Research Statement

Gabriel RADANNE

Part I. Report on past activities

Programming is difficult. Even more difficult is the task to write *correct* programs. Many techniques have been proposed to establish the correctness of programs such as testing and verification. All these techniques can be further improved by directly improving the main tool of programming: the programming language itself.

Unfortunately, most programming languages are insufficient for the many kinds of programs written in them: from concurrent programs written in C (whose memory model, while allowing efficient code, is very difficult to master) from large Web applications written in Javascript (who provides neither encapsulation nor abstraction facilities, thus hindering many modularity aspects), programming is a constant struggle to torture the code the programmer wants to write into the code allowed by the programming language. Furthermore, most programming languages do not help in any way in ensuring the correctness of programs.

Improving programming languages can lead to many benefits. For instance, rich type systems have been used to check properties of increasing complexity and diversity, directly while programming. Carefully designed programming languages are also more amenable to static analyses or elaborate optimisations. Improving languages is not only useful for correctness: modern languages provide statically checked documentation, accurate tab-completion for methods in modern editors or “fearless refactoring”: the ability to do complex refactoring simply by following the compiler errors. In all these cases, the overall safety and ergonomics or the language is improved by designing it hand-in-hand with associated analyses, either type systems or static verification.

This ability to help programmers write programs correctly has lead to a recent resurgence of languages supporting strong static checking for many different domains. Facebook introduced Hack and Flow [3, 2] to statically type their gigantic PHP and Javascript code-bases and Infer [9] to statically verify their mobile applications (in many different languages, from Java to Objective-C). Mozilla created Rust to help design the new generation of browsers (which rivals operating systems in complexity nowadays). Microsoft introduced Typescript and F# to write rich Javascript and native applications. JetBrains designed Kotlin, a JVM language designed from the ground up to support excellent IDE integration through various static analyses. All these languages feature rich type-systems or analyses that have been tailored for their use cases, from ownership in Rust to separation logic in Infer.

My research activities are centered around programming languages: how to make them more expressive, safer, easier to use, and scalable to larger ecosystems. To this end, my research activities can be summarized as follows:

- I design Domain Specific Languages (DSLs) that extend existing languages with new constructs appropriate to the domain at hand. In this context, I often focus on two aspects: providing expressivity, safety and ease-of-use through *rich type systems* and ensuring good scaling capabilities of the associated ecosystem through *encapsulation and modularity*.
  - As part of the Ocsigen team, I formalized and improved Eliom, an extension of OCaml for Web programming [38, 35]. We also described how such constructions can be used in practice for Web programming [37, 36].
  - With James Cheney, Sam Lindley and Philip Wadler, we designed a new way to integrate database queries in statically compiled programming languages [10].
  - As part of the MirageOS team, I developed a configuration DSL for "free-standing" applications (that do not need an underlying kernel). The resulting DSL is now used pervasively in the MirageOS ecosystem.
- I also design analyses for existing languages in order to improve safety.
  - With Laure Gonnord and David Monniaux, we proposed a novel and efficient algorithm for checking termination of imperative sequential C programs, which are often used in embedded programming [19].
  - With Peter Thiemann, we are currently designing a linear type system that integrates well in languages of the ML family. This type system is particularly appropriate to write safe implementations of network protocols.
- To be really successful, any domain needs an ecosystem. As such, I also design libraries that complement the language.
  - With Peter Thiemann, we developed a fast and productive algorithm to generate test-cases for regular expression engines [34].
  - I developed a library to write typed regular expressions that are safer and easier to use, without compromising performances [33]. These typed regular expressions have been used notably for URL routing and log parsing.
  - I made many contributions to the OCaml ecosystem. One of my area of focus is to make embedded DSLs easier to define through syntax extensions.

Most of my research also lead to associated software productions. To reflect this, many sections are annotated with software self-evaluations following the INS2I model described in `https://csins2i.irisa.fr/files/2016/09/FicheLogicielsCliquableCSI-INS2I.pdf`.

1. Modularity and expressivity for tierless Web programming

My PhD thesis focused on the design, formalization and development of the Eliom language, an extension of OCaml for Tierless Web Programming. The theoretical aspects were published part in APLAS 2016 [38], part in my thesis [32]. Practical studies have been presented in IFL 2016 [37] and WWW 2018 [36].

Context Web applications are usually decomposed into tiers: one program runs in the browser, usually written in Javascript and one program runs on a server, which can be written in many languages, from Java to PHP. Tierless languages allow programmers to write one single program that embodies all the aspects of a Web application: client, server, databases, ... This allows to put together related code, regardless of where it will be executed. Tierless languages can also check properties for the complete application, such that client and server agree on the type of their communications. The tierless program will be automatically decomposed into its various parts, either statically by a compiler, or dynamically during execution.

Tierless languages are generally new standalone languages, making the set of available libraries limited. Furthermore, analyses used for tier separation are often either dynamic, difficult to marry with a static type system, or highly non-modular. Modularity and separate compilation are nevertheless critical in Web programming workflow as they shorten the iteration cycle during development with faster compilation and provide separation of concerns for libraries.

The Ocsigen project aims to equip OCaml with tools and libraries for Web programming. OCaml is an industrial strength functional language with a rich static type system which can enforce many guarantees. The Ocsigen project leverages this type system in many way, for instance to ensure that HTML is correct, avoid dead links, or guarantee that forms are well typed. To go even further, Ocsigen also contains Eliom, an extension of OCaml with specific constructs for Web programming. OCaml comes with a rich language and a sizable ecosystem, but also a set of constraints: OCaml is a strongly typed language with static separate compilation. As pointed out before, tierless languages are usually incompatible with these features. Furthermore, the original implementation of the Eliom language was unsound and lacked proper integration with OCaml's type and module systems.

1.1. A new Eliom calculus

During my thesis, I reformulated the Eliom language, formalized it as a calculus that could be reasoned about and solved the incompatibilities between tierless and modularity by providing novel semantics and type and module systems that draw inspiration from staged-metaprogramming. So far, Eliom is the only tierless language that provides both static typing and static separate compilation. Using together rich type and module systems allows programmers to easily write tierless libraries containing client and server code in a modular and encapsulated way.

When Web programming rhymes with meta-programming In Eliom, programmers can interleave client and server pieces of code. This corresponds naturally to the idea of staged meta-programming where server code contains pieces of client code that can be composed, similarly to Lisp quotations. One traditional interpretation of stage languages is partial evaluation: the first stage (here the server program) is executed first and emits a second program (here the client program), which can be evaluated next. However, in the context of Eliom, we would like to separate statically client and server and executes them as two independent programs. Such separation should also ensure that communications between client and server are feasible and sound, a problem also known as "cross-staged persistence" in the meta-programming world.

