European Research Council

ERC Starting Grant 2021
Research proposal [Part B1]

Certified Refinement Types

CRETE

- Principal investigator (PI): Niki Vazou
- Host institution: IMDEA Software Institute
- Full title: Certified Refinement Types
- Proposal short name: CRETE
- Proposal duration: 60 months

Type-theory formal verifiers (e.g., Coq, Agda) have been extensively used to prove program properties (e.g., compiler correctness). Such verifiers are carefully designed to implement core systems (variants of Martin-Löf's type theory) that are known to be sound, i.e., if verifier accepts the program, then it generates a mathematical proof. But, type-theory verifiers are not typically used to develop general purspose programs. They require explicit proof terms and provide limited support for real program features (like non-termination) rendering real world program development impractical.

Refinement types (e.g., Liquid Haskell and F$\star$) were designed as a practical alternative to type-based verification and have been successfully used to verify real world applications (e.g., memory safety and security policy enforcement). Such systems start from an actual programming language, extend it with logical predicates to specify properties of the language, and automatically validate these specifications using SMT solvers. Yet, refinement types suffer from the disadvantage that they do not generate proofs machine checked by a core calculus. In practise, both Liquid Haskell and F$\star$ have many times been found unsound.

The goal of this proposal is the design of a sound and practical refinement type system. This requires building a verification system that is as practical as refinement types and constructs machine-checked mathematical proofs. The system will be implemented on refinemement types systems of mainstream programming languages (i.e., Haskell and Rust) and will be evaluated on verification of real applications, such as web security and cryptographic protocols.

This proposal is of high risk since it aims to develop a novel program logic in which SMT automation co-exists with real world programming features (e.g., diverging and effectful programs). Yet, it is of high gain since it will render sound, type-based verification accessible to mainstream programmers.

Section a: Extended Synopsis of the scientific proposal

a.1 Vision, State of the Art, and Motivation

Refinement types [14] are a modern software verification technique that extends types of an existing programming language with logical predicates, to verify critical program properties not expressible by the existing type system. For example, consider the function `get xs i` that returns the `i`th element of the list `xs`. The existing type below states that `get` takes a list of `a`s, an integer and returns an `a`[1].

Existing Type: `get :: [a] → Int → a`
Refinement Type: `get :: xs:[a] → i:{Int | 0 ≤ i < len xs} → a`

The type of `get` gets *refined* to enforce in-bound indexing, a property that the existing type system cannot encode. Concretely, the refinement `0 ≤ i < len xs` on the index `i` ensures that `get` will only be called with indices in the bounds of the input list, preventing memory violation bugs like Heartbleed.

Refinement types are designed to be practical. The specifications are naturally integrated within the existing language and the verification happens automatically by an SMT solver [2]. For example, it is trivial to verify that any natural number `i` is a good index for the list that contains `i+1` zeros.

```
type ℕ = {i:Int | 0 ≤ i}
example :: i:ℕ → Int
example i = get (replicate (i+1) 0) i -- replicate :: i:ℕ → a → {xs:[a] | len xs == i}
```

Checking `example` uses the decidable theories of equality, uninterpreted functions and linear arithmetic to essentially confirm that $i < i + 1$, which is trivial for SMT solvers. This SMT automation on top of an existing language, that comes with efficient runtimes, optimized libraries, and development tools, render the refinement type based verification practical and accessible to mainstream programmers. Refinement type systems have been developed for various programming languages (Ocaml [9], Haskell [21], Ruby [15], Scala [11]) and have been used to enforce sophisticated properties (for example about cryptographic protocols [3], reference aliasing [10], resource usage [12], and web security [17]).

