SAND2024-13620

SANDIA REPORT

Printed September 2024

Sandia National Laboratories

The Essence of Cryptol : A Denotational Cryptol Interpreter in Coq for Foundational Assurances for Quantum Resistant Cryptosystems

Zach Sullivan, Philip Johnson-Freyd, and Jon Aytac

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

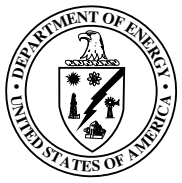
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods



ABSTRACT

Systems of the utmost consequence need a means to establish authenticity of software and data. Cryptosystems implement authentication, but can be vulnerable to cryptographic and implementation attacks. With the threat of quantum cryptographic attacks, "post-quantum" cryptosystems (PQCs) must be henceforth used in these systems. However, the new cryptography needs new ways to, rigorously and machine-checkably, prove systems free of vulnerabilities. We propose a retargetable capability to rapidly instantiate proven correct postquantum cryptosystems through novel proof-carrying synthesis and proof-automation technique, extending those proven successful on existing systems. This capability is crucial to meeting the cryptographic requirements for future high-consequence systems.

Since specifications for high consequence cryptography are presently captured in a domain specific language known as Cryptol. While this can enable convenient fully automated reasoning about Cryptol specificaitons and implementations via the Software Analysis Workbench (SAW), Cryptol has expressivity gaps, so that cryptosystems with probabilistic programming features like Falcon cannot be fully expressed in the language. Moreover, SAW's automation fails for programs and specificaitons with inductive and recursive structure, as in the Sphincs+ PQC. Finally, Cryptol and SAW together represent some 200,000 lines of unverified Haskell, so that the any guarantees about high consequence cryptography are presently contingent on a large, unverified, yet trusted computing base.

The first step of the larger project of agile, assured crpytography is therefore to provide a formal, mechanized semantics for Cryptol, so that the specifications expressed by cryptographers in Cryptol can be reasoned about and compiled into performant implementations with a foundational, machine checkable certificate of correctness.

This report describes our work on this first step, culminating in the design of a certified denotational interpreter, in Coq, for core Cryptol.

This page intentionally left blank.

CONTENTS

This page intentionally left blank.

SUMMARY

The possibly imminent threat of quantum computing demands the deployment of new quantum resistant cryptography to authenticate software and data, both in high consequence systems and in infrastructure. Presently, the specification and assurance of these new cryptosystems relies on Cryptol and SAW.

Cryptol is a domain specific language for expressing cryptography specifications, which features the parametric polymorphism, type constraints, and recursion necessary to specify post quantum cryptosystems like Sphincs+. However, while it is the language of choice for the formalization of cryptography, there is no formalization of its semantics- certainly not one encompassing these key features, and certainly not one which has been mechanized. Morevoer, it suffers some expressivity gaps. The specification of lattice cryptosystems like Falcon require probabilistic programming language features, so much so that key fragments of the Falcon specificaiton are left empty.

The assurance of high consequence cryptogrpahy relies on SAW, a Software Analysis Workbench. However, this analysis comes without any machine checkable certificate of soundness, so the entire enterprise suffers from what Ken Thompson called the problem of trusting trust. Moreover, the automation fails to produce results for recursive specifications, such as those in Sphincs+.

We present, here, a design for a certified, denotational Cryptol interpreter which translates Cryptol programs into the mathematics they describe. In the near term, these mathematical specifications may then be used in machine checkable proofs of security and functional correctness in a full spectrum proof assistant. In the long run, the mechanized semantics developed here can serve as the foundation for certified compilation of Cryptol into machine language or hardware.

This page intentionally left blank.

ACRONYMS & DEFINITIONS

Cryptol a domain specific language for specifying cryptographic algorithms

Coq a full spectrum proof assistant

Denotational Interpreter A program translating a source language into equivalent programs in the host language - in our case, the host language will be the mathematics of domains we've constructed in Coq.

Recursion a recursive definition invokes itself

Parametric Polymorphism Ad hoc polymorphism overloads the definition of an expression or operation to apply to many types, but promises no relationship between the meanings of the operation at different types. Parametric polymorphism defines expressions and functions so that they be applied over at all possble types with essentially the same behavior at all types.

Type Constraints An expression or function carrying a type constraint is applied only at types satisfying the constraint.

Domains A Scott domain is a mathematical model of recursion.

This page intentionally left blank.

1. INTRODUCTION

1.1. Motivation: The Need for Agile and Assured Post Quantum Cryptography

1.1.1. The Needs of the Mission and the State of the Art

Next generation cryptosystems for high consequence applications must be invulnerable against cryptographic and implementation attacks for their often decades-long lifespans. At such timescales, systems must withstand attacks by yet-to-be-realized, crypto-relevant quantum computers. But quantum-resistant algorithms are still in flux. In the past years, new cryptographic attacks were discovered for every NIST-approved PQC algorithm family, dramatically reducing their security guarantees. To control risk, we need a capability to rapidly deploy new, proven-correct cryptography implementations as we respond to emerging challenges from cryptographic attack, threat model, and device physics.

Presently, in most applications, the strongest level of assurance is to validate cryptographic primitives with respect to Cryptol reference implementations. Not only does this formal approach provide higher assurance, it also makes qualification substantially cheaper and faster, as a single formal artifact covers all possible cases. Cryptosystems come in parametric families - e.g., for SPHINCS+, the parameters are the Winternitz parameter, the height of the hypertree, the number of layers, the number of trees, the number of leaves, and the choice of hash function. One of the wise design decisions of the Cryptol language is to support the specification of entire families of cryptosystems via *parametric polymorphism*.

A language *ad hoc polymorphism* allows an expression to have many different implementations for different types, e.g. as in the operator overloading by which plus might have type plus : int → int → int as well as plus : float → float → float. However, ad hoc polymorphism is counterproductive for cryptography. Cryptosystem descriptions have parametric polymorphism, so that their behaviors are essentially the same at all possible parameter types. However, these parameters aren't allowed to be completely arbitrary- there are relations on the parameters, specifying those combinations of parameters for which the cryptosystem can provide some security guarantees. Where Sphincs+ specifications are polymorphic in the following parameters:

n : the security parameter in bytes.

w : the Winternitz parameter

h : the height of the hypertree

d : the number of layers

k : the number of trees in the Forest of Random Subsets.

t : the number of leaves of a Forest of Random Subsets tree.

The parameters are restricted to have specific *kinds* and satisfy specific *type constraints*:

```
parameter
  type h : #
  type constraint (fin h)
  type d : #
  type constraint (d >= 2, h%d == 0, h/d >= 1)
  // the following constraint is needed because
  // h - h/d bits should define a tree address
  type constraint (3*4*8 >= h - h/d)
  type k : #
  type constraint (fin k, k >= 1)
  // instead of t, we consider a = log t as the parameter since
  // it is a that is instantiated.
  type a : #
  type constraint (fin a, a >= 1)
```

Thus a cryptographer can use Cryptol's support for parametric polymorphism and type constraints to specify an entire family of cryptosystems, and use SAW's automation to get almost instant results about correctness properties of the specification, or about the equivalence of an implementation to the specification with some particular choice of parameters.

However, in practice, this means implementations and reference specifications are iteratively refactored in an effort to make their equivalence tractable for as much of the primitive as possible. This iterative, by-hand process dominates the deployment time for new cryptosystems, and prevents agile responses to new challenges. Moreover, it can only prove bit-wise equivalences, and cannot leverage algebraic reasoning to make parametric families of equivalences tractable. Crucially, it can't reason about cryptographic properties.

