Teaching and Research statement

Ambroise Lafont

This document provides a summary of my previous work, a research project, and teaching statement.

Part I
Summary of previous work

My research focuses mainly on the notion of programming languages and safe compilations between them.

Pre-doc I was an intern in the Gallium team, working with Xavier Leroy on a verified compiler for an untyped functional programming language. My internship focused on verifying the closure conversion phase, which was already implemented in Coq. In this phase, free variables are eliminated in local abstractions by replacing them with accesses to fields of the environment given as an additional argument to the function. Using the so-called step-indexing technique, my internship essentially focused on the case of mutual recursive functions.

PhD My PhD started with looking for an appropriate notion of specification for syntax with equations [3, 4], with results formalised in Coq. Then, I focused on accounting for the (operational) semantics of programming languages [5, 14]. It culminated with the notion of programming languages as transition monads, detailed in Section 2.

Side projects During my PhD, I have also been interested in

- a categorical setting for Howe's method, used to prove that bisimilarity coincides with contextual equivalence [9], detailed in Section 3;
- a compilation phase in the setting of dependently-typed theories consisting in eliminating inductive-inductive datatypes (Section 4, [20]), working with the Agda proof assistant;
- a Coq formalisation of some definitions of weak ω-groupoids in Coq, then proving internally that dependent types are weak ω-groupoids (Section 5).

Post-doc I am working on a concrete instance of a certifying compiler translating functional programs written in Cogent to C and Isabelle, and generating along a formal proof of correspondence between them (Section 6).

Attached publications Three publications are attached with this application:

1. one relating to my PhD topic [5], introduced in Section 2;

2. one introducing a categorical framework for Howe's method [9], explained in Section 3;

3. one explaining how inductive-inductive types can be eliminated [20] in a dependent type theory, detailed in Section 4.

Before delving into the details, in a first section I introduce the general framework of Initial Semantics that I have systematically used for specifying mathematical objects.

1 Introduction to Initial Semantics

My work involves notions of specification in the spirit of *Initial Semantics* or *Algebraic Specification* [18]. This has been popularised by the movement of *Algebra of Programming* [8] and is a standard way of characterising an object of a given type (typically, of a given category C) through an initiality property. The general methodology of Initial Semantics can be described according to the following steps:

1. introduce a notion of signature (for the category C);

2. introduce a notion of associated model, organising into a category with a functor U to the category C (typically, a model is an object of C equipped with additional structure);

3. define the object specified by the signature as $U(Z)$, where Z is the initial model, if it exists (the signature is then called *effective*);

4. find a sufficient condition for a signature to be effective[1].

The models of a signature delimit the scope of the principle of recursion, which is induced by initiality of the object specified by the signature.

A simple example: specifying $\mathbb{N}$ and defining addition by recursion Endofunctors on the category of sets yield a possible notion of signature: a model of an endofunctor F is just a F-algebra. The set $\mathbb{N}$ of natural numbers, equipped with its constant $1 \xrightarrow{0} \mathbb{N}$ and successor function $S : \mathbb{N} \to \mathbb{N}$, is the initial algebra for the endofunctor $X \mapsto 1 + X$. Addition can be defined by recursion as a function $\mathbb{N} \to \mathbb{N}^{\mathbb{N}}$ (mapping n to $q \mapsto n + q$) by giving an algebra structure on $\mathbb{N}^{\mathbb{N}}$ as follows:

- we pick the identity function $1 \xrightarrow{id} \mathbb{N}^{\mathbb{N}}$;
- we define $\mathbb{N}^{\mathbb{N}} \to \mathbb{N}^{\mathbb{N}}$ as mapping f to $n \mapsto f(n) + 1$.

The induced function $\mathbb{N} \to \mathbb{N}^{\mathbb{N}}$ is an algebra morphism, which ensures that the expected recursive equations are satisfied:

$$0 + n = n \qquad Sp + q = S(p + q).$$

[1] In the literature, the word signature is often reserved for the case where such a sufficient condition is automatically satisfied.

