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 program properties 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 (e.g., Haskell, C, Ruby, Scala, and the ML-family) to verify sophisticated properties of real world applications, e.g., safety of cryptographic protocols, memory and resource usage, and web security.

The weakness of refinement types is that they do not meet the soundness standards set by theorem provers. A sound verification system accepts as safe only those 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 though cannot be directly applied to refinement type checkers that aim to practically verify real world programs.

The goal of CRETE is to design a sound and practical refinement type system.

This is an ambitious goal that entails the development of 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 for mainstream languages (i.e., Haskell and Rust) and will be evaluated on real-world code, such as web applications 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, CRETE is high-gain since it proposes 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

Software verification validates that programs satisfy certain properties to ensure the absence of critical software bugs. The more costly software bugs become, in terms of money or debugging time, the more willing software developers are to ensure their absence using software verification techniques.

Refinement types [18] 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 a 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 [3]. 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}
```

Type checking the `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 [35], C [27], Ruby [19], Scala [16]) to verify sophisticated properties (e.g., about cryptographic protocols [4], reference aliasing [15], and resource usage [17]).

However, 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 (via unsafe FFIs) 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 specification, thus it is not sound. This concrete problem has an easy solution: F⋆ [31] and `Stainless` [16] 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 [33] 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

[1] We use the syntax of Haskell and Liquid Haskell [34].
[2] Stainless documentation on "Limitations of Verification":https://epfl-lara.github.io/stainless/limitations.html

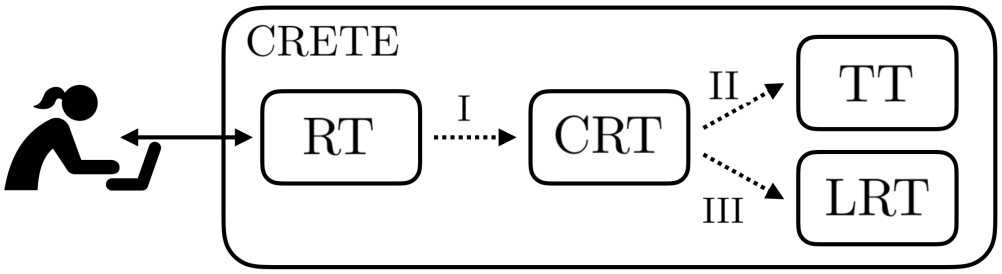

Figure 1: Objectives of CRETE. Starting from practical refinement types (RT, e.g., Liquid Haskell or Liquid Rust) we generate certified refinements (CRT; Objective I). CRT translate to type theory (TT, e.g., Coq; Objective II) and the logic of refinement types (LRT, variant of HOL; Objective III).

trusts the 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 big code bases that inevitably contain bugs and can potentially lead to unsound verification. Indeed, the implementations of F$\star$, `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 the existence of a consistent mathematical model. For example, Isabelle/HOL [25] has a core kernel of 5K lines of code that implements higher order logic (HOL [1]) whose consistency is known because there is a set theoretic model [1]. Coq [5]'s kernel is 14K lines of code that implement Calculus of Inductive Constructions (CIC [10]), a calculus also shown consistent by a set theoretic model [32]. Still, "on average, one critical bug has been found every year in Coq" [30]. So, one can only imagine how many critical bugs exist in the implementation of practical refinement type checkers, which have fewer users than Coq, no core kernel, and no strong mathematic support.

However, theorem provers are not practical for program verification. The first inconvenience arises 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`. In refinement types that is implicitly performed by the external SMT solver, without littering the executable code with proofs. 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 [22] to Ocaml or Haskell. This execution via extraction though, is not attractive to program developers that put extra effort and use idiomatic code to optimize program's runtimes.

The goal 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 secure web 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 mechanically 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; thus 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; thus 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; thus preventing *SoU2*, i.e., the assumption of inconsistent axioms. The final objective is to implement CRETE as 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 [23, 6] and tested against the language's runtime semantics by custom test generators [9, 20].

