# 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 alternates between theoretical investigations on the notion of programming language, and practical work related to safe compilation.

**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 mutually recursive functions.

**PhD** My PhD started with looking for an appropriate notion of specification for syntax with equations [2, 3], with results formalised in Coq. Then, I focused on accounting for the (operational) semantics of programming languages [4, 17]. It culminated with the notion of programming languages as transition monads, detailed in Section 1.

**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 2;
- a compilation phase in the setting of dependently-typed theories consisting in eliminating inductive-inductive datatypes (Section 3, [26]), working with the Agda proof assistant;
- a Coq formalisation of some definitions of weak $\omega$-groupoids in Coq, then proving internally that dependent types are weak $\omega$-groupoids (Section 4).

**Post-doc** I am working on a concrete instance of a certified compiler translating programs written in the COGENT functional programming language to C and Isabelle. I have in particular extended the formal proof of correspondence between them generated by the compiler, and exploit these changes to implement and verify drivers (Section 5).

# 1 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 $\lambda$-calculus.

## 1.1 An example of transition monads: untyped $\lambda$-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 $\lambda$-calculus. Then, I illustrate the semantic aspect of transition monads for this example.

**Syntax** The syntax of $\lambda$-calculus can be modelled as a monad $L$ on the category of sets, where $L(X)$ is the sets of $\lambda$-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 $\lambda$-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 $\sigma$. 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 $\lambda$-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 $\sigma$. This completes the definition of $\lambda$-calculus as a transition monad.

## 1.2 Syntax with equations

As explained above, it is well-known that syntaxes with variable binding, such as $\lambda$-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 $\lambda$-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 $\lambda$-calculus. With Benedikt Ahrens, André Hirschowitz, and Marco Maggesi, we introduced in [2] 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 [3], as a variant of Fiore and Hur's second-order equational logic. With the help of Benedikt Ahrens, I formalised[1] our main results within the UniMath system [38], a Coq implementation of homotopy type theory. This was the opportunity to contribute to UniMath.

## 1.3 Semantics

In [4], we tackle the specification of the operational semantics of a programming language such as $\lambda$-calculus with $\beta$-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 1.1.

With André and Tom Hirschowitz, we finally extend this setting in [17], introducing transition monads, which generalise to the simply-typed case and take into account several important examples, such as call-by-value $\lambda$-calculus (big-step or small-step semantics), $\overline{\lambda}\mu$-calculus, $\pi$-calculus, differential $\lambda$-calculus, and (positive) GSOS specifications. The generalisation allows in particular bipartite graphs, which can then account for heterogeneous 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 [36] does not account for higher-order languages such as $\lambda$-calculus, while the settings of Hamana [16], T. Hirschowitz [19], or Ahrens [1] always enforce congruence of reductions: head reductions are thus out of reach of these frameworks.

## 1.4 De Bruijn encoding

De Bruijn encoding [13] is a popular technique to represent syntax, which avoids quotients by renaming of bound variables. In [18], André Hirschowitz, Tom Hirschowitz, Marco Maggesi, and I, bridged the gap between the standard theory of syntax with variable binding [15] and this common practice.

The key insight consists in identifying an adequate skew monoidal structure accounting for De Bruijn substitution, and exploit the generalisation of

[1] `https://github.com/UniMath/largecatmodules`

Fiore, Plotkin, and Turi's monoidal setting [15] performed in my work with Tom Hirschowitz on Howe's method (see next section).

Although the resulting theory is simpler than the standard presheaf-based approach [15], I was able to establish a strong formal link with it. Our theory easily incorporates simple types and equations between terms, as well as an adapted notion of transition monads. I formalised [2] our syntactic setting in Coq in the untyped case, to demonstrate its simplicity.

## 2 Howe's method in a categorical setting

A fundamental requirement for any notion of equivalence between programs is that it is compatible with the various syntactical constructs of the programming language. In [21], Howe introduced a method which can be applied to prove congruence properties for a variety of different forms of program equivalence. How general this technique is remains unclear.

