ERC Starting Grant 2021
Part B2
CRETE: Certified Refinement Types

Section a. State-of-the-art and objectives

a.1 Introduction

Refinement types [44] 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` of 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 [6]. 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 type systems have been developed for various programming languages (Ocaml [36], Haskell [85], Ruby [45], Scala [41]) and have been used to enforce sophisticated properties (for example, about cryptographic protocols [9], reference aliasing [39], resource usage [42], and web security [50]).

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

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

For this concrete problem there is an easy solution: F⋆ [79] and `Stainless` [41] 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. 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. Finally, unlike type-theory based verifiers, refinement type checkers do not rely on a core kernel. Usually, the implementation of refinement type checkers trusts the compiler of the underlying language to generate an intermediate representation language (IRL), the type checking rules (adapted to accommodate the IRL) to generate logical verification conditions (VC) and the SMT to validate the VCs. These three trusted components are prone to implementation bugs that can lead to unsound verification.

In short, while practical refinement types have been extensively used to verify sophisticated properties of real world programs, their soundness is debatable.

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

[1] We use the syntax of Haskell and Liquid Haskell [85].

	DML	SAGE	F⋆	Liquid Haskell	Stainless
Existing Language	✓	✓	✓	✓	✓
Decidably Typed	✓	✓	×	✓	✓
Statically Typed	✓	×	✓	✓	✓
Type Inference	✓	×	×	✓	×
Expressive	×	✓	✓	✓	✓
Predicate Subtyping	×	✓	✓	✓	×
Polymorphism	×	×	✓	✓	×
Data Type Subtyping	×	×	×	✓	✓

Table 1: Desired features of practical refinement types and systems that implement them.

a.2 State-of-the-art

Next we discuss the state-of-the-art in refinement types. First (§ a.2.1), we compare the type-based verification approach of refinement types with assertion-based verification. Then, we present characteristic features (§ a.2.2) and soundness claims (§ a.2.3) of existing refinement type checkers. Finally (§ a.2.4), we compare the latter with soundness claims of theorem provers.

a.2.1 Type-based *vs.* Assertion-based verification

Refinement types is a type-based alternative to assertion-based verification systems, e.g., SPEC#[5], ESC/Java[34], Dafny[51], and more recently OpenJML[22] and Prusti[3]. Such systems, similar to refinement types, start from an existing programming language and automatically discharge assertions with an external solver. But, unlike refinement types, they directly encode the target language into the solver's logic, without taking advantage of the type system to generate verification conditions.

Refinement types use the abstractions of types systems in two key ways to facilitate program verification. First, the polymorphism – inherent in refinement type systems – leads to "theorems for free"[89]. For example, consider the function `(++)` that appends two lists:

```
(++) :: xs:[a] → ys:[a] → {v:[a] | len v = len xs + len ys }
```

Calling `(++)` with two lists of integers greater than a variable `x`, i.e., `xs,ys :: [{v:Int | x < v }]`, returns a list that preserves the same property `xs ++ ys :: [{v:Int | x < v }]`, because of refinement type instantiation of the type variable. Such reasoning, taken for free in type-based verification, requires qualification in assertion-based systems that quickly leads to unpredictable verification [52].

Second, the abstraction of refinement type specifications provides a clear separation between the function definition and the properties relevant to verification. For instance, refinement type-based verification of the callers of `(++)` is unaware of the `(++)`'s recursive definition. On the negative side, this approach is imprecise and unable to deduce that `2` belongs to the list `[1] ++ [2]`. On the positive side, it can trivially reason about sophisticated length invariants on lists, since the type of `(++)` precisely captures how the function modifies the length of lists. Thus, refinement type based verification, as opposed to assertion based systems, provides the ground for modular and decidable verification.

a.2.2 Practical Refinement Types

Figure 1 summarizes the main ingredients of refinement types and lead systems that implement them.

The term refinement types was introduced in 1991 [**?**] at a system, later named Dependent ML (`DML`) [92], that syntactically refines ML[59]'s data types into more precise subsets. For example, `singleton` can be defined by the user as a `list` that contains exactly one element. This form of refinements, now known as *data refinements* [30], combined with indexed types [94] is implemented in `DML` that has decidable type inference, via abstract interpretation and linear programming, to automatically verify data structure invariants [29]. Importantly, `DML` set the first objectives for practical refinement types. They need to be integrated with an existing programming language, enjoy decidable type checking (that can be automated by an external SMT solver) and type inference so that the user does not need to explicitly annotate every intermediate term.

The SAGE system [40] generalized refinements from data types to any base type, introducting the syntax $\{v{:}t \mid r\}$, where t is any base type (e.g., `Int`, `Bool`) and r is any pure, boolean expression of the underlying language. SAGE uses semantic, *predicate subtyping* [69] to generate verification conditions and an SMT solver (Simplify [27]) to validate them. In SAGE subtyping, and thus type checking, is undecidable and checked in a *hybrid* manner: partly at compile time using SMT solvers and partly at run-time via dynamic contract checks [33].

F7 [9, 11], later renamed to F$\star$ [78], changed the language r of refinements into expressions of the underling solver to restore static type checking. In their design, refinements can include qualifiers, thus type checking is undecidable (that in practice is controlled with SMT hints), but the system gets expressive enough to verify real world applications, among which cryptographic routines of web-browsers [96, 65]. F$\star$ now also includes dependent types [79], effects [66] and metaprogramming [55].

Liquid types [67], now used by Liquid Haskell [86], made decidable type checking a priority. Their design restricts the language r of refinements to logics that can be efficiently decided by SMTs (e.g., equality, uninterpreted functions, linear arithmetic, data types, but no qualifiers), for the shake of decidable type inference and predictable verification. To restore expressiveness, Liquid Haskell allows refinement and subtyping of polymorphic types and refinement reflection [88] that allows controlled reasoning about pure, terminating functions of the underlying language. Liquid Haskell has been used to verify a wide range of real world applications, including resource usage [42], security metaproperties [60], and properties of distributed code [53].

Refinement types is a promising practical verification technology that in the last decade has spread to mainstream languages. [68] and [19] show how to verify C and JavaScript programs by refining a low-level language of locations [74]. [46] show how refinements can be integrated within Racket's occurrence based type system [81]. [45] integrate refinements in Ruby's type system using just-in-time type checking. Finally, [**?**] present a refinement-type based verifier for higher-order Scala programs.

a.2.3 Soundness of Refinement Types

Data refinements (i.e., the DML-style refinement types) comes with strong metatheoretic principles [93]: [54] defines a logical framework where refinements are proof-irrelevant predicates and [57] gives a categorical interpretation of refinement types. Yet, this line of work studies only refinements of data types and does not trivially generalize to practical refinement type systems.

F$\star$ [38] followed the LCF [63] approach to set the principles of refinement types, where an external, clearly defined logic is used to validate subtyping. This and similar approaches [1] though, do not define the correspondence between the logic and the verified language. Further, no LCF-style approach is defined to allow refinements on type variables or data types, features used by practical systems.

SAGE came with a novel soundness proof [47] that shows soundness of refinement type systems with respect to operational semantics. The denotation of the refined type $\{v{:}t \mid r\}$ is defined to be the set of all expressions e, with unrefined type t, for which $r[e/v]$ evaluates, using operational semantics to true, i.e., $r[e/v] \hookrightarrow^\star \texttt{true}$. Denotational subtyping is also defined via operational semantics over all possible instantiations of the typing environment and is undecidable. In practise, an external SMT solver is used to approximate this undecidable subtyping. Soundness in this setting is defined using denotational inclusion: is e has type τ, then e belongs in the denotation of τ. For example, since $2 : \{v{:}\texttt{Int} \mid 0 < v\}$, then $0 < 2 \hookrightarrow^\star \texttt{true}$. This proving methodology provides a deep intuition on the semantics of refinement types and has been extensively used to formalize gradual types [8, 73]. But when used to formalize static verification of refinements types, it turns out to be insufficient.