In order to provide static and type safe slicing, I placed careful restrictions on the interleaving of client and server code. I then defined two semantics: a dynamic one which is close to the notion of partial evaluation and easy to explain, and a compiled one which is more complex but efficient. I also equipped the language with a novel notion of "converters" and an associated type-system to ensure that cross-stage communications are safe and easy to customize to the programmer's need. Finally, I proved that the compiled efficient semantics is equivalent to the easy-to-understand semantics. This equivalence is particularly important, as it means that the simple semantics can be explained to programmers to reason about their code, while the implementation uses the efficient compiled semantics. An initial version of the expression language was presented at APLAS 2016 [38], a more complete version is available in my thesis [32].

Tierless modules After ensuring that Eliom's expression language was compatible with static compilation and strong typing, another question appeared: how to scale it up? Programs are composed of many modules and each could contain both client and server parts. We could like to compose such modules in a way that is modular and encapsulated, but still preserves the type-safety and efficiency of the core language. For this purpose, I introduced tierless modules which combines an ML-like module system with tierless annotations. This module system integrates seamlessly in the host system, ensuring that it can use all its libraries. In practice, this means that Eliom programs have access to the complete OCaml ecosystem transparently. This new module system also supports abstraction and separate compilation and preserves the semantic equivalence discussed above. This module language is available in my thesis [32].

Eliom in practice As Robert W. Floyd said, "To persuade me of the merit of your language, you must show me how to construct programs in it"[16]. The Ocsigen team and I developed examples and libraries that use Eliom's new constructs to implement common Web programming idioms such as RPCs, HTML manipulation and functional reactive programming in a concise, safe and efficient way [37]. We also showed how Eliom modules allow to organize tierless programs in modular components that preserve separation-of-concerns [36].

1.2. Contributions to the Ocsigen project (A5 SO4 SM4 EM4 SDL5/DA2 CD3 MS3 TPM3)

While my research focused on the Eliom language, the Ocsigen project is composed of many different pieces to compose a complete Web programming framework. Since 2014, I have been involved in most pieces of the Ocsigen ecosystem, including a Webserver, the compiler from OCaml to Javascript, the original implementation of the Eliom language and the set of related Web programming libraries. As part of the Ocsigen team, I developed features and maintained many of these libraries. These contributions have improved both programming and research aspects, allowing me to tailor the Eliom language to support writing these libraries and also improve the libraries to best use Eliom's strengths. Additionally, I am the main developer of two particular pieces of this ecosystem.

TyXML (A5 SO3 SM5 EM4 SDL5/DA4 CD4 MS4 TPM4) TyXML [47] is a library to write HTML and SVG documents whose validity is guaranteed by the OCaml type system. While the idea of encoding validity of HTML documents in typing is not novel, TyXML is the only largely-used library which cover the complete specification and is easily accessible. Since 2014, I am the main maintainer and developer of TyXML. My main contributions were to make TyXML more accessible by diversifying its application domain (to build Dom trees, reactive HTML, . . .) and by providing a syntax extension that allow to directly use the regular HTML syntax. TyXML is commonly used in the OCaml community to write HTML Websites and packaged in large distributions such as Debian.

A new implementation of the Eliom language (A2 SO4 SM2 EM2 SDL1/DA4 CD4 MS4 TPM4) The original implementation of Eliom was unsound and had limited inference. During my thesis, I created a new implementation of the language by directly modifying the OCaml compiler [14]. While still a prototype, this new implementation is feature-par with the original version, is sound, and supports new features such as tierless modules. It also features streamlined compilation model and type system that are more in line with the theory. Some of the improvements made while developing this new prototype were contributed back to both the original Eliom implementation[1], and to the OCaml compiler[2] and are now in released versions. At the time of writing, the main limitation of this new prototype is that the surrounding tools, in particular build systems, have not been ported to the new compilation model implemented in the prototype.

2. Integrated database queries

One additional tiers relevant to Web programming are database queries. Queries for databases such as SQL are often used as part of a larger program and thus called from a host programming language. Two main approaches are available to integrate queries in a programming language: either identify through typing or static analysis which part of the language can be turned into a query, or allow quoted queries in a general language. The former is nicer to use as it allows the programmer to write queries like any other function of the language thus providing excellent integration, but is is unclear how to compile and optimize such queries statically. The later however is very easy to compile statically.

With James Cheney, Sam Lindley and Philip Wadler, we showed that the two formulations are equivalent, and provided code transformation from one to the other. I participated in the design of the transformation from integrated queries to quoted queries using normalisation-by-evaluation, implemented it in the Links [11] compiler and measured that this transformation does not compromise efficiency of the queries. This work was published in PEPM 2014 [10].

3. Modular operating systems

As part of the MirageOS team, I designed a configuration DSL for MirageOS [27, 26], a library-operating-system to create free-standing applications that do not require an underlying kernel. The resulting DSL and tools are now the main mean of creating MirageOS applications and is the central piece of the ecosystem. The associated article will be soon sent for review.

Context Modern operating systems are large: they contain many drivers and interfaces to handle modern hardware. In the context of services exposed to Internet, this size increases the weight of the service and its attack surface. Unikernels [25] are applications that directly contain all the pieces of operating system they need, such as network stack or file system drivers. Unikernels can be optimized for efficiency and size and can minimize their attack surface by shedding away unnecessary components.

MirageOS is a “library-operating-system” to build unikernels in a modular way. MirageOS unikernels can be instantiated on various architecture (Unix, Xen, bare-metal ...). A key idea of MirageOS is to leverage ML-style modules to describe complex components as functors: modules parameterized over other modules. A network stack is implemented in term of a network interface and an IPv4 resolver. Alternatively, a network stack could also be implemented directly with a socket on Unix. The finaly application usually does not care how the network stack is implemented, simply that it exists. At the time of writing, the MirageOS ecosystem contains around 70 modules, a dependency depth of up to 10 and more than 5 targets. Given the size of the ecosystem, such a flexibility creates a combinatorial explosion in configuration choices that can not be handled manually. One proposal to tame this complexity is to use a description language where users can express their configuration choices with the appropriate level of details.

[1]For instance Eliom pull-request #387
[2]For instance OCaml pull-requests #1699, #1703 and #1704

A DSL for configuring unikernels I designed a new configuration DSL that allow to specify rich configurations easily. The configuration space is described by the user as an OCaml file using an embedded DSL. A tool then allows to choose concrete instances in this space at *configuration-time*, for instance “I want to target Xen and use a network stack with a DHCP”. Users can provide concrete values (for instance, the IP address of the service) both at configuration or runtime through a new notion of “keys”. They can also define new custom keys tailored to their usecases. The main insight of the DSL is that configuring unikernels is in fact a meta-programming problem: A configuration is a meta-program that will emit the glue code necessary for the application to run. The difficulty here is to ensure that the end state is consistent, in particular when configuration choices and keys are intertwined, and to make the resulting description language accessible to novice OCaml programmers. Thanks to careful restrictions in the dependencies between configurations and keys, the tool can derive static knowledge on the configuration, including documentation. This DSL now covers the complete MirageOS ecosystem and is used by all MirageOS users, including operating system specialists that have limited familiarity with OCaml.