But, refinement types are not sound. For example, calling `example` with the maximum (fixed precision) integer will quickly break the `example`'s (verified) specification because of overflows, leading to memory access violations (Heartbleed Bug) or runtime exceptions (if `get` is partially defined).

```
> example maxBound
*** Exception: Non-exhaustive patterns in function get
```

For this concrete problem there is an easy solution, F⋆ [19] and `Stainless` [11] both encode fixed-precision integers as bit-vectors to reason about overflows. But, the unsoundness problems are deeper. First, the correspondence between program and SMT expressions is difficult in general and currently there are no guarantees that the refinement type checker developers implemented it correctly. For example, `Stainless` documentation[2] explicitly mentions errors in program and SMT correspondence as potential sources of unsoundnes and documents the encodings of unbounded data types as a known error. Second, the logic of refinement types is not well understood, leaving it unclear for the users what assumptions are safe to be made and which lead to inconsistencies, and thus unsound verification. For example, the PI recently discovered [20] that function extensionality had been encoded inconsistently in Liquid Haskell for many years. The inconsistent encoding seemed natural and was assumed by both the devopers and users of Liquid Haskell, but under its assumption Liquid Haskell could prove `false`. Finally, unlike type-theory based verifiers, refinement type checkers do not rely on a core kernel. Usually, the implementation of refinement type checkers trusts compiler of the underlying language to generate an intermediate program representation (IPR), the type checking rules (adapted to accommodate the IPR) to generate logical verification conditions (VC) and the SMT to validate the VCs. All these three trusted components are prone to implementation bugs that can lead to unsound verification. The implementations of F⋆, `Stainless`, and Liquid Haskell respectively consist of 1.3m, 185.3K, and 423K lines of code, without the existence of a concretely specified trusted core kernel bugs at such big code bases definitely exist and can potentially lead to unsoundness. For example, in Liquid Haskell there are found approximately 5 unsoundess errors per year due to implementation bugs.

[1] We use the syntax of Haskell and Liquid Haskell [21].
[2] Stainless documentions on "Limitations of Verification":https://epfl-lara.github.io/stainless/limitations.html

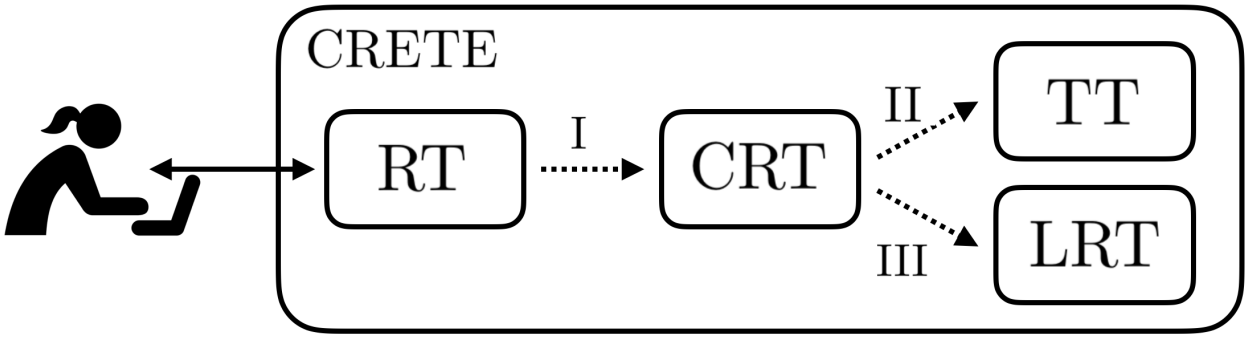

Figure 1: Objectives of CRETE. Starting from practical refinement types (RT, e.g., Liquid Haskell or Liquid Rust) it generates certified refinements (CRT; Objective I). CRT translates to the logic of refinement types (LRT, variant of HOL; Objective II) and type theory (TT, e.g., Coq; Objective III).