Worse still, Cryptol's automated reasoning often fails for the correctness and equivalence of recursive programs. Since postquantum cryptosystems are recursive, Cryptol therefore fails precisely for those cryptosystems for which, by virtue of their novelty, complexity, and consequence, the need for formal assurances is most dire, and the only present recourse is to certification of cryptography implementations via the slower, more expensive, incomplete means of testing.

1.1.2. Cryptol Specs for PostQuantum Cryptography and the Shortcomings of Cryptol *and* SAW

1.1.2.1. Sphincs+

Helpfully, Galois has written a Cryptol refrerence specification for the NIST approved Post Quantum Cryptosystems. Focusing on the signature algorithms in particular, we find reference specifications for Spincs+, Kyber – CRYSTALS, and FALCON, expressing not only the abstract, parametric

specifications not only of the cryptographic primitives for these systems, but also some simple correctness properties about these specificaitons.

For instance, a signature algorithm should be able to validate as authentic messages signed with its key (for brevity, let's call this the dogfood property). Cryptol's automation succeeded at proving this simple correctness property for the Winternitz One Time Signature, or wots.

```
wotsCorrectness : [n]Byte -> [n]Byte -> Address -> [n]Byte -> Bit
property wotsCorrectness SKseed PKseed ADRS M = (pk == pk_sig) where
pk = wots_PKgen(SKseed, PKseed, ADRS)
sig = wots_sign(M, SKseed, PKseed, ADRS)
pk_sig = wots_pkFromSig(sig, M, PKseed, ADRS)
```

Remarkably, SAW checks this properly successfully! Sphincs+ is a postquantum cryptosystem for securely signing many messages given one secure wots. Can SAW check the dogfood property for Sphincs+?

```
CorrectnessSPHINCSPlus: ([n]Byte, [n]Byte, [n]Byte) -> Bit
property CorrectnessSPHINCSPlus(SKseed, SKprf, PKseed) = spx_verify`{10}(M, SIG, PK)
where
(SK, PK) = spx_keygen(SKseed, SKprf, PKseed)
M = zero
SIG = spx_sign`{10}(M, SK)
```

In fact, this fails- where the signature validation check passes for the underlying wots, Cryptol – SAW cannot prove the same signature validation correctness property for Sphincs+.

```
# Known Issues
- The Correctness of the overal SHPINCS+ primitive is currently failing
  although correctness of its intermediate WOTS passes.
```

Why does the automation fall over for Sphincs+ when it succeeded for wots? The answer is recursion. Classical hash-based signature algorithms construct signatures for multiple messages through Merkle trees with one-time signature key pairs on their leaf nodes. Moder signature schemes like XMSS and Sphincs+ are both stateless and more efficient, achieving greater cryptographic security with smaller parameters through the use of hypertrees, trees of trees linked together through one-time signatures. As this english language description suggests, hypertrees are highly inductive types, so the algorithm which walks over them and computes signatures is recursive:

```
treehash : ([n]Byte, [32], Integer, [n]Byte, Address) -> [n]Byte
treehash(SKseed, s, z, PKseed, ADRS) =
if z == 0 then
- leaf case
wots_PKgen(SKseed, PKseed, ADRS0)
```

```
else
- internal node case
H(PKseed, ADRS', hashL # hashR)
where
 ADRS0 = setKeyPairAddress(setType(ADRS, WOTS_HASH), toInteger(s))
 ADRS' = setTreeHeight(setTreeIndex(
 setType(ADRS, TREE), toInteger(s >> z)), z)
 z' = z - 1
 hashL = treehash(SKseed, s, z', PKseed, ADRS)
 hashR = treehash(SKseed, (s + (1<<z')), z', PKseed, ADRS)
```

The red term treehash is defined (via the blue terms hashL and hashR) in terms of itself. SAW has difficulty reasoning about general recursion, and so can't prove even the simple dogfood property, let alone reason about the equivalence of an implementation of the Sphincs+ primitive to its specificaiton.

1.1.3. A Certified Denotational Interpreter for Cryptol: Why?

Thus our initial motivation is to circumvent the limitations of SAW by interpreting Cryptol into Coq. Such an interpreter could be used

1. in game-based security proofs to reason about the properties of these specifications,
2. as specifications of cryptographic primitives to be used in program logics to reason about the equivalence of implementations to specifications, and,
3. by the same token as item 2., to ground compiler correctness theorems for certified compilation of Cryptol programs.

observation 1 *Since the role of a program logic is to prove an imperative program implements a function specification, task 2 motives us to choose a* denotational *interpreter, mapping terms* Cryptol *terms into their denotations as mathematics.*

observation 2 *Since* Cryptol *programs like* Sphincs+ *and* FALCON *are recursive, and, so, possibly non-terminating, the category of mathematical objects in which our denotations will reside will be the category of* pointed Scott domains. *As we will briefly recount later, this is the category providing a model of recursion.*

observation 3 *All of these usecases are about equivalences, so, when we say that we would like a* certified Cryptol *compiler, we mean that we would like machine checkable proof certificates about the equivalence of the notion of equivalence implemented in the compiler's reduction semantics to the notion of equivalence implemented in our denotational semantics. But what about our denotational semantics would we like to prove? When we eventually implement a certified* Cryptol *interpreter (in the preceding list, task 3.), we will implement a* reduction *semantics- a collection of rewriting rules describing the operational semantics of compilation. We will then be able to prove*

whether the notions of equality given by this reduction semantics is equivalent to that implemented by our denotational semantics. This amounts to the following three properties about the reduction and denotaitonal semantics:

1. ***Compositionality****: When two terms have equal denotation, they should have equal denotation* when envoked in any well-formed context.
2. ***Soundness****: When the reduction semantics reduces a term to a value, the denotation of the term should equal the denotation of the value.*
3. ***Adequacy****: When a term of type of basekind is equal in denotation to a value, the reduction semantics should run that term to that value.*

1.1.4. A Certified Denotational Interpreter for Cryptol: Why Isn't There Already One?

For the past decade, Cryptol specifications have served as the foundation of the assurance story for high-consequence cryptography. We were surprised that we could find in the literature neither a formalization of Cryptol's semantics nor a denotational interpreter. Galois does have a repository (https://github.com/GaloisInc/cryptol-semantics) implementing a mechanized big step semantics, however it falls short of providng a suitable solution for our purposes. Specifically, they did not solve the following three problems:

1. **They didn't implement a semantics for type constrained parametric polymorphism**. Parametric polymorphism and type constraints are subtle features of the language, and are key to the interpretation of cryptography specifications - and they are not implemented in the reference interpreter.
2. **They didn't implement a semantics for recursive Cryptol**: Since they have committed to a big-Step semantics, their seman- tics can't help us reason about possibly non-terminating, recursive Cryptol programs like Sphincs+.
3. **They did not implement a certified interpreter:** They helpfully implement a big step semantics via eval_type and eval_expr. However, the interpreter in Interp.v is unimplemented, and, so, they provide no certified interpreter. But this is not as grave as the next point.

.

