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

Refinement types are a type-based, static verification technique designed to be practical. They enrich the types of an existing programming language with logical predicates to specify properties of the language and automatically validate these specifications using SMT solvers. Refinement types are a promising verification technology that in the last decade has spread to mainstream languages (including ML-family, Haskell, C, Ruby, Scala) to verify sophisticated properties (e.g., safety of cryptographic protocols, memory and resource usage, and web security) of real world applications.
The weakness of refinement types is that they do not meet the soundness standards set by theorem provers. A sound verification system only accepts as safe programs that never violate their specifications. Refinement type checkers (e.g., Liquid Haskell, F*, and Stainless) approximately report five unsoundness bugs per year, as opposed to only one reported by the Coq theorem prover. This rarity of unsoundness bugs in Coq is unsurprising since Coq is designed to soundly machine check mathematical proofs. Coq's soundness design recipe thought cannot be directly applied to refinement type checkers that aim to practically verify real world programs.
The goal of CRETE 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 refinement type systems of mainstream programming languages (i.e., Haskell and Rust) and will be evaluated on real world applications, such as web security and cryptographic protocols.
CRETE is high risk since it aims to develop a novel program logic in which SMT automation co-exists with real world programming. Yet, this proposal is high gain since it develops a low-cost, high-profit approach to formal verification that aims to be integrated in mainstream software development.

Section a: Extended Synopsis of the scientific proposal

a.1 Motivation and Goal

Refinement types [22] 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` requires that `get` can only be called with indices in the bounds of the input list. Such assertions are checked statically and can be used to prevent critical, real-world bugs, e.g., the memory violation of the infamous Heartbleed bug, without the need of runtime checks that increase program's execution time.

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 [4]. 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, renders refinement type based verification practical and accessible to mainstream programmers. Refinement types is a promising verification technology that in the last decade has spread to mainstream languages (Haskell [40], C [9], Ruby [23], Scala [19]) to verify sophisticated properties (e.g., about cryptographic protocols [5], reference aliasing [18], resource usage [20], and web security [25]).