2 Programming languages as transition monads

In the context of the theory of programming languages, there are different ways to describe the syntax and operational semantics of a language. My PhD focuses on high-level mathematical descriptions allowing to generate syntax and semantics from basic data, with automatic support for reasoning modulo renaming of bound variables. In this section, I detail the work (starting from my PhD) that led to the definition of transition monads, as a notion of programming language, taking into account both the syntactic and the semantic aspects, in the simply-typed case. This approach relies on the categorical notions of monads and modules, for a systematic account of substitution. I first start with a simple example of transition monad: untyped λ-calculus.

2.1 An example of transition monads: untyped λ-calculus

A programming language typically consists in two components: the syntax and the semantics. I first briefly recap how monads nicely capture the notion of syntax with the example of λ-calculus. Then, I illustrate the semantic aspect of transition monads for this example.

Syntax The syntax of λ-calculus can be modelled as a monad L on the category of sets, where $L(X)$ is the sets of λ-terms taking free variables in X, modulo renaming of bound variables (i.e., $\lambda x.x$ and $\lambda y.y$ are identified). The unit of the monad embed variables into terms $X \to L(X)$, while the bind operation $(X \to L(Y)) \times L(X) \to L(Y)$ is *parallel (or simultaneous) substitution*: it maps a pair consisting of a substitution $\sigma : X \to L(Y)$, associating to each variable x in X a term $\sigma(x)$, and a λ-term t taking free variables in X, to the substituted term $t[\sigma]$, which is t where all the variables have been replaced according to σ. The monadic laws express various expected substitution properties:

$$x[\sigma] = \sigma(x) \qquad t[x \mapsto x] = t \qquad t[\sigma][\delta] = t[x \mapsto \sigma(x)[\delta]].$$

Semantics For each set X of free variables, there is a graph of reductions defined as follows: vertices are λ-terms taking free variables in X, and arrows are reductions between them. Furthermore, these graphs are compatible with substitution: if t reduces to u, then $t[\sigma]$ reduces to $u[\sigma]$, for any relevant σ. This completes the definition of λ-calculus as a transition monad.

2.2 Syntax with equations

As explained above, it is well-known that syntaxes with variable binding, such as λ-calculus, can be formalised as monads on the category of sets (in the untyped case), hence capturing the notion of well-behaved parallel substitution. They can be specified with binding signatures, which are mere lists of operation symbols with associated arities. These arities not only tell how many arguments an operation takes as input, but also how many variables are bound in each argument. For example, the λ-calculus has two operations: application, specified as a usual binary operation, and abstraction, as a unary operation binding one variable in its argument.

However, these specifications are not sufficient for syntaxes that are defined modulo some equations, such as differential λ-calculus. With Benedikt Ahrens, André Hirschowitz, and Marco Maggesi, we introduced in [3] a notion of quotient of these binding signatures. Commutative operations can be specified as such, but not associative ones. This motivates our proposed setting for specifying general equations in [4], as a variant of Fiore and Hur's second-order equational logic. With the help of Benedikt Ahrens, I formalised[2] our main results within the UniMath system [30], an implementation of homotopy type theory within the Coq proof assistant. This was the opportunity to contribute to UniMath.

2.3 Semantics

In [5], we tackle the specification of the operational semantics of a programming language such as λ-calculus with β-reduction. To this end, we introduce the notion of reduction monads, which are roughly graphs whose vertices are terms of the syntax (as a monad) and arrows are the reduction steps as exemplified above in Section 2.1.

With André and Tom Hirschowitz, we finally extend this setting in [14], introducing transition monads, which generalise to the simply-typed case and take into account several important examples, such as call-by-value λ-calculus (big-step or small-step semantics), $\overline{\lambda}\mu$-calculus, π-calculus, differential λ-calculus, and (positive) GSOS specifications. The generalisation allows in particular bipartite graphs, which can then account for heterogenous reductions, as it happens in big-step semantics (terms reduce to values). We also provide a notion of signature for specifying transition monads, in the spirit of Initial Semantics.

The range of examples that transition monads capture goes beyond the scope of the related work. For example, the theory of bialgebraic operational semantics à la Plotkin-Turi [27] does not account for higher-order languages such as λ-calculus, while the settings of Hamana [13], T. Hirschowitz [15], or Ahrens [2] always enforce congruence of reductions: head reductions are thus out of reach of these frameworks.