Functoria, a library for configuration DSLs (A5 SO3 SM4 EM4 SDL5/DA4 CD4 MS3 TPM3) The idea of using a description language to orchestrate modular ecosystems using functors is in fact very general and can be adapted to other contexts. To demonstrate this, I designed an abstract notion of configuration DSL that is independent of MirageOS. I then created Functoria [17], a library to build such configuration DSLs for various ecosystems. The designer of the domain-specific ecosystem simply needs to inform Functoria of its runtime semantics to obtain a fully-featured configuration DSL. This library is not only used to build MirageOS’s own DSL, but was applied as prototype to other projects such as Owl [49], a scientific computing library, and LuaML [18, 39], a library to build modular Lua interpreters. Such a general method to build configuration DSLs could be used to inform and direct the construction of new modular ecosystems by directly providing advanced tooling to many different domains.

4. Termination checker for C

With Laure Gonnord and David Monniaux, we investigated the question of program termination for numerical imperative programs. Our algorithm was implemented in Termite[3], a tool to check termination of C programs, and published in PLDI 2015 [19].

Context The C language is often considered both too limited and too hard to use safely. However, in contexts such as embedded programming, C is the right tool for the job as it allows to manipulate the underlying architecture precisely. Nevertheless, we would still like to improve the safety of embedded programs. One of the big project of the last few decades has thus been to statically verify that C programs verify as many properties as possible, as demonstrated by tools like Frama-C. One property of particular interest is termination. In contexts such as real-time software or cryptography, showing that a given procedure terminate is a first step in showing that such procedures respect critical time properties. Of course checking termination is undecidable in general, but it can be checked in some cases. One tractable class of programs are numerical imperative programs with *linear* ranking functions. Ranking functions are integer measures that are positive and decrease during execution of the program. By definition, if such

[3]https://termite-analyser.github.io/

function exists, the program terminates. Unfortunately, previous methods to infer linear ranking functions had difficulties with large programs.

Termite (A2 SO4 SM3 EM1 SDL4/DA4 CD4 MS4 TPM4) We proposed a novel algorithm based on incremental generation of constraints that is both more precise and faster than the state of the art. The idea is to proceed in an iterative manner: if at some point we generate a counter-example, the program doesn't terminate. If we explore all the cases, it terminates. We showed the correctness and termination of the algorithm. I extended the algorithm to support programs with multiple control points and to work on the Single State Assignment representation which is often used in modern compilers. I then implemented the extended algorithm as a verification pass for LLVM, a mainstream C compiler. The resulting implementation was precise enough to verify significant parts of various verification test-suites and several orders of magnitude faster than most concurrent analyzers. In particular, it was able to instantly check the termination of subtle algorithms like heapsort.

5. Linear types for ML

With Peter Thiemann, we are working on linear types for languages with full type inference. The system has been implemented as a prototype type checker and the associated article will be soon sent for review.

Context One difficulty in programming is handling the lifetime of resources such as file system handles or network connections. Resources are created, used, then destroyed. Resources should of course not be used after having being destroyed, and this is the source of many bugs (so-called "use after free" errors). One lead to facilitate the correct handling of resources is to use type-systems to verify that such resources can not be used by different parts of the program. This is the base of "linear" type systems and spawned many different adaptations, notably the Rust language.

Linear types are also essential to implement "session types". The goal of session types is to record the correctness of protocols in types. For instance, a channel could have a type mandating that the user must first send a number, then receive either "accept" or "deny". Duplicating such channel would allow users to send a number twice, which would not conform to the protocol.

Using a rich language of the ML family to describe protocols is very desirable and has in fact already been largely accomplished as part of the MirageOS ecosystem (Section 3). While ML type systems already prevent large classes of bugs, they don't prevent linearity errors. Unfortunately, linear type systems are often difficult to adapt to languages of the ML family which are strict, impure and support type inference and a module system. Notably, full type inference (where types do not need any annotation, even when declaring functions) is often sacrificed in the profit of richer systems in the context of linear types.

Inference for linear types With Peter Thiemann, we are working on designing a linear type system for ML languages. In particular, I worked on designing a complete and sound type inference algorithm for our system that is suitable for retro-fitting on top of an existing ML-like language. While not as powerful as some other languages, it supports specific programming styles that are in need of linear types, such as protocol implementations with session types. To marry linear typing and full type inference, I leveraged type checking by constraints with a novel use of kinds and qualified types.

6. Practical tools for regular expressions

Regular expressions are one of the most commonly used instrument in a programmer's toolbox. One of my side interest is to design tools and libraries to improve the ease-of-use and safety of regular expressions. With Peter Thiemann, we developed an algorithm to generate test-cases for regular expression engines which was published in GPCE 2018 [34]. I also designed a library for efficient type-safe regular expressions which was published in PEPM 2019 [33].

6.1. Testing regular expression engines (A3 SO4 SM3 EM3 SDL5/DA4 CD4 MS4 TPM4)

Given the history on finite state automatons, the field of regular expression matching is surprisingly flourishing: the research on matching algorithms is active and new algorithms are regularly implemented. As such, one might want to test such regular expression engines by giving examples that should or should not be accepted. This is not so easy: random generation is not sufficient (the chance of hitting a string accepted by `a*` is 0 at the limit) and would require an external oracle. All these issues make it difficult to design automatic testing for regular expression and forces authors to manually write tests.

To solve all these issues, Peter Thiemann and I proposed an algorithm to generate the language recognized by *extended* regular expressions with a complement operator [34]. The presence of a complement operator allow to easily generate words both accepted and rejected by a given regular expression. The key insight was to rely on the series interpretation of languages using a length-lexicographic ordering. I also investigated the use of appropriate data-structures to improve the performance of the algorithm. We implemented our algorithm in Haskell (by Peter Thiemann) and in OCaml (by me) and showed it was efficient in practice. The OCaml implementation was distributed as the Regenerate library [40] and is now used to test various regular expression engines, including a production-ready one.