Type-theory based verifiers are designed to be sound. As an outstanding example, Coq [7] has a core small kernel that implements the well studied logic of Calculus of Inductive Constructions (CIC [6]). All programs verified correct using Coq are checked by the small kernel, which is the only TCB of Coq. But soundness comes at the cost of practicality, since the user needs to explicitly provide proof terms. For instance, to call the function `get [1,2,3] 2` the user needs to additionally provide a proof term `proof2≤3 :: 2<3` that can be inductively defined to prove that `2<3`. Such a proof is easy, but explicit proof construction becomes strenuous as the complexity of verification increases. As a characteristic example, Compcert [18] is a C compiler formally verified in Coq whose development is 36,5K lines of code, 12.5% of which is code. In other words, the Coq verification is 8 times larger than the program verified and as the author notes, "this percentage would be highly dropped if Coq supported SMT automation".

The ideal verification type system will combine the practicality of refinement types with the soundness of type-theory based verifiers.

a.2 The Grand Challenge and Objectives

The grand challenge of CRETE is to construct sound proofs of correctness for software verified using SMT-automated, *practical* refinement types. We will define certified refinement types that capture SMT proofs as explicit certificates and use them to derive HOL-style and Coq proofs of the original software. We will implement CRETE and use it to *soundly* verify real world code, e.g., cryptographic protocols in Rust and web security applications in Haskell.

To address this challenge certified refinement types (CRETE), is divided in four scientific objectives summarized in Figure 1 and explained below.

I: Certified Refinement Types The first objective is to define CRT, a certified refinement type system that isolates all automation of programs that refinement type check into explicitly certified terms. This isolation gives two advantages. First, the explicit certificates can be independently validated or tested using runtime execution of the underlying language. Second, decoupled from external automation, CRT will serve as the core calculus of refinement typing whose metatheory can be formally developed and proved sound.

The Approach: The first step to derive certified terms is to encode the SMT generated proofs in the core system. For instance, when the term `get [1,2,3] 2` refinement type checks, the SMT generated proof of `0 ≤ 2 < len [1,2,3]` will be encoded in the explicit term `proof-of(0 ≤ 2 < len [1,2,3])`. Such terms will be encoded using ideas from proof reconstruction [4, 1] to extract proof terms from the SMT implicit proofs. Next, we will translate the original program into a certified one, i.e., a program

with explicit proof terms. For instance, the call `get [1,2,3] 2` will now be encoded as `get [1,2,3] (2,proof-of(0 ≤ 2 < len [1,2,3]))`. This translation is not trivial, since the exact program points where SMT is performing proofs are not evident to the user. But, it can be directed by the refinement type derivation tree of the original program, i.e., explicit proof terms will be inserted at all the leaves of the derivation tree when SMT is invoked. Finally, the last step is the design of a type systems that checks the certified terms, so that the certified programs type check if and only if the original program type checks. Such a system will not perform type inference and will not depend on an external SMT, so the TCB will be minimal compared to the original automated refinement type system.

II: Logic of Refinement Types The second objective is to establish soundness of CRT by employing the classical methodology of program semantics. We will define LRT, the logic of refinement types that approximates CRT, intuitively, if $\Gamma \vdash e : \{v\!:\!a \mid p\}$, then $\Gamma \vdash p\ e$ is provable in LRT. We will show consistency of LRT, thus getting consistency of CRT. Importantly, LRT will be a variant of higher order logic (HOL) which is well studied and understood. Thus, this objective will give us a deeper understanding of refinement types and clarify the expressive power and the consistent assumptions of our system, in practise, preventing future inconsistencies that lead to unsoundness.

The Approach: The first step for this objective is to encode and prove soundness of the minimal, toy calculus of a certified refinement type system using a mechanized theorem prover. The soundness theorem will be expressed using denotational semantics (i.e., that if the program e has type t, then e belongs in the denotations of the type t) since this approach is common in the refinement types [22, 16]. The next step is to gradually extend the toy calculus with language constructs (e.g., recursive definitions, user-defined data types, polymorphism) required to verify realistic programs. A final and most challenging step, would be to use soundness of certified refinement types and encode soundness of refinement types, i.e., that if an expression e has some type t and the elaboration of e into a certified expression type checks, then e indeed belongs to the interpretation of the refined type t.