With Tom Hirschowitz and Peio Borthelle, we propose a general categorical setting in which Howe's method applies. Our framework covers the case of applicative bisimilarity for call-by-name and call-by-value variants of $\lambda$-calculus. It requires generalising the standard framework for syntax with variable binding [15] to the skew monoidal case, which I formalised in Coq [3]. Beyond this mechanisation, I significantly contributed to this work by extensively simplifying our categorical setting [20].

## 3 Compiling inductive-inductive types

Inductive-inductive types (IITs) are a generalisation of inductive types in type theory. They allow the mutual definition of types with multiple sorts where later sorts can be indexed by previous ones. They offer a way of formalising type theory in type theory [11], supporting the claim that type theory is powerful enough to be used as its own metatheory.

In this section, I am describing some work published in [26] with Ambrus Kaposi and András Kovács: we precisely define what it means for a type theory to support IITs, and show that, in fact, it is essentially enough to support indexed 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 Uniqueness of Identity Proofs (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.

[2] `https://github.com/amblafont/binding-debruijn`

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

**QIITs in type theory** Our work relies on a previous work of Kaposi and Kovács [25]. 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 $\mathcal{T}_{\text{QIIT}}$. 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, $\mathcal{T}_{\text{QIIT}}$ can be considered as a set theoretic QIIT. If this QIIT exists internally, then:

1. the notion of signature, as a context of $\mathcal{T}_{\text{QIIT}}$, is internalisable;

2. it can be proved internally that any other QIIT exists (see [25]).

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

It is not obvious that such a QIIT can be constructed without the need for quotients. My main contribution consisted in showing that this is actually the case, by providing an internal construction of $\mathcal{T}_{\text{IIT}}$ as a QIIT, with the help of the Agda proof assistant. Concretely, I define the untyped syntax of $\mathcal{T}_{\text{IIT}}$ with its typing judgements, using inductive types. Then, I show that this induces a model for the QIIT signature associated to $\mathcal{T}_{\text{IIT}}$. 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.

# 4 Formalisation of weak $\omega$-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 $\omega$-groupoid [37, 29]. Later, Brunerie [10] proposed a definition of weak $\omega$-groupoids which looks amenable to internalisation, and indeed, Nuo Li formalised it in the proof assistant Agda [6]. 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.

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 $\omega$-groupoid, in a *2-level type system.*

**Bridging the extrinsic and the intrinsic definition** In his PhD [10], Brunerie defines a dependent type theory *extrinsically*, i.e. through an untyped syntax with typing judgements. Then he defines weak $\omega$-groupoids as models (in a suitable sense) of this type theory. Brunerie performs a recursion on his type theory to prove that types are weak $\omega$-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 3, 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 [5], 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 $\omega$-groupoid.

# 5 Certifying compilation of Cogent

The seL4 project [28] has shown that verification of systems code is costly unless we invest a huge amount of proof engineering. Cogent [33] is a restricted functional language designed to mitigate this issue. From a Cogent program, the compiler generates a C program, an Isabelle specification, and a formal proof of correspondence between them, based on the Isabelle/HOL proof assistant.

I am the main contributor to the verification effort taking into account a new feature called Dargent [12] that enables the programmer to specify how Cogent data types are laid out in memory. Dargent is meant to bridge the gap between the C data structures used in the operating system, and the algebraic data types used by Cogent.

## 5.1 Overview of Dargent

Dargent offers the possibility to assign any (heap allocated) record type a custom layout describing how the data is mapped into a flat array of data. A record assigned with a custom layout is essentially compiled to such an array. 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 {foo : $\mathsf{A}, \ldots$} , the custom getter for the field foo takes an array as input and outputs a value of type $\mathsf{A}$. The custom setter for the same field takes an array of words and a value of type $\mathsf{A}$, and updates the array so that it represents the updated record.