6.2. Typed regular expressions (A4 SO3 SM3 EM4 SDL5/DA4 CD4 MS4 TPM4)

A common issue faced by programmers is input validation: "Is my input data in the format that it should?" Another question immediately follows: "How to extract the interesting parts?" Regular expressions are often used to parse input, but do a poor job at the second task. Most regular expression engines simply return a list of captured strings without information regarding repetitions and branches. Some algorithm return the tree parsed by a regular expression, but none are implemented in efficient popular engines. I created a method to implement typed extraction on top of mainstream regular expression engines by using 2-layers regular expressions [33]. A typed layer, for which we rebuilt the complete parsing tree, sits on top of a untyped layer. This method does not compromise performances and also provides printing and multi-regex matching. I implemented this method as an OCaml library [46] which is now used by external users, notably for URL routing and log parsing. My technique was also adapted to create an Haskell library [45].

7. Involvement in the OCaml ecosystem

I am a prolific contributor to the OCaml ecosystem and have contributed, either through small patches or as a main developer, to many different libraries. I have already detailed some of these libraries in the previous sections. Here are some of my most important contributions that are not directly related to the previous activities. Many of these contributions aim to improve the support of domain specific languages in OCaml, notably through syntax extensions.

The OCaml compiler (A5 SO3 SM4 EM4 SDL5/DA2 CD3 MS2 TPM1) Due to my work on the new Eliom implementation presented in Section 1.2, I am very familiar with the internals of the OCaml typechecker. Since then, I have made several contributions, mainly in two areas: make the syntax of the OCaml language more flexible and more amenable to syntax extensions, and cleanup the internals of the compilers to make it easier to extend. Thanks to my expertise in OCaml's type and module systems, I have also been one of the main reviewers for changes to the typechecker made either by core OCaml developers or external contributors.

The ocaml-community project Since 2018, I'm one of the founders and stewards of the ocaml-community project[4] which aims to adopt largely-used OCaml package that lack maintainers and ensure that they receive essential updates and fixes. The project has now adopted 9 projects that are used pervasively in the OCaml ecosystem, such as `utop` or `calendar`.

The great PPX migration (A5 SO1 SM4 EM4 SDL5/DA4 CD3 MS4 TPM3) Historically, OCaml used a system called "camlp4" to create syntax extensions. In 2015, this system was deprecated in favor of a more stable and integrated system called "PPX". I was one of the early proponent to push this new system by migrating many old syntax extensions (`js_of_ocaml`, `lwt`, `tyxml`, . . .) and created or contributed to new ones (`ppx_tyre`, `sedlex`, . . .). I have also made significant contributions to the compiler to generalize and ease the writing of PPXs. The extensions I implemented are now commonly used in the OCaml ecosystem.

ocp-browser (A5 SO2 SM3 EM3 SDL5/DA3 CD4 MS4 TPM3) `ocp-browser` is a tool to interactively explore modules available on the programmer's system. It leverages the tooling infrastructure for `ocp-index`, an auto-completion engine for text editors. In 2013, I reimplemented `ocp-browser` based on an early prototype by Louis Gesbert to make it usable, accessible and fast. Since then, it has been part of standard tooling for OCaml, and is often recommended to new programmers.

Polyglot OCaml bytecode (A- SO3 SM3 EM3 SDL4/DA4 CD4 MS4 TPM4) File formats such as PNG, PDF or binary executables are all defined by languages that describe their internal data. Unfortunately, such languages are often under-specified through vague specifications written in English. Consequently, parsers for these languages are common sources of security vulnerabilities. For instance, at the time of writing, there are more than 700 vulnerabilities regarding PDF.

A playful way to explore these file-format languages is to create polyglots: files that are valid in multiple formats (for instance, a file that is both a PDF file and a PNG picture). This idea has been notably explored by Ange Albertini [4]. I created a tool [8] that takes an arbitrary PDF file, an arbitrary OCaml bytecode and creates a single file that is both a valid PDF and a valid bytecode. This tool exploits the internal structure of the PDF and bytecode file formats in unexpected ways to make an interleaving that is valid for both. This tool has been used on "real world" files (notably, research articles and their associated prototypes). The underlying technique will be published in the next issue of POC||GTFO[5].

[4] https://github.com/ocaml-community/

[5] https://www.alchemistowl.org/pocorgtfo/

Part II. Research Project: Domain Specific Type-systems

In my previous work, I developed new programming language features such as type systems or static analyses to create safe and convenient languages adapted to various domains. Indeed, Domain-Specific Languages (DSL) are instrumental in exploring new complex use cases through novel programming techniques. DSLs have been at a forefront of recent computer science innovations, from blockchains (contract languages [24]) to quantum computing (Quipper [20], Q#, ...), including machine learning (TensorFlow [13], ...). DSLs in general can be decomposed into two families.

- **Standalone** DSLs form a separate language. They are often highly-specialized to their domain, and can benefit from semantics, type systems and analyses that would not be usable for general-purpose languages. Concrete examples include GPU kernels (often expressed in a constrained subset of C), SQL, or grammar specifications in Yacc.
- **Embedded** DSLs extend an existing programming language. They should propose excellent integration into their host language, but requires their host language to be powerful and extensible. Embedded DSLs are closely related to libraries. Indeed, in many ecosystems, libraries can leverage some forms of meta-programming to provide convenient APIs adapted to the domain. These DSLs can be very distinct from their host language such as LINQ (Query sublanguage in C#), or tightly integrated like LMS (metaprogramming in Scala) or openMP (parallel code in C through custom pragmas).

Developing DSLs with custom semantics is reasonably accessible by creating libraries, either with or without a custom syntax. Such DSLs can however strongly benefits from special purposed type systems to improve both correctness and usability. Defining custom type systems, notably for embedded DSLs, is particularly challenging and generally done in an ad-hoc manner. Another recurring key question is how to integrate such DSLs into existing ecosystems and how to support modularity and encapsulation. Programming nowadays is mostly about using existing libraries. To be useful, a DSL should integrate with an existing host ecosystem seamlessly.

My research project is to promote and develop the use of statically typed DSLs by making them more powerful, easier to use and easier to define while, following my previous activities, demonstrating them on concrete application domains. In the long term, I would like to make DSL design as accessible as possible. On the standalone side, while each design tends to be ad-hoc, I would like to develop a toolbox of DSL techniques, notably regarding modularity, abstraction and integration with external ecosystems. On the embedded side, my goal is to make the definition of integrated DSLs as easy as declaring regular libraries, while still preserving their specificities in term of type system and execution model. This also makes my work not only applicable to regular DSLs, but also to libraries that requires complex interactions with the host language. While I intend to develop theoretical tool and methods that are applicable to many languages, I also want to implement them concretely. For this purpose, I often use OCaml as an experimental platform for my research ideas.

Foremost, my project is to develop tools and methods to create new DSLs. This notably involves exploring various specific domains in details to investigate their needs, as presented in Section 8. To support the definition of such statically typed DSLs, new techniques must be created to ease their developments, notably in two directions.