3 Howe's method in a categorical setting

Applicative bisimilarity is a coinductive characterisation of observational equivalence in call-by-name lambda-calculus, introduced by Abramsky [1]. Howe [17] gave a direct proof that it is a congruence.

With Tom Hirschowith and Peio Borthelle, we abstract over this result in [9]: we introduce a categorical framework for operational semantics, in which we define substitution-closed bisimilarity, an abstract analogue of the open extension of Abramsky's applicative bisimilarity. We furthermore prove a congruence theorem for substitution-closed bisimilarity, following Howe's method. We finally demonstrate that the framework covers the call-by-name and call-by-value variants of λ-calculus in big-step style. As an intermediate result, we generalise the standard framework for syntax with variable binding [12] to the skew-monoidal case. My main contribution was the Coq formalisation of this intermediate result[3], Theorem 2.15.

[2] `https://github.com/UniMath/largecatmodules`
[3] `https://github.com/amblafont/Skew-Monoidalcategories`

I have then spent some time simplifying the framework and the proofs, with the help of Tom Hirschowitz. The updated results are available in a preprint [16].

4 Compiling inductive-inductive types

An *inductive-inductive type* (IIT) allows the specification of a data type through a mutual inductive definition of a type A and a type family B indexed over that type A. In this section, I am describing some work published in [20] with Ambrus Kaposi and András Kovács: we precisely define what it means for a type theory to support IITs, and show that this is the case of intensional type theory with Uniqueness of Identity Proofs (UIP), functional extensionality, and inductive types.

Internalisation Our work relies on *internalisation*, that is, adapting an *external* set theoretic definition into an *internal* type theoretic one, more specifically in our case, in intensional type theory with UIP, functional extensionality and inductive types. The internal definition should become equivalent to the external one when taking the standard interpretation of type theory into set theory.

QIITs in type theory Our work relies on a previous work of Kaposi and Kovács [19]. They give an external notion of signature for *quotient inductive-inductive types* (QIIT), an extension of IIT with quotients. For any such signature, there is an associated category of models. The internalisation of this category yields a definition of a QIIT, as the initial object in this category. Indeed, the initiality property gives rise to the expected elimination principle.

The signatures for QIITs are defined as contexts of a domain specific dependent type theory $\mathcal{T}_{\mathrm{QIIT}}$. It happens that this type theory itself can be characterised as an initial object in the category of models of a specific QIIT signature. In this sense, $\mathcal{T}_{\mathrm{QIIT}}$ can be considered as a set theoretic QIIT. If this QIIT exists internally, then:

1. the notion of signature, as a context of $\mathcal{T}_{\mathrm{QIIT}}$, is internalisable;
2. it can be proved internally that any other QIIT exists (see [19]).

Our work: IITs in type theory Similarly, we give an external notion of signature for IITs as contexts of a domain specific type theory $\mathcal{T}_{\mathrm{IIT}}$, a subset of $\mathcal{T}_{\mathrm{QIIT}}$. For any such signature Σ, we define an IIT to be the initial object of the internal category of models associated to Σ. As it is the case for $\mathcal{T}_{\mathrm{QIIT}}$, the type theory $\mathcal{T}_{\mathrm{IIT}}$ can be considered as a set theoretic QIIT. Again, if this QIIT exists internally, then, it can be proved internally that any IIT exists.

My main contribution is to provide an internal construction of $\mathcal{T}_{\mathrm{IIT}}$ as a QIIT, with the help of the Agda proof assistant. Concretely, I define the untyped syntax of $\mathcal{T}_{\mathrm{IIT}}$ with its typing judgements, using inductive types. Then, I show that this induces a model for the QIIT signature associated to $\mathcal{T}_{\mathrm{IIT}}$. To prove that this model is initial, I first specify the relation enjoyed by the initial morphism. Then I show that it is indeed functional and that any morphism out of this model satisfy the same functional relation.

5 Formalisation of weak ω-groupoids

I have worked on the internalisation (using the proof assistant Coq) of Brunerie's proof that any *fibrant type* of a Martin-Löf type theory is a weak ω-groupoid, in a *2-level type system*.