Liquid Haskell [86] used SAGE's methodology to formalize its core calculus. Such formalizations are insufficient for three reasons. First, they do not formalize the SMT approximation of the underlying programming languages, easily leading to unsoundness (§ a.1). Second, unlike the LCF-style formalization, they do not clarify the logic used by refinement types, leaving it uncertain for the users what assumptions can be consistently made. For example, the PI recently detected that function extensionality had been encoded inconsistently in Liquid Haskell for many years [86]: the user was able to prove false each time the function extensionality axiom was used, invalidating all their verification efforts. Third, formalizations of small core calculi usually leave out sophisticated features supported by the actual implementation. Stainless [**?**] comes with a formalization of the complete refined system F, but, by design, Stainless is not using semantic subtyping and refined polymorphism as Liquid

Haskell and F⋆, thus it cannot be directly applied for these systems. Ideally, a complete formalization of refined System F_ω with semantic subtyping is required to prevent future and understand known inconsistencies of refined systems. For example, Liquid Haskell is known to be inconsistent in the presence of recursive negative types [2]. In research track III, we will develop the logic of refinement types to address these inconsistencies.

Such a theoretical development though, would not be sufficient to guarantee soundness of practical refinement type checker implementations. The implementations of F⋆, `Stainless`, and Liquid Haskell respectively consist of 1.3M, 185.3K, and 423K lines of code, and none of them concretely isolates a trusted core kernel. So, bugs that are inevitable in such big code bases can potentially lead to unsoundness. For example, in Liquid Haskell there are reported approximately 5 soundness bugs per year. Thus, to ensure soundness of the actual implementation a small trusted kernel is required.

a.2.4 Design of Sound Theorem Provers

The recipe to implement a sound theorem prover is known. The prover should consist of a small trusted kernel than implements a set of deductive rules that are proved consistent by reduction to a consistent mathematical theory. For example, Isabelle/HOL [61] has a core kernel of 5K lines of ML code that implements LCF [58, 63] whose consistency is known because of a sound translation to the domain theory and continuous functions [71]. Coq's kernel is 14K lines of ML code that implement CIC [24, 23] a calculus shown consistent by reduction to the theory of sets and functions [43]. Still, "on average, one critical bug has been found every year in Coq" [75]. So, one can only imagine how many critical bugs exist in the implementation of practical refinement type checkers.

Sadly, the soundness recipe cannot be applied to refinement type checkers. Their implementation highly depends on external solvers for automation and on the compiler of the underlying programming language for extraction of the intermediate representation language (IRL). Developed to support evolving programming languages, IRLs are subject to change, e.g., to facilitate runtime optimizations or support of novel language features. Thus, the development of a stable core kernel in a practical refinement type checker is not feasible. In research track II we aim to address this problem by reducing refinement to dependent type checking, and thus use Coq's soundness guarantees.

a.2.5 The Design Spot: Between Legacy Programming Languages and Theorem Provers

In theory, and due to Curry–Howard isomorphism, theorem provers can be used as general purpose programming languages and vice versa. Theorem provers support program execution via code extraction (e.g., Coq) or native runtime semantics (e.g., Agda, Idris), while some legacy programming languages (e.g., Haskell and Scala) support dependently typed programming thus, theorem proving. In practise though, there is a clear distinction between theorem provers and languages used for industrial software development, based on the design and maintenance decisions taken by their developers.

Haskell is a characteristic example of an industrial programming language that is being extended with theorem proving facilities. Haskell's industrial users (e.g., Github and IOHK) have been sponsoring the development and maintenance of the GHC Haskell compiler and expect that the new releases of the compiler will satisfy their implicit requirements, including backward compatibility of the new releases, no runtime slowdowns, and addition of new features. These requirements do not facilitate the extension of GHC with dependently typed features. In 2012 [31] Eisenberg et al envisioned a GHC with CIC-style, dependent types, which is actively developed until now [18]. This development is tremendously slowed down by the code reviewing and backward compatibility requirements of an industrial programming language. Further, by design, dependent types in GHC cannot meet the small, core kernel requirement of a sound theorem prover.

Lean [25], on the opposite direction, was designed as a theorem prover and now allows for general purpose programming. In 2013, Lean 1 was developed as highly automated theorem prover in which implicit proofs elaborate, via metaprogramming and tactics, into a dependent type theory (a version of CIC) and checked by a small kernel. Now, Lean 4, implemented in Lean 4, is a general purpose programming language that preserves the requirements of a sound theorem prover but also has a compiler that generates C/LLVM code and supports multithreading. But, this is not sufficient to gain

[2] Liquid Haskell unsoundly allows subtyping on data types github:issues/159 while F⋆ doesn't allow it github:issues/65.

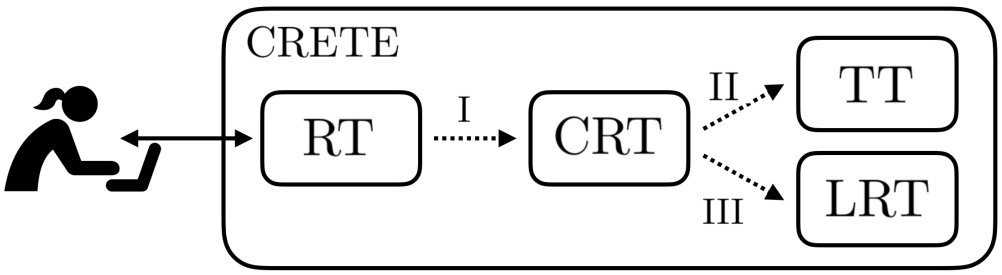

Figure 1: Objectives of CRETE. Starting from practical refinement types (RT, e.g., Liquid Haskell or Liquid Rust) we generate certified refinements (CRT; Objective I). CRT translate to type theory (TT, e.g., Coq; Objective II) and the logic of refinement types (LRT, variant of HOL; Objective III).

industrial users. As stated by the developers[3], "We constantly break backward compatibility. The users should not expect we will fix bugs and/or add new features for their project." This clarifies that Lean is not indented for industrial development.

Refinement types occupy a unique design spot between industrial programming languages and sound theorem provers. Implemented externally from the language compilers, refinement type checkers do not need to abide by the strict maintenance requirements of industrial programming languages. Thus refinement type checkers can follow more flexible, research-oriented release circles while still being used by industrial developers. Currently, refinement type checkers do not meet the soundness standards set by theorem provers, which we aim to address in this proposal.

a.3 CRETE Scientific Objectives

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.

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.

II: Translation of CRT to Type Theory The second objective is to translate CRT programs into Coq, advancing both the theory and practice of refinement types. In theory, this objective will formalize the relationship between refinement types and constructive type theory as verification techniques. In practice, 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.

III: Logic of Refinement Types The third 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

[3] `https://leanprover.github.io/lean4/doc/faq.html`

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.

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.

Section b. Methodology

Each of the four objectives is addressed by a research track. Next, for each research track of CRETE, we present the required tasks and summarize the importance and challenges. § b.5 summarizes the organization of the research tracks and § b.6 provides concluding remarks.

b.1 Research Track I: Certified Refinement Types

In this research track we develop CRT, a certified refinement typed language that replaces the SMT solver with certified terms (Task I.1). The certificates are validated in Task I.2 and tested in Task I.3.