## 5.2 Extending the verification framework with Dargent

Dargent affects the formalisation of the type system, and beyond C code generation, it also affects the generated correspondence proof with the Isabelle specification and the generated C code. Changes (see my Cogent branch at `https://github.com/amblafont/cogent/tree/dargentfull`) to the verification framework were necessary to take Dargent into account regarding various statements:

1. the correspondence between Cogent record values to C flat arrays;
2. the correspondence between record operations in Cogent and C;
3. the compositional properties of generated getters and setters.

I implemented specific Isabelle/ML tactics to take care of these proofs.

## 5.3 Applications

To demonstrate the improved readability of programs that Dargent offers, I reimplemented[4] the power control system for the STM32G4 series of ARM Cortex micro controllers by ST Microelectronics, based on a C implementation from LibOpenCM3[5], an open-source low-level hardware library for ARM Cortex-M3 micro controllers. The original C file[6] is about one hundred lines long.

I also verified a Cogent timer driver[7], using Dargent, based on a sixty lines long C implementation. The formalisation took advantage of the fact that the shallow embedding remains simple, as the layouts are fully transparent to the functional semantics of the Cogent program.

[4] `https://github.com/amblafont/dargent-examples/tree/master/libopencm3/STM32G4_pwr`

[5] `http://libopencm3.org/`

[6] `http://libopencm3.org/docs/latest/stm32g4/html/pwr_8c_source.html`

[7] `https://github.com/amblafont/timer-driver-cogent`

# Part II
# Research project

Formal verification of programs has significantly progressed since the last century. Let us mention the verified compiler CompCert [30] and the Verified Software Toolchain [7] for the C language, as well as Iris [24], a program logic implemented in Coq used to certify numerous large programs, in multiple languages such as Rust [23] and OCaml [32].

My research project aims at contributing to this domain by a fruitful mix of theoretical investigations and concrete practical implementations. It focuses on two different research directions inherited from my PhD and my postdoctoral position:

1. Constructing a mathematical theory of programming languages (Section 6), i.e., finding solid and stable mathematical notions of programming languages and compilation, so that various results, proof methods, and notions can be abstracted and generalised. An important effort will be devoted to finding adequate notions of signatures to specify and present programming languages and compilations, as well as implementing tools to offer specific computer-aided reasoning.

2. Certifying programs in a heterogeneous environment, focusing in particular on the concrete case of the COGENT programming language (Section 7), a language for certified systems programming, which thus inherently faces interoperability issues.

## 6 Theory of programming languages

The development of a theory of programming languages can be split into two research directions: first, devising a general notion of signature to specify programming languages and compilations between them; second, generalising well-known proof methods and algorithms.

### 6.1 Theory of signatures

**Linearity** Linear types have been used [27] to enforce memory safety properties at compile time, for example by forbidding referring twice to the same memory location. A general notion of programming language is expected to account for them. Let us consider the paradigmatic case of linear lambda-calculus. There, all variables must be used exactly once. As a consequence, terms do not support arbitrary substitution, such as non injective renamings. The mathematical notion of operads handles such cases [35], although the simply-typed setting has not been worked out yet. Programming languages that feature linear variables typically support non-linear variables as well. This happens for example in the quantum lambda-calculus, or in the COGENT programming language, where linearity of variables depend on their types. The operadic setting needs to be investigated to accommodate such mixed syntax. Beginning with the untyped case, I am looking for a monoidal product such that monoids support parallel substitution respecting linearity. Recent work in this direction by Hyland-Tasson [22] seems promising, although they are interested in slightly different linearity constraints. The next step will be to investigate how syntax of programming languages can be specified as monoids, using the standard device of endofunctors with strengths [15].