In his PhD [10], Brunerie defines a dependent type theory *extrinsically*, i.e. through an untyped syntax with typing judgements. Then he defines weak ω-groupoids as models (in a suitable sense) of this type theory.

Types are weak ω-groupoids One of the discoveries underlying the recent homotopical interpretation of Martin-Löf type theory is the fact that, for any type, the tower of its iterated Martin-Löf identity types gives rise to a weak ω-groupoid [28, 22]. Later, Brunerie proposed a definition of weak ω-groupoids which looks amenable to internalisation, and indeed, Nuo Li formalised it in the proof assistant Agda [7]. However, no explicit argument has been provided for Li's definition to be well-founded as Agda's termination checker was unable to testify and needed to be switched off.

Bridging the extrinsic and the intrinsic definition Brunerie performs a recursion on his type theory to prove that types are weak ω-groupoids. However, the invoked recursion principle does not come straightforwardly from the definition of his type theory through an untyped syntax and typing judgements, when internalising the argument in type theory. This challenge is similar to the construction of $\mathcal{T}_{\mathrm{IIT}}$ described in Section 4, but it has the following additional difficulty: we don't want to work in a setting with UIP as it would trivialise the weak omega-groupoidal structure of a type.

Two-level type theory I have found a solution to this problem by working in a 2-level type theory, i.e., a type theory with two equalities: a strict one satisfying UIP, and a homotopical one, which does not satisfy UIP. This kind of type theory was devised by Altenkirch, Capriotti and Kraus [6], following ideas of Voevodsky's homotopy type system (HTS). Voevodsky indeed invented HTS to solve a similar dilemma when defining semi-simplicial types in a Martin-Löf type theory.

A 2-level type theory comes with a notion of *fibrant type*: roughly, a type is fibrant if it does not talk about the strict equality. The elimination rule of the homotopical equality is then restricted to fibrant types. If this condition is not enforced, then the two equalities are provably equivalent, which is unsatisfactory as only the strict equality should satisfy UIP.

Axiomatising homotopical equality in Coq, I was able to adapt Brunerie's proof to show that any fibrant type, equipped with its tower of iterated homotopical equalities, is a weak ω-groupoid.

6 Certifying compilation of Cogent

Cogent [23] is a restricted functional language with a certified compilation to C: the Cogent compiler generates a C program, an Isabelle program, and a formal proof of correspondence between them, based on the Isabelle/HOL

proof assistant and the AutoCorres library that parses C files and construct a representation of the C functions in Isabelle.

I have extended this proof of correspondence to take into account a new feature called DARGENT [24] that enables the programmer to specify how COGENT data types are laid out in memory. This feature is meant to bridge the gap between the C data structures used in the operating system, and the algebraic data types used by COGENT.

6.1 Overview of Dargent

DARGENT offers the possibility to assign any (heap allocated) record type a custom layout describing how the data is mapped into an array of words. A record assigned with a custom layout is essentially compiled to an array of words. The compiler generates custom field getters (and setters) in C, retrieving or setting the field values from the array, according to the given layout. Roughly, for a record $\{\text{foo} : \mathsf{A}, \ldots\}$, the custom getter for the field foo takes an array of words as input and outputs a value of type A. The custom setter for the same field takes an array of words and a value of type A, and updates the array so that it represents the updated record.

6.2 Extending the verification framework with Dargent

With DARGENT, getting or setting a field in COGENT corresponds to a C call of custom getter and setter functions generated by the compiler. The following changes (visible on my COGENT branch at `https://github.com/amblafont/cogent/tree/dargent-isa`) to the verification framework were necessary to take DARGENT into account:

1. adapting the record lemmas, which are elementary correspondence statements about getting and setting a field in COGENT and the C compiled version (no longer a direct field access or update of a C structure);

2. adapting the value relation between C values and COGENT values, since a COGENT record should now relate to a C array of words, rather than a C structure;

3. generating lemmas stating some expected properties about the custom getters and setters, such as setting a field and then getting a different field yields the same as getting this field without prior setting.

