoni. Figure 2 gives a schematic presentation of the team that matches the research areas as rendered in Figure 1. Details are deferred to Section (c).

Figure 2 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 address the fundamental question of what type theories are, which will also lead us to rethink them. Our approach is novel and ambitious in that it aims at an algebraic framework that will generalise to type theories all aspects of our current understanding of algebraic theories. The scale at which we will be attempting this is unprecedented, systematically exploring a wide spectrum of key type-theoretic features.

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

We will tackle semantic problems at the forefront of current understanding. In the context of type theories, we will conduct investigations around a main open problem in the area: to reconcile extensional equality and intensional identity (which roughly correspond to the mathematical and computational notions of sameness). In the context of logical systems, we will develop novel calculi suggested by mathematical models and new models for so far intractable calculi encompassing aspects of resource management and computational effects. A distinguishing novelty of our approach is that it will be informed by the logical notion of polarisation.

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

We will aim at evolving the term and type structure of current type theories, which in essence has not changed since the late 1960s. Pursuing the view that further evolution will come from mathematical input, we will research type theories as formal languages of (higher-dimensional) categorical structures.

4 Programming: *Design and implementation of novel computational languages.*

Guided by mathematical theories and pragmatics, we will look into the engineering of principles and concepts for programming, exporting them to language designs and implementations. We will specifically target experimental languages, supporting indexed data structures, computational effects, and metaprogramming.

b Methodology

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

FINISH

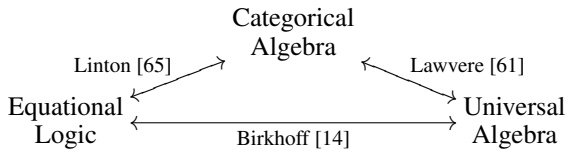
b-1 Foundations

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 [14], 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 3.

Figure 3 Algebraic Trinity



Equational Logic is the metalanguage of algebraic theories, while Universal Algebra is its model theory. They were both introduced by Birkhoff [14], 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 [61] and Linton [65]. 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).

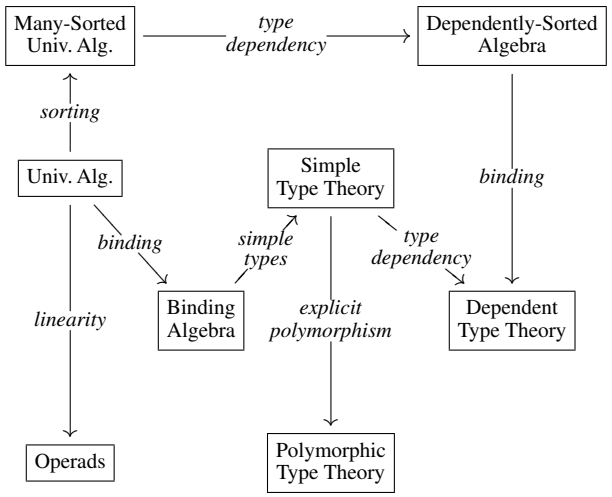
Each of the approaches to algebraic theories in the Algebraic Trinity gives a different viewpoint of the subject. Thus, it is important to consider them all. For instance, Equational Logic provides the deductive system for formal reasoning about equations from axioms; Universal Algebra provides a general notion of model from which one derives an abstract notion of syntactic structure by means of freely generated models; 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 4, where the *linearity* dimension could be transported along or mixed with all the other ones.

This research rests on fundamental work, old and recent, of Fiore *et al.* [42, 35, 40, 39, 41], that started with the first algebraic model of abstract syntax with variable binding and progressively led to an extension of the Algebraic Trinity to the realm of Binding Algebra (*viz.* algebraic languages with binding operators). The further extension to simple type theories should follow along similar lines further taking into account the algebraic structure of types. As for the other points

Figure 4 Algebraic Type Theory research space



of the research space in Figure 4, 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-2 Models

b-2.i Equality and identity in dependent type theory. We have briefly mentioned intensional and extensional type theory in Section (a-2.ii). This section examines these two flavours of type theory and presents a research programme geared to investigate the main open problem in the area:

[1] Reconcile Intensional and Extensional Type Theory in a computational framework.