In a sense, it isn't so surprising Galois didn't provide a solution to the above three problems: searching the literature, we could not find a reference solving all three of the above problems! This, in turn, isn't so surprising, since this would require first solving the following technical challenges:

1. Since we would like to run our interpreter, we would like to be able to *extract* from a constructive Coq definition of the denotational interpreter into a *program*. If the terms by which we prove the well-formedness of our interpreter can be, by finitely many rewrites, rewritten to a normal form, than the normalization-by-evaluation lets us extract a runnable program from our proofs. However, our interpreter needs to provide a denotation for possibly non-terminating programs, so our denotational semantics takes values in the category of *pointed*

domains. If we were to define these constructively, this would entail defining coinductive terms with non-terminating rewriting sequences, which are by necessity not normalizable. Thus we must solve a difficult technical problem:

Problem 1 *How can we extract a denotational interpreter of possibly non-terminating programs?*

2. The second difficult technical problem:

   Problem 2 *How can we design a certified denotational interpreter for parametric polymorphism with type constraints?*

   This problem could have been as large as a certified denotational interpreter of SystemFC, that is, the problem of designing a certified denotational interpreter for all of Haskell.

In subsequent chapters we will describe the design decisions which allow us to solve these problems for Cryptol.

2. WHAT IS.... CRYPTOL?

2.0.1. Simplifying Observations on the Core Semantics of Cryptol

Cryptol has three base kinds.

1. A numeric base kind #. Types of this base kind are type level $\mathbb{N}^\star$, the extended natural numbers.
2. a type-like base kind $\star$. Types of this base kind are bits, integers, and rationals.
3. a base kind Prop of propositions. This kind allows Cryptol to express proof irrelevant constraints.

Arbitrary kinds are constructed inductively as functions amongst these kinds. For instance, if you ask Cryptol the type of the type constraint Literal,

```
Cryptol> :type `(Literal)
     ...
    Inferred: # -> * -> Prop
```

Thus Literal is a type constraint constraining the type in the second argument to be the type level extended natural given in the first argument. This has some the surprising upshot that extended naturals are interpreted by Cryptol as qualified type schemes, e.g.

```
Cryptol> :type 2
2 : {a} (Literal 2 a) => a
```

This behavior arises since, given a term, Cryptol's type inference is simultaneously inferring generalizations to polymorphic type schema with type constraints.

```
Cryptol> let twoPlusXY (a, b) = a + ab where ab = a*b
Cryptol> :type twoPlusXY
twoPlusXY : {a} (Ring a) => (a, a) -> a
```

As the extended naturals are type schemes, so too are operations on the naturals. E.g. addition:

```
Cryptol> :type (+)
(+) : {a} (Ring a) => a -> a -> a
```

One can even define constrained type schema parametrized not only by type but by constraint, as in the following example

```
type constraint PredSyn (a : *) (p:Prop) = (Zero a, Cmp a, p)
```

This seems to suggest the possibilty of higher-order constraints! Indeed, type inference on the above definition gives:

```
Main> :type `(PredSyn)
[error] at <interactive>:5:9--5:16:
  Incorrect type form.
    Expected: a numeric type
    Inferred: * -> Prop -> Prop
```

However, the interpreter only allows accepts the parametrizing constraint to be an atomic constraint:

```
someFunc : {a} (PredSyn a (Logic a)) => a
someFunc = ~zero

Main> :type someFunc
someFunc : {a} (Zero a, Cmp a, Logic a) => a
```

Thus we can't actually express arbitrary type constraints- rather, we can only express lists/conjunctions of constraints built from a finite collection of constructors. This dramatically simplifies the situation!

observation 4 *Thus* Cryptol *can be thought of as a language with parametric polymorphic and type constraints, but whose constraints can only be conjunctions/lists of constraints built from a* fixed *collection of constructors.*

Moreover, inspection of the source reveals there is only a fixed collection of derivations of those predicates. That is,

observation 5 *Moreover, we observe that there is only a* fixed *collectionof axioms.*

Not only are the constraints restricted to lists of atomic propositions, the types are also limited:

observation 6 *User defined types can be, at most, records built out of a fixed collection of type constructors, and function of such types.*

We will make all this precise in the subsequent section, but the intuition is that

observation 7 Cryptol *is as if* Haskell *were restricted to a fixed collection of typeclasses, and user defined types were restricted to be type synonyms. That is,* Haskell *with a fixed class context, and with* ***no type level recursion****, only* ***term level recursion****.*

This observation will greatly simplify the denotational interpreter.

2.0.2. Core Cryptol: Syntax

The core language of Cryptol is essentially Haskell but with a fixed class context. **The core Cryptol syntax** is as follows:

$x, y, z \in$	*Variable*		
$\alpha, \beta \in$	*Type Variable*		
$\underline{\alpha}, \underline{\beta} \in$	*Normal Type Variable*		
$\mathsf{A}, \mathsf{B}, \mathsf{C} \in$	*Constructor*		
$\underline{\mathsf{A}}, \underline{\mathsf{B}}, \underline{\mathsf{C}} \in$	*Normal Constructor*		
$\mathsf{L}, \mathsf{L}_N, \mathsf{L}_T \in$	*ConstraintConstructor*		
$n \in$	$\mathbb{N}$		
$\underline{\kappa} \in$	*Base Kind*	$::= \star$	Standard Types
		$\mid \#$	Numeric
$\kappa \in$	*Kind*	$::= \underline{\kappa}$	
		$\mid \underline{\kappa} \to \kappa$	
$\kappa_{\gamma_N} \in$	*Numeric Constraint Kind*	$::= \epsilon \mid \# :: \kappa_{\gamma_N}$	
$\kappa_{\gamma_T} \in$	*Type Constraint Kind*	$::= \epsilon \mid \star :: \epsilon \mid \underline{\kappa} :: \kappa_{\gamma_T}$	
$\kappa_\gamma \in$	*Constraint Kind*	$::= \epsilon :: \underline{\kappa} :: \kappa_\gamma$	
$\tau \in$	*Type*	$::= \mathsf{C}\ \tau_0 \cdots \tau_n$	
		$\mid \alpha$	
$\underline{\tau} \in$	*Normal Type*	$::= \underline{\mathsf{C}}\ \underline{\tau}_0 \cdots \underline{\tau}_n$	
		$\mid \underline{\alpha}$	
$\gamma \in$	*Constraint*	$::= \mathsf{L}_N \tau \mid \mathsf{L}_T \tau$	
$\underline{\gamma} \in$	*Normal Constraint*	$::= \mathsf{L}_N \tau \mid \mathsf{L}_T \underline{\tau}$	
$\rho \in$	*Constrained Type*	$::= \tau \mid \gamma \Rightarrow \rho$	
$\sigma \in$	*Polytypes*	$::= \tau \mid \forall \alpha : \kappa . \sigma$	
$\pi \in$	*Selector*	$::= n \mid x$	
$M, N, L \in$	*Expression*	$::= [M_0, \ldots, M_n] : \tau$	List
		$\mid (M_0, \ldots, M_n)$	Tuple
		$\mid \{x_0 = M_0, \ldots, x_n = M_n\}$	Record
		$\mid M.\pi$	Projection
		$\mid \{M : \tau; \pi = N\}$	Update
		$\mid$ `if` M `then` N `else` L	
		$\mid$ `case` M `of` $\{\mathsf{A}_0 \to M_0, \ldots, \mathsf{A}_n \to M_n\}$	
		$\mid x$	Variable
		$\mid \Lambda \alpha :: \kappa . M$	Type Abstraction
		$\mid M\ \tau$	Type Application
		$\mid \lambda x : \tau . M$	Term Abstraction
		$\mid M\ N$	Term Application
		$\mid M$ where $x = N$	Let binding
		$\mid \mu x . M$	recursion