The automation of the proofs of these last lemmas is work in progress. The value relation mentioned in the second item in the final correspondence statement between the C and the cogent code, which roughly says that the output C value of the compiled C program should relate to the output value of the compiled Isabelle program.

Part II
Research project

A major breakthrough in the field of compilation correctness is the pioneering verified C compiler CompCert developed by Leroy's team. Mostly implemented within the Coq proof assistant, its correctness is guaranteed by a mechanised proof, avoiding the need for testing. This provides a particular instance of verified compilation. However, it does not currently fit in any proposed theoretical definition of programming languages and their compilations: mathematical frameworks such as Plotkin and Turi's mathematical operational semantics [27] are still far from matching the practice.

This research project focuses on answering the following question: What is a programming language? The goal is to develop a mathematical understanding of programming languages, compilations between them, and their metatheory in order to reduce the gap between theory and practice. The long-term objective is to provide a Coq-based tool that would take a specification of a programming language as input and generate the corresponding mathematical object, together with associated expected properties. Such a tool would require solid and stable mathematical definitions: this is the main focus of this research project.

Relevance of my profile A large part of this project revolves around the mathematical notion of transition monad as programming language, that I have introduced with Tom and André Hirschowitz [14], based on the last chapter of my PhD thesis. Besides, I have worked on various formalisations. In particular, I formalised the syntactical part of the setting of transition monads (see the report on previous work). This experience will prove useful when it comes to developing the Coq-based device mentioned above.

Plan Transition monads suffer from some limitations, explained in the first section, that suggest a few extensions. In the second section, I list some more ambitious challenges, beyond the scope of transition monads.

7 Extending the setting of transition monads

Transition monads[4] provide a mathematical notion of programming language, taking into account both syntax and operational semantics. Transition monads are designed to systematically account for substitution invariance, using the categorical notions of monad and modules over them. They can be specified by algebraic presentations, in the spirit of Initial Semantics: there is a notion of signature for specifying them; to each signature is associated a category of models, consisting of transition monads equipped with additional structure (e.g., some operations with specified arities). The transition monad *specified* by a signature is (by definition) provided by the initial object in the category of models.

[4]See the first part of this document on previous work for a more introductory description motivating this notion.

Although transition monads capture a number of important examples such as λ-calculus (big- or small- step semantics, call-by-value or call-by-name), some variant of π-calculus and differential λ-calculus, they suffer from some limitations. For example, the definition of π-calculus as a labelled transition system is out of reach, and the metatheory is not well developed. These defects suggest some extensions that are described in the next subsections.

7.1 Addressing specific programming languages

7.1.1 Differential λ-calculus (short term)

The differential λ-calculus can be organised into a transition monad without too much pain. However, its specification remains challenging, as we now explain. Recall that a transition monad specified by a signature comes with a recursion principle induced by its initiality property in an appropriate category of models. In previous work [5], we provided a notion of signatures and models for particular cases of transition monads, called reduction monads. There, the induced recursion principle enables the simple construction of compilations targeting a language with a different syntax. This remains to be adapted to the more general setting of transition monads, for which the current notions of signatures and models generate a recursion principle that only targets languages with the same syntax (and possibly different semantics). Although this extension is not expected to raise any serious difficulty, it involves changes to the notion of signature that would make the differential λ-calculus harder to specify. The difficulty with this calculus is that the operational semantic relies on two intermediary operations constructed by recursion in the initial model: it is not obvious how to replicate this construction in any model of the syntax.

7.1.2 Computational λ-calculus (short/medium term)

Another example of a transition monad which is not obvious to specify is some variant of the computational λ-calculus with big-step semantics [21], where terms reduce to semantic values. The precise notion of semantic value depends on a chosen semantic monad T (for example, the state monad). In this example, the difficulty lies in the sequential reduction rule for composing effectful terms, which does not fit in the current notion of signature for transition monads because of universal quantification on semantic values in the second premise:

$$\frac{M \Downarrow X \qquad \forall V,\ \ldots \Downarrow \ldots}{\ldots \Downarrow \ldots}$$

One option to explore is to think of this universally quantified premise as a possibly infinite set of premises, although this would open the door to infinitary reduction rules.