**Dependent types** Whereas there is a standard notion of binding signature for simply-typed syntax, the case of dependently typed languages is still an active field of research. I plan to investigate this question by focusing particularly on simultaneous substitution. In the simply-typed case, the monadic approach nicely accounts for them, but it does not straightforwardly apply to the dependently typed case, because of the nested dependent structure of contexts. Monadicity fails for simple examples, if one takes as basic notion of context a set of *types* $Ty$ together with, for each type $A \in Ty$, a set $Tm_A$ of *terms of type* $A$. Indeed, the category of contexts equipped with a type universe $\boldsymbol{U}$ and an explicit coercion from terms of type $\boldsymbol{U}$ to types is not monadic. However, the forgetful functor can be understood as the composition of two monadic functors, by considering the intermediate category of contexts equipped with an additional type $\boldsymbol{U}$. Such decomposition suggests to consider nested monads, or more precisely, compositions of monadic functors where the last one targets the category of basic contexts. The idea is that each layer adds a new construction to the dependently typed language. The link with other semantic models of type theories needs to be worked out. It is already clear that the domain category of this composition of monadic functors has a structure of category with families [14], under some technical conditions. But the main challenge consists in designing a notion of specification for these nested structures. In particular, stating an adequate recursion principle is intricate because naive notions of signatures are not functorial, contrary to the simply-typed case.

## 6.2 Generalising results and algorithms

Beyond providing a mathematical notion of programming languages, this project aims at accounting for standard algorithms (such as compilers) and results.

**Compilation** The general notion of programming languages and compilations that this project develops is meant to handle in the first place Plotkin's CPS translations [34], compilation of lambda-calculus to the Zinc or Krivine machine, or to combinatory logic. This requires further work on the available recursion principle. Indeed, in my recent work on a theory of syntax suited for De Bruijn encoding [18], the recursion principle is not powerful enough to compile easily between languages that do not share the same simple type system. Even the type erasure function that turns a well-typed term into a raw, untyped one is not straightforward. The setting of transition monads [17] can similarly be improved: the recursion principle can only be used to compile a programming language into itself, but with different evaluation rules. Such a case occurs for example in the COGENT programming language, which is given two operational semantics, and a correspondence theorem between those. However, compilations typically involve different source and target syntax.

I plan to exploit the recent work of Arkor-Fiore [8] that enables recursion between syntax with different type systems. In particular, I would like to adapt their work to support arbitrary simultaneous substitution, rather than unary substitution, as they did. Regarding the case of operational semantics, I will take advantage on my previous work on reduction monads [4] to generalise the recursion principle.

Other algorithms that I plan to explore are type soundness results for common typing algorithms, such as bidirectional type systems, in particular for system F and some of its extensions (such as recursive types).

**Congruence of bisimilarity** One motivation for the quest of a formal notion of programming language is that it then becomes possible to state precise theorems about a well-defined class of languages, instead of introducing proof methods whose scopes are not always clearly delimited. On this matter, I have worked on a general setting for Howe's method to prove that applicative bisimilarity is a congruence [9, 20]. It remains to be extended to cover more programming languages, such as the higher-order pi-calculus, or lambda-calculus with algebraic effects and handlers, or other useful notions of bisimilarity such as the normal or environmental one.

## 6.3 Local computer-aided reasoning

In a proof, difficulties are usually confined to specific particularly technical fragments. In this project, I propose to design computer tools to help proving correctness of such critical phases, in the particular field of operational semantics, and more generally programming language theory.

Such tools typically build upon successful proof assistants such as Coq, Agda, or Isabelle/HOL, but not exclusively.

**Graphical reasoning** Categorical proofs often involve showing commutation of diagrams of morphisms. Pen and paper are inconvenient for large diagrams, since the exercise may require some layout reorganisation. Moreover, translating such proofs in a text-based proof assistant such as Coq is tedious and repetitive. Motivated by this state of affairs, I have started implementing a commutative diagram editor[8]. The current preliminary version merely allows the user to generate latex from drawn nodes and arrows, but the goal is to interact with Coq, by generating proof scripts from a commutative diagram or conversely, by

[8] `https://amblafont.github.io/graph-editor/index.html`

building a commutative diagram helped by proved Coq theorems. Alternative kinds of diagrams will be investigated as well, such as string diagrams, e.g., when reasoning about monads.