Task I.1: Certificate Generation

In this task, we will define a program transformation that introduces (trusted) certificates to replace SMT invocations during refinement type checking. For example, consider the `test` definition below:

```
test :: xs:[a] → i:{ℕ | i < len xs - 1} → a
test xs i = get xs (i + 1)
```

Typechecking of `test` will first introduce `test`'s two arguments into the typing environment $\Gamma \doteq \{\ldots, xs : \texttt{[}a\texttt{]}, i : \{i{:}\mathbb{N} \mid i < \texttt{len}\ xs - 1\}\}$ and then will check the below function application rule.

$$\frac{\Gamma \vdash \texttt{get}\ xs\ : \{v{:}\mathbb{N} \mid v < \texttt{len}\ xs\} \to a \qquad \Gamma \vdash i+1 : \{v{:}\mathbb{N} \mid v < \texttt{len}\ xs\}}{\Gamma \vdash \texttt{get}\ xs\ (i+1) : a}$$

To decide that the index `i+1` is a good index for `get`, i.e., it satisfies `get`'s precondition from § a.1, type checking will generate a subtyping rule, that will get validated by the SMT solver.

$$\frac{\Gamma \vdash i+1 : \{v{:}\texttt{Int} \mid v = i+1\} \qquad \dfrac{\texttt{SMTValid}(0 \le i < \texttt{len}\ xs - 1 \Rightarrow v = i+1 \Rightarrow 0 \le v < \texttt{len}\ xs)}{\Gamma \vdash \{v{:}\texttt{Int} \mid v = i+1\} \preceq \{v{:}\mathbb{N} \mid v < \texttt{len}\ xs\}}}{\Gamma \vdash i+1 : \{v{:}\mathbb{N} \mid v < \texttt{len}\ xs\}}$$

The safety and soundness of this derivation depends on the language representation of `Int` and their encoding to SMT's terms. For example, in F⋆ type checking will soundly succeed, since `Int` denotes unbounded integers but would fail if the base integer type was of fixed precision (e.g., `UInt32.t` represented in as bitvector in SMT). In Liquid Haskell, `Int` is fixed-precision, but for precise verification it gets represented to SMT unbounded integers, thus type checking will unsoundly succeed. In short, the encoding of language to SMT expressions is difficult and should not be trusted.

In this task, we will use the type derivation tree to replace the (untrusted) SMT invocations by an (again untrusted) certificate. So, in our example, we generate the below certified derivation tree.

$$\frac{\ldots \qquad \dfrac{}{\Gamma \vdash_{CRT} \texttt{cert}(i+1, \{v{:}\texttt{Int} \mid 0 \le v < \texttt{len}\ xs\}, \Gamma) : \{v{:}\texttt{Int} \mid 0 \le v < \texttt{len}\ xs\}}}{\Gamma \vdash_{CRT} \texttt{get}\ xs\ \texttt{cert}(i+1, \{v{:}\texttt{Int} \mid 0 \le v < \texttt{len}\ xs\}, \Gamma) : a}$$

That is, the certificate will replace the subtyping rule and will get propagated in the original expression, leading to the following transformed expression, that can now be type-checked without the SMT.

$$\lambda xs, i.\texttt{get}\ xs\ (i+1) \hookrightarrow \lambda xs, i.\texttt{get}\ xs\ \texttt{cert}(i+1, \{v{:}\texttt{Int} \mid 0 \le v < \texttt{len}\ xs\}, \Gamma)$$

To generalize, we will define the CRT variant of an original refinement typed language RT by extending RT's expressions to include the *trusted* primitive $\texttt{cert}(e, \tau, \Gamma)$ and by modifying RT's typing rules to replace SMT invocations with the below trusted T-Cert rule.

$$\frac{}{\Gamma \vdash_{CRT} \texttt{cert}(e, \tau, \Gamma) : \tau} \text{ T-Cert}$$

The goal is to define the type-preserving transformation $\bullet \hookrightarrow \bullet$ from RT to CRT terms:

Goal 1 *If* $\Gamma \vdash e : \tau$ *and* $e \hookrightarrow e'$*, then* $\Gamma \vdash_{CRT} e' : \tau$*.*

The definition of the translation will be directed by the original type derivation, following the techniques of hybrid and gradual types [33, 90, 8, 28, 2]. Unlike these systems that convert the certificates into runtime casts, which unavoidably slows down runtime [80], our certificates need to explicitly carry the typing environment Γ, to permit static certificate validation and testing.

Task I.2: Certificate Validation

In this task we will validate the explicit certificates using the proof certificate generation and validation technology [56, 14] provided by the SMTs, under various program-to-SMT encodings.

For example, the certificate validation of the example in Task I.1 will generate the following error.

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

test xs i = get xs (i + 1)
                   ^^^^^^^
```

The generated certificate will be validated by both CVC4 and Z3 under two different encodings that map `Int` to SMT `Int` and `BitVec`. Since validation failed only on the fixed-precision encoding the system will infer potential solutions and let the user decide if the runtime efficiency offered by fixed-precision `Int` is more important than the lack of overflows offered by `Integers`.

To generalize, we will implement certificate validation with three features. First, certificates can be validated under different encodings of the CRT to SMT language (that will be designed by the developers). Second, validation will use more than one SMT solvers, which has shown (by Sledgehammer [13]) to be extremely useful in cases where solvers diverge. Third, and most importantly, we will generate and externally validate SMT proof certificates (e.g., CVC4 generates proof certificates that are checked in the logical framework LFSC [77]), thus drastically reducing the trusted code base.

Task I.3: Certificate Testing

In this task we will use property based testing [21] to test, instead of validating, the generated certificates. In our example, the certificate will be falsified with the $\texttt{maxBound} = 2^{63} - 1$ as a counterexample.

To do so, we will design custom test generators that properly cover the derived certificates. For instance, a random generator for `Int`s will not find the `maxBound`, corner case counter-example. Instead, we need to design a custom generator that favors tests close to the maximum and minimum corners. To design such generators we will use coverage guided test generation techniques [49] combined with techniques to generate tests for function and polymorphic [20, 10] certificates.

Importance & Challenges

This research track addresses two key problems. First, it reduces the trusted components of refinement type checkers by external certificate validation. Second, it addresses the language-to-SMT correspondence problem by the provision of multiple encodings and by testing the certificates. Even though our overflow example is trivial and has a known solution, as noted in the `Stainless` documentation[4], the definition of a "precise correspondence between runtime execution semantics and the semantics used in verification" is a difficult problem, that only gets worse in the presence of limited memory, equality, and higher-order features [82]. Testing the certificates will not only bypass the potentially unsoundness of this difficult encoding, but also will generate counter-examples that the user can much easier understand compared to verification errors.

There are three main challenges in this track. First, practical refinement type systems incorporate various sources of automation of top of SMT solvers (e.g., inference or evaluations) that are expected to complicate the certificate derivation. The PI has long years of experience on building and understanding such automations, thus she is one of the few experts that is capable to isolate them in explicit certificates. The second challenge is how to expose to the user the various certificate validation and testing options in a user friendly way. To address this challenge, we plan to follow the paradigm of Isabelle/HOL and build an editor integrated user interface that presents the validation options and results. The PI plans to hire a research programmer that will help with the development and maintenance of this interface. The final challenge is the developement of a customized testing suite for each certified expression. The PI has experience on the relevant fields of test generation [72] and gradual types [87], but is not an expert in the area of custom test generation. For this task, the PI will collaborate with Leonidas Lampropoulos who is an expert in the field of testing.

b.2 Research Track II: Translation of CRT to Type Theory

In this research track we will define a translation of CRT to type theory, with the goal to attain sound, machine-checked proofs of the original CRT program. Concretely, we will define the translation as a (partial) function $\bullet \rightsquigarrow \bullet$ from CRT to Coq (distinguished using red and green color when required). We describe the translation on data types (Task II.1), specifications (Task II.2), and terms (Task II.3) and aim to produce correct-by-construction Coq programs (Goal 2).

Task II.1: Refined to Inductive Data Types

In this task we translate refined data types to Coq's inductive definitions. For example, lists refined to capture their length will get translated into inductive vectors.