Various aspects of what follows arose and are under discussion with Warren (IAS School of Mathematics, Princeton).

Central to a type theory is its associated equational theory. Equational theories of dependent type theories may be specified in different (though typically equiv-

alent) forms (see *eg.* [1]). For the purpose of our informal discussion here it will be enough to assume that they determine a congruence relation henceforth referred to as judgmental equality.

In Pure Intensional Type Theory, judgemental equality is generated by β rules of computation (recall Section (a-1.ii)). As for Extensional Type Theory, the terminology will be used here to refer to extensions of Pure Intensional Type Theory with extensionality features. In their simplest form, these consist of incorporating η rules of extensionality (recall Section (a-1.ii)) into judgemental equality. Such extensions will be referred to here as Weak Extensional Type Theory.

Already for these basic kind of systems, our knowledge of the subject is limited. For instance, the concrete mathematical models (as opposed to syntactic ones) of Pure Intensional Type Theory that researchers work with (like groupoids, strict ω-categories, and simplicial sets) interpret dependent sums and dependent products as adjoints and hence are necessarily models of Weak Extensional Type Theory. A first question arises:

[2] Are there natural mathematical models of Pure Intensional Type Theory?

In examining this one should also consider:

[3] What is a model of Pure Intensional Type Theory?

In this respect, we note that the current categorical formalisms (see *eg.* [54]) are not flexible enough (as they are either based on adjoint interpretations or on ad hoc variations thereof). On the other hand, the approach of Dybjer on Internal Type Theory [29] will provide models within Cartmell's Generalised Algebraic Theories [17]. Their relationship to developments in the Algebraic Type Theory of Section (b-1.i) will be investigated.

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 [77]. 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 thus important to understand the interaction of modularly extending type theories, central to which is the notion of conservative extension. Informally, an extension T' of T is conservative if the restriction of the judgemental equality of T' to T coincides with the judgemental equality of T.

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

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

This will lead to sophisticated model constructions that will deepen our understanding of the subject. In particular because it was already the case in proving the conservative extension of the Simply-Typed Lambda Calculus extended with (non-dependent) extensional strong sums [37].

The discussion of further notions of extensionality requires the consideration of Identity Types

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

that are meant to provide a notion of equality internal to the type theory.

In Strong Extensional Type Theory, one requires that Identity Types completely internalise judgemental equality (see [68]). Since for rich type theories this leads to undecidable type checking, Pure Intensional Type Theory has become the foundational core of proof assistants for constructive mathematics; like Automath, Coq, and Agda. However, as extensionality lays at the core of mathematics and its practice, the quest for a computational framework in between Pure Intensional Type Theory and Strong Extensional Type Theory continues.

In this direction, Martin-Löf proposed a notion of Identity Type (see *eg.* [77]) which is given as an inductively generated family [28] with a reflexivity constructor. These Identity Types are currently receiving renewed special attention, stemming from connections with homotopy theory and higher-dimensional category theory [7]. For them some fundamental matters are still to be understood. For instance,

[5] Does extending (Pure Intensional or Weak Extensional) Type Theory for dependent sums and dependent products with Identity Types produce a conservative extension?

Also, 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 this is not so for the latter.

A recent interesting approach to extensionality being intensively investigated is the extension by Voevodsky of Pure Intensional Type Theory with a so-called Univalence Axiom motivated by homotopical foundations. This axiom has been shown consistent from the outset, but:

[7] Is the extension of (Pure Intensional or Weak Extensional) Martin-Löf Type Theory with the Univalence Axiom conservative?

It is an open problem, referred to as Voevodsky's Main Computational Conjecture, as to how to give a constructive interpretation of the Univalence Axiom. The question arises as to whether one could circumvent this problem by concentrating instead on a notable consequence of the axiom; namely, that it somehow endows Identity Types with logical character, very roughly in that the Identity Type constructor logically commutes with all the other constructors up to equivalence. Therefore we will rather aim to:

[8] Design a constructive type theory with built-in

Logical Identity Types.