- To encapsulate properties specific to each domain, typed DSLs need to extend existing type-systems with new domain-specific rules. Section 9 presents leads on extending type systems to ease definition of typed DSLs.
- DSLs must be included in their host ecosystem. Section 10 aims to improve the support of modularity for programming languages in the context of modern, large ecosystems.

The area of domain specific languages and programming language theory are strangely often put apart. Indeed, people with domain-specific knowledge often discover strong use cases which lead to the creation of new languages that could benefits from programming language theory. The opportunity to develop new domain-specific languages will only grow further, as programming is now used in nearly all other scientific domains. Given my contributions to both domain specific ecosystems and programming language design, this project can be seen as an attempt to bridge these two worlds. As such, I will label my various projects with two labels of varying intensity: "DSL" indicates the need for some domain-specific knowledge (which I may not yet possess, but could be complemented by the host team), and "PLT" (Programming Language Theory) indicates some theoretical work on programming languages. Projects are also annotated with an estimated timing scope.

8. Domain-specific programming language constructs

The best way to know what DSLs need is to investigate concrete use-cases and domains that would benefit from such approaches. This approach lead me to investigate domains such as Web Programming or Unikernels in the past. Similarly, I want to explore various application domains and creates both generic and specific language constructs that make such DSLs easier to write and more powerful. For many domains, both standalone and embedded DSLs can be considered (for instance, database can be used through SQL queries and through embedded ORMs). As such, I will often investigate both approaches, depending on the particular task at hand.

8.1. Typed Manipulation of Foreign Data

Short-medium term (**DSL**, PLT)

Many generic data formats such as XML or Json have a type system. This type system is sometime explicit (schemas for SQL tables and XML, GraphQL, . . .) and sometime implicit with relation to the data. Unfortunately, the data's type is not always available directly to the programmer. By making the host language aware of the data's specific type system, we can improve both usability and safety. This line of thinking has been well explored for some datatype (for instance, the CDuce project[7] for XML) but many modern data-format remained unexplored, in particular in the context of Web programming.

Unfortunately, this work is very specific to each data format and is difficult to make general. Some generic-purpose features such as F# type-provides attempts to ease the introduction of externally-specified format into the host language, but do not replace a complete specification for a given data format. I plan to investigate type systems for such foreign data by leveraging extendable type-checkers, as described in Section 9.1, for concrete data-formats used in modern web programming and databases.

8.2. Protocol formats and Security

Medium-long term (**DSL**)

While generic data-formats can rely on schemas (or their equivalent) for typing information, other data-formats are simply specific to a given task. A particularly interesting format family are binary formats used in security protocols. In the recent years, these formats have been identified as recurrent sources of vulnerabilities due to poor specifications and hand-written insecure implementations. Infamously, the Hearthbleed bug was caused by improper input validation in the openSSL's TLS parser. Similarly, numerous vulnerabilities have been found in ASN.1-based parsers [1], a description language used by many protocols. For instance, lack of validation in X.509 parsers in popular browsers have been shown to lead to serious security vulnerability in HTTPS [6].

Parser generators are a promising lead to write efficient yet safe parsers for such formats. This approach has found some leeway in the LangSec community [23, 30, 22]. Unfortunately, such formats often do not conform to grammar classes where efficient parser generators are known. This is due to the reliance on fields that specifies the lengths or offsets of other fields.

I want to develop a description language to specify binary formats. Such description language could be used to statically verify formats, experiments with new formats and generate fast and safe parsers and printers. Developing such a language would involve several tasks such as describing new classes of grammars that are more adapted to binary formats and can be parsed efficiently, defining analysis to check the validity of a given descriptions and implementing descriptions of real-world formats. The main goal would be to ensure absence of bugs that lead to common security issues while still preserving efficiency. One (ambitious) application would be to replace parsers contained in current security libraries.

8.3. First Class Optimisations

Medium term (DSL, **PLT**)

A well known issue of optimizing compilers is their lack of predictability. Compilers are mostly black boxes that will or will not apply optimisations depending on the precise shape of the code, the current information available and the temperature of the room. To alleviate this issue, compilers often propose annotations, or pragmas, to direct the compiler more precisely. The programmer must choose which optimisations should be applied and how, and encode those decisions in the (often limited) pragma language.

In the context of highly specific libraries and embedded DSLs, these annotations suffer from severe shortcomings. First, to be effective, these annotations should not be in the implementation of the DSL, but directly in the user's code, forcing the user of the DSL to learn about internal details of the implementation. Second, such annotations are often specific to the compiler and not phrased in term of the domain at hand. They often become some magical invocation that must be inserted ("You shall always enable `ftree-vectorize`" . . .), which makes the code more verbose and less flexible. Finally, DSL authors are limited to optimisations provided by the compiler. Complex domain-specific optimisations are unlikely to be provided. These limitations can be circumvented through preprocessors, as often seen in C (GPGPU kernels, openMP). These approaches are often unprincipled or require special compiler support.

I propose to put the design and control of domain-specific optimisations back into the hand of the library author through the use of staged metaprogramming. Staged metaprogramming enables programmers to manipulate code as first class, similarly to lisp quotations. Library authors can then design code transformations and optimisations simply as functions. Since such functions are directly part of the language, DSL users are then able to use and customize such optimisations in term of the specific domain.

In the past, I used staged metaprogramming in both standalone DSLs (for MirageOS's configuration) and embedded DSLs (Eliom's expression language). In the context of standalone DSLs, such first class optimisations can be integrated directly in the language through compiler-provided meta-compilation. In the context of embedded DSLs, it can be provided through frameworks such as LMS, Template Haskell or MetaOCaml, or direct compiler supports as in D. In all these cases, writing concrete code transformations for a given specific domain is still difficult. I plan to explore ways to make such definitions easier and more accessible to authors of DSLs, while applying it to concrete use-cases, notably in the context of High Performance Computing.

9. Rich type systems for embedded DSLs

As said previously, embedded DSLs such as the ones presented in Section 8 require a tight integration with their host language and are often restricted by the expressivity of the type system of the host language. The holy grail, in this case, would be to extend the type system of the host language easily, safely, and in modular way. Of course, this dream is mostly inaccessible: type systems are known for being very tightly specified and not easy to extend, even less so in a modular way. However, one potential (limited) solution is to use type checking by constraints. The idea of type checking by constraints is to decompose type checking in two phases: a constraint *generation* phase, which emits constraints based on the shape of the program, and a constraint *solving* phase, where the constraints are solved by a constraint solver. Extending the constraint generator is then easy: as long as the constraint solver can handle them, we can emit whichever additional constraints we want.