```
data [a] where                                  Inductive vec (A:Type) : ℕ → Type :=
  ([]) :: {l:[a] | len l = 0 }                    | nil  : vec A 0
  (:)  ::  a → xs:[a]                             | cons : ∀ len, A → vec A len
       → {l:[a] | len l = len xs + 1}                    → vec A (S len).
```

$\rightsquigarrow$

On the left, the list algebraic data type (ADT) is refined to axiomatize its length, i.e., the length of `[]` is zero, while the length of `(x:xs)` is one plus the length of `xs`. It's translation on the right returns a vector which is indexed by a natural number `len` so that the index on `nil` is zero, while the index on `cons x xs` is the successor of the index on `xs`.

With this definition, the translation will turn lists $x : [a]$ to vectors $x :$ `vec` A $\texttt{len}_x$ and calls to lengths `len` x into x's index $\texttt{len}_x$:

Refined List Translation			*General ADT Translation*			
$x : [a]$	$\rightsquigarrow$	$x :$ `vec` A $\texttt{len}_x$	$x :$ `T` a_1 … a_n	$\rightsquigarrow$	`T` A_1 … A_n f_{1x} … f_{mx}	
`len` x	$\rightsquigarrow$	$\texttt{len}_x$	$f\ x$	$\rightsquigarrow$	f_x	

This behavior naturally generalizes (above; on the right) to ADTs `T` with n type parameters that are refined with m inductive refinements.

The translation will fail on various not inductively defined data types. For example, in data definitions with negative recursive definitions, the translation will fail with a descriptive error message.

```
data Evil a where                                Error: Evil doesn't have a strictly
  Very :: (Evil a → a) → Evil a                   positive occurrence
```

$\not\rightsquigarrow$

The definition of `Evil` in a refinement program can lead to unsoundness (§ a.2), thus failures of the translation (with descriptive failing messages) already signals potential inconsistencies.

A translation between refined and inductive data types is an active research area. For example, [4] studies this systematic translation via categorical theoretic perspectives and [73] compares the two representations to conclude that both are equally expressive while each requires different proving techniques. Yet, both approaches focus on small examples. In practical applications multiple properties can refine the same ADT (e.g., lists can be refined with length and nullity properties) while the same ADT can satisfy different refinements (e.g., [84] lets you treat the same list as either increasing or decreasing). More interestingly, refinement types let you freely refine codata. The goal of this task is,

based on the existing work, to implement a systematic translation for refined ADTs of real-world code (e.g., the benchmarks of [84]) and address the challenges and requirements of practical applications.

Task II.2: Classical Refinement Types to Constructive Subset Types

In this task we will translate refinement type specifications to Coq specifications using subset types. For example, the specification of `get` from § a.1 will be translated as follows.

```
get :: xs:[a] → i:{ℕ | i < len xs} → a
```

⇝

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

Translation of the list and length follows Task II.1, while refinement types turn into subset types.

Subset types [17, 26] in Coq contain a value `x` combined with a predicate `P` that `x` satisfies.

```
Inductive sig (A : Type) (P : A → Prop) : Type := exist : ∀ x : A, P x → sig P
```

Coq actually comes with with defined notation of subset as "refinement" types.

```
Notation "{ x : A | P }" := sig (fun x : A => P)
```

So, the type `{i: ℕ | i < len_xs}` used to index `get` is notation for `sig (fun i : ℕ => i < len_xs)`.

Inhabiting subset types is not trivial. The constructor `exist` needs three components: a predicate `P`, an expression `x`, and a proof that `P x` holds. The requirement of an explicit proof term is the critical difference between inhabitants of refinement and subset types, while in Task II.3 we discuss how the former can be converted to the latter using the explicit certificates.

Subset types are not commonly used in Coq because the inhabitant generation is complicated. For instance, [76] specifies `get` using an inductive predicate `less len_xs` that captures all natural numbers less than `len_xs`. Such an alternative encoding would complicate program transformation. The term `i:len_xs` declares the existence of an in-bounds `i`, but (unlike subset types) does not carry the concrete value for `i`, thus the translation would need to reconstruct the concrete value of `i` when used by the CRT original program.

Both approaches share the challenge that the logical connectives have different semantics in the two systems. Concretely, implication and negation are classical in the refinement types but intuitionistic in dependent types. To address this challenge we will use the CPS translation of classical to intuitionistic logic [37] to translate the classical connectives of the refinements into constructive in Coq.

Task II.3: Certified to Proof Terms

In this task we will translate the CRT into Coq terms. For the example, the certified `test` function of Task II.1, will get translated as follows (for brevity we write _ instead of the certificate's environment):

```
test :: xs:[a] → i:{ℕ | i < len xs - 1} → a
test xs i = get xs (cert (i + 1) {v:ℕ | v < len xs} _)
```

⇝

```
Lemma lemma : ∀ (i : ℕ) (n : ℕ), (i < n - 1) → (i + 1 < n).
Proof. smt. Qed.