In this context, we will:

[9] Investigate possible connections to Strachey's notion of parametric polymorphism [92] in computer science as formalised by Reynolds [83] 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.iii), 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 [55].

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.

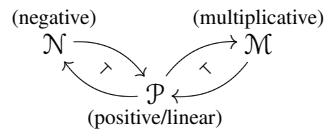
Initial work along this line has been done by Jaber, Tabareau, and Sozeau [53], who formalised the presheaf construction on a partial order in the Calculus of Constructions, and then re-interpreted back the Calculus of Constructions in the internal presheaf model. However, not only a wide spectrum of 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 [26]), and thereby introduces new linear structure. This is important in many applications, for which the reader may consult [34], that will be at the core of item **b-2.ii [3]** above.

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.v)). This research is work under discussion with Curien and Munch-Maccagnoni (Laboratoire PPS, Université Paris Diderot - Paris 7). A distinguishing novelty of our approach is that it will be pursued as informed by the logical notion of polarisation.

The notion of explicit polarisation in logical systems was introduced by Andreoli [4] 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 [48]. 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 structures.

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



(5)

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

When the effect structure is trivial, the above restricts to models of Linear Logic [70]; while, when the resource structure is trivial, one recovers the models of Call-By-Push-Value [64]. One is thus led to the following programme.

[1] Devise a sound and complete logical system for the model theory (5), and relate it to existing polarised systems [76, 24].

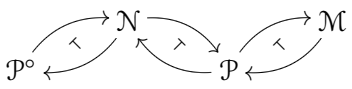
[2] A first model-theoretic result shows that the effect structure on $\mathcal{P}$ lifts to one on $\mathcal{M}$. Show that this corresponds to an encoding of Call-By-Push-Value in the devised logical system.

[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 [81], and (ii) revisiting and refining Filinski's result on representing monads [33] that reduces general monadic effects to the storage and escape effects.

[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 (5) can be further specialised to models with enough internal structure so as to extend

the picture as follows



(6)

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 (6) 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 [71], where the effect structure is collapsed. Our programme for (5), *ie.* items **b-2.iii [1–4]**, will be then also pursued for (6) 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 theoretically, 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 [66]. In particular, the Kleisli category of a computational monad provides a model of call-by-value [75]; while the coKleisli category of a linear exponential comonad provides one of call-by-name (*ie.*, a cartesian closed category) [90].

The programming languages corresponding to the metalanguages of the polarised models (5) and (6) 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 challenging. 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 than that of a category.

[6] What would then be the mathematical theory of universal constructions for these structures giving rise to logical connectives? Note again from considering LC that 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. Recall from Section (a-2.v) that 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.

Here, we are specially interested in grounding it through categorical models (see Figure 1), that have 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 [16]; Artemov's Logic of Proofs [5] and related systems; Mendler's Multimodal CK [72]; and Park and Im's Calculus S_{Δ} [78].

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	

Surprisingly, this has not changed since de Bruijn's Automath in the late 1960s. We firmly 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.

b-3.i Generalised Type Theory. We present work discussed with Gambino (DMI, Università degli Studi di Palermo) and Hyland (DPMMS, University of Cambridge) with whom the development will be undertaken.

Section (b-2.ii) considers presheaf categories in the large as mathematical universes of discourse (or gros 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) [12]. Lawvere [63] 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.

Success in this endeavour will yield a system where the mathematical calculations done with profunctors

(*eg.* in the context of the coherence of profunctor composition [12], the mathematical theory of substitution [34], generalised species of structures [38], and generalised polynomial functors [36]) can be established formally.

Intuitively, the coend and end constructions mentioned in the table above are 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]** to a fully-fledged, possibly higher-dimensional, dependent type theory.

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

$$\mathrm{Id}_T(x,y)$$

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

$$\mathrm{Di}_T(x,y)$$

would be to classify the possible ways in which an element of a type T may evolve to another one in a possibly irreversible manner; as it happens, for instance, in computation. Thereby leading to not necessarily equivalent types $\mathrm{Di}_T(x,y)$ and $\mathrm{Di}_T(y,x)$, 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 that allow 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 $\mathrm{Di}_T(x,y)$ and $\mathrm{Di}_{T^\circ}(y,x)$ 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 [49] and aim to:

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

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, use, and distribution.