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 [37] and related work [13] 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 [29] 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 eliminating 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, 11], 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 [12] 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. ; also proved with the lia tactic

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` or linear arithmetic tactics that construct 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 [36] and various master thesis projects [26, 38]). In practice, 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 (from § a.1). We will define the logic of refinement types LRT, i.e., a deductive system of inference rules and axioms similar to HOL. 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 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 [14]) with the categorical interpretations of STLC [28]. 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 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 the axioms that the user can admit while preserving consistency. So, on the practical side, it would prevent inconsistencies that currently exist on practical refinement type checkers and jeopardize all user verification effort. Importantly, LRT (like CIC) will define a consistent deductive system with strong mathematic foundations that in the future can serve as the core calculus of the next-generation theorem provers.

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` [24] and `STORM` [21] 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 assumed. 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` [39]).

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 collaborating with new students on big code bases thus has learned techniques that minimize the entry costs, e.g., the development of isolated interfaces. The second challenge is that our certificate generation might not be possible for arbitrary code developed in our target languages, 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 failure messages helping both the developers and the users understand whether failure is due to implementation errors or due to inherent system limitations 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 synergistic 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.

Organization Three PhD students will respectively work on each of the first three objectives. The students will implement and evaluate their results in the Haskell development, that will be maintained by a research engineer. The Rust development will be lead by the PI's current PhD student. The PI, together with a postdoc, will supervise and ensure the smooth interaction of all the objectives.

a.3 The risks and difficulties

The CRETE project is high-risk because (1) it proposes a derivation of Coq proofs from real programs and (2) it proposes the design of a logic for refined System F_ω. To ameliorate risk (1) we will carefully engineer the proposed translation and we will impose the necessary 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 extensive experience with the design and implementation of practical refinement types, thus she 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 (in particular 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 attractive, 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] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory*. Springer, 2002. URL https://www.springer.com/gp/book/9781402007637.

[2] R. Atkey, P. Johann, and N. Ghani. Refining Inductive Types. In *Logical Methods in Computer Science (LMCS)*, 2012. URL https://lmcs.episciences.org/957.

[3] 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. URL http://www.smt-lib.org/.

[4] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement Types for Secure Implementations. In *Transactions on Programming Languages and Systems (TOPLAS)*, 2011. URL http://doi.acm.org/10.1145/1890028.1890031.

[5] Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004. URL https://www.springer.com/gp/book/9783540208549.

[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 *Certified Programs and Proofs (CPP)*, 2011. URL https://doi.org/10.1007/978-3-642-25379-9_15.

[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). In *International Conference on Functional Programming (ICFP)*, 2018. URL https://doi.org/10.1145/3236784.

[8] A. Chlipala. *Certified Programming and Dependent Types*. MIT Press, 2013. URL http://adam.chlipala.net/cpdt/.

[9] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *International Conference on Functional Programming (ICFP)*, 2000. URL https://doi.org/10.1145/351240.351266.

[10] T. Coquand and G. Huet. The Calculus of Constructions. In *Information and Computation*, 1988. URL https://doi.org/10.1016/0890-5401(88)90005-3.

[11] B. Delaware, C. Pit-Claudel, J. Gross, and A. Chlipala. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Principles of Programming Languages (POPL)*, 2015. URL https://doi.org/10.1145/2676726.2677006.

[12] B. Ekici, A. Mebsout, C. Tinelli, C. Keller, G. Katz, A. Reynolds, and C. W. Barrett. SMTCoq: A Plug-In for Integrating SMT Solvers into Coq. In *Computer Aided Verification (CAV)*, 2017. URL https://doi.org/10.1007/978-3-319-63390-9_7.

[13] C. Flanagan. Hybrid Type Checking. In *Principles of Programming Languages (POPL)*, 2006. URL https://doi.org/10.1145/1111037.1111059.