Definition test {A len_xs} (xs:vec A len_xs) (r:{i : ℕ | i< len_xs - 1}) : A :=
  match r with
    | exist _ i pf  => get xs (exist (fun v: ℕ => (v < len_xs)) (i + 1) (lemma i len_xs pf))
  end.
```

The type of `test` is translated following Task II.2. Here we describe the translation of `test` definition. Originally, the translation follows the CRT definition, until the certificate is found, i.e., bidirectionally. At that point the certificate `cert (i + 1) {v:ℕ | v < len xs} _` gets converted into the subset term `exist pred e proof` and translation reduces to the derivation of the three components `pred`, `e`, and `proof`. The expression `e` exactly matches the certified expression `i+1`. The predicate `pred` is simply derived by turning the refinement type of the certificate `{v:ℕ | v < len xs}` into the predicate `fun v: ℕ => (v < len_xs)`. This step clarifies the importance of certified expressions in this research track:

if the translation was attempted directly from refinement types to Coq, the derivation of such valid (intermediate) predicates would be challenging. Instead, now the main challenge is to derive adequate proof terms. Again, directed by the certificate, we seek the proof that `(i + 1 < n)`. The translation derives a lemma that `(i < n - 1)` $\rightarrow$ `(i + 1 < n)`. To call the `lemma` we need the proof that `(i < n - 1)` which exists in the subset argument `r`, thus the whole definition is wrapped in a pattern match. To prove the `lemma` we use SMTCoq [32] that allows verified SMT tactics within Coq.

The crux of this task is the systematic extraction of proof terms that validate the CRT certificates. Essentially, each certificate will get translated to a subset term as follows

$$\texttt{cert}(e, \{v{:}t \mid p\}, \Gamma) \rightsquigarrow \texttt{exist}\ (\texttt{fun}\ v : t' \Rightarrow p')\ e'\ (\texttt{make-proof}\ e'\ p'\ \Gamma')$$

where $e \rightsquigarrow e'$, $t \rightsquigarrow t'$, $p \rightsquigarrow p'$, and $\Gamma \rightsquigarrow \Gamma'$. The challenge in this task is to systematically generate proof terms for the type $p'\ e'$. Since the certificates were generated (in Task I.1) based on SMT proofs then using SMTCoq (as in the `test` example), it will always be possible to systematically reconstruct the proofs. To do so, we will define the metafunction `make-proof` $e\ p\ \Gamma$ that systematically constructs the proof that $p\ e$ directed by the environment Γ.

In short, the goal of the task is to translate CRT into correct-by-construction Coq programs. Let P be a CRT program that contains refined data types, type specifications and function definitions. If the program type checks in CRT, i.e., $\vdash_{CRT} P$, and the translation succeeds, i.e., $P \rightsquigarrow P$', then the derived program should be accepted by Coq:

Goal 2 *If* $\vdash_{CRT} P$ *and* $P \rightsquigarrow P$', *then* $\vdash_{Coq} P'$.

The translation will combine bidirectional program synthesis [35] with the SMTCoq [32] generation of set type inhabitants, as described in the `test` example. As part of the research track IV, we will implement the translation and evaluate it in real world applications.

Importance & Challenges

This research track is important for both the theory and practise of refinement types. Our translation will formalize the relation between classical refinement types and constructive type theory, that until now has been a frequent question answered only by case-study comparisons (from the PI [83] and various master thesis projects [62, 91]). In practise, in this track we will translate programs verified using *practical*, SMT-automated refinement types into the *sound*, mechanically-checked Coq proofs.

The challenge of this research track depends on the complexity of CRT programs we will target. Some programs will just be impossible to translate, e.g., programs that contain non-positive data types like `Evil` of Task II.1. We will gradually increase the complexity of the programs we target and ensure that the translation provides accurate error messages in case of failure, so it is usable from the early stages. Based on the `hs-to-coq` [16] project that translates significant portions of Haskell's real-world library into Coq, we expect our translation to be applicable to real code. Of course, in `hs-to-coq` the Coq proofs are performed manually, but in this research track the proof derivation is directed by the CRT certificates and will be using SMTCoq and further tactics as required. At the first stages, we will allow the user to manually edit and generate proofs terms that the translation fails to construct. The PI plans to collaborate on this track with Joachim Breitner, lead contributor of `hs-to-coq` and Coq expert.

b.3 Research Track III: Logic of Refinement Types

In this research track we define LRT, the logic for refinement types. In Task III.1 we present an LCF-style, extensible methodology that consistently defines LRT. We discuss extensions of our methodology to ADTs (Task II.2) and eventually System F_ω (Task II.3).

Task III.1: Definition of a consistent LRT

Figure 2 summarizes our methodology to define LRT, the logic of refinement types. We begin with an existing programming language PL with expressions e, (unrefined) types t and an typing relation $\Gamma \vdash_{PL} e : t$. In this task we pick PL to be simply typed lambda calculus (STLC) and later extend it. We define LRT on top of PL (à la LCF [71]), that is the syntax of propositions ϕ in LRT includes expressions of PL and quantifiers over these expressions:

$$\phi \doteq \forall_{x \in PL}\ \phi \mid \exists_{x \in PL}\ \phi \mid e$$

We define the judgement $\Gamma \vdash_{LRT} \phi$ that based on inference rules of higher-order logic (HOL [15]) decides the validity of ϕ based on assumptions captured by the environment Γ. We use this judgement to define REF, a refinement type system that relies on LRT to refine types with predicates. That is, an expression e gets a type refined by a predicate ϕ when the predicate $\phi[e/v]$ is decided valid by LRT, as in rule T-Ref.

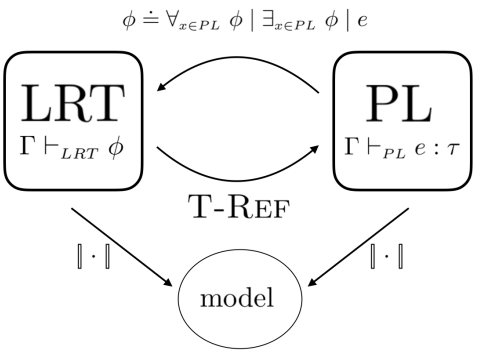

Figure 2: Definition of the logic of refinement types LRT with respect to the programming language PL.

$$\frac{\Gamma \vdash_{PL} e : t \quad \Gamma \vdash_{LRT} \phi[e/v]}{\Gamma \vdash_{REF} e : \{v{:}t \mid \phi\}}\ \text{T-Ref} \qquad \frac{\Gamma \vdash \tau_l \preceq \tau_r}{\Gamma \vdash \{v{:}\tau_l \mid \phi\} \preceq \tau_r}\ \text{T-LRef} \qquad \frac{\Gamma \vdash \tau_l \preceq \tau_r \quad \Gamma, v : \tau_l \vdash_{LRT} \phi}{\Gamma \vdash \tau_l \preceq \{v{:}\tau_r \mid \phi\}}\ \text{T-RRef}$$

The left subtyping rule T-LRef allows weakening, i.e., the refinement ϕ can be ignored as long as it is well formed, while the right subtyping rule T-RRef allows strengthening using LRT to decide validity. These rules (that are similar to RCF [38]) come with the benefit that the logic of REF is LRT which is a variant of HOL and as such provides an ideal framework to study the metatheoretical properties of refinement types. Yet, in REF all refinements are decided solely using higher order inference rules of quantified expressions and not following the structure of the expressions. Thus, type checking in REF (as in RCF [38]) is undecidable and as such cannot be used to define a practical system.

The first goal of this task is to define REF as a sound approximation of the refinement type system CRT. That is, type checking in CRT implied type checking in REF.

Goal 3 (Sound Approximation) *If* $\Gamma \vdash_{CRT} e : \tau$, *then* $\Gamma \vdash_{REF} e : \tau$.

To prove this goal we need to carefully define the inference rules and axioms of LRT to sufficiently encode the reasoning of the (expression-directed) typing rules of CRT. As in research track II the explicit certificates of CRT will help clarify where the calls to LRT are required.

The second goal of this task is to show that the rules of LRT we defined are consistent.

Goal 4 (Consistency of LRT) *There is no proof of* $\emptyset \vdash_{LRT} false$.

To show consistency we will follow the technique of [71, 43] and interpret both LRT and PL to the same consistent model. Concretely, first, we we will pick a consistent mathematical model of PL. Since in this task PL is STLC we can choose a consistent cartesian closed category (CCC) as our model [71], for instance Scott domains [70] or ZFC [48]. Next, we will show that each element of the model satisfies the inference rules and axioms of LRT. These two steps, combined with the known consistency of the mathematical model show that LRT is consistent.

The Goals 4 and 3 combined imply consistency of the CRT typing rules.

Task III.2: Extension of LRT with Refined Data Types

In this task we will extend LRT with refined ADTs. As explained in Task II.1, ADT definitions can be refined to express invariants of data types. These invariants are checked at construction and assumed at destruction of the data type. The checking is easy, data constructors are (injunctive) functions with the refined types provided at the `data` definition. The destruction is more complicated. The typing

rule for case analysis of monomorphic lists in practical refined systems, like CRT, is the following.

$$\frac{\begin{array}{c}\Gamma \vdash l : \{v\texttt{:[Int]} \mid p\} \quad \Gamma, l : \{v\texttt{:[Int]} \mid p \wedge p_{nil}\} \vdash e_{nil} : \tau \\ \Gamma, x : \{x\texttt{:Int} \mid p_x\}, xs : \{xs\texttt{:[Int]} \mid p_{xs}\}, l : \{l\texttt{:[Int]} \mid p \wedge p_{cons}\} \vdash e_{cons} : \tau \\ \Gamma \vdash \texttt{[]} : \{l\texttt{:[Int]} \mid p_{nil}\} \quad \Gamma \vdash \texttt{(:)} : \{x\texttt{:Int} \mid p_x\} \to \{xs\texttt{:[Int]} \mid p_{xs}\} \to \{l\texttt{:[Int]} \mid p_{cons}\}\end{array}}{\Gamma \vdash \texttt{case } l \texttt{ of } \{\texttt{[]} \to e_{nil};\ x : xs \to e_{cons}\} : \tau}\ \text{T-Case}$$

The rule performs case analysis on l to return e_{nil} if l is `[]` and e_{cons} otherwise. Each subexpression is checked in an environment that is appropriately strengthened both by the refinement of l and also by the refinements of the constructors (i.e., p_{nil} and p_{cons}) as retrieved by the environment. This strengthening permits *case sensitivity*, a critical features to practical refinement types. For instance, in the below (left) definition of `head` type checking will deduce that the call to `error` is provably dead code, because it is checked under an environment where `xs` has length both positive (by the precondition) and zero (by the strengthening on the case).