Three directions for this research are presented. These will be considered as 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.iv)).

The main problem to be addressed here is to:

[1] Design a language for programming indexed data structures that will inform the design of the next generation of programming 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.

Most of the work in this area has concentrated on exploring the top-down approach (see *eg.* [8] and [69]). Here we will pursue research on the bottom-up approach, which has not been explored systematically. A strong pragmatic reason for this is that our uppermost interest is in a language targeted to programmers. This does not mean that we are not interested in the activity of proving as in constructive proof assistants. However, our main goal, under the Propositions-as-Types paradigm, is:

[2] To prove properties as a by-product of programming; rather than to extract programs from proving properties.

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 $\mathsf{F}_\mathfrak{i}$: an extension of System F_ω with static type-indexing structure.

System $\mathsf{F}_\mathfrak{i}$ 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 the main feature pulling us away from traditional dependently-typed formalisms.

The logical consistency of System $\mathsf{F}_\mathfrak{i}$ 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 [73], 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 [28]. 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 termination checking.

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

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

Indeed, this is the main open problem in dependently-typed programming language theory.

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 the calculus L mentioned in Section (a-2.vi). A first main question that arises is:

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

For instance,

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

[3] How can the close connection between the system L and abstract machines be exploited? For instance, in formal compiler correctness.

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

[4] Establish a Propositions-as-Types trinity (see (3)) between classical sequent calculi and computational languages with effects and rich type structure through adjoint interpretations.

As for polarisation, we believe that programmers will be able to intuitively assimilate the eager *vs.* lazy modes of computation and data structures 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 [10] on the experimental programming language Eff.

Eff is based on the algebraic theory of effects and handlers of Plotkin *et al.* [81, 82]; 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. Interestingly, as it has transpired in conversation between Fiore and Staton, the consideration of the richer Second-Order Algebraic Theories of Fiore *et al.* [40, 41] 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 programming languages with 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

feel free to change any of this – Tim

In a metaprogramming system there are two languages of interest. The object-language which is the object of study and the meta-language which manipulates object programs as data. The purpose of the meta-language is to define algorithms for the purpose of constructing or analyzing object-programs. Metaprograms play a role in many kinds of systems

Generic programming. One style of generic programming defines a datastucture that reflects how users define new types. For example, a value of type `Def Tree` describes how a new `Tree` datatype is defined. A generic program then analyzes this structure of `Def` types to write code parametric over any user defined type. For example a generic equality might have type `Def t -> t -> t -> Bool` and a generic marshalling function might have type `Def t -> t -> List Bit`.

Reflection. One kind of reflection is where the object-language and the meta-language coincide

c Resources

Principal investigator. As Principal Investigator (PI), ***Marcelo Fiore*** would be concerned with and engaged in all areas of the project, heading the team.

- For the PI, salary is requested for eight months (75%) per year for the five years of the duration of the project.

The reduction acknowledges that the PI will concentrate on the project while continuing with some teach-

ing to attract new PhD students and still have some departmental administration duties.

Senior Visiting Researchers. The Senior Visiting Researchers (SVRs) Pierre-Louis Curien, Peter Dybjer, and Tim Sheard are renown researchers in their respective field of expertise.

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

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

Pierre-Louis Curien is a CNRS researcher at Laboratoire PPS, Université Paris Diderot - Paris 7, a laboratory dedicated to research on logic in computer science that he established in 1998 and directed thereafter for 10 years.

Curien's field of research is programming-language semantics, with contributions that have ranged broadly into Category Theory, Logic, and Type Theory. In direct relation to the present proposal, his work includes influential studies in: the implementation of functional programs, parametricity in polymorphism, substitution in abstract machines, and coherence in type theory.