Such approach has been shown to scale to fairly large languages (notably, Haskell), and many features (value restrictions, GADTs, type elaboration, including my work on linear types through constraints). Nevertheless, before using these ideas to define typed DSLs as those shown in Section 8, there remain many issues to explore regarding extensibility, ease of formalization and scaling to complex features.

9.1. Extensible and Customizable Type Checker — Long term (**PLT**, DSL)

While constraint-based typecheckers might seem easier to extend than traditional techniques, there remains many questions: how to extend the system without compromising its soundness, or how to handle type errors in the embedded DSL? Another important aspects is the actual API used to extend the type system. A very low level API would be to emit constraints directly. However, this would restrict DSL design to experts with a very good understanding of the typechecker implementation. Another proposal would be to use a slightly higher-level API that shield the DSL author from difficult notions such as unification and generalization. Similar ideas have been used to provide customize error messages for embedded DSLs [44], but not to extend the type system itself.

I want to develop type-systems (and their associated checker) that can be extended as part of an embedded DSL. Such type-systems would allow to have embedded DSLs developed almost like libraries, and be invaluable to develop the languages presented in Section 8 or for Eliom's extended type-system. Such approach could also open the door to niche or experimental features without directly integrating them in the language, as usually done in compilers like GHC.

9.2. Ease formalisation of new type systems

Medium-long term (DSL, **PLT**)

Formalizing a type-system usually aims to prove it correct with respect to a specified semantics, ie. prove that well-typed programs do not crash. How to mechanize semantics is a well explored field and various fairly generic methods have been proposed [5, 29]. Mechanizing type-systems in theorem provers, however, is still done in ad-hoc manner that often requires a large amount of boilerplate. Furthermore, such formalization often only covers the core type system, without mentioning type-checking or type-inference algorithms. Formalizing every prototype domain-specific type-system in such a manner would be a prohibitive time-investment.

Some frameworks permit writing typing rules in a free-form manner without verification [15, 43]. Other methods attempt to provide a more principled system through constraints [31]. Yet other methods propose to directly export to theorem provers, but with little guidance towards a type-checking algorithm [28]. None of these frameworks support writing type-systems in a descriptive manner that can be used both for execution and reasoning.

One interesting lead is the notion of scope graphs [48] where terms are represented as graphs and typing rules are represented by graph-matching. So far, this has been used to implement typing rules, but not formalize them. Furthermore, it is not clear how to extend a type-system implemented that way.

I aim to develop a description language to specify domain-specific type-systems through constraints. This descriptions would specify checking and inference rules that are both executables and amenable to mechanized formalisation in a theorem prover. One particular challenge is to make such descriptions both reusable, for dissemination and experimentation purposes, and extendable, following Section 9.1.

9.3. Typechecking OCaml by constraint: theory and practice

Medium term (**PLT**)

OCaml's type system is very rich, it features parametric polymorphism, row polymorphism, objects, GADTs, modules, ... Writing an efficient type-checker for such a language is complex. Unfortunately, the current implementation reflects that complexity: it is difficult to understand, to maintain and to modify. An idea, supported by several core OCaml developers, is to renovate the implementation of the typechecker using typechecking by constraints. This would also be the occasion to integrate extensions which were too difficult to implement on the current typechecker. For instance modular implicits [50] integrates ad-hoc polymorphism in OCaml (to support multipurpose print or "+" functions, for instance) and would be significantly easier to implement based on constraints. Combined with Section 9.1, this would also provide an even better integration for Eliom than my previous prototype implementation.

Such a redesign of the typechecker would involve several distinct tasks: first, expand the set of features that can be typechecked using constraints to the various elements found in OCaml. Second, design a constraint language suitable for OCaml and an associated efficient constraint solving algorithm. Finally, implement it all in OCaml. Naturally, I do not aim to complete this task alone and would seek collaboration with OCaml core maintainers such as Jacques Garrigue and the OCaml foundation headed by Michel Mauny. Personally, I want to contribute both to its theoretical and practical aspects. In particular, I'm interested by the interaction between constraints and modules in the context of "first class modules", and by the notion of extendable type system, as described in Section 9.1.

10. Modularity in the modern programming landscape

Modern programming ecosystems can be characterized by one common factor: their ridiculously large scale. NPM (the Javascript package manager) is estimated to contain 800000 distinct packages[6]. Even more modest ecosystems, such as Haskell, OCaml or Rust, still count over 10000 packages, each with many modules. Large-scale software projects have similarly grown to scales that are difficult to grasp[7]. This has two consequences: on one hand, new languages must be able to leverage some existing ecosystems. To most working programming, the elegance and quality of a language only come second to library availability. DSLs will, at some point, need to call or be called from other contexts, and thus need to integrate into existing ecosystems. On another hand, languages must provide tools to tame this scale and complexity.

10.1. Domain-specific modules Medium term (DSL,**PLT**)

Modularity (the ability to split large programs in smaller pieces) and encapsulation (which ensures that internals of these smaller pieces are not exposed to the outside word) make software behave like Lego: once molded, Lego pieces are undivisable, and many pieces can be combined together in many different ways, as long as the pegs fit. In programming languages, these properties have many benefits such as separate compilation (for fast project recompilation), link safety guarantees (if my library is compilable by itself, linking with it will not fail) and the ability to abstract over internal implementation details.

There are nearly as many module systems as there are programming languages, from plain library-level hierarchy in Java to the very expressive ML modules. The design of the module system strongly influences the usability of the language. For instance, Rust is currently struggling to add incremental compilation, which is particular hard to marry with a module system that supports arbitrary dependency cycles.

Following my work on Eliom's module system and MirageOS, I'm interested in ensuring that my DSL designs and language features interact well with modularity. In general, this concern all areas of language design. In particular, module systems should take into account multi-facet ecosystems where several languages interact in the same module system. Notable examples include the JVM ecosystem or the C ABI. In such an environment, various languages including domain-specific ones can integrate easily. These examples however have simple module and type systems and only limited abstraction capabilities. Drawing support from extensible type systems (Section 9.1), I'm interested in designing module systems which can integrate DSLs while supporting module-level abstraction, similarly to ML or Scala.

10.2. Scaling up the ML module system Short-medium term (**PLT**)

The module system in languages from the ML family provides very strong forms of modularity and encapsulation by adding a second language separate from the expression language. This language extends usual module languages by providing module types (called signatures) and functions from modules to modules (called functors). In some languages of the ML family, like OCaml, modules are even first-class entities that can be manipulated as part of the program.

Unfortunately, such rich capabilities are only available inside files and do not extend naturally to libraries. For instance, making a whole library parameterized by a given module is very cumbersome and would imply turning each individual module in the library into a functor.

[6] http://www.modulecounts.com/
[7] https://informationisbeautiful.net/visualizations/million-lines-of-code/

Module systems in many languages experience the opposite issue: they can be used to organize libraries, but not to manipulate modules directly in the language. A similar problem is often found in object-oriented languages such as Scala where the class language is much richer than the package language.