[14] A. D. Gordon and C. Fournet. Principles and applications of refinement types. In *Logics and Languages for Reliability and Security*. IOS Press, 2010. URL https://doi.org/10.3233/978-1-60750-100-8-73.

[15] C. S. Gordon, M. D. Ernst, and D. Grossman. Rely-Guarantee References for Refinement Types over Aliased Mutable Data. In *Programming Language Design and Implementation (PLDI)*, 2013. URL https://doi.org/10.1145/2491956.2462160.

[16] J. Hamza, N. Voirol, and V. Kuncak. System FR: Formalized Foundations for the Stainless Verifier. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2019. URL https://doi.org/10.1145/3360592.

[17] M. A. T. Handley, N. Vazou, and G. Hutton. Liquidate Your Assets: Reasoning about Resource Usage in Liquid Haskell. In *Principles of Programming Languages (POPL)*, 2019. URL https://doi.org/10.1145/3371092.

[18] R. Jhala and N. Vazou. Refinement Types: A Tutorial. Under review, 2020. URL https://arxiv.org/abs/2010.07763.

[19] M. Kazerounian, N. Vazou, A. Bourgerie, J. S. Foster, and E. Torlak. Refinement Types for Ruby. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2017. URL https://doi.org/10.1007/978-3-319-73721-8_13.

[20] L. Lampropoulos, M. Hicks, and B. C. Pierce. Coverage Guided, Property Based Testing. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2019. URL https://doi.org/10.1145/3360607.

[21] N. Lehmann, R. Kunkel, J. Brown, J. Yang, D. Stefan, N. Polikarpova, R. Jhala, and N. Vazou. STORM: Refinement Types for Secure Web Applications. To appear in USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2021.

[22] P. Letouzey. Extraction in Coq, an Overview. In *Conference on Computability in Europe (CiE)*, 2008. URL https://doi.org/10.1007/978-3-540-69407-6_39.

[23] A. Mebsout and C. Tinelli. Proof Certificates for SMT-Based Model Checkers for Infinite-State Systems. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2016. URL https://dl.acm.org/doi/10.5555/3077629.3077652.

[24] J. Parker, N. Vazou, and M. Hicks. LWeb: Information Flow Security for Multi-Tier Web Applications. In *Principles of Programming Languages (POPL)*, 2019. URL https://doi.org/10.1145/3290388.

[25] L. C. Paulson, T. Nipkow, and M. Wenzel. From LCF to Isabelle/HOL. In *Formal Aspects of Computing*, 2019. URL https://doi.org/10.1007/s00165-019-00492-1.

[26] G. Petrou. Verification of Algorithmic Properties in Liquid Haskell, 2018. URL http://artemis.cslab.ece.ntua.gr:8080/jspui/handle/123456789/17036. Diploma Thesis at National Technical University of Athens.

[27] P. Rondon, M. Kawaguchi, and R. Jhala. Low-Level Liquid Types. In *Principles of Programming Languages (POPL)*, 2010. URL https://dl.acm.org/doi/abs/10.1145/1707801.1706316.

[28] D. S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. In *Theoretical Computer Science*, 1993. URL http://www.sciencedirect.com/science/article/pii/030439759390095B.

[29] E. L. Seidel, N. Vazou, and R. Jhala. Type Targeted Testing. In *European Symposium on Programming (ESOP)*, 2015. URL https://doi.org/10.1007/978-3-662-46669-8_33.

[30] M. Sozeau, S. Boulier, Y. Forster, N. Tabareau, and T. Winterhalter. Coq Coq Correct. Verification of Type Checking and Erasure for Coq, in Coq. In *Principles of Programming Languages (POPL)*, 2020. URL https://doi.org/10.1145/3371076.

[31] 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 *Principles of Programming Languages (POPL)*, 2016. URL https://doi.org/10.1145/2837614.2837655.