Curien made a very early contribution to the transfer of ideas from category theory to programming languages that turned out to be of practical importance. He observed that the interpretation of the λ-calculus in a cartesian closed category could be seen as a compilation into a 'low-level' language of categorical combinators, that could be then executed in a Categorical Abstract Machine [22]. Since then he regularly returned to the theme of abstract machines, most recently within the polarised version of the duality of computation [24] in the setting of system L, which features in Sections (a-2.vi & b-4.ii).

Peter Dybjer is Professor and Head of the Division of Computing Science in the Department of Computer Science at Chalmers University of Technology.

Dybjer's main research area is Type Theory, including its connections to Category Theory and Logic, and its implementation in proof assistants. He has made several fundamental contributions to the field, especially to Martin-Löf's Intuitionistic Type Theory.

Many of Dybjer's contributions are of direct relevance to this proposal. For instance: (i) his notion of Category with Families [29], a categorical notion of model of dependent types with a close connection to the syntax (which features in Section (b-2.i)); (ii) his exploration of the syntax and semantics of Inductive Families in type theory [28] (which features in Section (b-4.i)); (iii) his result with Clairambault [20] properly establishing the biequivalence between Martin-Löf type theory and locally cartesian closed categories (which is relevant to Sections (b-1.i & b-2.i)); (iv) his powerful notion of an inductive-recursive definition [30], that definitions and is useful for understanding universes and reflection principles in type theory (and is relevant to Sections (b-4.i & b-4.iii)).

Dybjer is a member of the team at Chalmers which develops the dependently typed programming language (and proof assistant) Agda.

Tim Sheard is a professor of computer science at Portland State University in Portland Oregon. Tim's interests include programming language design, functional programming, and the intersection of programming and logic. He is best known for a number of experimental languages, which have been widely distributed, and whose main ideas have been incorporated into other languages. Perhaps the best known example is MetaML[102], a staged language, where programs write programs which are run in later stages. The best ideas of MetaML have been incorporated into the O'Caml system and into the GHC in its Template Haskell[101] extensions. Another example is the language Omega, which pioneered the use of GADTs[99, 100] and indexed types (the theory of which is discussed in section **b-4.i** and on which the new language Nax is based). These features are now supported in many other languages. The most recent example is FunLog which supports the high level embedding of external decision procedures (such as SAT solvers, SMT solvers, Integer programming tools, and Markov Logic Networks) into a host language.

Research Associates. The Research Associates (RAs) 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 from the start.

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

Ki Yung Ahn is a PhD candidate at the Department of Computer Science, Portland State University. He is currently writing up his dissertation on the design and implementation of Nax, a language for indexed programming. This topic is at the core of Section (b-4.i); as so is his published work [3] on Mendler-style recursion schemes.

Ahn has been actively contributing to the Haskell community. His contributions include the open-source Haskell libraries yices and logic-TPTP, a publication in the Haskell symposium [2], and a Korean translation of Hutton's Haskell programming textbook [52].

His general interest is to promote research on programming-language theory into real-world software development practices. This is important for the proposed investigations in Sections (b-4.ii & b-4.iii).

Nicola Gambino is University Researcher in Mathematical Logic at the University of Palermo since December 2008. He has a consistent record of publications in leading journals in theoretical computer science and mathematics, and has been one of the plenary invited speakers at the 2010 Logic Colloquium. He has held visiting positions at several research institutions. The most recent one, as a member of the IAS School of Mathematics (Princeton) in the Autumn of 2011. There, he worked in contact with Voevodsky on Homotopy Type Theory, one of the areas of the proposed research (see Section (b-2.i)).

Dr Gambino has expertise and experience in Type Theory and Category Theory, two of the research areas of the proposal (see Figure 1). During his EPSRC Postdoctoral Fellowship at the DPMMS, University of Cambridge, he collaborated successfully with the PI on a project that forms part of the proposed research in Section (b-3.i).

Guillaume Munch-Maccagnoni is a PhD student at Laboratoire PPS, Université Paris Diderot - Paris 7. He is currently writing up his dissertation, where he contributes to the Propositions-as-Types correspondence in the context of classical logic and calculi with effects and rich type structures. In [76], he gave an original way of representing proofs in linear logic (including the modalities) in a way that turns out to match closely the model theories (5) and (6). This is relevant to the proposed investigations in Section (b-2.iii), and for further research on modalities as proposed in Section (b-2.iv).