```
head :: xs:{[Int] | 0 < len xs } → Int
head []     = error "provable dead code"
head (x:_) = x
```

```
induction :: l:[Int] → {v:() | 0 ≤ len l }
induction []     = ()
induction (_:xs) = ()
```

The T-Case rule combines two more features. First it permits type weakening at merging point, i.e., the types of the two branches can be different as long as they are both subtypes of the type that `case` return, and it encodes the induction principle on lists. For example, the `induction` function on the right above provides a proof that the length of every list is not negative and the proof goes by case splitting and at each case it is performed in a strengthened environment.

The goal of this task is to extend the LRT with refined data types so that the practical aspects of the rule T-Case are preserved while the induction principle is consistency added to the logic. That is, our extension should preserve the two Goals of Task II.1.

Task III.3: Extension of LRT to System F_ω

The ultimate goal of this task is to extend LRT with type polymorphism (setting PL to System F) and type operators (setting PL to System F_ω) while preserving subtyping and consistency. This goal is very ambitious but it the only way to define and understand the principle logic of practical refinement type systems that support these features, like Liquid Haskell.

The scene for System F is challenging but approachable. System FR [**?**] of `Stainless` formalizes a refined System F, but it does not serve our purposes because the formalization is with respect to evaluation (à la `SAGE`; see a.2), `Stainless` by design does not support semantic subtyping, and does not permit refinement of type variables, i.e., $\{v\texttt{:}\alpha \mid \phi\}$. As [8] highlights refinement of type variables is a common source of unsoundness: if type variables are refined then any type that can instantiate them should also be refined. Otherwise, after instantiation the refinement held by the type variable would get ignored, leading to unsoundness. To circumvent this unsoundness there are three alternatives: 1. no refinements are allowed to type variables [86], which leads to unpractical refinement type systems; 2. all types are refined [8], which opens up the interesting metatheoretical questions about the meaning of refinements in functions, 3. the development of a kind system that separates the type variables that can and cannot be refined [**?**]. The last approach was proposed by the PI but has not yet been formalized. In this task we aim to formalize this approach using the technique of Task II.1. To show consistency (Goal 4) we will experiment with variants of Plotkin and Abadi's logic for parametric polymorphism [64] and its proposed model [12].

The scene for System F_ω is less explored. A refined system for System F_ω and semantic subtyping would require reasoning about polarities as, currently, unsoundly implemented by the PI in Liquid Haskell 2, but never formalized. Our sky-high goal is to define the REF rules that use LRT to reason about type operations with polarities (Goal 3) and show consistency (Goal 4) using System F_ω's categorical model [43].

> **Importance & Challenges**
>
> The formalization of the LRT is of high importance for both the theory and practise 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 have been studied since the '70s and are well understood. Our methodology serves a different purpose than the soundness with respect to operational semantics approach that is now commonly used to formalize practical refinement type systems. It aims to define a consistent, mathematical model for refinement types that would clearly define 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.

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, after facing various unsoundness on these systems, is convinced that the route of classic program logics is an one way street to set the foundations of refinement types. To conduct this research track the PI will collaborate with Michael Greenberg and Alex Kavvos, who are experts in the theory and semantics of programming languages.

b.4 Research Track IV: Implementation and Evaluation on Applications

In this research track we will implement certificate generation in refinement type checking of Haskell to verify web applications (Task IV.1) and Rust to verify cryptographic protocols (Task IV.2).

Task IV.1: Certified Liquid Haskell for Secure Web Applications

In this task we will implement the CRT certificate generation as a back-end to Liquid Haskell and use it to target existing Haskell code. Liquid Haskell is a practical refinement type checker that verifies CoreSyn, GHC's intermediate representation that is a variant of System F_ω. The goal of this task is to extend Liquid Haskell's implementation with the sufficient book-keeping required to generate the certificates of CRT. This extension will be activated using a Liquid Haskell language pragma:

```
{-@ LIQUID "--certified:option" @-}
```

The option will determine if the certificates are validated or tested, as described in research track 1, or a complete translation to Coq is attempted following research track 2. Importantly, when the extension is not activated, there should be no effect in the efficiency of Liquid Haskell.

The extension will be developed early and serve as the play ground to quickly prototype and gradually extend all the research ideas of the proposal. This way all the developed ideas can get immediately applied in real world Haskell code. At the beginning, we plan to target Haskell's `containers` libraries. These libraries have already been verified by Liquid Haskell [84] and translated to Coq via `hs-to-coq` [16], giving us confidence that the certificate generation for these libraries is feasible and also providing a reference against which we can compare our translated Coq code.

Secure Web Application `STORM` [50] is a web framework, verified in Liquid Haskell, that allows developers to build model-view-controller applications with compile-time enforcement of centrally specified data-dependent security policies. `STORM` has been used to develop web secure implementations of real applications including `DISCO` for video conferences and `VOLTRON` for in-class collaboration on programming assignments. Yet, the Liquid Haskell verification of `STORM` is flirting with unsoundness. It uses sophisticated, unformalized features (e.g., contravariant type checking of higher ranked type constructors) that the PI implemented to accommodate `STORM` verification, but are not formalized. The generation of a formal, i.e., certified, verification of `STORM` while keeping all the user-end automation is the goal of this task. On the pursuit of this goal we will apply the Coq generation of research track 2 and the semantic understanding of research track 3 to derive a sound verification of the `STORM` and similar web security frameworks.

Task IV.2: Certified Liquid Rust for Secure Cryptographic Protocols

In this task we will implement certificate generation as a back-end of Liquid Rust, a refinement type checker for Rust programs that the PI has recently started developing with her PhD student following the design principles of practical refinement types described in § a.2. At this early stage, refinement type checking rules are still being developed to accommodate Rust specific features, including memory

management, aliasing and sharing. We expect that the certificate generation can also be adjusted to accommodate these features, yet the concrete adjustments are under active development. In the duration of this proposal, the certificate generation and Liquid Rust projects will, in parallel mature and integrate. This integration plan is important to direct a certificate generation technology that is general and adjustable enough to be used by multiple refinement type checkers.

Secure Cryptographic Protocol Rust combines memory and type safe, reasoning about over- and underflows, and generation of efficient and predictable LLVM code. This unique combination renders Rust the language that cryptographers choose to implement protocols that are not vulnerable to side channel (e.g., time and memory) attacks. For example, `RustCrypto` [95] is a suite of cryptographic algorithms carefully implemented in Rust that is trusted and used for industrial development. But, Rust's type system is not expressive enough to enforce functional safety of the protocols, thus the implementations rely on various runtime assertions. The first milestone of this task is to employ Liquid Rust to statically verify (and thus eliminate) the assertions on `RustCrypto`. The long term goal is to develop formal cryptographic proofs for the `RustCrypto` implementation: the certificate generation back-end of Liquid Rust will generate concrete proof obligations that can be proved employing existing crypto formal proofs (e.g., from`CertiCrypt` [7]). The achievement of this goal would lead to efficient and formally verified cryptographic protocols that are easy to develop.

Importance & Challenges

This research track is critical to ensure that the theoretical developments of the first three tracks are applicable to real software. Task Task IV.1 will ensure, from early stages, that the usefulness of our methodology, while task Task IV.2 will ensure generality. Importantly, this task implements a verification framework where the developer develops real world applications that are automatically verified using SMT and gain formal, machine-checked proofs, delivering the holy grail of practical and sound software verification.

The two tasks come with different set of challenges. The first challenge of Task IV.1 comes by the choice to directly incorporate our novel theoretical developments in Liquid Haskell instead of building a prototype implementation. Both the maintenance cost and the entry cost of new PhD students into a big existing codebase is challenging. The PI has already collaborated with many students and has the experience of developing herself the proper interface to minimize the entry costs. Further, she plans to hire a research programmer that will handle most of the maintenance and engineering requirements of this task. The second challenge of Task IV.1 is that certificate generation for the complete calculus of Liquid Haskell might not be impossible. 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 inherent limitations of the system and thus potential unsoundness.

Task IV.2 is challenging because the typing rules of Liquid Rust are still in a developing state, thus the certificate generation requires careful design. The synergetic development of these projects will encourage the generality of certificate generation and the development of Liquid Rust under novel sound foundations. Further, this task is challenging because it requires deep understanding of refinement types, the Rust compiler internalds, and cryptography. The PI has started this project with her PhD student who is an active contributor of the Rust compiler and Gilles Barthe and Dario Fiore, both cryptography experts.

b.5 Organization

Figure 3 summarizes the gantt chart of CRETE. All the research tracks rely on the development of the certified terms (Task I.1) and will be prototyped as a back-end to Liquid Haskell (Task IV.1), so in the first year we will focus on the proper design and implementation of CRT. The certificate validation (remainder of research track I) will start on the second year and continue until the forth year. The research tracks II and III will start after the first year and will last at least four years. The Liquid Rust project (Task IV.2) has already been initiated and will be developed using the CRETE's sound foundations.

Research Tracks	Year 1				Year 2				Year 3				Year 4				Year 5			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
I: CRT	Task I.1																			
II: TT																				
III: LRT																				
IV.1: CLHaskell																				
IV.2: CLRust																				

Figure 3: Gantt chart of CRETE.

b.6 Conclusion

In summary, CRETE will elaborate programs that refinement type check with explicit proof certificates. These certificates will be independently validated and tested or used to derive sound type-theoretic proofs for the original program. Further, CRETE will define the logic of refinement types, aa higher order logic that approximates reasoning of refined System F_ω with semantic subtyping, setting the principles of refinement type checking. Finally, our approach will be applied to reason about cryptographic protocols in Rust and web security applications in Haskell, evaluating the claim that it achieves both practical and sound verification.

High Risk CRETE is high risk because the derivation of Coq proofs will require careful engineering and potentially critical restrictions on the original system. Similarly, the definition of the logic for refined System F_ω is challenging given the complex of the system. To ameliorate the risk 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. Further, she has often encountered the soundness limitations of refinement types and is certain that they can be addressed using program logics and type theoretical approaches. Finally, IMDEA, without any teaching obligations and with expert colleagues, provides a perfect work environment to carry out such a risky experiment.

High Impact CRETE will impact both the academic and industrial communities. The deeper understanding of the logic of refinement types combined with the explicit certificates will shed new light in the areas of error reporting and inference of type specifications, both open research problems whose solution is critical for usable verification. Importantly, CRETE, via the 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.

References

[1] A. Aguirre, G. Barthe, M. Gaboardi, D. Garg, and P.-Y. Strub. A Relational Logic for Higher-Order Programs. In *International Conference on Functional Programming (ICFP)*, 2017.

[2] A. Ahmed, D. Jamner, J. G. Siek, and P. Wadler. Theorems for Free for Free: Parametricity, with and without Types. In *International Conference on Functional Programming (ICFP)*, 2017.

[3] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust Types for Modular Specification and Verification. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2019.

[4] R. Atkey, P. Johann, and N. Ghani. Refining Inductive Types. In *Logical Methods in Computer Science (LMCS)*, 2012.

[5] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: The spec# experience. *Commun. ACM*, 54(6):81–91, June 2011.

[6] 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.

[7] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella-Béguelin. Formal certification of ElGamal encryption. A gentle introduction to CertiCrypt. In *5th International Workshop on Formal Aspects in Security and Trust, FAST 2008*, volume 5491 of *Lectures Notes in Computer Science*, pages 1–19. Springer, 2009.

[8] J. F. Belo, M. Greenberg, A. Igarashi, and B. C. Pierce. Polymorphic contracts. In G. Barthe, editor, *Programming Languages and Systems*, pages 18–37, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[9] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement Types for Secure Implementation. In *Transactions on Programming Languages and Systems (TOPLAS)*, 2011.