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 -- maxBound = 2^63 − 1
*** Exception: Non-exhaustive patterns in function get
```

The refinement type checker accepts code that at runtime breaks its specifications, thus it is not sound. This concrete problem has an easy solution, F⋆ [37] and `Stainless` [19] both encode fixed-precision integers as bit-vectors to reason about overflows. But the *sources of unsoundness* (*SoU*) are deeper. *First (SoU1),* the correspondence between program and SMT expressions is difficult in general and currently there are no guarantees that the developers of refinement type checkers implement it correctly. For example, `Stainless` documentation[2] explicitly mentions errors in program and SMT correspondence as potential sources of unsoundness and documents the encodings of unbounded data types as another *known* error. *Second (SoU2),* 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 [38] that function extensionality had been encoded inconsistently in Liquid Haskell for many years. The inconsistent encoding seemed natural and was assumed by both the developers and users of Liquid Haskell, but under the assumption of functional extensionality Liquid Haskell could prove `false`, invalidating all the user's verification effort. *Finally (SoU3),* unlike traditional theorem provers (e.g., Coq and Agda), refinement type checkers do not rely on a small, trusted, 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 consist of

[1] We use the syntax of Haskell and Liquid Haskell [40].

[2] Stainless documentation on "Limitations of Verification":https://epfl-lara.github.io/stainless/limitations.html

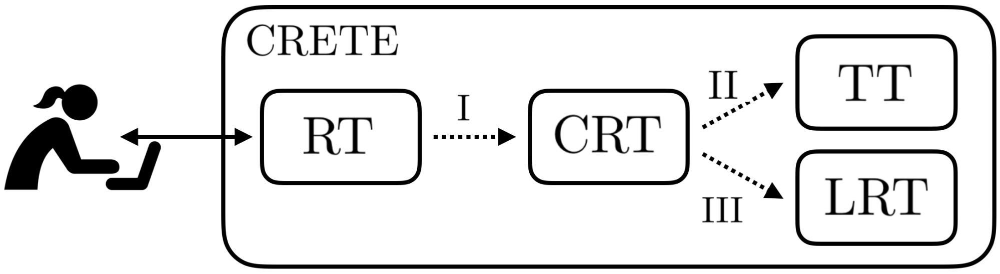

Figure 1: Objectives of CRETE. Starting from practical refinement types (RT, e.g., Liquid Haskell or Liquid Rust) we generate certified refinements (CRT; Objective I). CRT translate to type theory (TT, e.g., Coq; Objective II) and the logic of refinement types (LRT, variant of HOL; Objective III).

big code bases that inevidably contain bugs and can potentially lead to unsound verification. For example, the implementations of F⋆, `Stainless`, and Liquid Haskell respectively consist of 1.3M, 185.3K, and 423K lines of code, and none of these verifiers isolates a trusted kernel. Approximately five unsoundness bugs per year are reported in each system.

Theorem provers are designed to be sound. The recipe to design a sound verification system is known and followed by theorem provers, i.e., tools designed to machine check and automate proofs of mathematical theorems. These tools consist of a small trusted kernel that implements a deductive system that is proved consistent by reduction to a consistent mathematical theory. For example, Isabelle/HOL [31] has a core kernel of 5K lines of code that implements LCF [28, 33] whose consistency is known because of a sound translation to the domain theory and continuous functions [34]. Coq [15]'s kernel is 14K lines of code that implement Calculus of Inductive Constructions (CIC [11, 12]) a calculus shown consistent by reduction to the theory of sets and functions [21]. Still, "on average, one critical bug has been found every year in Coq" [36]. So, one can only imagine how many critical bugs exist in the implementation of practical refinement type checkers, that have far less users than Coq.

But, theorem provers are not practical for program verification. The first inconvenience comes because theorem provers require programs to be explicitly annotated with proof terms. For example, to `get` the `i`th element from a list of length `1+i`, a Coq definition would require an explicit proof of `i<1+i`, that in refinement types is implicitly performed by the external SMT solver, without littering the executable code with proof annotations. The major inconvenience comes because theorem provers, designed to implement a proof system, do not come with runtime execution semantics but rely on extraction mechanisms to generate executable code. For example, code verified by Coq can only get executed after extraction [26] to Ocaml or Haskell. This execution via extraction though, is not attractive to mainstream program developers that pay extra attention and use idiomatic code to optimize program's runtimes.

The grand challenge of this proposal is to develop a both practical and sound verification system.

a.2 Objectives and Methodology

Certified refinement types (CRETE) will construct *sound* proofs for software verified by SMT-automated, *practical* refinement types. We will define explicit certificates that capture the SMT proofs and use them to derive Coq and HOL-style proofs of the original software. CRETE will be used to verify real world code, such as cryptographic protocols and web security applications.

Figure 1 summarizes the workflow and scientific objectives of CRETE. The user will interact with SMT-automated refinement types to verify code developed in an existing programming language. The first objective of CRETE is to internally annotate the programs accepted by the refinement type checker with explicit certificates that will be independently validated or tested against the program's runtime semantics; addressing *SoU1*, i.e., the discrepancies between program and SMT semantics. The second objective is to translate the certified program into a sound theorem prover, e.g., Coq, in order to construct proofs that will ultimately be machine-checked by Coq's small core kernel; eliminating *SoU3*, i.e., potential unsoundness due to implementation bugs. The third objective is to develop the logic of refinement types a higher order logic that will formalize the inference rules of refinement types; preventing *SoU2*, i.e., the assumption of inconsistent axioms. The final objective is to implement CRETE as the the back-end of the Haskell and Rust refinement type checkers to obtain both practical and sound verification of real world software.

Next, we present the methodology we will use for each objective, their challenges and importance.

I: Certified Refinement Types The first objective is to define CRT, a certified refinement type system that uses term certificates to explicitly capture the implicit SMT proofs of programs that refinement type check into explicitly certified terms. The explicit certificates will be validated using SMT proof certificate generation and validation technologies [27, 13, 3, 6] and tested against language's runtime semantics by custom test generators [10, 24].

In the `example` of § a.1, the index `i` will be wrapped by a certificate that ensures that `i` has the type `{v:ℕ | v < i + 1}` in the typing environment `{i:ℕ}`, so `i` is a good index for `get`.