Munch-Maccagnoni visited the PI at the Computer Laboratory, University of Cambridge, during March–May 2011. Discussions on topics of Sections (b-2.iii) started then. These pave the way for the proposed research in Section (b-4.ii).

Additional costs.

- Travel funding for the PI and RAs is requested for international, european, and national meetings (conferences, workshops, research visits, *etc.*), and for the PI to visit the SVRs at their sites.
- Funding is also requested for inviting research collaborators to the Computer Laboratory, University of Cambridge, to work with the team.
- Further funds are requested for notebooks for the PI and RAs, and the cost of books and broadband for the PI.

REFERENCES

1 R. Adams (2006). Pure type systems with judgmental equality. *Journal of Functional Programming*, 16(2):219–246.
2 K.Y. Ahn and T. Sheard (2008). Shared subtypes: subtyping recursive parametrized algebraic data types. In *Haskell'08*.
3 K.Y. Ahn and T. Sheard (2011). A Hierarchy of Mendler style Recursion Combinators. In *ICFP'11*.
4 J.-M. Andreoli (1992). Logic programming with focusing proof in linear logic. *J. of Logic and Computation* 2.
5 S. Artemov (2001). Explicit provability and constructive semantics. *Bulletin for Symbolic Logic*, 7(1):1–36.
6 M. Artin, A. Grothendieck, and J.-L. Verdier, eds. (1972). *SGA 4*, LNM 269.
7 S. Awodey, T. Coquand, and V. Voevodsky (2012) Univalent Foundations of Mathematics. IAS Princeton.
8 L. Augustsson (1998). Cayenne — a language with dependent types. In *ICFP'98*, pp 239–250.
9 A. Barber (1996). Dual Intuitionistic Linear Logic. PhD thesis, University of Edinburgh.
10 A. Bauer and M. Pretnar (2012). Programming with Algebraic Effects and Handlers.
11 J. Beck (1967). *Triples, algebras and cohomology*. *Reprints in TAC* 2, 2003.
12 J. Bénabou (1967). *Introduction to bicategories*. In LNM 47.
13 N. Benton and P. Wadler (1996). Linear logic, monads, and the lambda calculus. In *11th LICS*.
14 G. Birkhoff (1935). On the structure of abstract algebras. *P. Camb. Philos. Soc.*, 31:433–454.
15 C. Boehm and A. Berarducci (1985). Automatic Synthesis of Typed Lambda-Programs on Term Algebras. *TCS* 39.
16 T. Borghuis (1998). Modal Pure Type Systems: Type Theory for Knowledge Representation. *Journal of Logic, Language, and Information*, 7:265–296.
17 J. Cartmell (1986). Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic* 32.
18 A. Church (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics* 58.
19 A. Church (1940). A formulation of the simple theory of types. *J. Symbolic Logic*, 5:56–68.
20 P. Clairambault and P. Dybjer (2011). The biequivalence of locally cartesian closed categories and Martin-Löf type theories. In *TLCA'11*.
21 P. Cohen (1966). *Set theory and the continuum hypothesis*.
22 P.-L. Curien, G. Cousineau, M. Mauny (1987). The Categorical Abstract Machine. *Science of Computer Programming* 8.
23 P.-L. Curien and H. Herbelin (2000). The duality of computation. In *ICFP'00*, pp 233–243.
24 P.-L. Curien and G. Munch-Maccagnoni (2010). The duality of computation under focus. In *IFIP TCS*.
25 H. Curry (1934). Functionality in Combinatory Logic. In *Proceedings of the National Academy of Sciences* 20.
26 B. Day (1970). On closed categories of functors. In LNM 137.
27 N.G. de Bruijn (1968). Automath, a language for mathematics. In *Automation and Reasoning*, pp 159–200, 1983.
28 P. Dybjer (1994). Inductive families. *Formal Aspects of Computing*, 6:440–465.
29 P. Dybjer (1996). Internal Type Theory. In *TYPES*, LNCS 1158, pp 120–134.
30 P. Dybjer (2000). A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory. *JSL* 65.
31 J. Egger, R. Møgelberg, and A. Simpson (2009). Enriching an Effect Calculus with Linear Types. In *CSL'09*.
32 S. Eilenberg and S. Mac Lane (1945). General Theory of Natural Equivalences. *Trans. Amer. Math. Soc.*, 58(2):231–294.
33 A. Filinski (1994). Representing monads. In *POPL'94*.
34 M. Fiore (2005). Mathematical models of computational and combinatorial structures. In *FOSSACS'05*, LNCS 3441.
35 M. Fiore (2008). Second-order and dependently-sorted abstract syntax. In *LICS'08*, pp. 57–68.
36 M. Fiore (2012). Discrete Generalised Polynomial Functors. In *ICALP'12*, LNCS 7392, pp 214–226.
37 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.
38 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.
39 M. Fiore and C.-K. Hur (2010). Second-order equational logic. In *CSL'10*, LNCS 6247, pp 320–335.
40 M. Fiore and C.-K. Hur (2011). On the mathematical synthesis of equational logics. *LMCS*-7(3:12).
41 M. Fiore and O. Mahmoud (2010). Second-order algebraic theories. In *MFCS'10*, LNCS 6281, pp 368–380.
42 M. Fiore, G. Plotkin and D. Turi (1999). Abstract syntax and variable binding. In *LICS'99*.
43 G. Frege (1879). *Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought*. In *[95]*.
44 G. Frege (1893/1903). *Grundgesetze der Arithmetik I/II*.
45 G. Gentzen (1935). Untersuchungen über das logische Schließen. Mathematische Zeitschrift 39:176–210.
46 J.-Y. Girard (1972). *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thèse de doctorat d'état, Université Paris VII.
47 J.-Y. Girard (1987). Linear logic. *TCS*, 50:1–101.
48 J.-Y. Girard (1991). A new constructive logic: Classical