III: Translation of CRT to Type Theory The third objective is to translate CRT programs into Coq. Currently, this translation is not feasible because standard refinement types do not support explicit proof terms required by Coq or constructive type theory, in general. But, the combination of the explicit term certificates of CRT with the development of SMT tactic in Coq (introduced by SMTCoq) makes the derivation of Coq proofs from CRT possible. In theory, this objective will formalize the comparison between refinement types and constructive type theory as verification techniques. In practise, the impact is greater. Coq is the most reliable software to mechanically check proofs. Connecting all the pieces, CRETE will translate programs – developed in a mainstream programming languages and verified using *practical*, SMT-automated refinement types – into Coq *sound* proofs thus, delivering the holy grail of practical and sound software verification.

The Approach: We plan to investigate two approaches on translating certified refinement types to Coq's dependent types, by using either subset or indexed types. Subset types [5] encode refinement types as dependent pairs, thus the call to `get` remains the same `get [1, 2, 3] (2, proof-2≤3)`, where `proof-2≤3 :: 2<3` is an inductive Coq proof term. The disadvantage of this approach is that the definition of `get` is not inductive, thus is difficult to define in Coq. Using indexed types we encoded lists of length `n` as `vec n a`, i.e., the length of the list is used to explicitly index the type. Then, the type and client of `get` becomes as follows:

Type of `get`: `get :: vec n a → i:ℕ → (i ≤ n) → a`
Client of `get`: `get [1, 2, 3] 2 (proof-2≤3)`

The challenge of this approach is that each refined list needs to explicitly carry information about its length, but we plan to address it, using proof information carried by the intermediate representation of the certified refinement types. We will generalize this example, to define a syntax directed translation from certified refinement to dependent types.

IV: Implementation and Evaluation on Applications The final objective is to implement CRT as a back-end to practical refinement type systems and evaluate the feasibility and impact of our

methodology. We will implement CRT as a back-end to Liquid Haskell to examine the soundness of verified, real-world, web security applications: was verification making inconsistent assumptions? To ensure that our methodology is general and can be used to develop sound and practical refinement type checkers, we will also incorporate CRT to Liquid Rust, a novel refinement type checker for Rust that we will use to verify cryptographic protocols.

The Approach: Our first step is to use certified refinement types as a back-end to existing refinement type systems, like Liquid Haskell, that is a mature refinement type checker of Haskell programs and Liquid Rust that is currently under construction. Next, we will use the implementation on real world code bases that are verified by the existing systems, including the approximate 2K unit tests of Liquid Haskell and commonly used Haskell libraries that enforce functional and safety properties (e.g., in-bound indexing and sortedness) and compare the verification time and user experience of verification with the two alternative systems. Finally, we will use the implementation to verify novel applications, for example, safety properties of smart contracts (e.g., in the implementation of Cardano [13]) or cryptographic protocols (e.g., the eccrypto [8] library), in which the soundness of the verifier is critical.

a.3 The risks and difficulties

This line of work is a promising, novel research area that as such, has many risks.

The first major risk is that the automatic derivation of a certified expressions is not possible for each program that has a refinement type derivation. This impossibility would be critical, as the majority of the proposal depends on it. Even though the risk is high, relevant work on proof reconstruction [4, 1] implies that certificate construction is feasible. So, even though I am not an expert in the area of proof construction, I will collaborate with experts in the field. Further IMDEA Software institute provides the right environment (i.e., no teaching obligations) and expert colleagues (e.g., Alexandar Nanevski who is a type-theory expert) to help me carry out such a risky experiment.

The second major risk is that the certified refinement type system could not be used as a back-end at refinement type systems of real programming languages, thus the derived system is actually sound, but not practical. Since I have been an active developer of Liquid Haskell for 8 years and have great experience with development and verification of real world software, I have the appropriate experience on real verification systems and knowledge to circumvent the problems that will appear during this process.

a.4 Potential Impact

Scientific Impact: The project aims to develop a novel type system that combines the advantages of both type-theory and refinement types and the objective is that this type system will be adopted by the verification and programming languages communities, both as a verification mechanism and as a novel research area.