7.1.3 π-calculus (short/medium term)

The π-calculus is a calculus of concurrent processes. When presented as an unlabelled reduction system, the π-calculus can be organised as a transition monad. However, practitioners tend to rely on a different presentation [26, Table 1.5 p38] based on labelled transition systems, which cannot. The difficulty is, for example, that the target of a reduction may refer to some channel that may

be either fresh or known, depending on the reduction rule: the current definition of transition monads is too strict to allow this. This calls for a generalisation of this notion.

7.2 Proving congruence of bisimilarity (short/medium term)

Transition monads provide guarantees about substitution (substitution laws, stability of transitions under substitution, ...), but not much beyond. Replaying some well-known metatheorems or methods is an obvious means of testing the relevance of these theoretical definitions, and at the same time could reveal an unthought level of generality for some already known results. For example, how can we adapt Howe's method, commonly used to show that bisimilarity is a congruence? I have recently worked on some categorical generalisation of this method [9] in a simpler setting, but it remains to explore how this can be adapted to the setting of transition monads.

7.3 Plotkin's CPS translations (short/medium term)

Morphisms of transitions monads provide a notion of compilation between programming languages, with forward simulation as correctness property: if a term reduces to another in the source language, then there is a reduction between the compiled terms. Other criteria may be considered (backward simulations, bisimulations), yielding alternative notions of morphisms between transition monads. For example, we can easily adapt our setting to allow translating one reduction to any finite sequence of reductions between compiled terms.

It would be useful to have mathematical tools for generating compilations with these alternative properties from elementary data. We propose to start with replaying the most standard academic compilations in the setting of transition monads: Plotkin's CPS translations of λ-calculus [25], translation of π-calculus to λ-calculus, translation of λ-calculus to the Krivine or Zinc machine. The goal is to test and refine the definitions of transition monads and their morphisms.

8 Long-term goals: beyond transition monads

8.1 Linear λ-calculus

In the context of my current postdoctoral position, I am working on Cogent [23], a functional programming language featuring linear types. The issue with linear types is that the syntax does not define a monad on the category of sets as usual: indeed arbitrary substitutions are not allowed, since they break linearity. For example, the linear λ-calculus is a programming language with substitution, which cannot be organised as a monad (and thus, not as a transition monad), but could be organised as an operad. This suggests introducing an operadic variant of transition monads.

Finally, the possibility of mixing both linear and non-linear variables (as in the Cogent programming language) should be investigated.

8.2 Dependently-typed languages

A type system for a programming language provides a way to check at compile time that the program cannot go wrong. Dependent type systems, such as the one underlying the Coq proof assistant are actively studied, in particular for their strong connection with proof theory, thanks to the fundamental Curry-Howard correspondence. For the simpler untyped case, monads on the category of sets provide a notion of syntax: the monad multiplication accounts for syntactic simultaneous substitution[5]. Although this idea can be straightforwardly extended to simply-typed syntax, the case of dependent types requires further investigation. The challenge here is to find a similar notion of monads and modules, that could be used to define and specify such complex syntaxes. This would provide an answer to the initiality conjecture formulated by Voevodsky [29], a recent problem related to the semantics of dependent type theories. This conjecture corresponds to the fact that the definition of a type theory as an extrinsic syntax, i.e., as an untyped syntax with typing judgements, is compatible with the intrinsic definition[6], as the initial object in the suitable category of models. This conjecture has been recently solved by de Boer-Brunerie-Lumsdaine-Mortberg [11], based on the notion of models as categories with families, for a specific dependent type theory. Our proposal would induce another solution for an alternative notion of models and for a whole class of dependent type theories.

8.3 Register allocation and Initial Semantics

Bridging the gap between mathematical methods and the complex algorithms used in practice is a tremendous challenge. This long-term goal does not directly relate to transition monads: it consists first in studying the possibility of addressing low-level compilation phases, such as register allocation, in the spirit of Initial Semantics. Some of these phases in the CompCert compiler are not even written in Coq, but in Caml. This is the case of graph colouring in register allocation, which is "difficult to prove correct directly": the output of the Caml program is validated on a per-program basis by a Coq verifier. This hints that the categorical understanding of such an algorithm is certainly even more challenging to achieve.