**Solver for morphism equality** To help in my recent research, I wrote a Coq tactic[9] that automatically proves equalities of morphisms involving monadic structure. It relies on the definition of monads as extension systems [31]: the tactic chooses an orientation of the monadic equations to compute a normal form from a given composition of morphisms. Testing equality of morphisms then amounts to comparing their normal forms. I have extended and used this tactic in the setting of a category with coproducts, equipped with a distributive law of monads, to show commutation of large diagrams in some of my recent research. Completeness of this tactic is not clear, and it would be helpful to understand what is its exact scope.

**Type-theoretic free monads** Here, the motivation comes from an experiment[10] in Agda to prove commutation of a large diagram. I axiomatised an endofunctor on the category of types, and then realised that its free monad could be axiomatised as well, by introducing a type-theoretic (dependent) induction principle. I was then able to prove the desired result more efficiently than by the usual diagrammatic reasoning. However, the current method only applies to the category of sets. The project is to generalise it to abstract categories with suitable structure, which involves (1) delineating the structure in question, and (2) proving the soundness, and hopefully completeness, of the method. This begs the question: what is the internal language of categories equipped with a monad? Investigating this matter will result in new means of proving and constructing morphisms in such settings, beyond providing theoretical grounds for my experiment.

## 7 Interoperability for Cogent

The Cogent programming language is designed to ease formal verification of systems code, thanks to its compilation to C which is certified through multiple intermediate stages.

Cogent programs are typically meant to operate with external C code, which may be verified independently using the AutoCorres library in Isabelle/HOL. Connecting such external formalisations with the proofs generated by the Cogent compiler raises some challenges. Beyond this interaction with C code, a device driver implemented in Cogent needs to communicate with the device following a well-specified layout.

As far as formal verification is concerned, these interactions raise some additional challenges, that we plan to solve for Cogent, before investigating at a more theoretical level.

[9] `https://amblafont.github.io/stuff/eqsolver.v`

[10] `https://amblafont.github.io/stuff/freemonad.agda`

## 7.1 Verified bitfields

Communicating with a peripheral device often requires careful data formatting, which is prone to error, and whose formal specification is tricky. In the context of my postdoctoral position, I have worked on an extension of the language allowing the user to provide a layout that specifies how Cogent data-types should be compiled. For example, a record consisting of eight booleans can be compiled to a single byte in C. Getting or setting one of the boolean field then compiles to a call to a generated getter or setter that fetches or updates the right bit, according to the layout. It is highly desirable to systematically provide a formal proof of this fact, which is crucial when communicating with peripheral devices whose configuration registers follow a rigid format. As a first step, I have already implemented automatic tactics that prove that getters and setters are only concerned with the bits mentioned in the field layout. In fact, it is not obvious how to state the expected property in general, because the field types (and values) can be arbitrarily complex and nested. However, this task should be possible by exploiting the deep embedding of Cogent in Isabelle/HOL, which provides an algebraic representation of Cogent types and values. A first application consists in extending my formal verification of a Cogent timer driver[11] with the proof that the implementation respects the format of the device register. Furthermore, register layouts sometimes feature non-standard sized integers, e.g., identifiers on 21 bits. A first naive attempt consists in representing them with standard 32-bit integers. This, however, breaks the refinement proof between the compiled C code and the functional specification. Roughly speaking, the reason is that one should never update a 21-bit field with an integer that does not fit on 21 bits, but this precondition to the field setter is not enforced with the naive attempt. A solution consists in extending Cogent with new types, for arbitrary sized integers. The verification framework needs then to accommodate these types and take the above mentioned precondition into account. A first application of this extension that I will consider consists in implementing and verifying a driver for a CAN network peripheral device.

## 7.2 Memory model

Cogent is not compatible with the common pattern used in systems programming consisting in casting a structure pointer into another, to simulate structure inheritance, even if it is done in external C code. The reason is that the Cogent verification framework relies on an optional AutoCorres feature, called the heap abstraction. This feature simplifies the memory model by assuming that each memory location relates to only one C type. Pointer casting typically breaks this assumption. In order to handle theses cases, it is desirable to remove the dependency of the verification framework on this feature. More precisely, what needs to be adapted is the refinement proof between the C code and Cogent stateful semantics. This task could benefit from revisiting the way the heap

[11] https://github.com/amblafont/timer-driver-cogent

abstraction is implemented, since the memory model of the stateful COGENT semantics is analogous to the heap abstracted one.