```
example :: i:ℕ → Int
example i = get (replicate (i+1) 0) (cert i {v:ℕ | v < i + 1} {i:ℕ})
```

This certificates will be validated by multiple SMT solvers (e.g., Z3 and CVC4) and under multiple encodings (e.g., mapping `Int` to both logical `Int` and bit-vectors) and tested against language runtime semantics to generate the following two errors:

-
  ```
  Error when Int is encoded as (_ BitVec 64) in CVC4 and Z3
  (Change Int to Integer or enable UnsafeNoOoverflows to suppress this error)
  ```

-
  ```
  Error counter-example found for i = 9223372036854775807

  example i = get (replicate (i+1) 0) i
                                      ^^^
  ```

The first error states that the certificate could not be verified when `Int` was encoded as bit-vector and suggests two alternative solutions (to change the language type or unsoundly suppress the error). The second error provides a counter-example (the `maxBound =` $2^{63} - 1$) that falsifies the certificate.

This objective has three main challenges. The first challenge is the definition and implementation of a type-preserving elaboration algorithm, especially in the presence of higher order functions and polymorphism. To address this challenge, the PI will use her expertise [41] and related work [16] on gradual typing that define similar elaborations. The second challenge is the development of custom test generators that cover the interesting (here, corner) cases of the tested certificates. The PI has early work on targeted testing [35] and will collaborate with testing experts on this task. The final challenge is the development of a GUI that presents the errors in a usable way. The PI has been involved in the past in the development of editor integrated error reporting and further plans to hire a research engineer to help develop a user friendly graphical interface.

This objective is important because it will externally validate the explicit certificates, thus reducing the trusted code base of the refinement type checker and it will test the certificates against the language semantics, thus eliminate the discrepancies between program and SMT semantics.

II: Translation of CRT to Type Theory The second objective is to translate CRT programs into type theory, and concretely, Coq. We will translate refined to inductive data types [2], refinement type specifications to subset types [8, 14], and the explicit certificates into Coq proof terms. For example, in the specification of `get` from § a.1 the list `xs:[a]` will be translated to the vector `vec A n`, where `n` is the natural number that captures the length of the vector and will replace `len xs` in the refinements.

```
get: ∀ A n, vec A n → {i : ℕ | i < n} → A
```

In Coq, the type `{i : ℕ | i < n}`, even though it shares the same syntax as refinement types, is a constructive subset type, i.e., it requires explicit proof terms. We will use the explicit certificates to direct the synthesis of such proof terms and the `smt` tactic, implemented by the SMTCoq [1] project, to conduct the proofs. For example, the `example` from § a.1, gets translated to the following.

```
Lemma lemma (i:ℕ): i < 1 + i. smt. Qed.