The normal type constructors mentioned above are as below:

$$
\begin{array}{rcl}
\mathsf{Array} & : & \star \to \star \to \star \\
\mathsf{Bit} & : & \star \\
\mathsf{Float} & : & \# \to \# \to \star \\
\mathsf{Fun} & : & \star \to \star \to \star \\
\mathsf{Inf} & : & \# \\
\mathsf{Integer} & : & \star \\
\mathsf{IntMod} & : & \# \to \star \\
\mathsf{Nat} & : & \mathbb{N} \to \star \\
\mathsf{Rational} & : & \star \\
\mathsf{Seq} & : & \# \to \star \to \star \\
\mathsf{Tuple} & : & \forall n : \mathbb{N}.\mathsf{nKind}\ \mathsf{n} \\
\mathsf{Record} & : & \forall ls : \mathsf{list\ name}, ls \to \mathsf{recordKind}
\end{array}
$$

In the *core* language of Cryptol, **the numeric Type operations** are simply operators on numeric types:

$$
\begin{array}{rcl}
(+) & : & \# \to \# \to \# \\
(-) & : & \# \to \# \to \# \\
(\times) & : & \# \to \# \to \# \\
(/) & : & \# \to \# \to \# \\
(\%) & : & \# \to \# \to \# \\
(\hat{}) & : & \# \to \# \to \# \\
\mathsf{width} & : & \# \to \# \\
\mathsf{min} & : & \# \to \# \to \# \\
\mathsf{max} & : & \# \to \# \to \# \\
\lceil - \rceil \% & : & \# \to \# \to \# \\
\lceil - \rceil / & : & \# \to \# \to \# \\
\mathsf{LenFromThenTo} & : & \# \to \# \to \# \to \#
\end{array}
$$

Of course, these operations are only partially defined on type level extended naturals. We do not introduce in the core language the type constraints restricting them to their proper domain of definition, since those are introduced via the compilation of the library https://github.com/GaloisInc/cryptol/blob/master/lib/C
For instance, for subtraction, we have the constraints:

```
primitive type
  {m : #, n : # }
  (fin n, m >= n) =>
  m - n : #
```

.

As these type declarations are evaluated into the constraint environment Σ (see the reduction rules below), the numeric type operations become suitably constrained type schema.

The Type Constraint Constructors:

$$
\begin{aligned}
\text{Literal} &: \# \to \star \to \text{Prop} \\
\text{LiteralLessThan} &: \# \to \star \to \text{Prop} \\
\text{FLiteral} &: \# \to \# \to \# \to \star \to \text{Prop} \\
\text{Zero} &: \star \to \text{Prop} \\
\text{Logic} &: \star \to \text{Prop} \\
\text{Ring} &: \star \to \text{Prop} \\
\text{Integral} &: \star \to \text{Prop} \\
\text{Field} &: \star \to \text{Prop} \\
\text{Round} &: \star \to \text{Prop} \\
\text{Eq} &: \star \to \text{Prop} \\
\text{Cmp} &: \star \to \text{Prop} \\
\text{SignedCmp} &: \star \to \text{Prop}
\end{aligned}
$$

We observe that, for open prgrams, it should be

numeric constraint constructors

$$
\begin{aligned}
\text{ValidFloat} &: \# \to \# \to \text{Prop} \\
\text{Equal} &: \# \to \# \to \text{Prop} \\
\text{Unequal} &: \# \to \# \to \text{Prop} \\
\text{GreaterThanOrEq} &: \# \to \# \to \text{Prop} \\
\text{Fin} &: \# \to \text{Prop} \\
\text{Prime} &: \$ \to \text{Prop}
\end{aligned}
$$

2.0.3. Core Cryptol: Reduction Rules

$$
\begin{array}{rcll}
\underline{\tau} \in & \textit{NormalMonotype} & := \mathsf{C}_N\ \tau_0 \cdots \tau_n \\
 & & \mid \underline{\alpha} : \star \\
\underline{\gamma_T} \in & \textit{NormalTypeConstraint} & := L_T \underline{\tau} \\
\underline{\gamma_N} \in & \textit{NormalNumericConstraint} & := L_N \underline{\tau} \\
\Delta \in & \textit{KindEnvironments} & ::= \epsilon | \Delta, \alpha : \kappa \\
\Gamma \in & \textit{TypeEnvironments} & ::= \epsilon | \Gamma, x : \tau \\
\Theta \in & \textit{PolyTypeEnvironments} & ::= \epsilon | \Theta, \sigma : \Lambda\alpha : \kappa \\
\Sigma \in & \textit{ConstraintEnvironment} & ::= \epsilon | \Sigma, \underline{\gamma_T}
\end{array}
$$

$$\boxed{\Theta|\Gamma|\Delta|\Sigma \vdash \sigma : \kappa}$$

$$\frac{\alpha : \kappa \in \Delta}{\Theta|\Gamma|\Delta|\Sigma \vdash \alpha : \kappa} \qquad \frac{C : \kappa}{\Theta|\Gamma|\Delta|\Sigma \vdash C : \kappa}$$

$$\frac{\Theta|\Gamma|\Delta|\Sigma \vdash \tau_1 : \kappa_1 \to \kappa_2 \quad \Theta|\Gamma|\Delta \vdash \tau_2 : \kappa_1 \quad \Theta|\Sigma \models \tau_1 @ \tau_2}{\Theta|\Gamma|\Delta|\Sigma \vdash \tau_1 \tau_2 : \kappa}$$

$$(\Rightarrow E)\frac{\Theta|\Gamma|\Delta|\Sigma \vdash \rho : \gamma \Rightarrow \tau \qquad \Theta|\Sigma \models e : \gamma}{\Theta|\Gamma|\Delta|\Sigma \vdash \rho e : \tau}$$

$$(\Rightarrow I)\frac{\Theta|\Gamma|\Delta|\Sigma, e : \gamma \vdash \rho : \tau}{\Theta|\Sigma \vdash \lambda e : \gamma . \rho : \gamma \Rightarrow \tau}$$

$$(\Lambda E)\frac{x \in \Lambda ts.(\llcorner ts) \Rightarrow \tau \quad \Theta|\Sigma \Vdash (\llcorner Ts)}{\Theta|\Gamma|\Delta|\Sigma \vdash x @ Ts = \tau[Ts/ts]}$$

$$(\forall E)\frac{\Theta|\Sigma \vdash \rho : \Lambda t.\tau}{\Theta|\Sigma \vdash \rho\alpha : [\alpha/t]\sigma}$$

$$(\forall I)\frac{\Theta|\Sigma \vdash \rho : \tau \qquad t : \text{freshName}}{\Theta|\Sigma \vdash \lambda t.\rho : \Lambda t.\tau}$$

$$\frac{\Theta, \alpha : \kappa|\Sigma \vdash \sigma : \kappa}{\Theta|\Sigma \vdash \forall \Lambda : \kappa.\sigma : \kappa}$$

$$(\mu)\frac{\Theta|\Gamma, x : \tau|\Delta|\Sigma \vdash M : \tau}{\Theta|\Gamma|\Delta|\Sigma \vdash \mu x.M = [\mu_x.M/x]M : \tau}$$