8.4 A Coq front-end for verified compilation

The overall long-term goal of the project is to devise a general Coq-based framework inspired by the particular CompCert compiler. Hence, as far as possible, the previous theoretical investigations will be formalised in Coq in an effective manner amenable to a Caml extraction. The user would simply provide a specification of some programming language, as a signature for transition monads. Then, the Coq tool would generate the transition monad matching the specification. A similar pattern could be considered for constructing compilations between programming languages.

Let us remind that the definitions and results regarding the syntactic part of transition monads and their specifications are already formalised, as mentioned

[5] See the first part of this document on previous work for the example of λ-calculus.

[6] See the report on previous work for a more detailed explanation on the differences between intrinsic and extrinsic definitions.

in the report on previous work.

Part III
Teaching statement

9 Teaching experience

In the following subsections, I detail the teaching I delivered during my PhD, mostly to engineering students of the French engineering school IMT Atlantique, where I did my PhD (2016-2019). During my master degree, I also

- designed and supervised Maple tutorials for students aiming at highly competitive national French examinations for entering business schools;
- mentored undergraduate students from the top-ranked French engineering school École Polytechnique in theoretical computer science and physics (special relativity).

9.1 Undergraduate level

- Python, 30h tutoring (including designing the material) at the UCO University, 2018
- Probability and statistics, 25h for students enrolled in an apprenticeship program at IMT Atlantique, 2018
- Co-supervision of bachelor student projects, 20h tutoring at IMT Atlantique, 2018
- Object-oriented programming in Java, 40h tutoring at IMT Atlantique, 2017

9.2 Postgraduate level

- Haskell, 12h tutoring at IMT Atlantique, 2018
- Linear programming, 7h tutoring at IMT Atlantique, 2018
- Algorithmic structures, 16h tutoring at IMT Atlantique, 2017

10 Teaching philosophy

I have always valued teaching as a way to pay back to society and make a difference. I feel deeply thankful and still admire some of my teachers that have enlightened me: I remember them as an ideal that I am always striving to achieve when I am in their position. As a mathematical teacher in high school, my mother gave me early the taste for teaching, often encouraging me to offer my help to younger students. After high school, I have been teaching since my master degree in a variety of subjects, from programming with Java, Python, Haskell, or Maple, to more theoretical subjects such as special relativity in physics, logic, or mathematics (probability and statistics). Especially since working in research, I have realised that teaching improves my own understanding, and thus consider it as a useful activity even for the sole purpose of research.

References

[1] Abramsky, S. The lazy lambda calculus. In *Research Topics in Functional Programming* (1990), D. A. Turner, Ed., Addison–Wesley.

[2] Ahrens, B. Modules over relative monads for syntax and semantics. 3–37.

[3] Ahrens, B., Hirschowitz, A., Lafont, A., and Maggesi, M. High-level signatures and initial semantics. In *27th EACSL Annual Conference on Computer Science Logic (CSL 2018)* (Birmingham, United Kingdom, Sept. 2018), pp. 1–20.

[4] Ahrens, B., Hirschowitz, A., Lafont, A., and Maggesi, M. Modular Specification of Monads Through Higher-Order Presentations. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)* (Dagstuhl, Germany, 2019), H. Geuvers, Ed., vol. 131 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 6:1–6:19.

[5] Ahrens, B., Hirschowitz, A., Lafont, A., and Maggesi, M. Reduction monads and their signatures. *Proc. ACM Program. Lang. 4*, POPL (2020), 31:1–31:29.

[6] Altenkirch, T., Capriotti, P., and Kraus, N. Extending Homotopy Type Theory with Strict Equality. In *CSL* (2016), vol. 62 of *LIPIcs*, Schloss Dagstuhl.

[7] Altenkirch, T., Li, N., and Rypáček, O. Some constructions on ω-groupoids. In *LFMTP* (2014), ACM.

[8] Bird, R. S., and de Moor, O. *Algebra of programming.* Prentice Hall International series in computer science. Prentice Hall, 1997.