Definition example (i:ℕ) : ℕ :=
  get (replicate (1+i) 0) (exist (fun v:ℕ => v < 1+i) i (lemma i)).
```

The term `exist (fun v:ℕ => v < 1 + i) i (lemma i)` has the type `{i : ℕ | i < 1 + n}`, as required by `get` and is constructed following the structure of the explicit certificate `cert i {v:ℕ | v < i + 1} {i:ℕ}`. The only information not provided by the certificate is the proof of `lemma` that we plan to construct using the `smt` tactic that constructs SMT-style proofs validated by Coq.

The challenges on this translation depend on the complexity of programs we will target. We will gradually increase the complexity of the programs and ensure that the translation in case of failure provides error messages useful for both the developers and the users, so our development is usable from the early stages. Based on the `hs-to-coq` [7] project that translates significant portions of Haskell's real-world library into Coq, we expect that our translation will be applicable to real code.

This objective is important for both the theory and practise of refinement types. The proposed translation will formalize the relation between classical refinement types and type theory, that until now has been a frequent question answered only by case-study comparisons (from the PI [39] and various master thesis projects [32, 42]). In practise, we will translate programs verified using *practical*, SMT-automated refinement types into the *sound*, mechanically-checked Coq proofs.

III: Logic of Refinement Types The third objective is to establish soundness of CRT by partially employing the soundness recipe of theorem provers. We will define the logic of refinement types LRT, i.e., a deductive system of inference rules and axioms that similar to higher order logic (HOL) decide predicate validity $\Gamma \vdash p$. Next, we will prove that LRT approximates CRT, intuitively, if $\Gamma \vdash e : \{v{:}a \mid p\ v\}$, then $\Gamma \vdash p\ e$ is provable in LRT and that LRT is consistent, by reduction to a consistent mathematical model. From these, we can deduce consistency of CRT. Our language of expressions we will start from simply typed lambda calculus (STLC) that we aim to extend with polymorphism (System F) and type operations (System F_ω).

The challenges of this objective depend on the complexity of the underlying system. For STLC the scene is clear and it requires the combination of existing principal formalizations of refined systems (e.g., RCF [17]) with the categorical interpretations of STLC [34]. The application of our methodology to polymorphic systems is very challenging, since both refinement types and higher order logic for polymorphic systems are currently under active research. Yet, the PI has a long experience and a deep understanding of polymorphism under practical refinement type systems and will collaborate on this challenging task with experts in the theory and semantics of programming languages.

This objective is of high importance since it will provide a deeper understanding of refinement types. On the theoretical side, it will, for first time, define the principles of refinement types using higher order logics and classic semantic techniques that have been studied since the '70s and are well understood. It will further define a consistent, mathematical model for refinement types that would clarify which axioms the user can add in the logic while preserving consistency. So, on the practical side, it would prevent inconsistencies that currently exist on practical refinement type checkers and jeopardize all the verification effort of their users.

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. Concretely, all the stages of CRETE will be developed as the back-end of Liquid Haskell, the practical refinement type checker that the PI developed and actively maintains. We will use certified Liquid Haskell to verify real world, secure web applications, like `lweb` [30] and `STORM` [25] that the PI, in collaboration with the University of Maryland and UC San Diego security experts, has already verified with (uncertified) Liquid Haskell. Since the existing verifications rely on unformalized features of Liquid Haskell this task will provide strong soundness guarantees that are currently only aspired. Further, we will incorporate CRETE to Liquid Rust, a novel refinement type checker for Rust that is under active development by the PI and her current PhD student, to statically verify the invariants of cryptographic protocol implementations (e.g., `RustCrypto` [29]).

There are two major challenges in this objective. First, our choice to directly incorporate our novel theoretical developments in real programming languages, instead of building a prototype implementation, comes with high maintenance and entry cost for new PhD students. The PI has a long experience in working with big code bases and collaboration with new students thus has learned techniques that minimize the entry costs, e.g., the development of isolated interfaces. The second challenge is that our certificate generation for the complete calculus of the target real languages might not be possible, especially since the two target languages support distinguished idiomatic features (memory safety in Rust and type operations in Haskell). To ameliorate this challenge, from the early stages, our implementation will emit descriptive feedback in case of failure helping both the developers and the users understand whether failure is due to implementation errors or due to inherent limitations of the system that flag potential unsoundness.