$$\boxed{\Theta|\Gamma|\Delta|\Sigma \vdash \gamma : \text{Constraint}}$$

$$\frac{\Theta|\Gamma|\Delta|\Sigma \vdash \tau_1 : \kappa_1 \to \kappa_2 \quad P|\Delta : \tau_2 : \kappa_1}{\Theta|\Gamma||\Delta|\Sigma \vdash \tau_1 @ \tau_2 : \text{Constraint}}$$

$$\frac{L : \underline{\kappa_l} \to \text{Constraint} \quad \Theta|\Gamma|\Delta|\Sigma \vdash \tau_l : \kappa_l}{\Theta|\Sigma \vdash L\underline{\tau_l} \to \text{Constraint}}$$

$$\frac{}{\Vdash \text{Zero Bit}} \qquad \frac{}{\Vdash \text{Zero Integer}}$$

$$\frac{\Vdash n : \mathbb{N} \quad \Vdash n \geq 1}{\Vdash \text{Zero IntMod}n} \qquad \frac{}{\Vdash \text{Zero Rational}}$$

$$\frac{\Delta|\Gamma \Vdash e : \text{Type} \quad \Delta|\Gamma \Vdash p : \text{Type} \quad \Sigma|\Theta|\Gamma|\Delta \Vdash \text{psValidFloat } e\ p}{\Sigma|\Theta|\Gamma|\Delta \Vdash \text{Zero (Float } e\ n)} \qquad \frac{\Delta|\Gamma \Vdash a : \text{Type} \quad \Delta|\Gamma \Vdash b : \text{Type} \quad \Sigma|\Theta|\Gamma|\Delta \Vdash \text{pZ}}{\Sigma|\Theta|\Gamma|\Delta \Vdash \text{Zero (Seq } a\ b)}$$

$$\frac{\Delta|\Gamma \Vdash a : \text{Type} \quad \Delta|\Gamma \Vdash b : \text{Type} \quad \Sigma|\Theta|\Gamma|\Delta \Vdash \text{pZero } b}{\Sigma|\Theta|\Gamma|\Delta \Vdash \text{Zero (Fun } a\ b)} \qquad \frac{\Delta|\Gamma \vdash es : [\text{Type}] \quad \Sigma|\Theta|\Gamma|\Delta \Vdash \forall e \in es.\text{pZero } e}{\Sigma|\Theta|\Gamma|\Delta \Vdash \text{Zero (Tuple } es)}$$

$$\frac{}{\Vdash \text{Logic Bit}} \qquad \frac{\Delta \Vdash \underline{\tau_1} : \# \quad \Delta \Vdash \underline{\tau_2} :: \star \quad \Sigma|\text{Logic } \underline{\tau_2}}{\Delta|\Gamma|\Sigma \Vdash (\text{Seq } \underline{\tau_1}\ \underline{\tau_2})}$$

$$\frac{\Delta \Vdash \underline{\tau_1} : \# \quad \Delta \Vdash \underline{\tau_2} : \# \quad \text{Logic } \underline{\tau_2}}{\Vdash \text{Logic } (\underline{\tau_1} \to \underline{\tau_2})}$$

$$\frac{n : \mathbb{N} \quad \Delta|\Gamma|\Theta \Vdash \prod_{\tau s : \{m|m<n\}} (\tau s\ i) : \star \quad \Sigma|\Theta|\Gamma|\Delta \Vdash \forall m < n.\text{Logic}(\tau s\ m)}{\Sigma|\Theta|\Gamma|\Delta \Vdash \text{Logic}(\text{Tuple } n\ \tau s)}$$

$$\frac{\Delta|\Gamma \Vdash ls : \text{listrecord\_field} \quad \Delta|\Gamma \Vdash (\tau s\ ls) : \underline{\tau} \quad \Delta \Vdash \underline{\tau} : \star \quad \Sigma|\Theta|\Gamma|\Delta \Vdash \text{Logic}(\tau s\ ls)}{\Sigma|\Theta|\Gamma|\Delta \Vdash \text{Logic}(\text{Record } ls\ \tau s)}$$

$$\frac{}{\Vdash \text{Ring Integer}}$$

$$\frac{\Delta|\Theta|\Sigma \vdash a : \#\quad \Sigma|\Theta|\Gamma|\Delta \Vdash (\text{Fin}a)\quad \Delta|\Gamma\Theta|\Sigma \Vdash (\text{GreaterThanOrEq } a\ (\text{Nat } 1))}{\Sigma|\Theta|\Gamma|\Delta \Vdash \text{Ring IntMod } a}$$

$$\frac{\Gamma|\Theta \vdash a : \#\quad \Gamma|\Theta \vdash b : \#\quad \Sigma|\Theta|\Gamma|\Delta \Vdash (\text{ValidFloat } a\ b)}{\Sigma|\Theta|\Gamma|\Delta \Vdash \text{Ring } (\text{Float } a\ b)}$$

$$\frac{\Gamma|\Theta \vdash a : \#\quad \Sigma|\Theta|\Gamma|\Delta \Vdash (\text{Fin}a)}{\Sigma|\Theta|\Gamma|\Delta \Vdash \text{Ring } (\text{Seq } a\ \mathit{Bit})}$$

$$\frac{\Gamma|\Theta \Vdash a : \#\quad \Gamma|\Theta \Vdash b : \#\quad \Gamma|\Theta \Vdash a \neq \text{Bit}\quad \Sigma|\Theta|\Gamma|\Delta \Vdash \text{Ring } b}{\Sigma|\Theta|\Gamma|\Delta \Vdash \text{Ring Seq } a \text{ Bit}} \ldots$$

$$\boxed{\Delta \vdash \Sigma}$$

$$\frac{}{\Delta \vdash \epsilon} \qquad \frac{\Delta \vdash \Sigma\quad \Sigma|\Delta \vdash \gamma}{\Delta \vdash \Sigma, \gamma}$$

$$\boxed{\Delta|\Sigma \vdash \Gamma}$$

$$\frac{}{\Sigma|\Delta \vdash \epsilon} \qquad \frac{\Delta|\Sigma \vdash \Gamma\quad \Delta|\Sigma \vdash \sigma : \star}{\Sigma|\Delta \vdash \Gamma, x : \sigma}$$

$$\boxed{\Sigma|\Theta|\Gamma|\Delta : \Gamma \vdash M : \sigma}$$

$$(var)\frac{(x : \sigma) \in \Gamma}{\Sigma|\Theta|\Gamma|\Delta \vdash x : \sigma}$$

$$(where)\frac{\Sigma|\Theta|\Gamma|\Delta \vdash M_1 : \sigma \qquad \Sigma|\Theta|\Gamma, x : \sigma \vdash M_2 : \tau}{\Sigma|\Theta|\Gamma \vdash M_2 \text{ where } x = M_1 : \tau}$$

$$(\rightarrow E)\frac{\Sigma|\Theta|\Gamma|\Delta \vdash M : \tau' \rightarrow \tau \qquad \Sigma|\Theta|\Gamma|\Delta \vdash M_2 : \tau'}{\Sigma|\Theta|\Gamma|\Delta \vdash M_1 M_2 : \tau}$$

$$(\rightarrow I)\frac{\Sigma|\Theta|\Gamma, x : \tau'|\Delta \vdash M : \tau}{\Sigma|\Theta|\Gamma|\Delta \vdash \lambda x : \tau'.M : \tau' \rightarrow \tau}$$

$$(\mu)\frac{\Theta|\Gamma, x : \tau|\Delta|\Sigma \vdash M : \tau}{\Theta|\Gamma|\Delta|\Sigma \vdash \mu x.M = [\mu_x.M/x]M : \tau}$$

3. A DENOTATIONAL CRYPTOL INTERPRETER

A denotational interpreter encodes the interpreted language in the host language. Since our goal is to give a semantics, the host language will be mathematics. Since we want to capture term-level recursion, this will be the mathematics of Scott domains, encoded in Coq in a manner we will describe a bit later.