Figure 5 Costs

logic. *Math. Struct. Comp. Sci.*, 1(3).
49 A. Grothendieck and M. Raynaud (1971). *SGA 1*, LNM 224.
50 R. Hindley (1969). The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.*, 146:29–60.
51 W. Howard (1969). The formulae-as-types notion of construction. In *[93]*, pp 479–490.
52 G. Hutton (2007). Programming in Haskell. Cambridge University Press.
53 G. Jaber, N. Tabareau, and M. Sozeau (2012). Extending Type Theory with Forcing. In *LICS'12*, pp 395–404.
54 B. Jacobs (1999). *Categorical Logic and Type Theory*.
55 P. Johnstone (2002). Sketches of an Elephant: A Topos Theory Compendium. Oxford University Press.
56 D. Kan (1958). Adjoint functors. *Trans. Amer. Math. Soc.* 87.
57 G. M. Kelly (1982). *Basic Concepts of Enriched Category Theory*. *Reprints in TAC* 10, 2005.
58 S. Kobayashi (1997). Monad as modality. In *TCS* 175.
59 J. Lambek (1968). Deductive systems and categories I. *J. Math. Systems Theory*, 2:278–318.
60 J. Lambek and P. Scott (1986). *Introduction to Higher Order Categorical Logic*. Cambridge University Press.
61 F. W. Lawvere (1963). Functorial Semantics of Algebraic Theories. *Reprints in TAC* 5, 2004.
62 F. W. Lawvere (1969). Adjointness in foundations. *Reprints in TAC* 16, 2006.
63 F. W. Lawvere (1973). Metric spaces, generalized logic and closed categories. *Reprints in TAC* 1, 2002.
64 P. Levy (2005). Adjunction models for Call-By-Push-Value with stacks. *TAC*, 14(5):75–110.
65 F. Linton (1966). Some aspects of equational theories. In *Proc. Conf. on Categorical Algebra at La Jolla*, pp 84–95.
66 S. Mac Lane (1971). Categories for the Working Mathematician. Springer Verlag, Second Edition 1998.
67 P. Martin-Löf (1975). An intuitionistic theory of types: predicative part. In *Logic Colloquium* 1973, pp 73–118.
68 P. Martin-Löf (1984). Intuitionistic Type Theory. Bibliopolis.
69 C. McBride. Epigram: practical programming with dependent types. In *AFP'04*, pp 130–170.
70 P.-A. Melliès (2009). Categorical Semantics of Linear Logic. In *Panoramas et synthèsis*, 27:1–196.
71 P.-A. Melliès and N. Tabareau (2010). Resource modalities in tensor logic. *APAL*, 161(5):632–653.
72 M. Mendler and S. Scheele (2011). Cut-free Gentzen calculus for multimodal CK. *Inf. & Comp.*, 209(12):1465–1490.
73 N. Mendler (1991). Inductive types and type constraints in the second-order lambda calculus. *APAL*, 51(1–2):159–172.
74 R. Milner (1978). A Theory of Type Polymorphism in Programming. *J. of Computer and System Sciences*, 17:348–375.
75 E. Moggi (1991). Notions of computation and monads. *Information And Computation*, 93(1).
76 G. Munch-Maccagnoni (2009). Focalisation and classical realisability. In *CSL'09*, LNCS 5771, pp 409–423.
77 B. Nordström, K. Petersson, and J. Smith (1990). Programming in Martin-Löf Type Theory: An Introduction. OUP.
78 S. Park and H. Im (2011). A modal logic internalizing normal proofs. *Information and Computation*, 209:1519–1535.
79 G. Plotkin (1975). Call-by-name, call-by-value and the