This objective is critical to ensure that the theoretical developments of CRETE are applicable to real software. The Liquid Haskell development will ensure, from early stages, that our methodology is useful, while the synergatice development of Liquid Rust and CRETE will ensure the generality of CRETE and the development of Liquid Rust under novel sound foundations. Importantly, this objective develops a verification framework where the user implements real world applications that are automatically verified using SMT and gets formal, machine-checked proofs, delivering the holy grail of practical and sound software verification.

a.3 The risks and difficulties

The CRETE project is of high risk because (1) it proposes a derivation of Coq proofs from real programs and (2) it proposes the definition of a logic for refined System F_ω. To ameliorate risk (1) we will carefully engineering the proposed translation and, as required, we will impose critical restrictions on the original system. To ameliorate the risk (2) we can restrict the system to a calculus that is sound but expressive enough to address the practical applications studied in CRETE. The PI is an active developer of Liquid Haskell for 9 years and has great experience with the design and implementation of practical refinement types, thus is in the unique position to distinguish the features of refinement type systems that are critical for real world applications. Further, the PI has active collaborations with experts in the areas of type theory and program semantics, e.g., Michael Greenberg, Alex Kavvos, and Gilles Barthe. Finally, IMDEA, without any teaching obligations and with expert colleagues (e.g., Aleksander Nanevski), provides a perfect work environment to carry out such a risky experiment.

a.4 Potential Impact

Scientific Impact: CRETE aims to develop the principles of refinement types and crystalize their connection with type theory. This deeper understanding will further encourage the adoption of refinement types systems by the verification and programming languages communities and shed new light to important, open research problems for example, error reporting and inference of type specifications.

Socioeconomic Impact: Refinement types have already been used in industrial software. Yet, the soundness problems are known and discourage further adoption. CRETE, via the current front-end of automated, practical refinement types, makes sound verification accessible to industrial developers. This low-cost, high-profit approach will encourage further adoption of formal verification and potentially refinement types will, by design, be integrated in future mainstream programming languages.

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 one postdoctoral researcher, three PhD students, and one research 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 UC San Diego and University of Maryland.

References

[1] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Wener. Verifying SAT and SMT in Coq for a fully automated decision procedure. In *PSATTT'11: International Workshop on Proof-Search in Axiomatic Theories and Type Theories*, Wroclaw, Poland, Aug. 2011. Germain Faure, Stéphane Lengrand, Assia Mahboubi.

[2] R. Atkey, P. Johann, and N. Ghani. Refining inductive types. *Logical Methods in Computer Science*, 7(2:9), 2012.

[3] C. Barrett, L. de Moura, and P. Fontaine. Proofs in satisfiability modulo theories. 07 2014.

[4] 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`.

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

[6] 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.

[7] J. Breitner, A. Spector-Zabusky, Y. Li, C. Rizkallah, J. Wiegley, and S. Weirich. Ready, set, verify. applying Hs-to-Coq to Real-World haskell code (experience report). *Proc. ACM Program. Lang.*, 2(ICFP), July 2018.

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

[9] R. Chugh, D. Herman, and R. Jhala. Dependent types for javascript. In *OOPLSA*, 2012.

[10] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, New York, NY, USA, 2000. Association for Computing Machinery.

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

[12] T. Coquand and G. Huet. Constructions: A higher order proof system for mechanizing mathematics. In B. Buchberger, editor, *EUROCAL '85*, pages 151–184, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

[13] L. de Moura and N. Bjørner. Proofs and refutations, and z3.

[14] B. Delaware, C. Pit-Claudel, J. Gross, and A. Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 689–700, New York, NY, USA, 2015. Association for Computing Machinery.

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

[16] C. Flanagan. Hybrid type checking. In *POPL*, 2006.

[17] A. D. Gordon and C. Fournet. Principles and applications of refinement types. In *Logics and Languages for Reliability and Security*, 2010.

[18] 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.

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

[20] 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.

[21] B. Jacobs. *Categorical Logic and Type Theory.* Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Amsterdam, 1999.

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

[23] 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.

[24] L. Lampropoulos, M. Hicks, and B. C. Pierce. Coverage guided, property based testing. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct. 2019.

[25] 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.

[26] P. Letouzey. Extraction in Coq, an Overview. In *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*, volume 5028 of *Lecture Notes in Computer Science*, Athens, Greece, June 2008. Springer-Verlag.