Fortunately, recent years have seen interesting advances such as mixins [42] and Haskell's linking calculus [21] that propose alternative module systems with different composition properties. My project is to investigate a notion of "open modules" which can be extended by any library. These open modules would allow programmers to easily functorize over complete libraries and to create namespaces, as found in many other languages, that can still be manipulated as modules. This work can be seen as allowing ML modules to have similar power as orchestration DSLs, such as the one I created for MirageOS.

10.3. Search modulo type isomorphism Medium term (PLT)

One recurring question of the working programmer is "Which function do I need to use ?". Often, the programmer has a fairly good idea of the signature of the function: "It should be a function that takes at least a car, a date, and returns a boolean". The programmer will then look at specific places likely to contain the right function (the `Car` module, for instance). This task grows harder with the size of the ecosystem, and one might like to directly search for such function using the programmer's intuited signature. Of course, such a search would have to ignore pesky details such as the order and number of arguments and focus on the shape of the types instead. The results would need to be sound (the function does match the signature), complete (we found all such functions) and fast to compute. This idea was initiated almost 25 years ago as "search modulo type isomorphisms" [41, 12]. Unfortunately, the algorithms developed for that purpose where appropriate to the ecosystems of statically typed functional languages of that time, which were very small. These algorithms do not scale at all to current ecosystems. Recent attempts at developing such systems, for instance Hoogle, are neither sound nor complete.

My project is to revisit the question of search modulo type isomorphism through a modern point-of-view. Using recent advances from proof-search, unification and databases we can develop efficient indexing and approximation techniques for search modulo type isomorphisms. Such techniques could be used not only for searching inside libraries, but also for type-directed code generation, as found in many dependently-typed languages. An application domain would be to search over whole ecosystems of statically typed languages, from small ones (Haskell, OCaml, Rust) to very large ones (Typescript).

11. Conclusion

As I like to design good tools for programming, I also enjoy programming, notably with statically typed functional languages. I have been a prolific contributor to both industrial-grade general purpose languages, like OCaml, and high-profile domain-specific ecosystems such as Ocsigen (web programming) or Mirage (operating systems). Most of my work has been implemented, either as prototypes to promote, experiment and distribute new ideas, or as production-ready software that can already be used to ease the work of software developers. These contributions have given me a practical experience with building both general-purpose and domain-specific languages. I believe that this practical experience, combined with my formal knowledge of programming languages and type systems, gives me a unique perspective to carry out this project to investigate domains in need of language support and develop new theoretical and practical tools regarding type systems and modularity to support writing such DSLs.

References

[1] Mitre: Common vulnerabilities and exposures for asn.1. URL https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=ASN.1.

[2] Flow typechecker. URL https://flow.org/.

[3] Hack programming language. URL https://hacklang.org/.

[4] Ange Albertini. Funky file formats. *Chaos Communication Congress*, 2014. URL https://media.ccc.de/v/31c3_-_5930_-_en_-_saal_6_-_201412291400_-_funky_file_formats_-_ange_albertini.

[5] Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 666–679. ACM, 2017. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837. URL http://dl.acm.org/citation.cfm?id=3009866.

[6] Alessandro Barenghi, Nicholas Mainardi, and Gerardo Pelosi. Systematic parsing of X.509: eradicating security issues with a parse tree. *Journal of Computer Security*, 26(6):817–849, 2018. doi: 10.3233/JCS-171110. URL https://doi.org/10.3233/JCS-171110.

[7] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. Cduce: an xml-centric general-purpose language. In Colin Runciman and Olin Shivers, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, pages 51–63. ACM, 2003. doi: 10.1145/944705.944711. URL https://doi.org/10.1145/944705.944711.

[8] bytepdf. *bytepdf*, 2018. URL https://github.com/Drup/bytepdf.

[9] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, volume 9058 of *Lecture Notes in Computer Science*, pages 3–11. Springer, 2015. doi: 10.1007/978-3-319-17524-9_1. URL https://doi.org/10.1007/978-3-319-17524-9_1.

[10] James Cheney, Sam Lindley, Gabriel Radanne, and Philip Wadler. Effective quotation: relating approaches to language-integrated query. In Wei-Ngan Chin and Jurriaan Hage, editors, *Proceedings of the ACM SIGPLAN 2014 workshop on Partial evaluation and program manipulation, PEPM 2014, January 20-21, 2014, San Diego, California, USA*, pages 15–26. ACM, 2014. doi: 10.1145/2543728.2543738. URL https://doi.org/10.1145/2543728.2543738.

[11] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *FMCO*, pages 266–296, 2006.

[12] Roberto Di Cosmo. A short survey of isomorphisms of types. *Mathematical Structures in Computer Science*, 15(5):825–838, 2005. doi: 10.1017/S0960129505004871. URL https://doi.org/10.1017/S0960129505004871.

[13] Jeff Dean. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*, 2015. URL http://download.tensorflow.org/paper/whitepaper2015.pdf.

[14] EliomLang. *EliomLang Sources*, 2018. URL https://github.com/ocsigen/eliomlang.

[15] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009. ISBN 978-0-262-06275-6. URL http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11885.

[16] Robert W. Floyd. The paradigms of programming. *Commun. ACM*, 22(8):455–460, 1979. doi: 10.1145/359138.359140. URL https://doi.org/10.1145/359138.359140.

[17] Functoria. *Functoria sources*. https://github.com/mirage/functoria/, 2017.

[18] Functoria Lua. *Functoria Lua*, 2018. URL https://github.com/Drup/functoria-lua.

[19] Laure Gonnord, David Monniaux, and Gabriel Radanne. Synthesis of ranking functions using extremal counterexamples. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 608–618. ACM, 2015. doi: 10.1145/2737924.2737976. URL https://doi.org/10.1145/2737924.2737976.

[20] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: a scalable quantum programming language. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 333–342. ACM, 2013. doi: 10.1145/2491956.2462177. URL https://doi.org/10.1145/2491956.2462177.

[21] Scott Kilpatrick, Derek Dreyer, Simon L. Peyton Jones, and Simon Marlow. Backpack: retrofitting haskell with interfaces. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 19–32. ACM, 2014. doi: 10.1145/2535838.2535884. URL https://doi.org/10.1145/2535838.2535884.

[22] Olivier Levillain. Parsifal: A pragmatic solution to the binary parsing problems. In *35. IEEE Security and Privacy Workshops, SPW 2014, San Jose, CA, USA, May 17-18, 2014*, pages 191–197. IEEE Computer Society, 2014. doi: 10.1109/SPW.2014.35. URL https://doi.org/10.1109/SPW.2014.35.