[10] J.-P. Bernardy, P. Jansson, and K. Claessen. Testing polymorphic properties. In *Proceedings of the 19th European Conference on Programming Languages and Systems*, ESOP'10, page 125–144, Berlin, Heidelberg, 2010. Springer-Verlag.

[11] G. Bierman, A. Gordon, C. Hriţcu, and D. Langworthy. Semantic Subtyping with an SMT Solver. In *International Conference on Functional Programming (ICFP)*, 2010.

[12] L. BIRKEDAL and R. E. MØGELBERG. Categorical models for abadi and plotkin's logic for parametricity. *Mathematical Structures in Computer Science*, 15(4):709–772, 2005.

[13] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending sledgehammer with smt solvers. In *Proceedings of the 23rd International Conference on Automated Deduction*, CADE'11, page 116–130, Berlin, Heidelberg, 2011. Springer-Verlag.

[14] 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.

[15] J. Bowen. *Towards Verified Systems*. 01 1993.

[16] 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.

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

[18] P. Choudhury, H. Eades III, R. A. Eisenberg, and S. Weirich. A Graded Dependent Type System with a Usage-Aware Semantics. In *Principles of Programming Languages (POPL)*, 2021.

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

[20] K. Claessen. Shrinking and showing functions: (functional pearl). *SIGPLAN Not.*, 47(12):73–80, Sept. 2012.

[21] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *International Conference on Functional Programming (ICFP)*, 2000.

[22] D. R. Cok. Openjml: software verification for java 7 using jml, openjdk, and eclipse. *arXiv preprint arXiv:1404.6608*, 2014.

[23] T. Coquand and G. Huet. Constructions: A higher order proof system for mechanizing mathematics. In *European Conference on Computer Algebra (EUROCAL)*, 1985.

[24] T. Coquand and G. Huet. The Calculus of Constructions. In *Information and Computation*, 1988.

[25] de Moura L., K. S., A. J., van Doorn F., and von Raumer J. The Lean Theorem Prover. 2015.

[26] 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.

[27] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. *J. ACM*, 52(3):365–473, May 2005.

[28] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: No more scapegoating. *SIGPLAN Not.*, 46(1):215–226, Jan. 2011.

[29] J. Dunfield. *A Unified System of Type Refinements.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2007.

[30] J. Dunfield. Extensible datasort refinements. *ESOP*, abs/1701.02842, 2017.

[31] R. A. Eisenberg and S. Weirich. Dependently Typed Programming with Singletons. In *ACM SIGPLAN Symposium on Haskell (Haskell)*, 2012.

[32] 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.

[33] C. Flanagan. Hybrid Type Checking. In *Principles of Programming Languages (POPL)*, 2006.

[34] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Programming Language Design and Implementation (PLDI)*, 2002.

[35] J. Frankle, P.-M. Osera, D. Walker, and S. Zdancewic. Example-directed synthesis: a type-theoretic interpretation. *ACM SIGPLAN Notices*, 51:802–815, 01 2016.

[36] T. Freeman and F. Pfenning. Refinement Types for ML. In *Programming Language Design and Implementation (PLDI)*, 1991.

[37] J.-Y. Girard. A new constructive logic: classic logic. *Mathematical Structures in Computer Science*, 1(3):255–296, 1991.

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

[39] 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.

[40] J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, pages 93–104, 2006.

[41] 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.

[42] 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.

[43] B. Jacobs. *Categorical Logic and Type Theory.* Elsevier, 1999.

[44] R. Jhala and N. Vazou. Refinement types: A tutorial. Under review, 2020.

[45] 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.

[46] A. M. Kent, D. Kempe, and S. Tobin-Hochstadt. Occurrence typing modulo theories. In *Programming Language Design and Implementation (PLDI)*, 2016.

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

[48] J.-L. Krivine. Typed lambda-calculus in classical zermelo-fraenkel set theory. *Archive for Mathematical Logic*, 40(3):189–205, 2001.

[49] L. Lampropoulos, M. Hicks, and B. C. Pierce. Coverage Guided, Property Based Testing. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2019.

[50] 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.

[51] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, page 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[52] R. Leino and C. Pit-Claudel. Trigger selection strategies to stabilize program verifiers. January 2016.

[53] Y. Liu, J. Parker, P. Redmond, L. Kuper, M. Hicks, and N. Vazou. Verifying Replicated Data Types with Typeclass Refinements in Liquid Haskell. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2020.

[54] W. Lovas and F. Pfenning. Refinement types for logical frameworks and their interpretation as proof irrelevance. *Logical Methods in Computer Science*, 6(4), Dec 2010.

[55] G. Martínez, D. Ahman, V. Dumitrescu, N. Giannarakis, C. Hawblitzel, C. Hritcu, M. Narasimhamurthy, Z. Paraskevopoulou, C. Pit-Claudel, J. Protzenko, T. Ramananandro, A. Rastogi, and N. Swamy. Meta-F*: Proof automation with SMT, tactics, and metaprograms. In *28th European Symposium on Programming (ESOP)*, pages 30–59. Springer, 2019.

[56] 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.

[57] P.-A. Melliès and N. Zeilberger. Functors Are Type Refinement Systems. In *Principles of Programming Languages (POPL)*, 2015.

[58] R. Milner. Models of LCF. *MEMO AIM- 186. Stanford University*, 1973.

[59] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[60] J. Parker, N. Vazou, and M. Hicks. LWeb: Information Flow Security for Multi-Tier Web Applications. In *Principles of Programming Languages (POPL)*, 2019.

[61] L. C. Paulson, T. Nipkow, and M. Wenzel. From LCF to Isabelle/HOL. In *Formal Aspects of Computing*, 2019.

[62] G. Petrou. Verification of Algorithmic Properties in Liquid Haskell, 2018. Diploma Thesis at National Technical University of Athens.

[63] G. Plotkin. LCF considered as a programming language. In *Theoretical Computer Science*, 1977.

[64] G. D. Plotkin and M. Abadi. A logic for parametric polymorphism. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, TLCA '93, page 361–375, Berlin, Heidelberg, 1993. Springer-Verlag.

[65] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan. Formally verified cryptographic web applications in WebAssembly. In *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[66] A. Rastogi, G. Martínez, A. Fromherz, T. Ramananandro, and N. Swamy. Layered indexed effects: Foundations and applications of effectful dependently typed programming, 2020. In submission.

[67] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.

[68] P. Rondon, M. Kawaguchi, and R. Jhala. Low-Level Liquid Types. In *Principles of Programming Languages (POPL)*, 2010.

[69] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in pvs. 1998.

[70] D. S. Scott. Domains for denotational semantics. In *International Colloquium on Automata, Languages, and Programming*, 1982.

[71] D. S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. In *Theoretical Computer Science*, 1993.

[72] E. L. Seidel, N. Vazou, and R. Jhala. Type targeted testing. In *European Symposium on Programming, (ESOP)*, 2015.

[73] T. Sekiyama, Y. Nishida, and A. Igarashi. Manifest Contracts for Datatypes. In S. K. Rajamani and D. Walker, editors, *Principles of Programming Languages (POPL)*, 2015.

[74] F. Smith, D. Walker, and J. Morrisett. Alias types. In *ESOP*, 2000.

[75] 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.

[76] M. Sozeau and C. Mangin. Equations Reloaded: High-Level Dependently-Typed Functional Programming and Proving in Coq. In *International Conference on Functional Programming (ICFP)*, 2019.

[77] A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli. Smt proof checking using a logical framework. *Form. Methods Syst. Des.*, 42(1):91–118, Feb. 2013.

[78] N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *International Conference on Functional Programming (ICFP)*, 2011.

[79] 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.

[80] A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek, and M. Felleisen. Is sound gradual typing dead? *SIGPLAN Not.*, 51(1):456–468, Jan. 2016.

[81] S. Tobin-Hochstadt and M. Felleisen. The Design and Implementation of Typed Scheme. In *Principles of Programming Languages (POPL)*, 2008.

[82] N. Vazou and M. Greenberg. Functional Extensionality for Refinement Types. Under review, 2021.

[83] 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.

[84] N. Vazou, P. Rondon, and R. Jhala. Abstract refinement types. volume 7792, pages 209–228, 03 2013.

[85] 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.

[86] 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.

[87] N. Vazou, E. Tanter, and D. Van Horn. Gradual liquid type inference. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2018.

[88] N. Vazou, A. Tondwalkar, V. Choudhury, R. G. Scott, R. R. Newton, P. Wadler, and R. Jhala. Refinement Reflection: Complete Verification with SMT. In *Principles of Programming Languages (POPL)*, 2017.

[89] P. Wadler. Theorems for free. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, page 347–359, New York, NY, USA, 1989. Association for Computing Machinery.

[90] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, page 1–16, Berlin, Heidelberg, 2009. Springer-Verlag.

[91] A. Westerberg and G. Ung. Comparing Verification of List Functions in LiquidHaskell and Idris, 2019. Degree Project at KTH Royal Institute of Technology.

[92] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, 1998.

[93] N. Zeilberger. Principles of refinement types. In *OPLSS*, 2016.

[94] C. Zenger. Indexed types. *TCS*, 1997.

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

[96] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. Hacl*: A verified modern cryptographic library. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1789–1806. ACM, 2017.