Socioeconomic Impact: The project aims to bring formal verification to mainstream programming, leading to higher quality software that is developed faster and is correct. An evidence of that impact is that Liquid Haskell, a refinement type checker for Haskell programs has already been used in industrial software by Well-Typed and Tweag-IO. Yet, the soundness problems of Liquid Haskell, and refinement types in general, are known, and discourage developers to use it. With CRETE, these problems will be addressed leading to further adoption of refinement type checking in mainstream development.

Educational Impact: The results of this project can be used for educational purposes. Refinement types have been taught in advanced undergraduate and graduate classrooms. This project aims to make software development supported by formal verification so attractable, that can be used as an aid on programming courses and promote verified programming as the de facto way to teach programming.

a.5 Commitment of the PI and Research Group

The PI will spend 75% of her time on this project. She will use the funding of this project to recruit 3 PhD, 2 Postdoctoral student, and 1 software engineer. Her goal is to build a research group at IMDEA that will be the leading group in the area of refinement type systems and collaborate with her strong international connections from UCSD and UMD.

References

[1] C. Barrett, L. De Moura, and P. Fontaine. Proofs in satisfiability modulo theories. In D. Delahaye and B. Woltzenlogel Paleo, editors, *All about proofs, Proofs for all*, Mathematical Logic and Foundations, pages 23–44. College Publications, jan 2015.

[2] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[3] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM TOPLAS*, 2011.

[4] S. Böhme, A. C. J. Fox, T. Sewell, and T. Weber. Reconstruction of z3's bit-vector proofs in hol4 and isabelle/hol. In J.-P. Jouannaud and Z. Shao, editors, *Certified Programs and Proofs*, pages 183–198, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[5] A. Chlipala. *Certified Programming and Dependent Types*. MIT Press, 2013.

[6] T. Coquand. Metamathematical investigations of a calculus of constructions. Technical Report RR-1088, INRIA, Sept. 1989.

[7] T.-C. development team. *The Coq proof assistant reference manual*, 2004. Version 8.0.

[8] M. Fourne. *eccrypto: Elliptic Curve Cryptography for Haskell*, 2019.

[9] T. Freeman and F. Pfenning. Refinement types for ml. *SIGPLAN Not.*, 26(6):268–277, May 1991.

[10] C. S. Gordon, M. D. Ernst, and D. Grossman. Rely-guarantee references for refinement types over aliased mutable data. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 73–84, New York, NY, USA, 2013. Association for Computing Machinery.

[11] J. Hamza, N. Voirol, and V. Kunčak. System fr: Formalized foundations for the stainless verifier. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct. 2019.

[12] M. A. T. Handley, N. Vazou, and G. Hutton. Liquidate your assets: Reasoning about resource usage in liquid haskell. *Proc. ACM Program. Lang.*, 4(POPL), Dec. 2019.

[13] IOHK. *cardano wallet*, 2020.

[14] R. Jhala and N. Vazou. Refinement types: A tutorial, 2020.

[15] M. Kazerounian, N. Vazou, A. Bourgerie, J. S. Foster, and E. Torlak. Refinement types for ruby. In I. Dillig and J. Palsberg, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 269–290, Cham, 2018. Springer International Publishing.

[16] K. Knowles and C. Flanagan. Hybrid type checking. New York, NY, USA, feb 2010. Association for Computing Machinery.

[17] N. Lehmann, R. Kunkel, J. Brown, J. Yang, D. Stefan, N. Polikarpova, R. Jhala, and N. Vazou. Storm: Refinement types for secure web applications. 2021.

[18] X. Leroy. A formally verified compiler back-end. volume 43, page 363–446, Berlin, Heidelberg, Dec. 2009. Springer-Verlag.

[19] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in f*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 256–270, New York, NY, USA, 2016. Association for Computing Machinery.

[20] N. Vazou and M. Greenbert. Function extensionality for refinement types. *Under Review*, 2020.