[27] A. Mebsout and C. Tinelli. Proof certificates for smt-based model checkers for infinite-state systems. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*, FMCAD '16, page 117–124, Austin, Texas, 2016. FMCAD Inc.

[28] R. Milner. Models of lcf. Technical report, Stanford, CA, USA, 1973.

[29] K. Mindermann, P. Keck, and S. Wagner. How usable are rust cryptography apis? pages 143–154, 07 2018.

[30] J. Parker, N. Vazou, and M. Hicks. Lweb: information flow security for multi-tier web applications. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, Jan 2019.

[31] L. C. Paulson, T. Nipkow, and M. Wenzel. From LCF to isabelle/hol. *CoRR*, abs/1907.02836, 2019.

[32] G. Petrou. Verification of algorithmic properties in liquid haskell, 2018. Diploma Thesis.

[33] G. Plotkin. Lcf considered as a programming language. *Theoretical Computer Science*, 5(3):223 – 255, 1977.

[34] D. S. Scott. A type-theoretical alternative to iswim, cuch, owhy. *Theoretical Computer Science*, 121(1):411 – 440, 1993.

[35] E. L. Seidel, N. Vazou, and R. Jhala. Type targeted testing. In J. Vitek, editor, *Programming Languages and Systems*, pages 812–836, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[36] M. Sozeau, S. Boulier, Y. Forster, N. Tabareau, and T. Winterhalter. Coq coq correct. verification of type checking and erasure for coq, in coq. *Proc. ACM Program. Lang.*, 4(POPL), Dec. 2019.

[37] 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.

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

[39] N. Vazou, L. Lampropoulos, and J. Polakow. A tale of two provers: Verifying monoidal string matching in liquid haskell and coq. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, Haskell 2017, page 63–74, New York, NY, USA, 2017. Association for Computing Machinery.

[40] 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.

[41] N. Vazou, E. Tanter, and D. Van Horn. Gradual liquid type inference. *Proc. ACM Program. Lang.*, 2(OOPSLA), Oct. 2018.

[42] A. Westerberg and G. Ung. Comparing verification of list functions in liquidhaskell and idris, 2019. Degree Project.

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 "rewriting 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-supervised 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 University 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.
Michael Hicks, David Van Horn on Applications of Verification to Security
Computer Science Department, University of Maryland, College Park, USA.
Michael Greenberg on Logics for Refinement Types
Computer Science, Pomona College, Claremont, USA.
Ranjit Jhala on Refinement Types
Computer Science and Engineering, University of California, San Diego, USA.

• FELLOWSHIPS AND AWARDS

2021 - 2023	Juan de la Cierva Incorporación Grant, by Spanish Ministry of Science and Innovation.
2020 - 2024	Atracción de Talento Fellowship, by Madrid Regional Government.
2018	Best paper award at ACM SIGPLAN Conference OOPSLA.
2017 - 2018	Victor Balisi Postdoctoral Fellowship, University of Maryland, College Park, USA.
2015	Graduate Award for Research, University of California, San Diego, USA.
2014 - 2016	Microsoft Research Graduate Research Fellowship.

Appendix: All current grants and on-goinng and submitted grant applications of the PI (Funding ID)

Current grants:

Project Title	*Funding source*	*Amount (Euros)*	*Period*	*Role of the PI*	*Relation to current ERC proposal*
Juan de la Cierva Incorporación	Spanish Ministry of Science and Innovation	93.000	2021 - 2023	PI	No overlap with CRETE.
Atracción de Talento Fellowship	Madrid Regional Government	80.000	2020 - 2024	PI	No overlap with CRETE.

On-going and submittedd grant applications: None

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 8 papers on CORE A* (flagship) conferences (2 ICFP, 2 POPL, 1 PLDI, 1 OSDI, and 2 OOSPLA). My h-index is 10 and I have 633 citations.

• **IMPORTANT PUBLICATIONS** Below I outline my 5 most important publications. Three of them are without the co-authorship of your PhD supervisor. 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. *205 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. *35 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.
- J. Parker, **N. Vazou**, and M. Hicks. Information Flow Security for Multi-Tier Web Applications. POPL, 2019. *19 citations*

  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.
- Martin Handley, **N. Vazou**, and G. Hutton. Liquidate your assets: Reasoning about resource usage in Liquid Haskell. POPL, 2020. *9 citations*

  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. *7 citations*

  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.

• INVITED PRESENTATIONS (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 WG 2.1: Algorithmic Languages and Calculu, Otterlo, Netherlands.
- 2019: Invited talk at IFIP WG 2.8: Functional Programming, Bordeaux, France.
- 2019: Invited talk at 12th Panhellenic Logic Symposium, Anogia, 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 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 conferences (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 UC San Diego and University 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 (e.g., in UC San Diego and 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 ones, the Atracción de Talento Fellowship (80K €), funded by Madrid regional government and Juan de la Cierva Incorporación Grant (93K €), funded by Spanish Ministry of Science and Innovation, both grants are focused to boost the career of young researchers.