[32] A. Timany and M. Sozeau. Cumulative Inductive Types in Coq. In *Formal Structures for Computation and Deduction (FSCD)*, 2018. URL https://doi.org/10.4230/LIPIcs.FSCD.2018.29.

[33] N. Vazou and M. Greenberg. Functional Extensionality for Refinement Types. Under review, 2021. URL https://arxiv.org/abs/2103.02177.

[34] N. Vazou, E. L. Seidel, and R. Jhala. LiquidHaskell: Experience with Refinement Types in the Real World. In *ACM SIGPLAN Symposium on Haskell (Haskell)*, 2014. URL https://doi.org/10.1145/2633357.2633366.

[35] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement Types for Haskell. In *International Conference on Functional Programming (ICFP)*, 2014. URL https://doi.org/10.1145/2628136.2628161.

[36] N. Vazou, L. Lampropoulos, and J. Polakow. A Tale of Two Provers: Verifying Monoidal String Matching in Liquid Haskell and Coq. In *ACM SIGPLAN Symposium on Haskell (Haskell)*, 2017. URL https://doi.org/10.1145/3122955.3122963.

[37] N. Vazou, E. Tanter, and D. Van Horn. Gradual Liquid Type Inference. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2018. URL https://doi.org/10.1145/3276502.

[38] A. Westerberg and G. Ung. Comparing Verification of List Functions in LiquidHaskell and Idris, 2019. URL https://kth.diva-portal.org/smash/get/diva2:1338661/FULLTEXT01.pdf. Degree Project at KTH Royal Institute of Technology.

[39] F. Ziegelmayer, A. Pavlov, N. Stalder, T. Arcieri, V. Filippov, and B. Warner. RustCrypto: Cryptographic algorithms written in pure Rust, 2021. URL https://github.com/RustCrypto. github organization.

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. - PhD candidate Lisa Vasilenko on "relational refinement types". - Master student Mustafa Hafidi on "tactics for Liquid Haskell'. - Undergrad student David Munuera on "Haskell to CIAO". - Research intern Zack Grannan on "rewriting for Liquid Haskell'. - Research intern Stefan Malewski on "gradual refinement types". IMDEA Software Institute, Madrid, Spain.
2017-2018	- PhD candidate James Parker on "verification of secure web applications", co-supervised with Michael Hicks and published on POPL'19 and OOPSLA'20. - PhD candidate 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 months/20 hours 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	1 week/40 hours course on Functional Programming, 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 and *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-going 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.

Note: Both current grants are fellowships that did not require concrete scientific proposal or tasks.

On-going and submitted 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 OOPSLA). My h-index is 10 and I have 635 citations.

• **IMPORTANT PUBLICATIONS** Below I outline my five most important publications. Three of them are without the co-authorship of Ranjit Jhala, my 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. *206 citations*

  This paper introduces Liquid Haskell, a refinement type checker for Haskell programs and uses Liquid Haskell to 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 portions 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 portions of the metatheory and evaluation.

- J. Parker, **N. Vazou**, and M. Hicks. Information Flow Security for Multi-Tier Web Applications. POPL, 2019. *20 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 part 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 communities 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 to present my work on industrial conferences (e.g., Scala and Haskell eXchange, organized by Skills Matters, one the world's 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 events I have organized appears in the CV. Here I am emphasizing that I have been a member of the organization committee of many A* programming languages conferences since 2018: as co-organizer of PLMW at ICFP'18, co-organizer of 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 young 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 closely collaborated with two PhD students that now have excellent positions: James Parker who is a research engineer at Galois and Milod Kazerounian who is a lecturer at Tufts University. In my two first years as an assistant professor at IMDEA, I have already and selectively attracted two PhD students and two 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 lead developers and maintainers of Liquid Haskell a refinement type checker for Haskell programs that has been used both for educational purposes in graduate and undergraduate classes (e.g., in UC San Diego and University of Maryland) and industrial companies (including Well-Typed, Tweag-IO, and 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.