[23] Stefan Lucks, Norina Marie Grosch, and Joshua Konig. Taming the length field in binary data: Calc-regular languages. In *2017 IEEE Security and Privacy Workshops, SP Workshops 2017, San Jose, CA, USA, May 25, 2017*, pages 66–79. IEEE Computer Society, 2017. doi: 10.1109/SPW.2017.33. URL https://doi.org/10.1109/SPW.2017.33.

[24] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 254–269. ACM, 2016. doi: 10.1145/2976749.2978309. URL https://doi.org/10.1145/2976749.2978309.

[25] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David J. Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: library operating systems for the cloud. In Vivek Sarkar and Rastislav Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 461–472. ACM, 2013. doi: 10.1145/2451116.2451167. URL https://doi.org/10.1145/2451116.2451167.

[26] Mirage tool. *Mirage tool*, 2018. URL https://github.com/mirage/mirage.

[27] MirageOS. *MirageOS*, 2018. URL https://mirage.io/.

[28] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 175–188. ACM, 2014. doi: 10.1145/2628136.2628143. URL https://doi.org/10.1145/2628136.2628143.

[29] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional big-step semantics. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pages 589–615, 2016. doi: 10.1007/978-3-662-49498-1_23. URL https://doi.org/10.1007/978-3-662-49498-1_23.

[30] Erik Poll. Langsec revisited: Input security flaws of the second kind. In *2018 IEEE Security and Privacy Workshops, SP Workshops 2018, San Francisco, CA, USA, May 24, 2018*, pages 329–334. IEEE, 2018. doi: 10.1109/SPW.2018.00051. URL https://doi.org/10.1109/SPW.2018.00051.

[31] François Pottier and Didier Rémy. *The Essence of ML Type Inference*. URL http://gallium.inria.fr/~fpottier/publis/emlti-final.pdf.

[32] Gabriel Radanne. *Tierless Web Programming in ML*. PhD thesis, Paris Diderot, November 2017. URL https://www.irif.fr/~gradanne/papers/phdthesis.pdf.

[33] Gabriel Radanne. Typed parsing and unparsing for untyped regular expression engines. In *Proceedings of the ACM SIGPLAN 2014 workshop on Partial evaluation and program manipulation, PEPM 2014, January 20-21, 2014, San Diego, California, USA*. ACM, 2019.

[34] Gabriel Radanne and Peter Thiemann. Regenerate: a language generator for extended regular expressions. In Eric Van Wyk and Tiark Rompf, editors, *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2018, Boston, MA, USA, November 5-6, 2018*, pages 202–214. ACM, 2018. doi: 10.1145/3278122.3278133. URL https://doi.org/10.1145/3278122.3278133.

[35] Gabriel Radanne and Jérôme Vouillon. Tierless Modules. Draft, March 2017. URL https://hal.archives-ouvertes.fr/hal-01485362.

[36] Gabriel Radanne and Jérôme Vouillon. Tierless web programming in the large. In Pierre-Antoine Champin, Fabien L. Gandon, Mounia Lalmas, and Panagiotis G. Ipeirotis, editors, *Companion of the The Web Conference 2018 on The Web Conference 2018, WWW 2018, Lyon , France, April 23-27, 2018*, pages 681–689. ACM, 2018. doi: 10.1145/3184558.3185953. URL https://doi.org/10.1145/3184558.3185953.

[37] Gabriel Radanne, Vasilis Papavasileiou, Jérôme Vouillon, and Vincent Balat. Eliom: tierless web programming from the ground up. In Tom Schrijvers, editor, *IFL 2016, Leuven, Belgium, August 31 - September 2, 2016*, pages 8:1–8:12. ACM, 2016. ISBN 978-1-4503-4767-9. doi: 10.1145/3064899.3064901. URL http://doi.acm.org/10.1145/3064899.3064901.

[38] Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. Eliom: A core ML language for tierless web programming. In Atsushi Igarashi, editor, *APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, volume 10017 of *Lecture Notes in Computer Science*, pages 377–397, 2016. ISBN 978-3-319-47957-6. doi: 10.1007/978-3-319-47958-3_20. URL https://doi.org/10.1007/978-3-319-47958-3_20.

[39] Norman Ramsey. ML module mania: A type-safe, separately compiled, extensible interpreter. *Electr. Notes Theor. Comput. Sci.*, 148(2):181–209, 2006. doi: 10.1016/j.entcs.2005.11.045. URL https://doi.org/10.1016/j.entcs.2005.11.045.

[40] Regenerate. *Regenerate*, 2018. URL https://github.com/regex-generate/Regenerate.

[41] Mikael Rittri. Using types as search keys in function libraries. *J. Funct. Program.*, 1(1):71–89, 1991. doi: 10.1017/S095679680000006X. URL https://doi.org/10.1017/S095679680000006X.

[42] Andreas Rossberg and Derek Dreyer. Mixin' up the ML module system. *ACM Trans. Program. Lang. Syst.*, 35(1):2:1–2:84, 2013. doi: 10.1145/2450136.2450137. URL https://doi.org/10.1145/2450136.2450137.

[43] Traian-Florin Serbanuta, Andrei Arusoaie, David Lazar, Chucky Ellison, Dorel Lucanu, and Grigore Rosu. The K primer (version 3.3). *Electr. Notes Theor. Comput. Sci.*, 304:57–80, 2014. doi: 10.1016/j.entcs.2014.05.003. URL `https://doi.org/10.1016/j.entcs.2014.05.003`.

[44] Alejandro Serrano and Jurriaan Hage. Type error diagnosis for embedded dsls by two-stage specialized type rules. In Peter Thiemann, editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 672–698. Springer, 2016. doi: 10.1007/978-3-662-49498-1_26. URL `https://doi.org/10.1007/978-3-662-49498-1_26`.

[45] T-rex. *T-rex*, 2018. URL `https://github.com/gdeest/t-rex`.

[46] Tyre. *Tyre*, 2018. URL `https://github.com/Drup/tyre`.

[47] TyXML. *TyXML*. `http://ocsigen.org/tyxml/`, 2017.

[48] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *PACMPL*, 2(OOPSLA):114:1–114:30, 2018. doi: 10.1145/3276484. URL `https://doi.org/10.1145/3276484`.

[49] Liang Wang, 2018. URL `http://ocaml.xyz/chapter/cgraph_intro.html`.

[50] Leo White, Frédéric Bour, and Jeremy Yallop. Modular implicits. In Oleg Kiselyov and Jacques Garrigue, editors, *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014.*, volume 198 of *EPTCS*, pages 22–63, 2014. doi: 10.4204/EPTCS.198.2. URL `http://dx.doi.org/10.4204/EPTCS.198.2`.