[21] N. Vazou, E. L. Seidel, and R. Jhala. Liquidhaskell: Experience with refinement types in the real world. *SIGPLAN Not.*, 49(12):39–51, Sept. 2014.

[22] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, page 269–282, New York, NY, USA, 2014. Association for Computing Machinery.

Section b: Curriculum vitae

PERSONAL INFORMATION

Family name, First name: Vazou, Niki
Researcher identifiers: orcid: 0000-0003-0732-5476, publons: 3145359, google scholar: ARcLTokAAAAJ
Date of birth: 20 July, 1987
Nationality: Greek
URL for web site: `https://nikivazou.github.io/`

• EDUCATION

2011-2017	PhD Computer Science and Engineering, University of California, San Diego, USA. Supervisor: Ranjit Jhala; Thesis title: "Liquid Haskell: Haskell as a Theorem Prover".
2005-2010	Diploma National Technical University of Athens, Athens, Greece.

• CURRENT POSITION

2018-present	Research Assistant Professor IMDEA Software Institute, Madrid, Spain.

• PREVIOUS POSITIONS

2017-2018	Postdoctoral Fellow Computer Science Department, University of Maryland, College Park, USA.
Summer 2016	Internship with Jeff Polakow Awake Networks, Mountain View, USA.
Summer 2014	Internship with Daan Leijen Microsoft Research, Redmond, USA.
Fall 2013	Internship with Dimitrios Vytiniotis Microsoft Research, Cambridge, UK.

• SUPERVISION OF GRADUATE STUDENTS AND POSTDOCTORAL FELLOWS

2019-present	PhD candidate Christian Poveda on "refinement types for Rust", co-supervised with Gilles Barthe. Master student Mustafa Hafidi on "tactics for Liquid Haskell'. Undergrad student David Munuera on "Haskell to CIAO". Research intern Zack Grannan "rewritting for Liquid Haskell'. Research intern Lisa Vasilenko on "relational refinement types". Research intern Stefan Malewski on "gradual refinement types". IMDEA Software Institute, Madrid, Spain.
2017-2018	PhD candidate James Parker on "verification of web security applications", co-supervised with Michael Hicks and published on POPL'19 and OOPSLA'20. Milod Kazerounian on "refinement types for Ruby", co-superrvised with Jeff Foster and published on VMCAI'18 and PLDI'19. Computer Science Department, University of Maryland, College Park, USA.
2018	Diploma student George Petrou on "verification with Liquid Haskell", co-supervised with Nikolaos Papaspyrou. National Technical Univarsity of Athens, Athens, Greece.

• TEACHING ACTIVITIES

Fall 2019	2 month/20 hour seminar on Advanced Functional Programming Computer Science Department, Universidad Politécnica de Madrid, Madrid, Spain.
Winter 2018	Lecturer at Advanced Functional Programming (CMSC498V) Computer Science Department, University of Maryland, College Park, USA.
Fall 2017	Co-Lecturer at Introduction to programming (CMSC330) Computer Science Department, University of Maryland, College Park, USA.
Summer 2015	one week/40 hour course on functional programmings) Clubes De Ciencia, Guanajuato, Mexico.

• ORGANISATION OF SCIENTIFIC MEETINGS

2021	Co-Chair, Artifact Evaluation, PLDI
2020	Virtualization Committee, POPL
2020	Co-Chair, Student Research Competition, POPL
2019	Chair, Student Research Competition, POPL
2019	Chair, Haskell Implementors' Workshop
2019	Co-Chair, Programming Languages and Analysis for Security
2018	Co-Organizer, Programming Languages Mentoring Workshop, ICFP
2018	Co-Chair, Workshop on Type-Driven Development

• REVIEWING ACTIVITIES

POPL'21, VMCAI'21, CAV'20, FCS'20, POPL'19, ESOP'18, ICFP'18, PEPM'21, PLS'21, TFP'20, Haskell'18, ML-Family Workshop'17, HOPE'17, PADL'17, Scala'17, Haskell'16, HiW'16, TFP'16, PADL'16, Scala'16, AEC@POPL'16, AEC@PLDI'16.