The point of our efforts is to interpret Cryptol programs into domains in order to reason about their behavior and their behavioral equivalences. If we later have the opportunity to develop a certified Cryptol compiler, the language will finally have a *reduction* semantics, instantiated by a collectino of rewriting rules with some nice properties.[1]

When we say that our future Cryptol compiler will be *certified*, we mean that there will be machine checkable proof certificates that the notion of equivalence implemented in our denotational semantics is equivalent to the notion of equivalence carried by our reduction semantics.

We described Cryptol's grammar in the preceding chapter, and the reduction rules presented there give a transition relation $\longrightarrow$, whose transitive closure we write as $\rightsquigarrow$. In terms of these, writing the denotation implemented in our interpreter as $[\![-]\!]$, we can state the three properties we will show of our future compiler.

1. **Compositionality**: When two terms have equal denotation, they should have equal denotation *when envoked in any well-formed context*,i.e. $[\![M]\!] = [\![M']\!] \Rightarrow \forall [\![\mathsf{C}[M]]\!] = [\![\mathsf{C}[M']]\!]$;
2. **Soundness**: When the reduction semantics reduces a term to a value, the denotation of the term should equal the denotation of the value,i.e. $M \rightsquigarrow V \Rightarrow [\![M]\!] = [\![V]\!]$;
3. **Adequacy**: When a term is equal in denotation to a value, the reduction semantics should run that term to that value, i.e. $\forall \tau : \underline{\kappa}.[\![M]\!] = [\![V]\!] \in [\![\tau]\!] \Rightarrow M \rightsquigarrow V$.

This denotation $[\![-]\!]$ will map the parametrically polymorphic, type constrained Cryptol terms into some target semantic category... but which one?

3.1. Models of Recursion: Domains

We ned a way to model recursive function types. That is, we need objects $\mathbb{D}$ which is isomorphic to its function types $\mathbb{D} \longrightarrow \mathbb{D}$:

$$\mathbb{D} = \mathbb{D} \longrightarrow \mathbb{D}.$$

[1] While Cryptol's general, term-level recursion means there will be a possibly non-terminating fragment of Cryptol, there will be a terminating fragment, for which we may prove strong normalization and confluence.

Where can we find such objects? They can't be sets, since the only set S whose cardinality $\|S\|$ is the same as the cardinality $|S^S| = |S|^{|S|}$ of the set of functions from that set S to itself is the singleton set $S = \{\bullet\}$. If we would like a non-trivial solution, we must look elsewhere.

Definition 1 *$\omega - CPO$ is a set and a partial order on that set:*

$$(S, \sqsubseteq)$$

and

$$\forall \{d_i \in S\}_{i \in \mathbb{N}}$$

such that $\forall i, d_i \sqsubseteq d_{i+1}$ and $\exists U\{d_i\}$ is the least upper bound of that chain.

These are the most basic thing in denotational semantics of recursive languages.

A morphism f

$$f : (S, \sqsubseteq) \to (T, \sqsubseteq')$$

is a function $f : S \to T$ such that $\forall x \sqsubseteq y\ f(x) \sqsubseteq f(y)$ and $\forall \{d_i\}_{i \in \mathbb{N}}$ and $f(U\{d_i\}) = U\{f(d_i)\}_{i \in \mathbb{N}}$. i.e. monotonic and preserves limits.

remark 1 *Its terminal object will be the element with one element. There is only one choice of partial orders. This order on products looks like*

$$(a, b) \sqsubseteq (a', b')$$

iff

$$a \sqsubseteq a' \text{ and } b \sqsubseteq b'.$$

What is the exponential? We order pointwise. That is,

$$f \sqsubseteq g \iff \forall x,\ f(x) \sqsubseteq g(x)$$

So,

$$T, \sqsubseteq^{(S, \sqsubseteq)} = \{f : S \to T \mid f \text{ is Scott-Continuous}\}.$$

These least-upper-bounds (lub)s are preserved, i.e. $U\{f_i\}(x) = U\{f_i(x)\}$.

There exists a non-trivial $\mathbb{D} : \omega - \mathsf{CPO}$ such that $\mathbb{D} \cong \mathbb{D}^{\mathbb{D}}$

Define

$$\mathbb{D}_0 = \{\bot, \top, F\}$$

$$\mathbb{D}_{i+1} = \mathbb{D}_i^{\mathbb{D}_i}$$

This is a chain because we can define

$$up_i : \mathbb{D}_i \to \mathbb{D}_{i+1}$$

$$down_i : \mathbb{D}_{i+1} \to \mathbb{D}_i$$

$$up_0(x) = y \mapsto x$$

$$up_{i+1} = down_i \implies up_i$$

$$down_0(f) = f(\bot)$$

$$down_{i+1}(f) = up_i \implies down_i$$

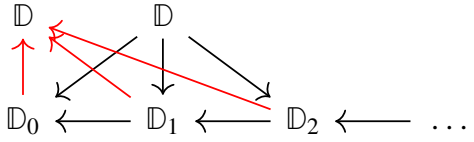
This is like defining a function by a taylor series expansion.

Lemma 1 *$\forall i$, $down_i up_i = \mathrm{id}_{D_i}$ and $up_i \cdot down_i \sqsubseteq \mathrm{id}_{D_{i+1}}$.*

Define a product

$$\mathbb{D} = \Pi_{i \in \mathbb{N}} \mathbb{D}_i \mid \forall i, down(\mathbb{D}_{i+1}) = \mathbb{D}_i.$$

We then get by universal property,



we get that $\mathbb{D} \simeq \mathbb{D}^{\mathbb{D}}$. To capture partiality, we can lift from the category $\omega - \mathsf{CPO}$ to the pointed category $\omega - \mathsf{CPO}_\perp$ by adjoining to each domain a bottom, and by taking as functions bottom preserving Scott continuous functions.

3.2. Making Use of the Simplicity of Cryptol

Giving a semantics to languages with parametric polymorphism and type constraints can be quite involved. The classic work of Wadler and Blott[16] translates constraints into dictionary arguments, while the classic work of Jones[8] translates such a language into a calculus with explicit evidence abstraction and application.

Since Cryptol is essentially Haskell with a fixed class context, type level normalization by evaluation produces normal instances of a polytype/type scheme σ can be given inductively by

$$\lfloor \tau \rfloor = \{\tau\} \tag{3.1}$$

$$\lfloor \gamma \Rightarrow \rho \rfloor = \begin{cases} \lfloor \rho \rfloor & \text{if } \Vdash \gamma \\ \emptyset & \text{otherwise} \end{cases} \tag{3.2}$$

$$\lfloor \forall t.\sigma \rfloor = \bigcup_{\tau \in \text{Normal Type}} \lfloor [\tau/t]\sigma \rfloor \tag{3.3}$$

Such an inductive derivation of normalized constraints is implemented in the constraint$_{\text{d}}$erivation definition found in our core semantics core.v included in the appendix.