Part III

# Teaching statement

## 8 Teaching experience

During my last postdoctoral position, I gave a few introductory talks about category theory for my team (about 4 hours in total).

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).

### 8.1 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

### 8.2 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

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

[2] 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.

[3] 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.

[4] 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.

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

[6] Altenkirch, T., Li, N., and Rypáček, O. Some constructions on $\omega$-groupoids. In *LFMTP* (2014), ACM.

[7] Appel, A. W. Verified software toolchain. In *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings* (2012), A. Goodloe and S. Person, Eds., vol. 7226 of *Lecture Notes in Computer Science*, Springer, p. 2.

[8] Arkor, N., and Fiore, M. Algebraic models of simple type theories: A polynomial approach. 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. 88–101.

[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] Chapman, J. Type theory should eat itself. *Electron. Notes Theor. Comput. Sci. 228* (2009), 21–36.

[12] Chen, Z., Jackson, V., Keller, G., Lafont, A., McLaughlin, C., O'Connor, L., and Rizkallah, C. Dargent, A Silver Bullet for Data Layout Refinement. Submitted at PLDI 2022 (`https://amblafont.github.io/articles/dargent.pdf`), Nov. 2021.

[13] De Bruijn, N. G. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. 381–392.

[14] Dybjer, P. Internal type theory. In *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers* (1995), S. Berardi and M. Coppo, Eds., vol. 1158 of *Lecture Notes in Computer Science*, Springer, pp. 120–134.

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

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

[17] 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.

[18] Hirschowitz, A., Hirschowitz, T., Lafont, A., and Maggesi, M. Variable binding and substitution for (nameless) dummies. Accepted at FOSSACS 2022 (long version available at `https://amblafont.github.io/articles/debruijn-extended.pdf`), Oct. 2021.

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

[20] Hirschowitz, T., and Lafont, A. A categorical framework for congruence of applicative bisimilarity in higher-order languages. Accepted at LMCS, under minor revision `https://hal.archives-ouvertes.fr/hal-02966439`, Mar. 2021.

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

[22] Hyland, M., and Tasson, C. The linear-non-linear substitution 2-monad. In *Proceedings of the 3rd Annual International Applied Category Theory Conference 2020, ACT 2020, Cambridge, USA, 6-10th July 2020* (2020), D. I. Spivak and J. Vicary, Eds., vol. 333 of *EPTCS*, pp. 215–229.

[23] Jung, R., Jourdan, J., Krebbers, R., and Dreyer, D. Safe systems programming in Rust. *Commun. ACM 64*, 4 (2021), 144–152.

[24] Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., and Dreyer, D. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program. 28* (2018), e20.

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

[26] 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.

[27] Klabnik, S., and Nichols, C. *The Rust Programming Language.* No Starch Press, USA, 2018.

[28] Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. sel4: formal verification of an operating-system kernel. *Commun. ACM 53*, 6 (2010), 107–115.

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

[30] Leroy, X. A formally verified compiler back-end. *J. Autom. Reason. 43*, 4 (2009), 363–446.

[31] Marmolejo, F., and Wood, R. Monads as extension systems - no iteration is necessary. *Theory and Applications of Categories 24* (01 2010), 84–113.

[32] Mével, G., Jourdan, J., and Pottier, F. Cosmo: a concurrent separation logic for multicore ocaml. *Proc. ACM Program. Lang. 4*, ICFP (2020), 96:1–96:29.

[33] 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.

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

[35] Tanaka, M. Abstract syntax and variable binding for linear binders. In *Mathematical Foundations of Computer Science 2000, 25th International Symposium, MFCS 2000, Bratislava, Slovakia, August 28 - September 1, 2000, Proceedings* (2000), M. Nielsen and B. Rovan, Eds., vol. 1893 of *Lecture Notes in Computer Science*, Springer, pp. 670–679.

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

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

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