• MEMBERSHIPS OF SCIENTIFIC SOCIETIES

2021 - present	Board Member of Haskell Foundation.
2019 - present	Visitor of IFIP Working Group 2.1 on Algorithmic Languages and Calculi.
2019 - present	Visitor of IFIP Working Group 2.8 on Functional Programming.
2017 - present	Member of The ACM SIGPLAN Haskell Symposium Steering Committee.
2016 - 2020	Member of Haskell.org Committee.
2015 - 2016	Event Coordinator at Graduate Women in Computing, UCSD.

• MAJOR COLLABORATIONS

Gilles Barthe on Refinement Types for Rust for Cryptographic Protocols
Max Planck Institute for Security and Privacy, Bochum, Germany.
Simon Peyton Jones on Verification of Haskell
Microsoft Research, Campridge, UK.
Michael Hicks, David Van Horn on Applications of Verification to Security
Computer Science Department, University of Maryland, College Park, USA.
Éric Tanter on Gradual Types
Computer Science Department, Uniiversity of Chile, Santigo, Chile.

• FELLOWSHIPS AND AWARDS

2018	Best paper award at ACM SIGPLAN Conference OOPSLA.
2017	Victor Balisi Postdoctoral Fellowship Computer Science Department, University of Maryland, College Park, USA.
2015	Graduate Award for Research Computer Science and Engineering, University of California, San Diego, USA.
2014	Microsoft Research Graduate Research Fellowship.

Section c: Early achievements track-record

• **RESEARCH ACTIVITIES** My research is on refinement type systems and concretely my goal is to make SMT-based, decidable, semi-automated verification an integral part of legacy programming languages and usable by mainstream programmers. I have developed Liquid Haskell, a refinement type checker for Haskell programs that is adopted by the industrial, educational, and research Haskell community (with 18K hackage downloads, many tutorials in industrial venues, projects and courseworks by students and various security applications). I have 7 papers on CORE A* (flagship) conferences (2 ICFP, 2 POPL, 1 PLDI and 2 OOSPLA). My h-index is 10 and I have 576 citations.

• **IMPORTANT PUBLICATIONS** Below I outline my 5 most important publications. As common in my field, the first author is the main contributor of the work presented in the paper.

- **N. Vazou**, E. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement Types for Haskell. ICFP, 2014. *190 citations*

  This paper introduces Liquid Haskell, a refinement type checker for Haskell programs and used it for verify 10K lines of real world Haskell code.

  I was the main contributor on this work, concretely, I developed all the metatheory section and conducted large portion of the implementation and the experiments.

- **N. Vazou**, A. Tondwalkar, V. Choudhury, R. Newton, P. Wadler, and R. Jhala. Refinement Reflection: Complete Verification with SMT. POPL, 2018. *34 citations*

  This paper presents how decidable SMT-based verification can be used to allow theorem proving of arbitrary (undecidable) properties via refinement types.

  I was the main contributor on this work, concretely, I came up with the novel idea presented in the paper and developed large portion of the metatheory and evaluation.

- Martin Handley, **N. Vazou**, and G. Hutton. Liquidate your assets: Reasoning about resource usage in Liquid Haskell. POPL, 2020.

  This paper shows how refinement types can reason automatically about resources (e.g., time complexity and memory allocation).

  This work is without co-authorship of my PhD supervisor. I closely supervised the first author (who was a PhD student during this work) and I conducted the metatheory of the system, the major portion of the experimental comparison with relevant systems, and most part of the paper writing.

- M. Kazerounian, S. N. Guria, **N. Vazou**, J. Foster, D. Van Horn. Type-Level Computations for Ruby Libraries. PLDI, 2019.

  This paper presents an expressive type system for Ruby with type-level computations used to type check database queries in commonly used Ruby libraries.

  This work is without co-authorship of my PhD supervisor. I closely supervised the first author (who was a PhD student during this work) for the development of the system presented in this paper and the metatheory.