The simplifying observations of Cryptol allow our semantic category to be Set indexed collection of $\omega - \mathsf{CPO}_\perp$'s, or **type frames**- that is, a $\mathsf{T}[\![-]\!] : \text{NormalTypes} \to \omega - \mathsf{CPO}_\perp$ such that

1. $\forall \tau \in \text{Normal Type}.\mathsf{T}^{\text{type}}[\![\tau]\!]$ is an interpretation of τ as a non-empty set.
2. $\forall \zeta = \Gamma | \Delta \vdash M : \tau$ and Γ compatible environment η, $\mathsf{T}^{\text{term}}[\![\zeta]\!]\eta$ is the interpretation of M in $\mathsf{T}^{\text{type}}[\![\tau]\!]$.
3. $\forall$ applications of an element of $\tau \to v$ to an element of τ, $\mathsf{T}_{\tau,v} : \mathsf{T}^{\text{type}}[\![\tau \to v]\!] \times \mathsf{T}^{\text{type}}[\![\tau]\!] \to \mathsf{T}^{\text{type}}[\![v]\!]$
4. $\forall f, g \in \mathsf{T}^{\text{type}}[\![\tau \to v]\!]$, if, $\forall x \in \mathsf{T}^{\text{type}}[\![\tau]\!]$, $\mathsf{T}_{\tau,v}(f,x) = \mathsf{T}_{\tau,v}(g,x)$, then $f = g$
5. $\mathsf{T}^{\text{term}}[\![-]\!]$ and $\mathsf{T}_{\tau,v}$ are such that
   a) If ζ derives $\Gamma \vdash x : \tau$, then $\mathsf{T}[\![\zeta]\!]\eta = \eta(x)$
   b) If ζ_{MN} derives $\Gamma \vdash MN : v$, ζ_M derives $\Gamma \vdash M : \tau \to v$, and ζ_N derives $\Gamma \vdash N : \tau$, then $\mathsf{T}[\![\zeta_{MN}]\!]\eta = \mathsf{T}_{\tau,v}(\mathsf{T}[\![\zeta_M]\!]\eta, \mathsf{T}[\![\zeta_N]\!]\eta) = \mathsf{T}[\![\zeta_M]\!](\eta[x \mapsto d])$

Then we can give the semantics of a polytype/type scheme as simply

$$\mathsf{T}^{\text{scheme}}[\![\sigma]\!] = \prod_{\tau \in \lfloor \sigma \rfloor} \mathsf{T}^{\text{type}}[\![\tau]\!] \tag{3.4}$$

The Coq listing implementing the key, difficult part of this approach- the normalization by evaluation- can be found in the appendix.

3.3. Modeling Recursino in Coq Through NonConstructive Reasoning

An entirely constructive definition of the category $\omega - \mathsf{CPO}$ can be given coinductively, as initially described by Capretta[4] and implemented in Coq by Benton[1]. However, the definition of the lifting monad, whose Kleisli category gives the category $\omega - \mathsf{CPO}_\perp$ of pointed domains, requires the use of corecursive, non-terminating terms.

Reasoning about the equivalence of corecursive types in Coq is particularly dangerous, since one can lose subject reduction and so soundness if one isn't careful. Thus we would like an alternative, *nonconstructive* definition of pointed domains.

Of course, we would like more than mere soundness - we would like to make reasoning about prpogram equivalences as simple and tractable as possible. Since Coq has decidable typechecking, definitional equalities are the easiest means of showing equivalences. We can win more definitional equalities if any two proofs of a proposition were equal, that is, if our propositions were *proof irrelevant*. In Coq, the type of propositions, Prop, is inhabited by proofs. Recently, a new type, SProp, was introduced into the type theory, whose inhabitants are *mere propositions*. We've built an entire type system for tracking proof irrelevance, so that we can ensure no dependencies of proof irrelevant properties on proof relevant properties, culminating in the library NonConstructiveDefinitions appearing in our definition of Domains.

```
Record Poset : Type :=     poset_Type :> Type;    poset_Order :: Order poset_Type;
#[export] Instance poset_IsPartialOrder NCR : NonConstructiveReasoning (P : Poset)
  : IsPartialOrder (poset_Order P).
```

THis allows, among other things, more convenient proofs of the Kleene fixpoint theorem

```
  Theorem kleene_fixpoint : forall P : OmegaCpoWithBottom (f : P ->c P),  f (fixpoint
  Proof using NCR.
    intros. unfold fixpoint.
    pose proof (@continuous NCR P P f (Continuous_IsContinuous P P f) (unwind_Monotoni
    simpl in *.
    rewrite -> H.
    assert ((unwind_Monotonic f >>m f) = (add1_Monotonic >>m unwind_Monotonic f)).
          apply equality_of_monotonic. apply functional_extensionality.      intro. si
    symmetry.
    pose proof lub_upper_chain_unchainged P (unwind_Monotonic f).
    simpl in *.
    assumption.
  Qed.
End Fixpoints.
```

This page intentionally left blank.

4. FUTURE WORK

4.0.1. Filling in Any Gaps

In this LDRD, we have formalized core Cryptol, but there remain some important features worth formalizing. Notably, we have not yet provided a semantics for Cryptol's module system. We have forseen the architectural changes needed to accomodating modules - this will involve formalizing the satisfaction of constraints for open Cryptol terms, where constraints become monotonically more satisfiable as the environment instantiating the module become more definite.

We can capture this Kripke monotonicity by wrapping our constraint predicates in the associated modalities. For instance, where we have presently

```
Definition normalize_constraint
  (kindEnv : kind_env)
  (c : top_constraint kindEnv)
  (H : constraint_vars_satisfy (supported_var kindEnv) c)
  : normal_constraint.
```

We should expect

```
Definition normalize_constraint
  (kindEnv : kind_env)
  (c : top_constraint kindEnv)
  (H : constraint_vars_satisfy (supported_var kindEnv) c)
  : possibility_wrapped normal_constraint.
```

where

```
Variant possibility_wrapped (S : Set) : Set :=
   | conditionally : S -> possibility_wrapped S
   | never : possibility_wrapped S
   | always : possibility_wrapped S.
```

4.0.2. A Frontend

We would like to write a frontend which consumes raw Cryptol. Since the Cryptol is total Haskell, we should be able to extract much if our frontend from Cryptol's existing Haskell frontend.

4.0.3. Certifying Compilation

We would like to compile (the terminating fragment of) Cryptol into Coq's Gallina programming language, along with a proof of contextual equivalence via logical relations between Cryptol and Coq programs.

4.0.4. Certified Compilation

The work we have presented here solves a key gap in the assurance story for high consequence cryptography, but, as we will briefly explain below, **high consequence applications may require still more agility**.

Each algorithm comes in a parametric family – e.g., for SPHINCS+, the parameters are the Winternitz parameter, the height of the hypertree, the number of layers, the number of trees, the number of leaves, the security parameter, as well as a choice of hash function. [3] Sets of parameters are selected for different applications based on their security level, itself based on estimates of the time an adversary must invest to defeat the cryptosystem with probability of some lower bound.

Unfortunately, every single candidate has been since shown to be susceptible to new cryptographic attacks reducing, sometimes dramatically, the effective security level for the blessed parameters[2][11][5]. Even SPHINCS+ (the least mathematically adventurous candidate) has been shown to be vulnerable to attack[12]. Thus it becomes crucial to heed NSM-10's "emphasis on cryptographic agility" to "allow for seamless updates for future cryptographic standards" in response to new cryptographic attacks[3][14].