λ-calculus. *Theoretical Computer Science*, 1:125–159.
80 G. Plotkin (1977). LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255.
81 G. Plotkin and A. J. Power (2003). Algebraic operations and generic effects. *Applied Categorical Structures* 11.
82 G. Plotkin and M. Pretnar (2009). Handlers of algebraic effects. *Programming Languages and Systems*, pp 80–94.
83 J. Reynolds (1983). Types, abstraction and parametric polymorphism. *Information Processing* 83, pp 513–523.
84 B. Russell (1902). Letter to Frege. In *[95]*, pp 124–125.
85 B. Russell (1903). *The principles of mathematics*. CUP.
86 D. Scott (1967). A proof of the independence of the continuum hypothesis *Theory of Computing Systems*, 1(2):89–111.
87 D. Scott (1969). A type-theoretical alternative to ISWIM, CUCH, OWHY In *TCS*, 121(1–2):411–440, 1993.
88 D. Scott (1970). Constructive validity. In LNM 125.
89 M. Schönfinkel (1924). On the building blocks of mathematical logic. In *[95]*, pp 355–366.
90 R. Seely (1989). Linear logic, *-autonomous categories and cofree coalgebras *Contemporary Mathematics* 92.
91 T. Sheard (2004). Languages of the future. In *SIGPLAN Notices*.
92 C. Strachey (1967). Fundamental Concepts in Programming Languages. In *HOSC*, 13:11–49, 2000.
93 J. Seldin and J. R. Hindley (1980). *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*.
94 A. Turing (1937). On computable numbers, with an application to the Entscheidungsproblem. *Proc. LMS*, 42:230–265.
95 J. van Heijenoort, ed. (1967). *From Frege to Gödel: A source book in mathematical logic, 1879–1931*.
96 J. von Plato (2012). Gentzen's proof systems: byproducts in a work of genius. *Bull. Symbolic Logic*, 18(3):313–367.
97 P. Wadler (2003). Call-by-value is dual to call-by-name. In *ICFP'03*, pp 189–201.
98 The Haskell Programming Language. `http://www.haskell.org/haskellwiki/Haskell`.
99 Tim Sheard. Languages of the future. October 2004. OOPSLA Companion Volume.
100 Tim Sheard and Emir Pasalic. Meta-programming with built-in type equality. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM'04)*. `http://cs-www.cs.yale.edu/homes/carsten/lfm04/`, pages 106–124, Cork, Ireland, July 2004.
101 Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. *ACM SIGPLAN Notices*, 37(12):60–75, December 2002.
102 Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM)*, pages 203–217, Amsterdam, 1997. ACM Press.