- J. Parker, **N. Vazou**, and M. Hicks. Information Flow Security for Multi-Tier Web Applications. POPL, 2019.

  This paper presents a framework for enforcing label-based, information flow policies in database-using web applications with a mechanized proof of non interference.

  This work is without co-authorship of my PhD supervisor. I closely supervised the first author (who was a research programmer during this work) and did the metatheory of the system and most of the paper writing.

• **INVITED PRRESENTATIONS (A SELECTION OF)** As an active member of the functional programming research and industrial community and a leader in the active area of refinement types, I have been invited to participate in prestigious events (e.g., IFIP working groups which are recognized by United Nations with a goal to encourage collaborations between international scientists) and present by work on industrial conferences (e.g., Scala and Haskell eXchange, organized by Skills Matters one the worlds largest communities of software engineers).

- 2020: Invited talk at IFIP Working Group 2.1: Algorithmic Languages and Calculu, Otterlo, Netherlands.
- 2019: Invited talk at IFIP Working Group 2.8: Functional Programming, Bordeaux, France.
- 2019: Invited talk at 12th Panhellenic Logic Symposium, Anogeia, Greece.
- 2018: Keynote at Haskell eXchange, London, UK.
- 2018: Keynote at Zurihac, Zurich, Switzerland.
- 2018: Invited talk at Scala eXchange, London, UK.
- 2017 Invited talk, Semantics of Effects, Resources, and Applications, Shonan, Japan.
- 2017: Invited talk at Lambda Days, Poland.
- 2016: Invited talk at Compose Conference, NY, USA.
- 2016: Seminar 16112: From Theory to Practise of Algebraic Effect Handlers, Dagstuhl, Germany.
- 2016: Seminar 16131: Language Based Verification Tools for Functional Programs, Dagstuhl, Germany.

• **ORGANIZATION OF INTERNATIONAL EVENTS** The explicit list of the organization appears in the CV. Here I am emphasizing that I have been a member of the organization committee of A* programming languages conferences since 2018: as co-organizer of PLMW at ICFP'18, co-organizer or Student Research Competition at POPL'19 and '20, member of virtualization committee at POPL'20, and artifact evaluation co-chair at PLDI'21.

• **ABILITY TO INSPIRE YOUNG RESEARCHERS** Because of my strong presence in the conferences (via organization, presentations, and tutorials) I am able to inspire and attract your researchers. In 2019 and 2020, I co-organized student research competition at POPL. This competition is organized by ACM and aims to expose early work of undergraduate and graduate students to the broader scientific community by presenting posters as well as partially cover student's travel expenses to conferences. In 2018, I co-organized programming languages mentoring workshop (PLMW) at ICFP. This workshop takes place the day before the main ICFP conference and includes panels and mentoring or scientific presentations from recognized members of the programming language scientific community that are targeted to young researchers. Even though my current position does not involve teaching young people, in Fall 2019 I organized a seminar where I taught advanced functional programming to students of UPM and I have given 2-5 hour tutorials at three major programming languages conferernces (ICFP'16, PLDI'17, and POPL'21).

During my postdoc at University of Maryland, I strongly collaborated with two PhD students that now have have good positions (James Parker who is research engineer at Galios and Milod Kazerounian who is a lecturer at Tufts University). In my two first years as an assistant professor at IMDEA, I have already attracted 2 PhD students and 3 interns, while I still have active collaborations with students from both UCSD and Univeristy of Maryland.

• **PATENTS AND SOFTWARE** I am one of the main developers and maintainers of Liquid Haskell a refinement type checker for Haskell programs that has been used both for educational purposed in graduate and undergraduate classes (in UCSD, University of Maryland) and industrial companies (including Well-Typed, Tweag-IO, Aware Security).

• **AWARDS** The list of awards is mentioned in the CV, here I want to emphasise the most recent one, the Atracción de Talento Fellowship, provided by Comunidad de Madrid, Spain to your researchers.