The State of the Art (SOTA) for high-assurance cryptography lacks the parametricity, flexibility, and automation to provide the agility necessary to rapidly re-develop, re-optimize, and re-certify postquantum cryptography in response to the emergence of new challenges from new cryptographic attacks, new threat models, or new device physics, forcing an unacceptable trade off between risk to deployment schedule and to cybersecurity.

We would therefore like to ultimately take entire families of post-quantum cryptosystems, expressed as parametric Cryptol specifications, and compile these specifications into performant, correct-by-construction implementations. This would facillitate an agile response to new mission demands and new threats, compiling implementations which would be deployable and proven correct as soon as specified.

Thus we anticipate the otherwise contradictory needs of cryptographic agility and foundational assurance to be resolved by certified compilation of all of Cryptol into machine languages with formal semantics like CLite or, possibly, some future variant of Koika or Kami.

4.0.5. Probabilistic programming language

FALCON is a Learnnig With Errors (LWE) based post-quantum cryptosystem promising extremely high effective security parameters at lower cost. However, the cryptosystem parameters upon which

the security guarantees most strongly relie are the covariance of a gaussian distribution over a lattice over a finite field. That distribution is produced by a recursive gaussian sampler. Therefore, the equivalence of an implementation to the sampling algorithm is of paramount importance.

However, in Galois's reference Cryptol spec for parametrized FALCON, the specification given for the fast fourier sampler is the zero function:

```
ffSampling : k, n (ispoweroftwo k n) => ([2](FFT n), falconTree k) -> [2](FFT n)
ffSampling(t, T) = z where
    [t0, t1] = t
    z = [z0, z1]
    z0 = zero
    z1 = zero
```

. Thus Cryptol seems to fall short in expressing the LWE based cryptography specifications. An extension of Cryptol with probabilistic programming language features would not only allow the expression of such specifications, but moreover provide a foundation for reasoning about the implementations with respect to their contextual equivalence *as probabilistic programs* to the specification[17]. This would provide a separation of concerns allowing implementations the freedom to implement sampling and conditioning optimally with respect to their given resources, as opposed to imposing implementation details in the specificaiton, as is the case for the reference KYBER – Crystals spec.

4.0.6. Codata types

Another limitation in Cryptol is that it can only express the specifications of *cryptographic primitives*- that is, the mathematical functions underlying the cryptosystem. However, to specify a cryptosystems in general requires, more than this, a specification of a *cryptographic protocol*, a *temporal* specification of the *reactive behavior* of devices participating in the protocol.

Cryptol lacks the expressivity needed to describe the *streams* of behaviors admitted by the protocol. The specification of such streams of behaviors can be expressed, not through *inductive data types*, but through *coinductive codata types*[6]. The addition of codata types to Cryptol would then allow the specification of entire cryptographic protocols.

This page intentionally left blank.

5. CONCLUSION

We've described in this report how we solved the two main technical problems described in the introduction.

- **Problem 3** *How can we design a certified denotational interpreter for parametric polymorphism with type constraints?*

  Solution 1 *We normalize the constraint derivations appearing in* Cryptol *type scheme into a set of derivations of normalized consrtraints, so that the semantics of a polytype is given as a product of type frames over a finite collection of ground types.*

- **Problem 4** *How can we extract a denotational interpreter of possibly non-terminating programs?*

  Solution 2 *To capture the semantics of general recursion, we give a nonconstructive definition of domains, using our own in-house type system for proof irrelevance.*

The ideas behind these solutions are implemented in the software artifacts, core.v and omega − CPO.v we've included in the appendix. Our development implements as software artifacts the lessons we've learned along the way:

1. **The essence of Cryptol**: We've learned quite a bit about what Cryptol is. While this may sound trivial, given the existence of papers and manuals colloquially describing Cryptol, the precise definition of the Cryptol language isn't a settled question, even to experts. There are issues, only a couple years old, in Galois's Cryptol repository, in which the developers uncover surprising behavior from their Cryptol interpreter when presented with higher-props and higher-kinds in their type schemes. Our work codifies the essence of Cryptol.
2. **A Convenient Semantics for Cryptol**: We learned that *the key features* Cryptol features needed for postquantum cryptography, namely, parametric polymorphism, type constraints, and term-level recursion- could nonetheless be given a simple semantics in terms of type frames indexed by a set of ground types.
3. **Convenient Category of Nonconstructive Domains** : We learned how to safely define a nonconstructive theory of pointed domains in Coq, made all the more powerful and convenient through proof irrelevance.

This page intentionally left blank.

REFERENCES

[1] Nick Benton, Andrew Kennedy, and Carsten Varming. Some domain theory and denotational semantics in coq. In *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22*, pages 115–130. Springer, 2009.

[2] Ward Beullens. Breaking rainbow takes a weekend on a laptop. In *Annual International Cryptology Conference*, pages 464–479. Springer, 2022.

[3] Joseph R Biden Jr. Improving the nation's cybersecurity. *Executive order*, 14028, 2021.

[4] Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1, 2005.

[5] Wouter Castryck and Thomas Decru. An efficient key recovery attack on sidh. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 423–447. Springer, 2023.

[6] Paul Downen, Zachary Sullivan, Zena M Ariola, and Simon Peyton Jones. Codata in action. In *European Symposium on Programming*, pages 119–146. Springer International Publishing Cham, 2019.

[7] Galois. A central repository for specifications of cryptographic algorithms in cryptol, 2024. accessed September 2024.

[8] Mark P Jones. *Qualified types: theory and practice*. Number 9. Cambridge University Press, 2003.

[9] Jeffrey R Lewis and Brad Martin. Cryptol: High assurance, retargetable crypto development and validation. In *IEEE Military Communications Conference, 2003. MILCOM 2003.*, volume 2, pages 820–825. IEEE, 2003.

[10] Tamara K. Locke. Guide to preparing SAND reports. Technical report SAND98-0730, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, May 1998.

[11] Daniel Apon Matzov, Daniel J Bernstein, Carl Mitchell, Léo Ducas, Martin Albrecht, and Chris Peikert. Improved dual lattice attack, 2022.

[12] Ray Perlner, John Kelsey, and David Cooper. Breaking category five sphincs+ with sha-256. In *International Conference on Post-Quantum Cryptography*, pages 501–522. Springer, 2022.

[13] Rolf Riesen. How to be conformant. *Psychology Today and Tomorrow*, 784(3):121–130, 2002.

[14] Bruce Schneier. Nist's post-quantum cryptography standards competition. *IEEE Security & Privacy*, 20(5):107–108, 2022.

[15] John E. Smith. On the use of dry erase markers in science. *Journal of Pen and Pencil*, 784(3):121–130, 2002.

[16] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, 1989.

[17] Mitchell Wand, Ryan Culpepper, Theophilos Giannakopoulos, and Andrew Cobb. Contextual equivalence for a probabilistic language with continuous random variables and recursion. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–30, 2018.

DISTRIBUTION

Email—Internal

Name	Org.	Sandia Email Address
Formal Methods Working Group	8741/2	wg-formal-methods-general@sandia.gov
Technical Library	1911	sanddocs@sandia.gov

Hardcopy—Internal

Number of Copies	Name	Org.	Mailstop
1	L. Martin, LDRD Office	1910	0359

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.