[9] Borthelle, P., Hirschowitz, T., and Lafont, A. A cellular howe theorem. In *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020* (2020), H. Hermanns, L. Zhang, N. Kobayashi, and D. Miller, Eds., ACM, pp. 273–286.

[10] Brunerie, G. *On the homotopy groups of spheres in homotopy type theory.* Phd, Université Nice Sophia Antipolis, June 2016.

[11] de Boer, M. A proof and formalization of the initiality conjecture of dependent type theory, 2020. Licentiate defense over Zoom.

[12] Fiore, M. P., Plotkin, G. D., and Turi, D. Abstract syntax and variable binding.

[13] Hamana, M. Term rewriting with variable binding: An initial algebra approach.

[14] Hirschowitz, A., Hirschowitz, T., and Lafont, A. Modules over monads and operational semantics. In *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)* (2020), Z. M. Ariola, Ed., vol. 167 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 12:1–12:23.

[15] Hirschowitz, T. Cartesian closed 2-categories and permutation equivalence in higher-order rewriting.

[16] Hirschowitz, T., and Lafont, A. A categorical framework for congruence of applicative bisimilarity in higher-order languages. working paper or preprint available at `https://hal.archives-ouvertes.fr/hal-02966439`, Oct. 2020.

[17] Howe, D. J. Proving congruence of bisimulation in functional programming languages. *Inf. Comput. 124*, 2 (1996), 103–112.

[18] J.A. Goguen, J. T., and Wagner, E. An initial algebra approach to the specification, correctness and implementation of abstract data types. In *Current Trends in Programming Methodology, IV: Data Structuring* (1978), R. Yeh, Ed., Prentice-Hall, pp. 80–144.

[19] Kaposi, A., Kovács, A., and Altenkirch, T. Constructing quotient inductive-inductive types. *Proceedings of the ACM on Programming Languages 3*, POPL (2019), 2.

[20] Kaposi, A., Kovács, A., and Lafont, A. For Finitary Induction-Induction, Induction Is Enough. In *25th International Conference on Types for Proofs and Programs (TYPES 2019)* (Dagstuhl, Germany, 2020), M. Bezem and A. Mahboubi, Eds., vol. 175 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 6:1–6:30.

[21] Lago, U. D., Gavazzo, F., and Levy, P. B. Effectful applicative bisimilarity: Monads, relators, and howe's method. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017* (2017), IEEE Computer Society, pp. 1–12.

[22] LeFanu Lumsdaine, P. Weak omega-categories from intensional type theory. *Logical Methods in Computer Science 6*, 3 (Sept. 2010).

[23] O'Connor, L., Chen, Z., Rizkallah, C., Amani, S., Lim, J., Murray, T. C., Nagashima, Y., Sewell, T., and Klein, G. Refinement through restraint: bringing down the cost of verification. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016* (2016), J. Garrigue, G. Keller, and E. Sumii, Eds., ACM, pp. 89–102.

[24] O'Connor, L., Chen, Z., Susarla, P., Rizkallah, C., Klein, G., and Keller, G. Bringing effortless refinement of data layouts to cogent. In *Leveraging Applications of Formal Methods, Verification and Validation. Modeling - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part I* (2018), T. Margaria and B. Steffen, Eds., vol. 11244 of *Lecture Notes in Computer Science*, Springer, pp. 134–149.

[25] Plotkin, G. D. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci. 1*, 2 (1975), 125–159.

[26] Sangiorgi, D., and Walker, D. *The π-calculus - a theory of mobile processes*. 2001.

[27] Turi, D., and Plotkin, G. D. Towards a mathematical operational semantics. pp. 280–291.

[28] van den Berg, B., and Garner, R. Types are weak ω-groupoids. *Proc. of the London Mathematical Society 102*, 2 (2011), 370–394.

[29] Voevodsky, V. Mathematical theory of type theories and the initiality conjecture, 2016. Research proposal to the Templeton Foundation.

[30] Voevodsky, V., Ahrens, B., Grayson, D., et al. UniMath — a computer-checked library of univalent mathematics. available at `https://github.com/UniMath/UniMath`.